UPDATE NOTICE

DECsystem–10/DECSYSTEM–20 Processor Reference Manual

AD–H391A–T1

June 1982

Insert this Update Notice in the *DECsystem–10/DECSYSTEM–20 Processor Reference Manual* to maintain an up-to-date record of changes to the manual.

Changed Information

The changed pages contained in this update package reflect addition of G format floating point, addition of one-word global byte pointer, numerous minor updates and corrections, deletion of special material for TOPS–20 Releases 1 and 2.

The instructions for inserting this update start on the next page.

INSTRUCTIONS
AD–H391A–T1

The following list of page numbers specifies which pages are to be placed in the *DECsystem–10/ DECSYSTEM–20 Processor Reference Manual* as replacements for, or additions to, current pages.

Title page
Copyright page

iii
iv

1–1
1–4

1–9
1–10

1–13
1–18

1–21
1–22.1

1–25
1–28

2–7
2–8

2–13
2–28.6

2–65
2–68.1

2–69
2–70

2–85
2–92.1

2–125
2–126

3–1
3–2.1

3–5
3–6

3–11
3–12

3–17
3–18

3–27
3–28

3–31
3–36

3–39
3–40

3–47
3–56

4–33
4–36

A–1
A–16

A–19
A–20

E–1
E–6

Entire
Index

PLEASE NOTE that the change bars in the outside margin and date printed at the bottom of the page indicate pages where technical information has changed.

KEEP THIS UPDATE NOTICE IN YOUR MANUAL TO MAINTAIN AN UP-TO-DATE RECORD OF CHANGES.

June, 1982

DECsystem–10
DECSYSTEM–20
Processor Reference Manual

AA–H391A–TK, AD–H391A–T1

June 1982

This document explains the machine language programming of the central processors used in the DECsystem–10 and DECSYSTEM–20.

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

Northeast/Mid–Atlantic Region

Digital Equipment Corporation
PO Box CS2008
Nashua, New Hampshire 03061
Telephone:(603)884–6660

Central Region

Digital Equipment Corporation
Accessories and Supplies Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone:(312)640–5612

Western Region

Digital Equipment Corporation
Accessories and Supplies Center
632 Caribbean Drive
Sunnyvale, California 94086
Telephone:(408)734–4915

digital equipment corporation • marlboro, massachusetts

First Edition, May 1968
Second Edition, December 1971
Third Edition, August 1974
Fourth Edition, February 1978
Fifth Edition, July 1980
Updated, June 1982

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

Contents

Preface

Chapter 1 Introduction

Chapter 2 User Operations

Chapter 1
Introduction

A DECsystem–10 or DECSYSTEM–20 is a general purpose, stored program computing system that includes at least one PDP–10 central processor, a memory with error-checking capability, and a variety of peripheral equipment. Each central processor is the control unit for an entire large-scale subsystem, in which it is connected by buses to random access storage modules and peripheral equipment, some of which may be shared with other central processors. Within a given system the central processor governs all peripheral equipment, either directly or indirectly, sequences the program, and performs all arithmetic, logical and data handling operations. But a given system may also contain other kinds of processors. A system based on the KL10 central processor contains a small PDP–11 front end processor; this acts as the system console and may also handle communications equipment and the unit record peripheral equipment via a Unibus. The DECSYSTEM–2020, the only system based on the KS10 processor, contains a microprocessor for handling console functions (with a terminal), and all of its peripheral equipment is handled over two or more Unibuses. Earlier model central processors have manual consoles and handle unit record equipment directly via an in-out bus. A system may also include 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, for control purposes, to some central processor, to which it appears as a peripheral device. The direct-access processor is also connected to its peripheral equipment by a device bus, and to memory either directly by its own memory bus or via a channel bus through the memory control part of the central processor. A DECSYSTEM–2020 cannot include direct access processors, but the Unibus adapters themselves have much of the capability of such processors: in particular an adapter can gain direct access to memory via the same KS10 system bus used by the processor. A system may also contaïn peripheral subsystems, such as for data communications, which are themselves based on small computers; from the point of view of the PDP–11, such a subsystem in toto is regarded as a peripheral device. Unless otherwise specified, the words "processor" and "central processor" refer to the large scale PDP–10 central processor.

At present there are four types of PDP–10 central processors, the KL10, the KS10, the KI10, and the KA10. The first, which exists in two versions, with and without extended addressing, is the fastest and most powerful, having the largest instruction set including string manipulation, double precision in both fixed point and floating point, and in later machines, expanded range floating point. The KS10 lacks expanded range floating point, lacks extended addressing, and is slower than the KL10; but it otherwise has the maximum instruction set, and it is considerably less expensive. All processors handle words of thirty-six bits. Earlier memories store these with a parity bit for detecting single-bit errors. In the newest MOS memories, available with the KL10 and KS10, each word is accompanied by a 7-bit code for correction of single errors and detection of double errors. Maximum memory capacity depends upon the physical addressing capability of the processor. However the physical capacity of the memory is not particularly relevant to a typical user programmer, as all recent processors are structured to operate in a sophisticated virtual memory environment. The fundamental virtual address is thirty bits, although no present processor is capable of using all of them. The virtual memory space is divided into sections of 256K each, whose locations are specified by the right eighteen address bits (the "in-section" address). Paging hardware further divides each section into 512 pages of 512 locations each. The actual size of the virtual address space for a given processor depends on how many out of the twelve possible section bits it implements. The addressing characteristics of the various processors are these.

	Extended KL10	*Single-section KL10*	*KS10*	*KI10*	*KA10*
Physical address (number of bits)	22	22	20	22	18
Physical memory capacity (number of locations)	4096K	4096K	512K	4096K	256K
Section bits implemented	5	0	0	0	0
Number of sections	32	1	1	1	1
Virtual address (number of bits)	23	18	18	18	18
Virtual address space (number of locations)	8192K	256K	256K	256K	256K

In an Extended KL10 whose operating system supports extended addressing only in executive address space, user space is the same as that in a single-section KL10.

The extended KL10, by using five section bits, has a virtual memory twice the size of the maximum physical memory. All other processor configurations currently use only the 18-bit in-section address, so all access is defined as being in section 0. This means that the KS10 has a physical memory that can be twice as large as the virtual space available to a single program; and the single-section KL10 and the KI10 can have a physical memory sixteen times as large. A virtual limitation of 256K is seldom critical however, as these processors, like the extended KL10, have features that allow for dynamic paging and working set management. KA10 memory management is limited to a basic one- or two-part protection and relocation scheme.

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 hold the words. The KL10 can also handle half words, doublewords, bytes, and strings.

> Half words are simply the two halves of a word, wherein the left half is bits 0–17, the right half, bits 18–35. In operations on half words, the two halves of a given word are handled independently; e.g. when both are incremented, no carry from right to left can occur (this is not true on the KA10, where incrementing both halves is done by adding 1000001 to the entire word).
>
> A doubleword is two adjacent words treated as a single 72-bit entity, where the word with the lower address is on the left. In some operations, such as the product in double precision multiplication, this concept is extended to multiple length operands involving more than two consecutive words. The direction from more to less significance is always from lower to higher addresses. (The KA10 cannot handle doublewords, except to the limited extent of double length products and dividends.)
>
> A byte is any contiguous set of bits within a word. It is identified by a byte pointer.
>
> A string is a sequence of bytes packed into and encompassing an arbitrary number of words. It is defined by its length in number of bytes and an initial value for a pointer that is incremented automatically for handling the bytes. (Both KI10 and KA10 lack string hardware.)

Registers specifically for holding addresses have a number of bits appropriate to the type of processor and whether the address is physical or virtual. Address bits are numbered according to the right-justified position of an address in a word. Thus the bits of an in-section address are numbered 18–35, and those of a 22-bit physical address are numbered 14–35. Words are used either as instructions in the program, as addresses, or as operands (data for the program).

Most of this introductory chapter is oriented toward a DECsystem–10 or DECSYSTEM–20 based on a KL10 processor, in both its single-section and extended forms, or a DECSYSTEM–2020, which is based on the KS10 processor. §§1.1 and 1.7 apply only to the KL10, and §§1.2 and 1.8 apply only to the KS10. Much of the information for the KL10 applies also to systems based on the KI10 and KA10. The final section of the chapter explains the ways in which those earlier processors differ from the architecture defined in the preceding sections. §1.3 is probably of interest only to system programmers.

1.1 KL10-based System Organization

The illustrations on the next three pages show the organization of the two types of computer systems based on the KL10 central processor and the internal organization of that processor. A KL10-based system is effectively a group of processors organized around an E or execution bus. The other processors (controllers, interfaces) generally act at the direction of the central processor but carry out those actions independently of it.

On the E bus of a DECSYSTEM–20 there may be up to four DTE20 interfaces, each of which connects to a PDP–11 front end processor, and up to eight RH20 Massbus controllers (Figure 1.1). An RH20 handles disks or tapes via a Massbus; although fundamentally under control of the KL10, the RH20 operates from its own command list in memory and uses a separate C or channel bus for data transfers to and from internal memory via the M box, bypassing the E box. All DECSYSTEM–20 memory is internal: the memory controllers with their storage modules are connected directly to the S or storage bus, and access to them is possible only through the M box.[1] Unit record equipment, such as line printers and card readers, and communication subsystems are handled by PDP–11 front end processors. The data path to memory for these is via the E bus, but it uses automatic features of the priority interrupt, thus interfering minimally with the KL10 program. Among the front end processors, one is master: it acts as the system console, bootstraps the system by loading the KL10 microcode from disk, and is also the system diagnostic facility (for which it has a direct connection to one of the disks on the RH20).

Figure 1.2 shows a typical DECsystem–10 based on a KL10. In terms of memory and peripherals, such a system is much like a KI10-based DECsystem–10, but it has the faster and more powerful central processor. Here external memory is on a KI10 memory bus interfaced to the S bus by a DMA20, and the peripherals are on a KI10 in-out bus interfaced to the E bus by a DIA20. Massbus devices are handled by an RH10, which maintains a direct path to external memory by way of a data channel. Such a system generally has only one front end processor, which acts as the console and diagnostic facility, and bootstraps the microcode from disk or DECtape. One version of the DECsystem–10 is more of a hybrid 10-20: a machine in the 1090 series has KI10 memory and in-out buses, but uses the RH20 Massbus controller, which is right on the E bus and maintains a path to external memory by way of the C bus through the M box.

There are also two versions of the operating system for use with the KL10: the TOPS–20 Monitor and the TOPS–10 Monitor. The Extended KL10 with both user and executive space extended is available only in TOPS–20 systems. In a TOPS–10 system, an Extended KL10 can have extended addressing only in executive space, and for this it must run microcode version 271 or greater (in which case, the TOPS–10 Monitor actually uses so-called "TOPS–20 paging"). In other words an Extended KL10, regardless of Monitor, has TOPS–20 paging; in a single-section KL10 the paging always matches the Monitor.

[1] MOS and core memory cannot be mixed on the same bus. If the system includes both, there must be two S buses.

Figure 1.1: KL10-based DECSYSTEM-20

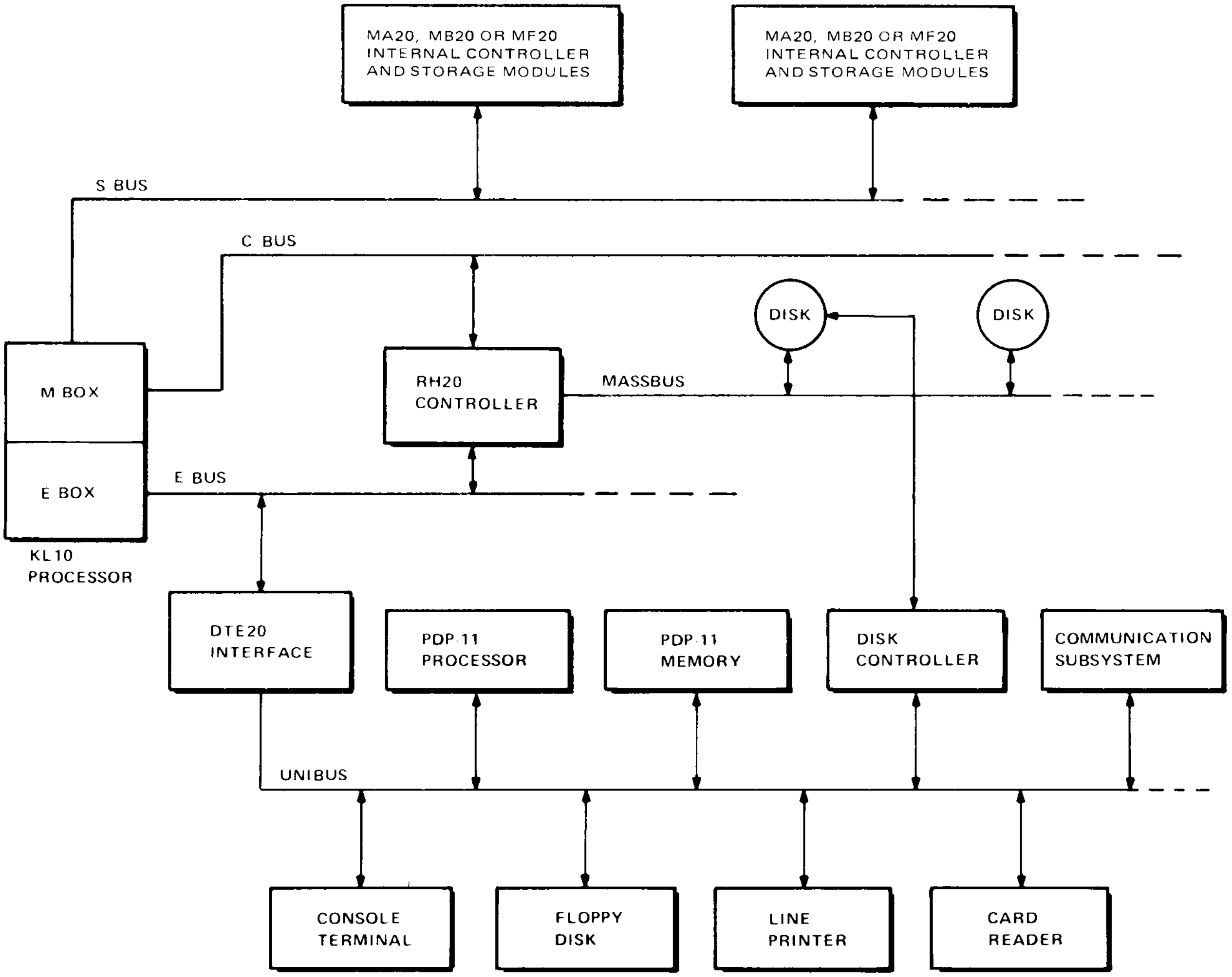


Figure 1.2: KL10-based DECsystem–10

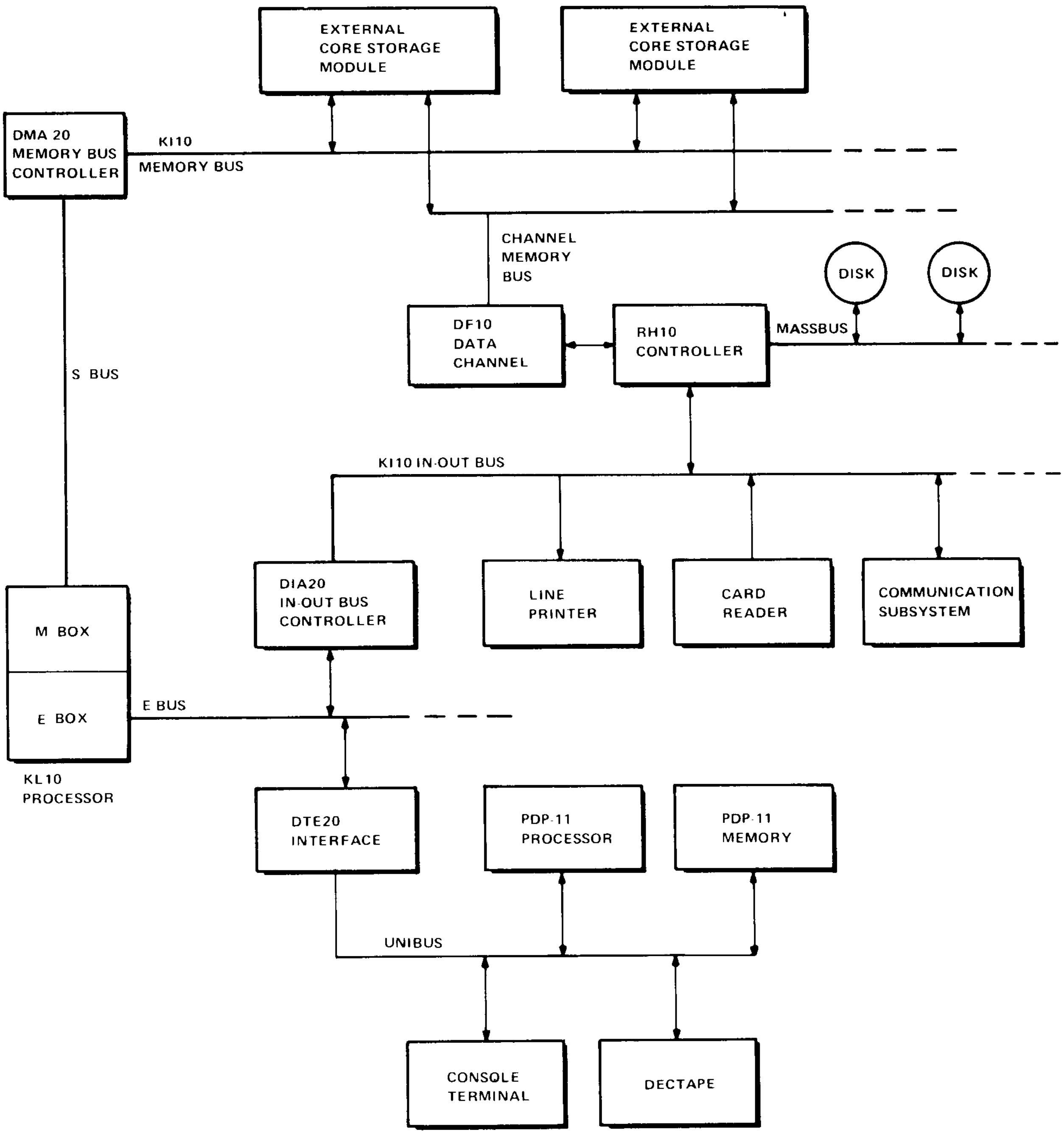
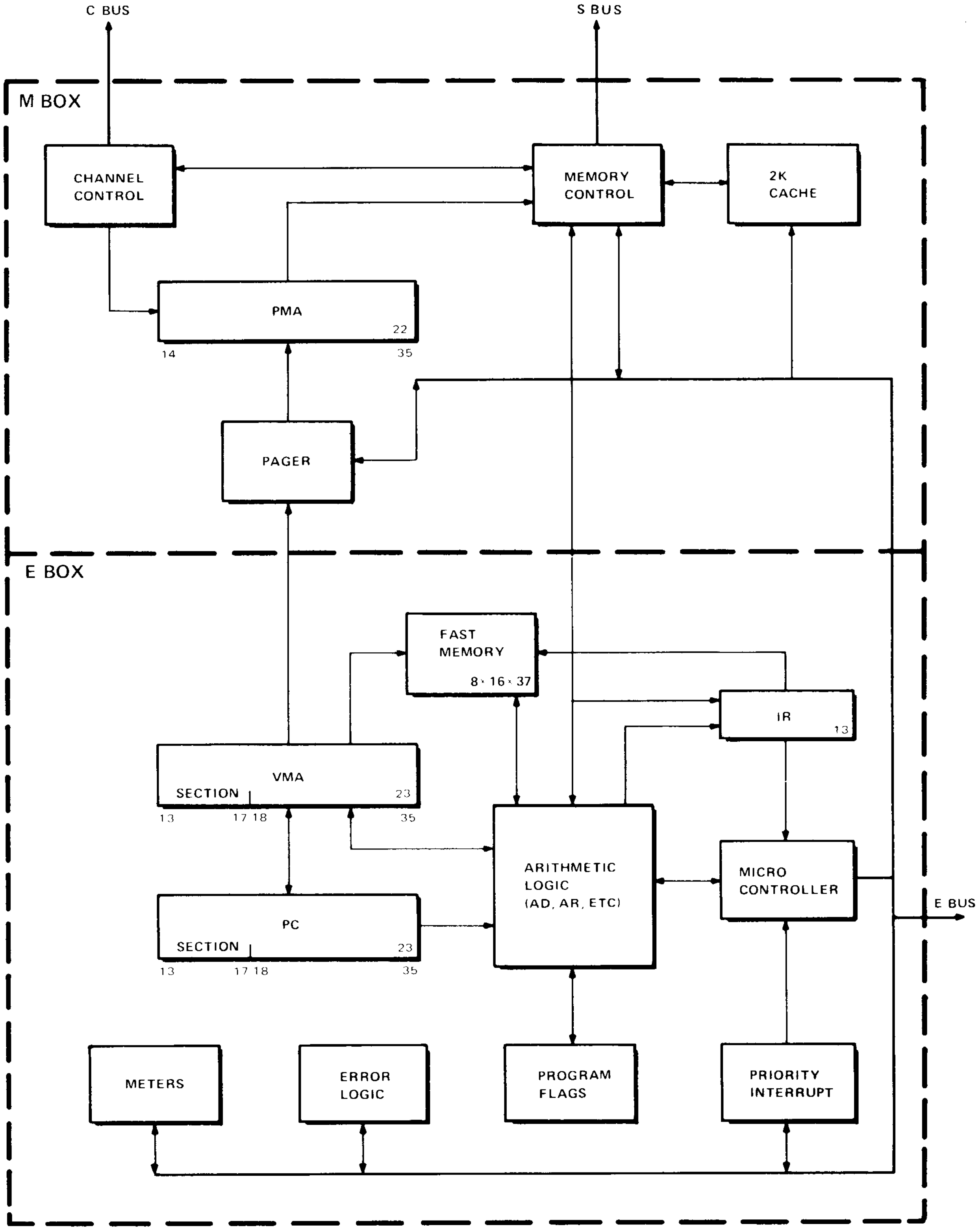


Figure 1.3: KL10 Processor Simplified



The KL10 Processor

Figure 1.3 shows the internal configuration of the KL10 processor. Of the registers shown, only PC, the program counter, is directly relevant to a typical user. The processor performs a program by executing instructions retrieved from the memory locations addressed by PC. For the normal program sequence, PC is regularly incremented by one so that instructions are taken from consecutive locations. 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. Throughout the text the phrase "jump to location *n*" means to load the value *n* into PC, and continue performing instructions in the normal counting sequence beginning at the location then specified by PC. Physically PC is not a counter at all — it just holds the program count, and the actual counting is done in the virtual memory address register VMA. The entire VMA functions as a counter, but no carry is allowed into the section part in program counting. Hence large data structures can arbitrarily cross section boundaries but the program cannot. The program count wraps around in the current PC section, which is specified by PC bits 13–17. For the program to go from one section to another requires an explicit transfer of control by jumping to another section. In a single-section KL10 all section bits are held at zero, so VMA and PC function as 18-bit registers. The virtual address from VMA, whether eighteen bits or twenty-three, is translated by the pager to a 22-bit physical address that is supplied to memory via PMA.

Each instruction retrieved from memory contains information identifying the operands and an instruction code specifying the operation to be performed using those operands. The code goes to the instruction register IR, from which it is decoded by the microcontroller, which in turn performs the instruction by manipulating all of the other E box elements and making the necessary requests to the M box. The microcontroller also executes the more fundamental operations of sequencing the program, handling TOPS–20 paging operations beyond the basic address translation made by the pager (TOPS–10 operations are built into the M box pager), processing interrupts, and so forth. (Not shown in the illustration is a multitude of control lines emanating from the microcontroller and extending throughout the machine.) The microcontroller operates from a microcode contained in a control store. This microcode bears the same relation to the microcontroller as the program does to the processor. But microprocessing is invisible to the programmer, and he need not be concerned with the microcode except to the extent of loading it at system initialization. The reader should however note an important implication of this type of processor implementation: a single KL10 processor can actually be a number of different processors merely by loading different microcodes.

The major working area of the processor is the arithmetic logic. This contains three full-word registers, arithmetic register AR, buffer register BR, and multiplier-quotient register MQ, the first two of which have 36-bit right extensions, ARX and BRX, for handling double length operands. Various combinations of these registers play a role in all arithmetic, logical and data handling operations, and in program control operations as well.

Also included in the arithmetic logic are an extremely fast double length adder AD–ADX, and smaller registers that handle floating point exponents and control shift operations and byte manipulation. But like the microcontroller, the arithmetic logic can be disregarded. Almost all of the operations necessary for the execution of a program are performed in it, but it never retains any information from one instruction to the next. Computations either affect control elements such as PC and the program flags, or produce results that are stored and must be retrieved if they are to be used as operands in other instructions. The program flags detect conditions of interest to the programmer, such as arithmetic and stack overflow, which can cause program traps.

An instruction word has only one 18-bit address field for addressing any location in the current PC section. 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 a memory address (a zero index register address specifies no indexing). Although all computations on both operands and addresses are performed in the arithmetic logic, 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 locations are not actually in the storage modules — they are in a fast 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, and provision is made for referencing them from nonzero sections. Moreover there are actually eight of these fast memory blocks (also referred to as "AC blocks"), but generally only one is available to a program at any given time. Blocks 6 and 7 are reserved specifically for the microcode; the Monitor usually assigns block 1 to user programs and reserves the others for itself.

An optional feature that speeds up memory access and increases the efficiency of storage module use is a cache. This facility has 2048 locations that temporarily substitute for a selection of the most frequently used storage locations. Hence the cache may be regarded in some respects as a set of general purpose registers. A program loop once read from storage and then resident in the cache may be executed hundreds of times without further instruction fetches from storage. Data produced by the program is written in the cache. Thus if the program sets up a location to be a counter, that location may be read and written thousands of times with no storage access, even initially. When the cache is present but does not contain the word the program wants, memory control gets a group of four adjacent words from storage, including the requested one, and places them in the cache, on the assumption the program will probably want the other three and can thus get them more quickly. This is a reasonable assumption, since the program counts sequentially and data manipulation is frequently sequential as well. Cache control has a mechanism for determining frequency of use, and it writes the least recently used word groups back into storage (or discards them if unchanged) when the cache space is needed for new references. The only address restriction on the 512 4-word groups is that the cache can have the same groups from at most four pages. There may be complete pages in the cache, but it is more likely to have a selection of groups from a selection of pages depending on frequency of use. Generally the cache contains words for the current user and for the Monitor, as well as for handling interrupts for many users. The reader should be aware that the cache contains representations of memory word groups, not necessarily the actual storage contents. For example, when the program writes a word, the contents of that cache location then differ from the contents of the corresponding storage location and the other words in the group may not even be in the cache. This caution is of interest however only to the operating system: a typical program simply makes memory references, and the more of these in which the cache substitutes invisibly for storage, so much the better.

Also included within the processor are a number of internal devices that are similar to external controllers in that they operate independently of the program but are controlled by it over the E bus. Some of these have already been mentioned: the program sets up the pager, instructs cache control to update storage, sets up the memory system, and gets diagnostic information from the memory controllers and storage modules. Other such "devices" are the error logic, the meters, and the priority interrupt. By means of the error logic, the program can monitor conditions in the processor. The meters provide a time base, an interval counter, and facilities for keeping track of program use of the system and analyzing system performance. The interrupt facilitates processor control of the entire system by means of a number of priority-ordered levels 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 level or doing some special operation specified by the device (such as incrementing the contents of a memory location). Assignment of levels to devices is entirely under program control. Two of the devices to which the program can assign levels are the error logic and the interval counter.

1.2 KS10-based System Organization

Figures 1.4 and 1.5 show the organization of the newest member of the DECSYSTEM family — the DECSYSTEM–2020 and the KS10 processor used in it. The overall system (Figure 1.4) comprises a number of major units or subsystems that communicate with one another over a bus built into the backplane. The minimal system has five subsystems: processor, MOS storage, console, and two in-out subsystems, each based on a Unibus. One Unibus adapter handles the disk system, the second handles all other peripheral equipment. Depending on the device, these adapters can make direct access to storage or request that the processor handle the transfer via the program. The console, which is based on a microprocessor, boots the system from disk and handles interaction of the operator or a remote diagnostic link with the other subsystems. The backplane bus and most other full word data paths are actually thirty-eight bits, having a parity bit for each half word. The system can run under either the TOPS–20 or TOPS–10 Monitor.

Figure 1.4: DECSYSTEM–2020

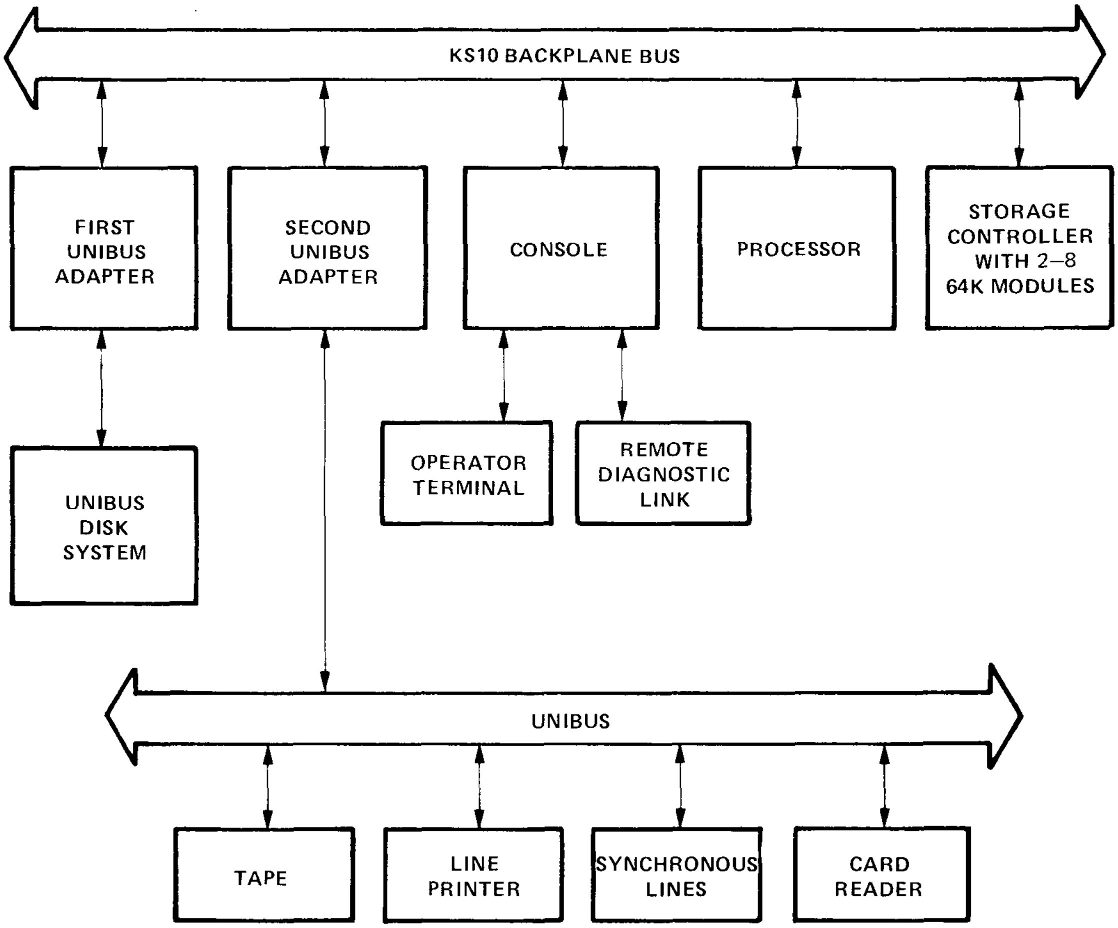
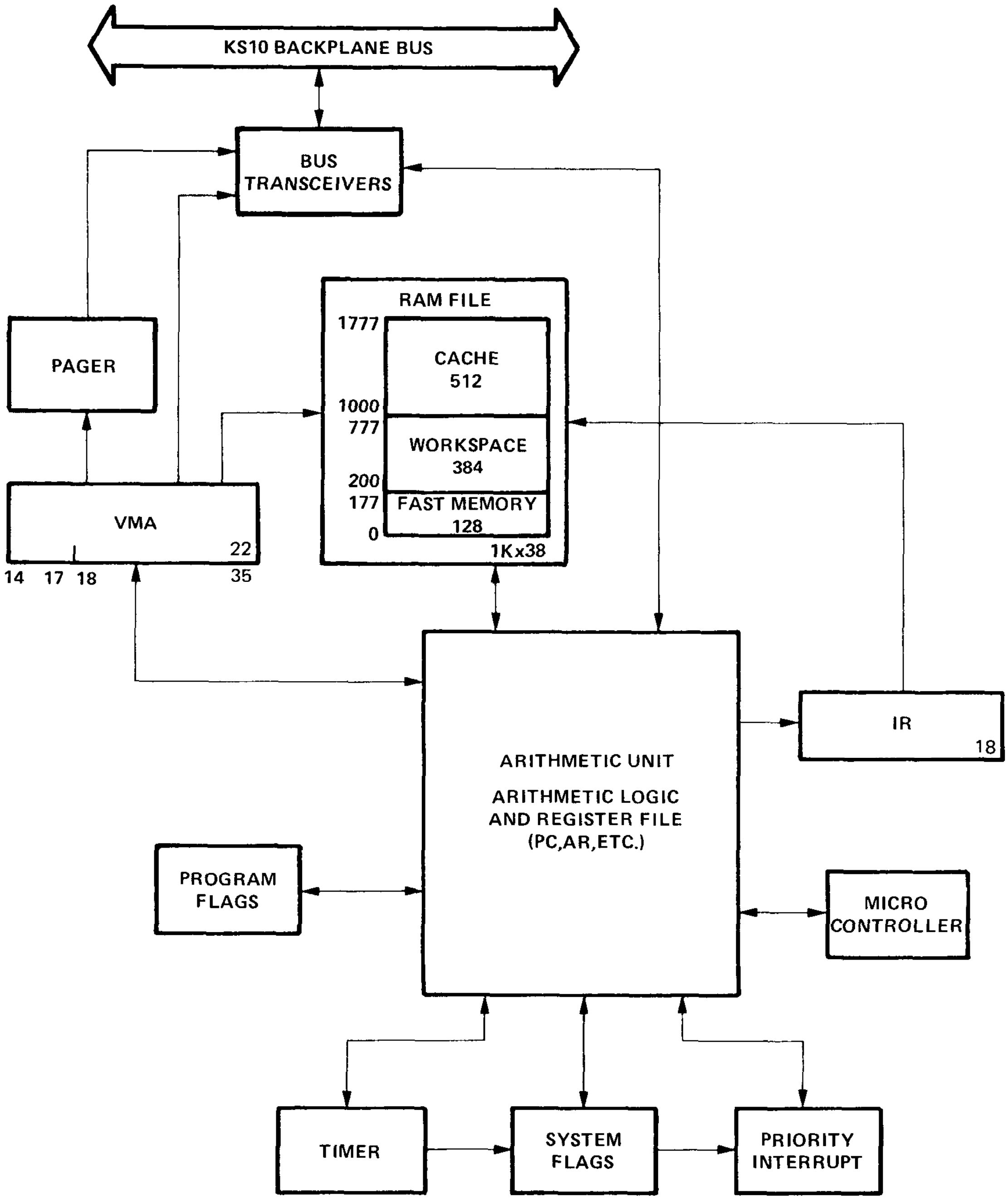


Figure 1.5: KS10 Processor Simplified



Of the elements shown in the processor illustration (Figure 1.5), only fast memory, the program flags, and the program counter PC are directly relevant to a typical user. The processor performs a program by executing instructions retrieved from the memory locations addressed by PC. For the normal program sequence, PC is regularly incremented by one so that instructions are taken from consecutive locations. 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. Throughout the text the phrase "jump to location *n*" means to load the value *n* into PC, and continue performing instructions in the normal counting sequence beginning at the location then specified by PC. Physically PC is not a counter at all — it is a register in the register file (described below). This register just holds the program address, and the actual counting is done by the arithmetic logic, which wraps the count around in eighteen bits because the virtual space is limited to section 0. Addresses from PC, or calculated by the arithmetic logic, go to the virtual memory address register VMA. Each virtual storage address from VMA is translated by the pager to a 20-bit physical address that is supplied to the storage subsystem via the bus. VMA actually has twenty-two bits, for handling not only physical storage addresses, but addresses for other types of bus transactions: with the console, in-out equipment, memory status.

Each instruction retrieved from memory contains information identifying the operands and an instruction code specifying the operation to be performed using those operands. The code goes to the instruction register IR, from which it is decoded by the microcontroller, which in turn performs the instruction by manipulating all of the other processor elements and making the necessary requests for bus transactions. The microcontroller also executes the more fundamental operations of sequencing the program, handling paging operations beyond the basic address translation made by the pager, processing interrupts, and so forth. (Not shown in the illustration is a multitude of control lines emanating from the microcontroller and extending throughout the machine.) The microcontroller operates from a microcode contained in a control store. This microcode bears the same relation to the microcontroller as the program does to the processor. But microprocessing is invisible to the programmer, and he need not be concerned with the microcode except to the extent of loading it at system initialization. The reader should however note an important implication of this type of processor implementation: a single KS10 processor can actually be a number of different processors merely by loading different microcodes.

The major working area of the processor is the arithmetic unit. Central to this unit is a set of ten 4-bit microprocessor slices, which together contain the full word arithmetic logic and a file of ten registers. The register file includes, besides PC, the arithmetic register AR, other associated registers used in manipulating data and performing arithmetic and logical operations, and registers that contain system addresses, status information and constants. The arithmetic logic includes a full word adder, shifter and mixers. It also contains complete 10-bit logic for direct manipulation of floating point exponents and standard 7-bit bytes, and also for controlling shifting and operations on bytes of other sizes. Multiple length operands are handled by separately manipulating their higher and lower order words using the registers in the file. But like the microcontroller, the arithmetic unit (except for PC) can be disregarded by the user. Almost all of the operations necessary for the execution of a program are performed in it, but it never retains any information from one instruction to the next. Computations either affect control elements such as PC and the program flags, or produce results that are stored and must be retrieved if they are to be used as operands in other instructions. The program flags detect conditions of interest to the programmer, such as arithmetic and stack overflow, which can cause program traps. (Several registers in the file do retain information of interest to the system programmer however.)

An instruction word has only one 18-bit address field for addressing any location in the virtual 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 a memory address (a zero index register address specifies no indexing). Although all computations on both operands and addresses are performed in the arithmetic unit, 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 locations are not actually in the storage modules — they are in a fast 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. Moreover there are actually eight of these fast memory blocks (also referred to as "AC blocks"), but generally only one is available to a program at any given time. Block 7 is reserved specifically for the microcode; the Monitor usually reserves block 0 for itself and assigns the others to user programs.

A feature that speeds up memory access and increases the efficiency of storage module use is a virtual cache. This facility has 512 locations that duplicate the contents of storage locations in current use in the virtual address space of the program. Every time a word is read from storage or written in storage, it is also written in the cache location selected by the right nine virtual address bits, which represent position within the virtual page. Provided there is no intervening reference to the same position in some other page, a subsequent read reference to the same virtual location can be made to the cache (referred to as a "cache hit") instead of going over the bus to storage. A program loop once read from storage and then resident in the cache may be executed hundreds of times without further instruction fetches from storage; and data produced by the program can be retrieved without requiring bus transactions. To a great extent the cache is also invisible: a typical program simply makes memory references, and the more of these in which a word is read from the cache instead of storage, so much the better. However a program that tends to settle in one virtual page at a time, instead of alternating references among a number of pages, will maintain a much higher cache hit rate, saving considerable time.

Fast memory and the cache are contained respectively in the bottom 128 and top 512 locations in a RAM file in the processor. The remaining 384 locations are a workspace used by the microcode as a scratch pad, and used for handy storage of various system quantities and constants that expedite the execution of the more complicated instructions. Also included within the processor are several elements, such as the pager already mentioned, that are similar to external controllers in that they operate independently of the program but are controlled by it. The timer provides a time base and an interval counter. By means of the system flags, the program can monitor various conditions throughout the system, and can interrupt the console or be interrupted by it. The interrupt facilitates processor control of the entire system by means of a number of priority-ordered levels 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 level or the source of the request. Assignment of levels is entirely under program control. Two levels can be assigned to each Unibus adapter, and one can be assigned to the system flags.

1.3 Timesharing

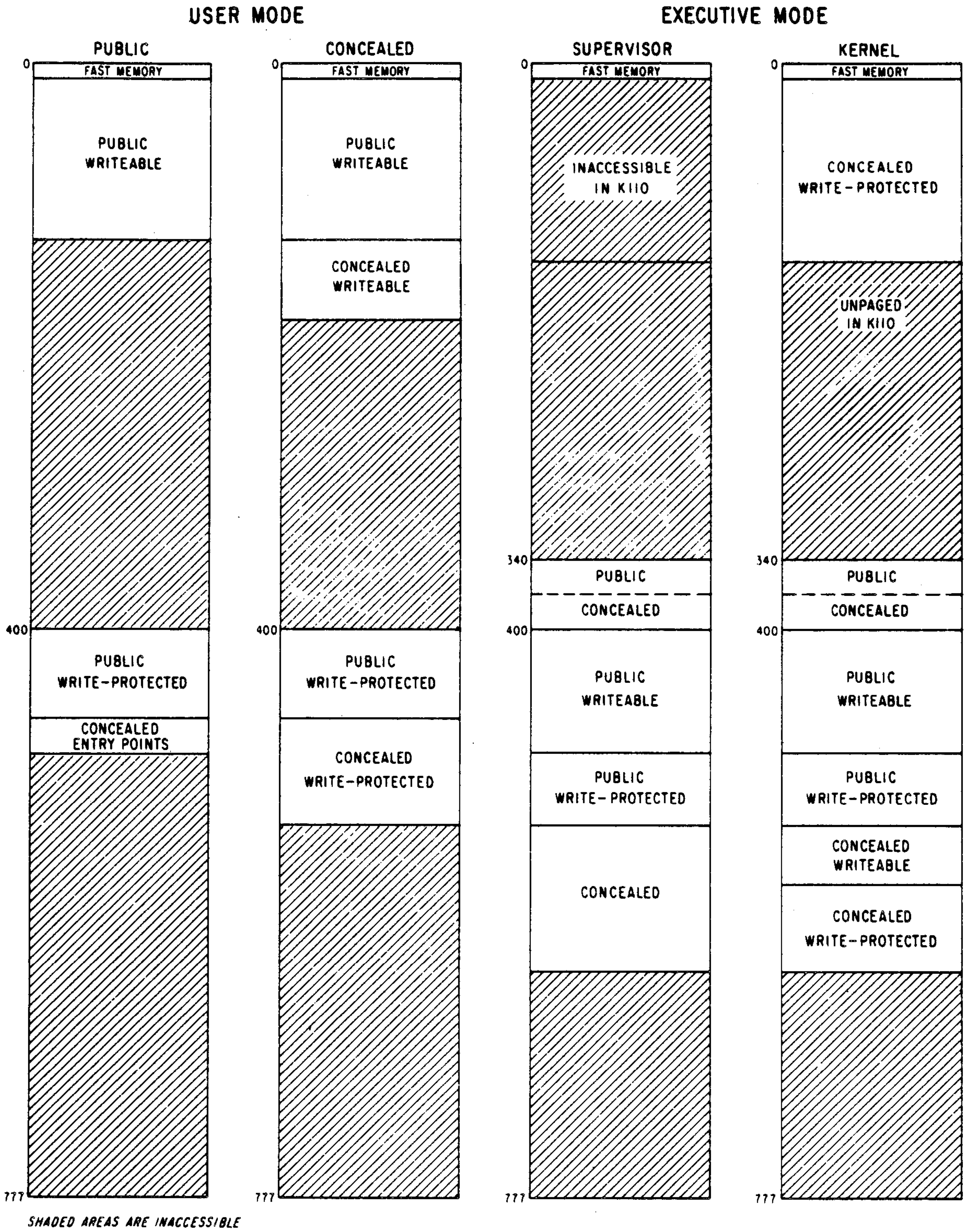
Inherent in the machine hardware are restrictions that apply universally: only certain instructions can be used to respond to a priority interrupt, and certain memory locations have predefined uses. But above this fundamental level, the timeshare hardware provides for different modes of processor operation and establishes certain instruction 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 type of 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 (unrestricted in-out with a limited number of devices is allowed for special real time applications). 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 operation in executive mode.

The paging hardware maps pages from the virtual address space into pages anywhere in physical memory. A page map for each program specifies not only the correspondence from vitrual address to physical address, but also whether an individual virtual page is accessible or not, alterable or not, and whether the cache can be used for references to it. In the KL10 and KI10, both user and executive modes are subdivided according to whether the program is running in a public area or a concealed area; these areas are distinguished by whether or not their pages are labeled public. Within user mode these submodes are public and concealed; within executive mode they are supervisor and kernel. 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 the KS10 all pages may be regarded as concealed, as none are labeled public; but in reality the concept of public *vs* concealed simply does not apply. KS10 executive mode is identical to kernel mode in that supervisor restrictions do not exist. In this treatment of timesharing, any mention of public as against private is irrelevant to the KS10, and functions indicated as being performed by the kernel or supervisor program are all handled by the KS10 executive.

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 turn off the pager to address memory directly, using physical addresses; the address space is then said to be unpaged. In paged address space, individual pages may be restricted as inaccessible or write-protected, but it is the kernel 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 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 affect, even though the locations are not write-protected in kernel mode. The kernel program generally assigns fast memory block 0 for ordinary use by the Monitor in either mode (especially in a TOPS–10 system — to be compatible with the KI10 where the hardware requires it). Typically, the Monitor assigns block 1 to all users and uses blocks 2 and 3 for handling interrupts (e.g. block 2 just for the highest priority level and block 3 for the others).

The most extensive hardware features for timesharing exist in the KL10 and KI10. The reason for this is that the newest software is much more sophisticated and thus requires less hardware to do the job — a fact that the KS10 takes advantage of to cut cost. An example of the use of the most extensive timeshare hardware is illustrated in Figure 1.6. This drawing shows the layout of a single-section KL10 address space that is configured to make full use of the various modes, to be used with a TOPS–10 Monitor, and to be compatible with earlier machines. The space is 256K, made up of 512 pages numbered 0–777 octal. Any program can address locations 0–17 as these are in fast memory and are completely unrestricted (although the same addresses may be in different blocks for different pro-

Figure 1.6: Possible TOPS–10 Virtual Address Space Configuration



grams). The public 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 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. In a TOPS–20 system, an area labeled "write-protected" might better be called "copy on write." Write protection is generally for pure code shared by a number of users. If one user attempts to alter it, the TOPS–20 Monitor will ordinarily make a separate copy for him in his alterable space, and keep the write-protected copy for the remaining users.

In our example write-protected user pages are in the high address half of the space for compatibility with the two-part protection and relocation scheme of the KA10. We define the supervisor program as confined to pages 340 and above, even though there is actually nothing to prevent it from reading that part of the kernel program shown in the lower numbered pages. The reason for specifying it this way is for compatibility with the KI10, where the bottom 112K of executive space is unpaged and accessible only in kernel mode. Part of the executive public area may be write-protected, and even though the supervisor can read concealed information, it cannot change a concealed location whether write-protected or not. For executive concealed areas, the distinction of writable as against write-protected applies only to kernel mode. As in the case of concealed user mode, when the kernel program fetches an instruction from a public area the processor returns to supervisor mode. With TOPS–10 paging, 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.

In executive space of an extended KL10, the interrupt code must be in section 0. The rest of the executive program is usually in section 1, but the two sections are mapped identically, so a given in-section address in either refers to the same physical location. Even with an extended user space, a single-section user program would ordinarily be run in section 0 for compatibility with an unextended space. For the multisection case, the program might be in section 1, special tables in section 2, and a large data structure, such as an immense matrix, might occupy sections 10–12. In terms of instructions implemented and procedures used, the KS10 acts like an extended processor that is confined to section 0.

To manage the system effectively, the Monitor keeps a special table for each process in each processor. These process tables are defined in physical memory; each requires a single page whose whereabouts must be specified by the Monitor, which keeps an executive table for itself and a user table for each user. With TOPS–10 paging, the first half of the table holds the page map for the process; with TOPS–20 paging, the process table contains a table of section pointers to page maps for whatever sections are in use. The hardware defines the use of many other locations in the process tables, especially in the KL10: these include locations that hold trap and interrupt instructions, control blocks for channels and front end processors, and various quantities associated with paging and the meters. Of course in the KS10 there are no control blocks as there are no channels or front end processors; moreover timing information and many of the words associated with paging are kept in the workspace instead of the process tables. Parts of a process table not used by or set aside for the hardware are available to the software. In each user process table the Monitor generally keeps a stack for use with the process, job tables, and various user statistics such as memory space and billing information. In the text the phrase "user process table" refers to the table currently specified by the Monitor as the one for the user even if that user is not currently running.

1.4 Number System

A 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 repertory includes instructions that 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 fixed-point arithmetic instructions use twos complement representations 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 thirty-five 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 $2^n - 1$ (i.e. from all 1s) is equivalent to performing the logical complement, i.e. changing all 0s to 1s and all 1s to 0s. Therefore, to form a 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.

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

0 35

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

0 35

A twos complement addition actually acts as though the words represented 36-bit unsigned numbers, i.e. the signs are treated just like magnitude bits. In the absence of a carry into the sign stage, adding two numbers with the same sign produces a plus sign in the result. The presence of a carry gives a positive answer when the summands have different signs. The result has a minus sign when there is a carry into the sign bit and the summands have the same sign, or the summands have different signs and there is no carry. Thus the program can interpret the numbers processed in fixed point addition and subtraction as signed numbers with thirty-five magnitude bits or as unsigned 36-bit numbers. A computation on signed numbers produces a result that 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.

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 numbers with each sign and zero has a plus sign, there is one more negative number than there are strictly positive numbers (nonzero numbers with a plus sign). This is the most negative number and it cannot be produced by negating any positive number (its octal representation is 400000 000000 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. Single precision 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 zero.

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 made equal to the sign. The range for double length integers and proper fractions is thus -2^{70} to $2^{70} - 1$ and -1 to $1 - 2^{-70}$. The double precision instructions actually use quadruple length numbers for products and dividends. But numbers of any length are just a further extension of the basic format: thirty-five additional bits of the number in each lower order word, and bit 0 made equal to the sign. Remember that truncating a multiple length negative requires an adjustment for the twos complement unless the part discarded is zero. The convention for bit 0 of lower order words is inconsistent with that used for floating point format (see below). This does not affect the arithmetic instructions themselves, as they ignore bit 0 in all lower order words. However instructions that negate a doubleword use the floating point convention. This means that if such instructions are used for fixed point numbers, a problem could arise when comparing one double precision number with another.

Floating Point Numbers

The floating point instructions provide for conversion between fixed and floating forms and handle both single and double precision floating point numbers. The same format is used for a single precision number and the high order word of a standard range double precision number. A floating point instruction that handles numbers with the standard exponent range (available in all machines) 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. For example, for the instruction that scales the exponent, the hardware interprets the integral scale factor in standard twos complement form but produces the correct ones complement result for the exponent.

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

=			
=	0	10 001 000	100 110 010 000 000 000 000 000 000
	0	1 8	9 35

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

=	1	01 110 111	011 001 110 000 000 000 000 000 000
	0	1 8	9 35

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 $1 - 2^{-27}$, about eight significant decimal digits. 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. With either precision the exponent range is –128 to +127, giving a decimal range of approximately 1.5×10^{-39} to 1.7×10^{38}.

The precaution about truncation given for fixed point multiplication applies to single precision floating point operations as they are done in extra length; but the programmer may request rounding, which automatically restores the high order part (the result) to twos complement form if it is negative. In double precision floating point instructions, all operands and results are double length, and all instructions calculate an extra length answer, which is rounded to double length with the appropriate adjustment for a twos complement negative. In 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, or over eighteen decimal digits. Bit 0 of the low order word is made 0 in a result but is ignored in all operands; e.g. the number $2^{18} + 2^{-18}$ has this two-word representation in standard range double precision 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
```

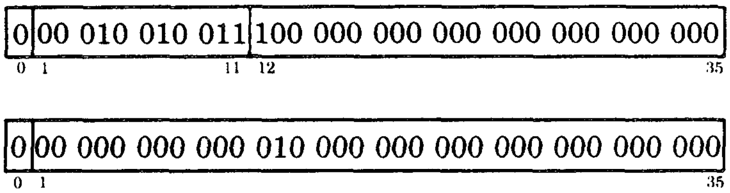
and its negative is

```
|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
```

Expanded Range Floating Point Numbers. Most KL10s have instructions for handling double precision floating point numbers with an expanded exponent range. This is accomplished by using three more bits for the exponent, thus increasing its range by a factor of eight at a cost of losing only one significant decimal digit in precision. Numbers of this type are referred to as being in "G format", for consistency with the VAX terminology (standard range single and double precision floating point correspond to the VAX F and D formats). These instructions are present in any KL10 with microcode version 271 or greater.

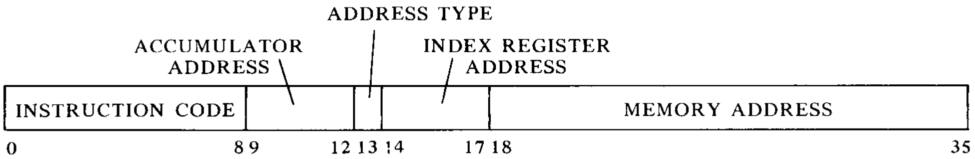
A G format number is like a standard range double precision number except that the high order word contains an 11-bit exponent and only the first twenty-four bits of the fraction. In other words the fraction starts at bit 12, and the contents of bits 1–11 are interpreted as an integral exponent in excess 1024 (2000_8) code. Exponents from −1024 to +1023 are therefore represented by the binary equivalents of 0 to 2047 ($0–3777_8$), resulting in this two-word representation for the number used in the preceding example.



These numbers give a decimal range of approximately 2.8×10^{-309} to 9×10^{307}.

1.5 Instruction Format

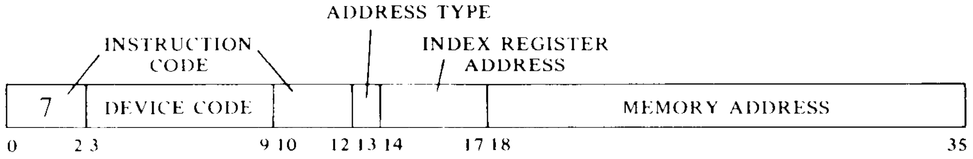
In the basic instruction format, the nine high order bits (0–8) specify the operation, and bits 9–12 address an accumulator. The rest of the instruction word supplies information for calculating the effective address, which is the actual address used to fetch the operand or alter program flow. Bit 13 specifies the type of addressing, bits 14–17 specify an index register for use



BASIC INSTRUCTION FORMAT

in address modification, and the remaining eighteen bits (18–35) address a memory location. In variations on this basic format, bits 9–12 may be used for addressing flags, or all thirteen high order bits (0–12) may be used for an expanded instruction code. The instruction codes that are not assigned as specific instructions are performed by the processor as so-called "unimplemented operations." Among the unimplemented operations are some that are specified as "unimplemented user operations" or UUOs (a mnemonic that means nothing to the assembler). Some of these are for the local use of a program (LUUOs) and some are for communication with the Monitor (MUUOs). In general, unassigned codes act like MUUOs.

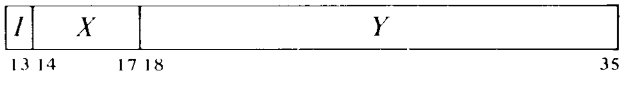
In the KL10 and earlier processors, three 1s in bits 0–2 indicate an input-output instruction, and these instructions have a different format. In all processors from the KS10 on, in-out instructions use the basic format, but for consistency they always do have 1s in the leftmost three bits (there are also non-IO instruction codes beginning with 7). In the IO instruction format used prior to the KS10, 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.



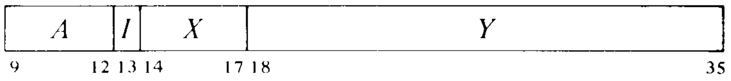
PRE-KS10 IN-OUT INSTRUCTION FORMAT

Of course post-KL10 IO instruction codes are opportunely chosen, so equivalent instructions generally have the same configuration in all processors.

Note that bits 13–35 have the same format in both types of instructions; in fact these bits are the same in every instruction whether it addresses a memory location or not. In the format illustrations throughout the manual this part of an instruction word is shown as



where bit 13 is represented by *I* for "indirect bit," i.e. the address type is either direct or indirect, where the latter is indicated by a 1. For every instruction the processor carries out an effective address calculation that results in a quantity referred to as *E*. This is the effective address of the instruction if indeed it is an address, whether for an operand or a jump. *E* may however be effective conditions, an effective shift, or something else, but the result of the calculation is always referred to as *E*. In illustrations for the basic instructions, bits 9–35 are almost always represented by

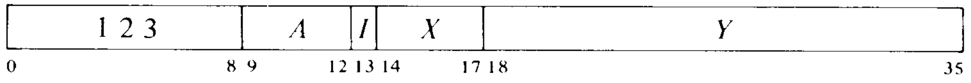


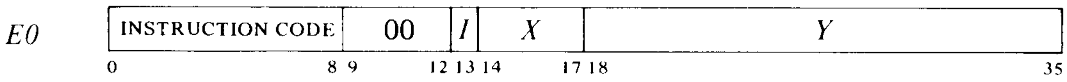
where *A* is the accumulator address.

NOTE

> Although the various parts of an instruction word are always labeled, in some instructions the result of the effective address calculation is not actually used. Unless otherwise specified, in such cases the *I*, *X* and *Y* parts of the word are reserved by Digital for possible future use, and they *must be zero* for compatibility with such use. Similarly when bits 9–12 are not used, they are also reserved and must be zero.
>
> A similar stricture holds for all the formats defined throughout the manual for address words, pointers, and all sorts of special words associated with system features. In words supplied by the program, unassigned bits are available for arbitrary use by the user only if specifically so indicated. Bits labeled "reserved" or simply left blank are reserved to Digital for future use by the hardware or use by the system software. In any word read by the program, unlabeled bits are read as 0s unless there is a specific indication otherwise.

The KL10 and KS10 have a feature that allows expansion of the instruction repertory by an extension of the basic format to two words. In a two-word instruction, it is only the first word that actually appears in the program sequence, i.e. that is referenced by PC; and the accumulator used by the instruction is that specified by the *A* field of the first word. But the instruction the processor actually executes is the second word, and it is found at location *E0*, which is the result of the effective address calculation for the first word. Moreover, the way the processor interprets the instruction code of the second word is entirely different from the way it would if that same word appeared in the program sequence as a one-word instruction. Thus use of a single instruction code in the first word effectively creates a whole new instruction set as large as the one the processor already has. At present there is only one such extended instruction set, and only a small number of the available extended codes are used. In extended instructions the first word is the extend instruction, which has code 123. The format illustrations for these instructions are like this.



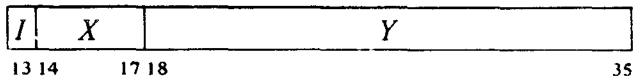


But remember: although the two words are shown together, they never appear one after the other in the program sequence. If they did, the processor might well perform the second word as a standard instruction after executing it as an extended instruction. As with all instructions, before executing the second word the processor calculates an effective address for it; this is referred to as *E1*, and its use depends on the instruction. Bits 9–12 of the second instruction word must be zero for compatibility with possible future use. Unassigned extended instruction codes are executed as MUUOs.

1.6 Effective Address Calculation

At system startup the pager is off, so all addresses are used as physical addresses for memory. In this case of course the program must not give addresses that lie outside the range determined by available memory. Also when the Monitor is setting up page maps, it must select appropriate physical translations. But for a running program, whether user or executive, in any normal circumstances, the relevant memory space is the virtual address space; and all address calculation should be viewed as being in virtual space. This is true even for fast memory, which every program regards as in its virtual space even though fast memory addresses are treated as physical and are not sent to the pager for mapping: instead they are supplied directly to the fast memory from VMA. For our discussion of the effective address calculation let us begin with the simpler case — a virtual space limited to a single section (all quantities eighteen bits).

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 *I* and *X* are both zero, *Y* is *E* — i.e. bits 18–35 contain the effective address. If *X* is nonzero, the contents of the right half of index register *X* are added to *Y* to produce an 18-bit 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 (referred to as an "indirect 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 the indirect bit; the 18-bit number calculated from the *X* and *Y* parts of this location is the effective address *E*. We have taken *Y* to be a memory address, but the program can just as well have an address in the index register, and have the *Y* part of any instruction or indirect word that references it be an offset or displacement. An instruction or indirect word is still an "address word"



even though it may not contain an address; and the quantity in an index register is still called an "index" even when it is an address instead of an offset. Note that throughout the procedure, no computed quantity is ever larger than eighteen bits. In the arithmetic operations overflow is discarded by disabling the carry from bit 18 to bit 17. Hence adding a large offset can be the same as subtracting a small one.

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 instruction 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, an offset for bytes in a string, 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 and no memory reference is required. 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 offset, 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. KS10 IO instructions do not use the result of the effective address calculation; instead they recompute an IO address by a similar procedure (§2.17).

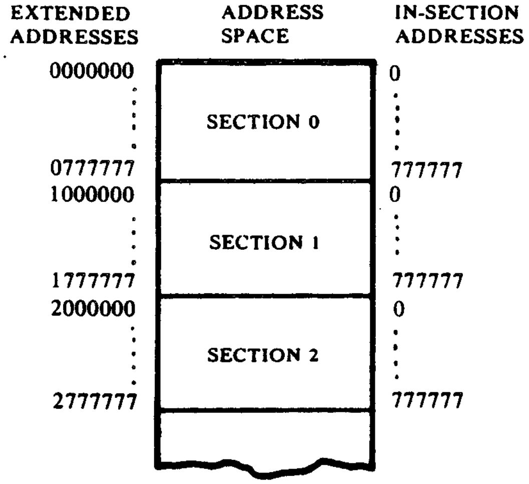
The important thing for the programmer to remember is that the same calculation is carried out for every instruction regardless of the type of information that must be specified for its execution, or even if the result is ignored. In the discussion of any instruction, *E* refers to the actual quantity 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, offset or conditions, or only part of it as in a shift number or scale factor.

PLEASE READ THIS

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

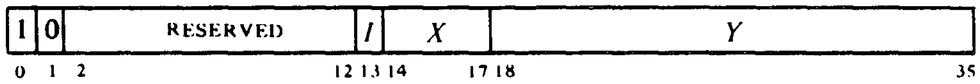
Extended Addresses

As explained at the beginning of this chapter, the address space of an unextended processor is limited to one section, which by definition is section 0. Such processors employ only in-section addresses, as no section number is necessary when there is only one section. In an extended processor the much larger address space is divided into sections of 256K each, and an individual location is identified by an address containing both a section number and an in-section part. There are still many circumstances, however, in which in-section addresses are used alone in an extended processor. The most obvious case is the address given directly by an instruction: this is limited to eighteen bits and is confined to the section from which the instruction is retrieved, being usually the section in which the program is currently running as determined by PC. And, of course, if user space is not extended, all of its addresses are in-section, being in section 0.



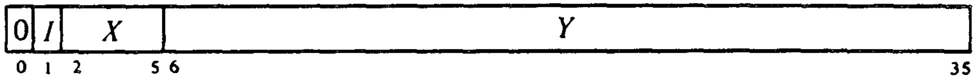
Even in an extended space, an effective address calculation performed in section 0 is done exactly as outlined above, with all addresses and displacements taken as 18-bit quantities contained in bits 18–35 of an instruction word, an index register, or an indirect word. In other words, when a program is running in section 0, *E* can never reference a nonzero section, for either an operand or a jump (although it can reference an operand that supplies an extended address). Moreover in terms of addressing, section 0 of an extended space is entirely compatible with the single section of an unextended space. But in nonzero sections, the effective address calculation can use extended addressing. To understand how extended addressing works, the reader must understand the following terms.

- A *local address* is an 18-bit address. The location it addresses must be in some section, which may be any section, but the section number is supplied by something other than the address.
- A *global address* is a 30-bit address, which therefore supplies its own section number. Of course only the right twenty-three bits (sections 0–37) are meaningful in a KL10 extended address space, but this does not mean that larger section numbers cannot be used for software purposes. In particular section number 7777 is reserved always to trap to the Monitor.
- A *local index* is an 18-bit unsigned displacement or address in bits 18–35 of an index register.
- A *global index* is a 30-bit unsigned displacement or address in bits 6–35 of an index register.
- A *local indirect word* is one containing a local address or displacement in this format.



For obvious reasons, an address word of this sort is also called an "instruction format indirect word". An instruction word is by definition a local address word.

- A *global indirect word* is one containing a global address or displacement in this format.



An address word of this type is also called an "extended format indirect word".

We can now state that an extended effective address calculation is carried out by essentially the same procedure as described above, with index and indirect steps depending on the values of *I* and *X* supplied by a sequence of address words. But now there are differences in the meanings of individual terms and in the way individual operations are performed. First, the indirect bit can be either bit 13 or bit 1 depending on whether it is supplied by a local or global address word (instruction or extended format). Second, there are several varieties of indexing: local and global, with two versions of the latter depending on whether the quantity being indexed is local or global.

- *Local indexing* occurs when the address word is local and either the left half of the index register is negative or the section number part of it (bits 6–17) is zero. In this case the operation is carried out just as in the unextended procedure, and the indexing produces a local address in the section from which the address word is taken (the PC section in the case of an instruction word). Note that this means the program can use local indexing in a nonzero section; and furthermore it can use the left half of the local index register for a control count that counts up through negative numbers to end an iterative process at zero.
- *Global indexing* occurs when the address word is global, or the address word is local but the left half of the index register is positive and bits 6–17 contain a nonzero section number. In either case the result is a global address. When the address word is global, the index is taken as global and is added to *Y* (bits 6–35 of the global indirect word). This is simply a global extension of local indexing: the address word may contain an address and the index register an unsigned offset, or vice versa; adding a large offset can be the same as subtracting a small one. The case where the address word is local is quite unlike local indexing: the index is assumed to be a global address, and the 18-bit *Y* is interpreted as a signed displacement (maximum magnitude 2^{17}), which is added to it algebraically.

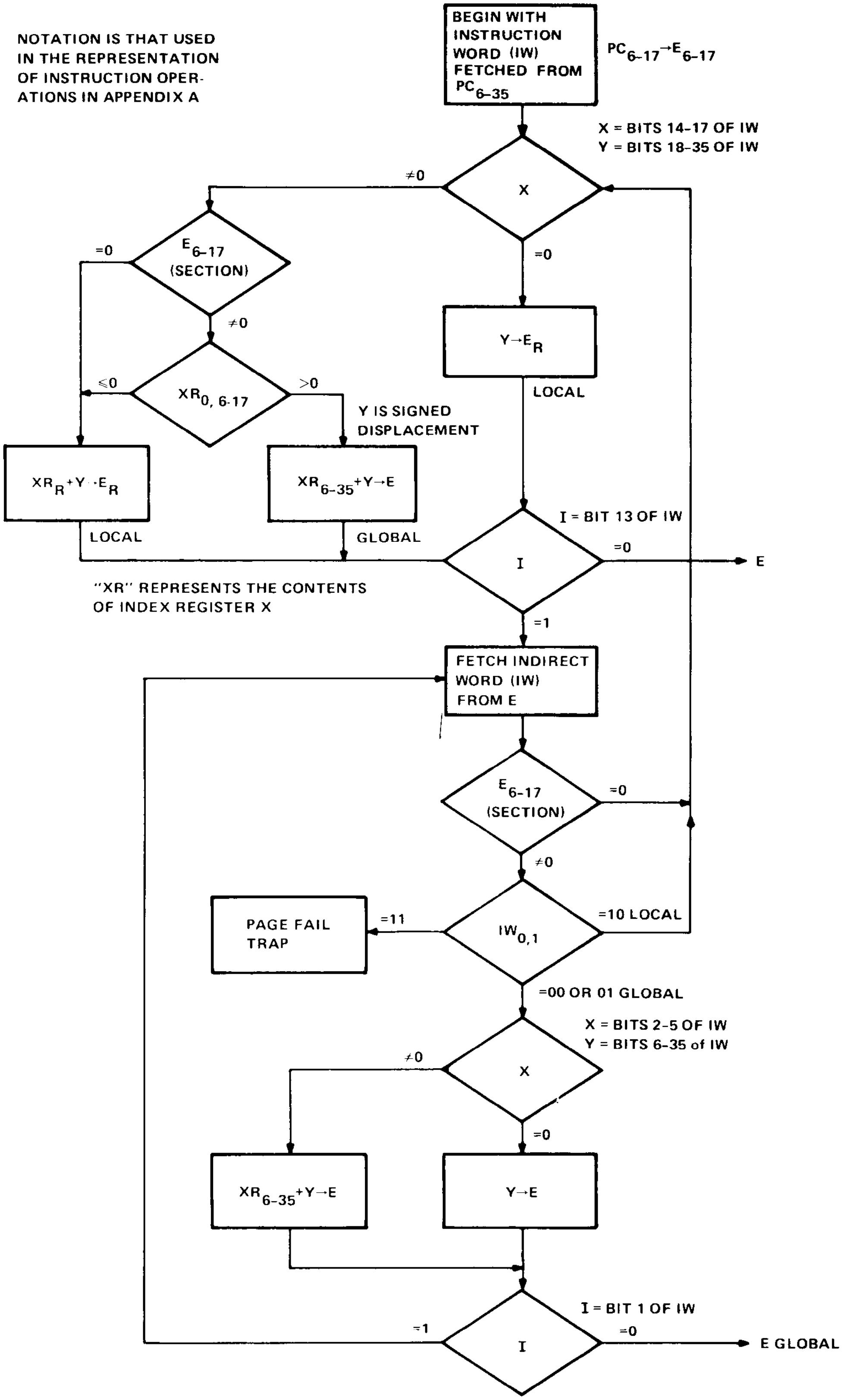
As shown in Figure 1.7, the effective address calculation begins in the section from which the first address word is taken. This is the "local" section for the given address word — the PC section in the case of an instruction word specified by PC. The calculation remains in the local section until the appearance of a global quantity (index or indirect word) changes the section number. So long as only local events occur, all addresses are interpreted as being in the same section (local indexing wraps around 256K). Note that either a local or global address can be used to fetch either a local or global indirect word; but indexing can change only a local quantity to a global one — it cannot modify a global address into a local one. No matter how long the procedure remains local, global indexing or retrieval of a global indirect word can switch to a new section. However if the procedure enters section 0 it can never get out. This is because the calculation then interprets all further quantities as local no matter what their format, i.e. no matter what the program may have meant by the information placed in the words containing them.

At the end if *E* is an address, then either it is a global address or it is interpreted as being in the last section specified. But in an instruction in which *E* is not an address, the section number is ignored and *E* is whatever number of bits is appropriate. In particular an immediate mode operand is always eighteen bits, except in two instructions that specifically handle an extended address as an immediate operand.

The accumulators are regarded as being in the local section of the instruction that addresses them. Hence unless otherwise specified, a local pointer taken from an accumulator addresses a location in the same section as the instruction.

Finally, there is the matter of fast memory reference. An address references a fast memory location if its in-section part is in the range 0–17, and either the address is supplied by PC, the section number is 0, the section number is 1, or the address is local. Note that if PC counts beyond the last in-section address, the wraparound causes instructions to be taken from the ACs. There are two means by which AC references can be made from any section: by using a local address, or by using what is specifically regarded as a global AC address, a section number of 1 combined with a fast memory in-section address.

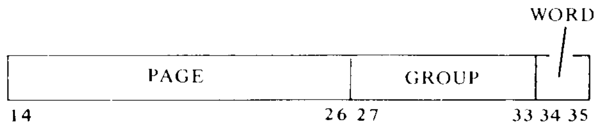
Figure 1.7: Extended Effective Address Calculation



1.7 KL10 Memory

When dealing with storage modules, the processor need not wait the entire memory cycle time. To read, it waits only until the information is available and then continues its operations, whatever the memory must do; to write, it waits only until the data is accepted, and the memory then performs an entire cycle to write that data. 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 storage module that may be needed by some other processor. Such instructions instead request separate read and write access. However, the above considerations apply only when the cache is not in use or is not present, thus requiring that the processor always deal with the storage modules and that it request one word at a time. With the cache in use for a given page, memory access is handled using the cache wherever possible, and when storage access is required, transfers are in 4-word groups. For a read request, the M box reads from the cache if the word is there; otherwise it initiates a storage-to-cache transfer, but this may require a prior cache-to-storage transfer to make room for the new data. For a write request, the M box always writes in the cache, and this too may require a cache-to-storage transfer to make room. Otherwise the M box writes in storage only when the cache is not in use, the Monitor specifically updates memory, or the data is supplied by an internal channel.

For handling storage transfers for a channel or with a cache, the M box interprets physical addresses in this format.



When the E box requests a word that is not in the cache, the M box gets the four words in the group specified by bits 27–33, or more specifically, gets whichever of them are not already in the cache. For the quickest possible service, the M box first gets the particular word requested; e.g. if the program requests word 2 in a group, the M box retrieves word 2 first, followed by words 3, 0, and 1. Even without a cache, channel transfers are always in groups of four, except perhaps for the first or last group in a block. Except with an MF20, the processor further increases the speed of memory operation by overlapping memory cycles: it can start one module to read a word before receiving a word previously requested from a different one. Such speedup is unnecessary with an MF20 as it is four words wide. Of course fast memory and the cache have no basic cycle; with them the processor reads or writes a word directly.

From the simple hardware addressing point of view, the entire physical memory is a set of 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 address is octal 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 storage modules of different capacities. Hence a given address actually selects a particular module and a specific location within it. For a 64K module with 22-bit addressing, the high order six address bits select the module, the remaining sixteen bits address a single location in it; selecting a 32K memory takes seven bits, leaving fifteen for the location. The times given below assume the addressed memory is idle when access is requested. The processor can avoid waiting for its own previously requested memory cycles to end by making consecutive requests to different storage modules. With an MF20 memory almost all transfers are of four words at a time, so there is seldom any conflict among requests. With other memories and provided a cache is in use, ordering requests among modules can be guaranteed by interleaving them in sets of four, in such a way that requests for the words in a group are cycled through the four modules in the set. Interleaving is effected by assigning four modules, each of *n* locations, to the same 4*n*-location area of the address space, and setting each module to respond only to one request out of the four in a group. Hence within the given area, all addresses ending in 0 or 4 are locations in one module, those ending in 1 or 5 are locations in a second, and so forth. Some of the earlier modules can be interleaved only in pairs, which is not as effective but is worthwhile. Without a cache, interleaving is not as effective, but it is still advisable since the program is sequential. Without interleaving or a cache, some alteration between modules is produced by keeping instructions in one and operands in another. Interleaving, assigning module numbers, and so forth, is done by the program for internal memories but by manual switch settings for external memories. Complete information is given in Appendix G.

The only physical locations uniquely defined by the hardware are those in fast memory, locations 0–17. All other hardware-defined addresses are relative to pages, such as the process tables, whose physical location is specified by the Monitor. Physical memory in a system is a constant unless a storage module is actually added or removed. The virtual address space accessible to a particular program is entirely a function of the way in which the Monitor sets up user operating conditions, except that any space and any restrictions must encompass an integral number of pages.

Memory Characteristics

The following tables give the characteristics of the various memories for the two types of KL10 processor. Times are in microseconds, and for external memories they include the delay introduced by 10 feet (3 meters) of cable. Read access for a single word or the first in a group is the time from the request until the word is in AR. For an entire 4-word group, read access is the time from the request untıl the last word is in the cache. Write access is the time from the request until the processor receives the memory acknowledgment, for either the first word or the fourth. Except for the MF20, these figures define the system access rates for storage modules with 4-way interleaving, as all memory operations are absorbed within them: by the time the processor receives the data or the acknowledgment, it can make a new request, which the memory will be ready for. Sizes given are those in which the units are available. Note that interleaving depends on the number of modules, not the number of units, most of which contain more than one module. Hence 4-way interleaving can be done with a single MA20 or MB20 memory, whereas it requires two MH10s or MG10s and four MF10s.

With MF20 memories, there is only one module per unit and interleaving is not used. (Each controller can handle three units or "groups".) The times given in the table are the actual times the processor must wait to get data or an acknowledgement, except that hitting a refresh cycle can cause a delay of up to 533 ns (refreshing requires about 3½% of total memory time). Following a read, the processor can make another request immediately. Following a write, it must wait from 467 to 867 ns before another request can be handled by the same controller. However since a single MF20 handles four words at once, one request following another within that time is unlikely.

Fast memory times are for referencing a memory location for an operand; a fast memory instruction fetch takes slightly more time than a cache access. When a fast memory location is addressed as an accumulator or index register, the access time is considerably shorter and usually takes no time at all, as it is done in parallel with instruction operations that are required anyway.

Physical Characteristics

	Number of Modules	*Size*
MF10 Core Memory	1	32K, 64K
MG10 Core Memory	2	64K, 128K
MH10 Core Memory	2	128K, 256K
MA20 Core Memory	4, 8	64K, 128K
MB20 Core Memory	4, 8	128K, 256K
MF20 MOS Memory	1	256K
KL10 Fast Memory		16
KL10 Cache		2K

The MF20 has a 7-bit error correction code; all other units have only a single parity bit. The MF20 also has a spare bit that can be substituted for a known bad bit.

Extended Processor Timing

	First or Single Word Access		*Four Word Access*	
	Read	*Write*	*Read*	*Write*
MF10 Core Memory	1.493	1.084	2.227	1.484
MG10 Core Memory	1.553	1.134	2.287	1.534
MH10 Core Memory	1.633	1.134	2.367	1.534
MA20 Core Memory	.883	.40	1.467	1.60
MB20 Core Memory	1.017	.40	1.60	1.60
MF20 MOS Memory	.800	.267	1.40	.667
KL10 Fast Memory	.067	.067		
KL10 Cache	.133	.133		

Single-section Processor Timing

	First or Single Word Access		*Four Word Access*	
	Read	*Write*	*Read*	*Write*
MF10 Core Memory	1.627	1.217	2.507	1.697
MG10 Core Memory	1.687	1.267	2.567	1.747
MH10 Core Memory	1.767	1.267	2.647	1.747
MA20 Core Memory	1.06	.48	1.76	1.92
MB20 Core Memory	1.22	.48	1.92	1.92
KL10 Fast Memory	.080	.080		
KL10 Cache	.160	.160		

1.8 KS10 Memory

Any subsystem can request use of the bus to write a word into storage or read a word from it. To save time in byte input operations, a Unibus adapter can also get the bus for a read-modify-write cycle. In this transaction a word goes from memory to the adapter, which inserts the byte and immediately sends the modified word back. A requesting subsystem may have to wait til the bus is free and it has priority, and even then there may occasionally be a further wait of up to 750 ns for memory refresh (which requires about 5% of total memory time). But otherwise reading from storage takes 900 ns, and writing takes 600 although the memory remains busy for an additional 300. Whenever the processor writes or reads a word in storage, that word is automatically written in the cache. Thus if the processor wishes to read the same word at a later time, retrieval requires only 300 ns. The cache hit rate is generally about 80%.

The following table gives the characteristics of KS10 memory with times in nanoseconds.

	Read	*Write*	*Size*	*Error Facility*
MOS Memory	900	600	128K–512K	7-bit correction code
Fast Memory	300	300	16	2 parity bits
Cache	300		512	2 parity bits

There is no cache write time as writing is automatic and is absorbed in storage access time. Fast memory times are for addressing as memory locations. Access to an accumulator or index register is made in a single microinstruction period of 150 ns, and frequently this represents no extra time, as the same microinstruction often performs other functions.

The memory array comprises from two to eight storage modules of 64K each. But from the hardware addressing point of view, the entire physical memory is simply a set of 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 address is octal 1777777, decimal 524,287. (Addresses are always in octal notation unless otherwise specified.)

At a halt the microcode places a halt code and PC in *storage* locations 0 and 1. The only other physical locations uniquely defined by the hardware are those in fast memory, locations 0–17. All other hardware-defined addresses, such as in the process tables or the halt status block, are relative to physical locations specified by the Monitor. Physical memory in a system is a constant unless a storage module is actually added or removed. The virtual address space accessible to a particular program is entirely a function of the way in which the Monitor sets up user operating conditions, except that any space and any restrictions must encompass an integral number of pages.

1.9 Programming Conventions

Two elements of system software intimately associated with the presentation in this manual are the assembler and the operating system. The manual explains the DECsystem–10 and DECSYSTEM–20 in terms of machine language programming. Such programming makes use of those basic characteristics of the MACRO assembler described here. The assembler naturally has many other features, such as use of predefined and user-defined pseudoinstructions. The overview of the system presented in the first two sections and the more detailed presentation of system operations in later chapters are in a sense a presentation of the sophisticated features of the operating system: its most impressive features related to the processor are essentially its capabilities for taking advantage of these sophisticated hardware characteristics. There are two versions of the operating system, the TOPS–10 Monitor and the TOPS–20 Monitor. The basic thrust of both is the timesharing of the system among a number of independent users, all of whom can make extensive use of all system facilities, including front end processing and the advanced file system.

MACRO 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 nine or thirteen bits (six in pre-KS10 in-out instructions). The assembler translates every statement into a 36-bit word, placing 0s in all bits whose values are unspecified. For example, 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.

NOTE

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

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

```
MOVNS    @2570
```

assembles as 213020 002570, and produces indirect addressing. Placing the number of an index register (1–17) in parentheses following the memory address causes modification of the address by the contents of the specified register. Hence

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

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

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

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

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

The device code in a pre-KS10 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, e.g.

```
        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. For example,

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

and

```
        ADD     12,A
        .
        .
        .
A:      7256004
```

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

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

is equivalent to

```
                      [777532000135]
```

A literal can encompass any number of lines of code, employing any of the programming conventions defined above, and to be assembled in consecutive locations. In fact a reasonable way to assemble the extended instructions is to give the individual extended instruction code and any necessary follow-up words as a literal in an extend instruction. The assembly of these two lines,

```
STRING: EXTEND  AC,[MOVSO OFF
                FILL]
```

produces, in location STRING, an EXTEND whose *Y* part (*E0*) points to the location containing the second instruction word MOVSO OFF. The *Y* part (*E1*) of the MOVSO contains the signed offset OFF, and location *E0*+1 contains the fill character FILL.

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

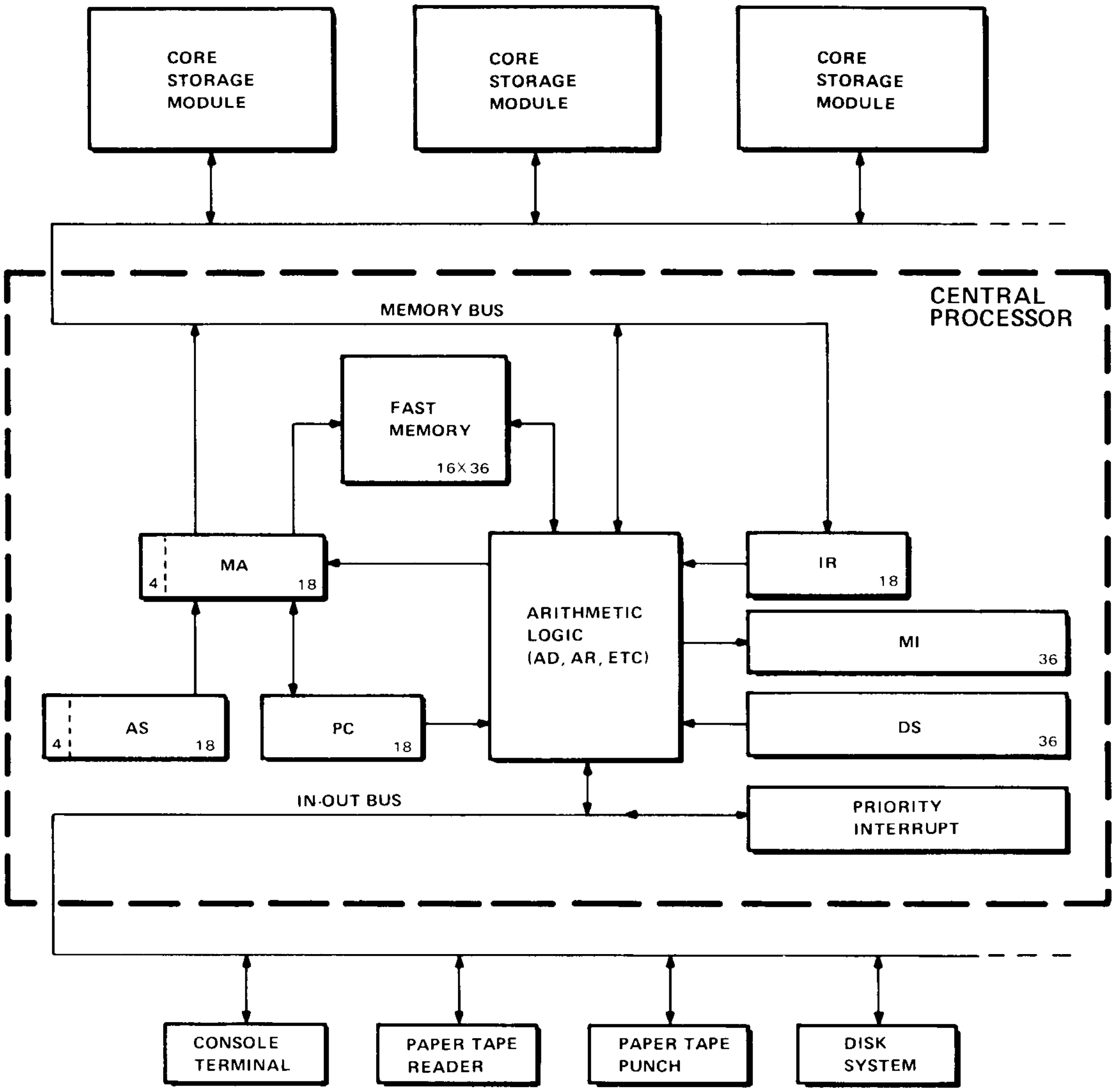
1.10 KI10 and KA10 Characteristics

The KI10 and KA10 are similar, even identical, to the KL10 in many respects, but their implementation is quite different: they have no microcontroller or microcode. They use the PDP–10 instruction set but not in its full variety as available in the KL10: neither earlier processor can handle strings or double precision fixed point numbers; the KA10 has no capability for handling doublewords or performing double precision floating point arithmetic, although it does have instructions (retained on all KL10 and KI10 TOPS–10 systems) for assisting the software in doing double precision floating point in a special software format.

Figure 1.8 illustrates the organization of a DECsystem–10 based on either of the earlier processors. The processor handles its peripheral equipment directly over an in-out bus, there is no cache, there is a real time clock but no meters, and all memory is external. The extra four bits shown on address registers are applicable only to the KI10. Both processors use an 18-bit internal address providing a virtual memory of one section that is compatible with section 0 of the KL10. But whereas the KA10 has a maximum physical memory equal in size to its virtual memory, which is organized by protection and relocation hardware, the KI10 has a physical addressing capability equal to that of the KL10 (22-bit address, 4096K) and has paging hardware. The KI10 virtual address space is the same as that of a KL10 with the TOPS–10 Monitor, except that in executive mode the first 112K of memory is unpaged (and thus not available to the supervisor program), and the Monitor can define a so-called "small user" whose accessible space must lie within the virtual ranges 0–37777 and 400000–437777. The KI10 has four fast memory blocks, of which hardware requires that the Monitor use block 0; the KA10 has only one block.

Both processors have manual operator consoles with facilities that are directly relevant to the programmer, although they are used mostly for manually stepping through a program to debug it. From the sense switches and the 36-bit data switch register DS, the program can read information supplied by the operator, and through the memory indicators MI, the program can display data for the operator. 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 supply a starting address for the program. In these processors IR contains the entire left half of the current instruction word, i.e. eighteen bits rather than thirteen. The memory address register MA supplies the address for every memory access. In the arithmetic logic of the KA10, there are only single length registers; but in the KI10, AR and AD have 28-bit left extensions for double precision floating point. The KA10 has no trapping mechanism: arithmetic and stack overflow signal the program by way of interrupts. Individual processor differences relevant to user programming are listed in Appendix E.

Figure 1.8: DECsystem–10 Based on KI10 or KA10



Memory

The following table gives the characteristics of the various memories available with the KI10 and KA10. Modify completion is the time to finish a read-modify-write cycle after the processor supplies the new data. Times are in microseconds and include the delay introduced by ten feet (three meters) of cable. Fast memory times are for referencing as a memory location (18-bit address); when a fast memory location is addressed as an accumulator or index register, the access time is considerably shorter.

	Read Access	*Write Access*	*Cycle*	*Modify Completion*	*Size*
MA10 Core Memory	.61	.20	1.00	.57	16K
MB10 Core Memory	.60*	.20*	1.65*	.97	16K
MD10 Core Memory	.83	.33	1.8	1.23	32–128K
ME10 Core Memory	.61	.20	1.00	.65	16K
MF10 Core Memory	.61	.20	1.00	.63	32K, 64K
MG10 Core Memory	.67	.23	1.00	.63	32–128K
MH10 Core Memory	.74	.23	1.18	.68	64–256K
KA10 Fast Memory	.21	.21			16
KI10 Fast Memory	.28	.0			16

* Add .1 in a multiprocessor system.

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

In a system with the greatest possible capacity, the largest KI10 address is octal 17777777, decimal 4,194,303; the largest KA10 address is octal 777777, decimal 262,143. All storage modules can be interleaved in pairs, and some of them in sets of four (see Appendix G). The KA10 cannot overlap memory access.

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

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.

140–161 Allocated to second processor if connected (same use as 40–61 for first processor). All information given in this manual about memory locations 40–61 for a KA10 applies instead to locations 140–161 for programming a second KA10 connected to the same memory.

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

Chapter 2
User Operations

This chapter describes all PDP–10 instructions that are generally available to the user. It also defines the types of in-out instructions, but does not discuss their effects when they address specific internal system elements or 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 extended instructions, the mnemonic given actually assembles to the word shown in the second format box; the first box shows the configuration of the EXTEND itself. The programmer must give EXTEND, and give the listed mnemonic either as a literal with EXTEND or place it in the source program at the location specified by the EXTEND effective address.

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; e.g. 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 satisifed 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. Letters representing modes are suffixes, which produce new mnemonics that are recognized as distinct symbols by the assembler. Following the description is a table giving the mnemonics and octal codes (bits 0–8) for the various modes.

In a description *E* refers to the effective address, half word operand, mask, offset, 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. In extended instructions *E0* and *E1* refer to the results of the effective address calculations for the first and second instruction words. E_R refers to the right eighteen bits of the effective address (i.e. the in-section part), but in a machine lacking extended addressing, E_R is equivalent to *E*. A reference to "location *E*,*E*+1" means the contents of the two locations are used together as a doubleword, such as a double length number. If the program is running in section 0 or the instruction gives a local address, the addresses wrap around so that when *E* is 777777, *E*+1 is 0.

PLEASE READ THIS

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

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 or more accumulators, these have addresses *A*, *A*+1, *A*+2, etc., which are interpreted modulo 20_8; e.g. *A*+1 is 0 when *A* is 17. In the text the various accumulators are referred to as AC, AC+1, and so forth. A pair of accumulators holding a doubleword is referred to as AC,AC+1. 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 memory. 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, §2.9 discusses trapping, §2.19 explains the restrictions on user programming, and Chapters 3 to 5 describe the special effects and restrictions associated with system operations in the various processors.

To minimize processor execution time the programmer should minimize the number of memory references and 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. The time depends upon which processor performs the instruction, and in many cases on the configuration of the operands and the number of iterative steps. 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, i.e. 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 extend or 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 §2.15.

2.1 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. Let us begin with a single instruction that simply interchanges the contents of an accumulator and a memory location.

EXCH Exchange

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

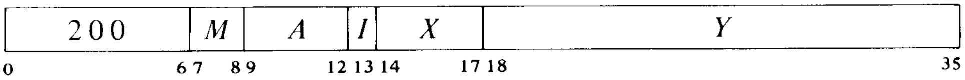
Move the contents of location *E* to AC and move AC to location *E*.

Move Instructions

This class of instructions consists of a group for general manipulation of single words and a special immediate mode instruction for handling an extended address. Each of the instructions in the standard move group handles one word, which may be changed in the process (e.g. 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

MOVE Move

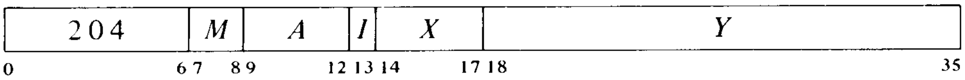


Move one word from the source to the destination specified by *M*. The source is unaffected, the original contents of the destination are lost.

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

Notes. MOVEI loads the word 0,*E* into AC. If *A* is zero, MOVES is a no-op that writes in memory; otherwise it is equivalent to MOVE except that it writes in memory.

MOVS Move Swapped

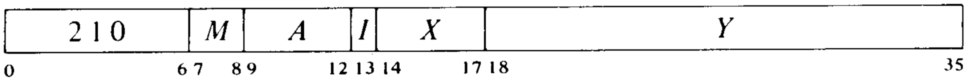


Interchange the left and right halves of the word from the source specified by *M* and move it to the specified destination. The source is unaffected, the original contents of the destination are lost.

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

Notes. Swapping halves in immediate mode loads the word *E*,0 into AC.

MOVN Move Negative



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

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

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

Notes. MOVNI loads AC with the negative of the word 0,*E* and can neither overflow nor carry.

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 Trap 1, Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} sets the flags, but this is not a normalized number.) The source is unaffected, the original contents of the destination are lost.

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

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

It is often convenient to keep a control count in the left half of an accumulator and a local address or displacement to be used for indexing in the right half. Suppose we wish to load 200 into the left half and 1400 into the right half of an accumulator that is addressed symbolically as XR. If the number 200 001400 is stored in location *M*, we can do this by giving the instruction

```
MOVE      XR,M
```

Of course the source program must somewhere define the value of the symbol XR as an octal number between 1 and 17. If the same word, or its negative, or with its halves swapped must be loaded on several occasions, each transfer still requires only a single move instruction that references *M*.

The following instruction makes the result of an effective address calculation available for use as a global address, even for accessing a fast memory location from any section.

XMOVEI Extended Move Immediate

4 1 5	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the program is running in a nonzero section, do one or the other of the following.

> If *E* is not a local AC address, clear AC bits 0–5 and place the global effective address *E* in AC bits 6–35.
>
> If *E* is a local AC address, put 1 in AC left and *E* in AC right.

If the program is running in section 0, this instruction is called SETMI, a Boolean instruction that performs an analogous function for section 0 (§2.4).

Notes. The form given a local AC address is that of a global AC address, which therefore still refers to fast memory no matter what section the address may be moved to or used in. Giving XMOVEI with an address 20 or greater without indexing or indirection places the current PC section number in AC left, and it can thus be used to determine what section the program is in.

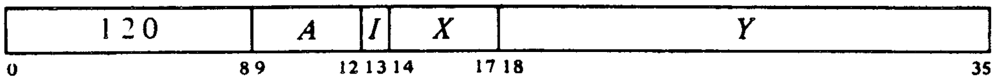
Double Move Instructions[1]

These four instructions are principally for manipulating the double length operands used in double precision arithmetic, fixed or floating. But they may be used to move or negate any doubleword, i.e. the contents of a pair of adjacent accumulators or memory locations. Two of the instructions are simple extensions of MOVE and MOVEM to doublewords, and for them the configuration of the operands is irrelevant. The other two are extensions of MOVN and MOVNM, with the operand interpreted as a double precision floating point number. They can just as well be used for fixed point numbers, but with a slight variation in the format. Namely a negative result has a 0 in bit 0 of the low order word instead of a copy of the sign. For arithmetic operations per se this difference is inconsequential, as all arithmetic instructions ignore bit 0 of all low order words. However it could cause a comparison of two equal double precision numbers to fail.

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), the memory locations have addresses *E* and *E*+1.

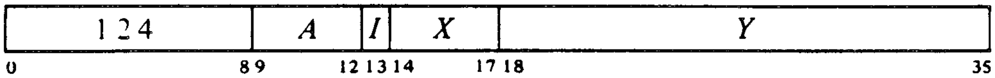
[1] In the KA10 these instructions are trapped as unassigned codes (§2.16).

DMOVE Double Move



Move a doubleword from location $E,E+1$ to AC,AC+1. The memory locations are unaffected, the original contents of the ACs are lost.

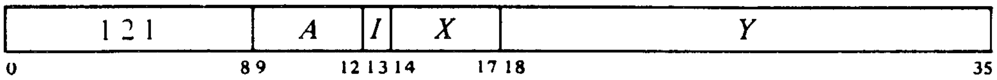
DMOVEM Double Move to Memory



Move a doubleword from AC,AC+1 to location $E,E+1$. The ACs are unaffected, the original contents of the memory locations are lost.

Notes. Do not use the instruction DMOVEM AC,AC+1 as its result is indeterminate. In the KI10 do not have E and X address the same (fast) memory location, as a page failure on the second word would then result in a different effective address calculation when the instruction is restarted.

DMOVN Double Move Negative

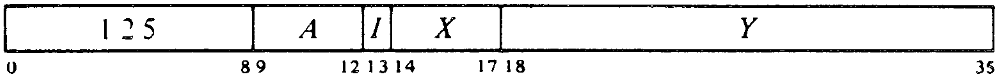


Negate the doubleword from location $E,E+1$ interpreted in double precision floating point and move it to AC,AC+1. If the memory doubleword is fixed point -2^{70}, set the Trap 1, Overflow and Carry 1 flags. (Negating the equivalent floating point with fraction −1 and the maximum exponent sets the flags, but this is not a normalized number.) If the memory doubleword is zero, set Carry 0 and Carry 1. The memory locations are unaffected, the original contents of the ACs are lost.

Note that the negation uses floating point conventions. Hence a negative fixed point result has the incorrect value in bit 0 of the low order word.

In the KI10 there is no overflow test as the KI10 lacks double precision fixed point instructions. For floating point the overflow test is really unnecessary, as negating a correctly formatted floating point number cannot cause overflow.

DMOVNM Double Move Negative to Memory



Negate the doubleword from AC,AC+1 interpreted in double precision floating point and move it to location $E,E+1$. If the AC doubleword is fixed point -2^{70}, set the Trap 1, Overflow and Carry 1 flags. (Negating the equivalent floating point with fraction –1 and the maximum exponent sets the flags, but this is not a normalized number.) If the AC doubleword is zero, set Carry 0 and Carry 1. The ACs are unaffected, the original contents of the memory locations are lost.

Note that the negation uses floating point conventions. Hence a negative fixed point result has the incorrect value in bit 0 of the low order word.

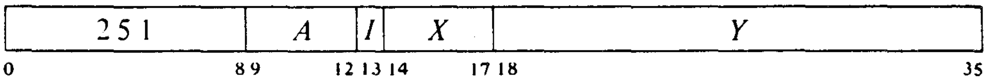
In the KI10 there is no overflow test as the KI10 lacks double precision fixed point instructions. For floating point the overflow test is really unnecessary, as negating a correctly formatted floating point number cannot cause overflow.

Notes. Do not use the instruction DMOVNM AC,AC+1 as its result is indeterminate. In the KI10 do not have E and X address the same (fast) memory location, as a page failure on the second word would then result in a different effective address calculation when the instruction is restarted.

Block Transfers

There are two instructions for moving blocks of data from one part of memory to another. One is restricted to acting within the section specified by the effective address. The other can be performed only in a nonzero section, but can move data arbitrarily anywhere in memory.

BLT Block Transfer



Beginning at the location addressed by AC left in the section specified by E, move words to another area in the same section 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 $E_R - AC_R + 1$. If $AC_R \geqslant E$, the BLT moves one word to location AC_R. If the source block is larger than $2^{18} - AC_L$, it is wrapped around to the beginning of the section.[2]

Provided AC is not in the destination block, then at the end in the KL10 and KS10, AC left and right respectively contain addresses one greater than those of the final source and destination locations referenced (or in the case of $AC_R > E$ in the KL10, the addresses of the locations that would have been referenced had the reverse order transfer actually taken place). In the KI10 and KA10, AC is indeterminate unless the interrupt system and the pager are both off, in which case it is unaffected. In any event, for program compatibility among processors, use of the resulting quantity in AC is strongly discouraged.

[2] *Caution:* In section 0 of a KL10 extended address space there is no wraparound, and the instruction inadvertently counts into section 1.

CAUTION

Should an interrupt or page failure occur during its execution, 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 *A* and *X* must not address the same register as this would produce a different effective address calculation upon resumption; and the instruction must not attempt to load an accumulator addressed either by *A* or *X* unless it is the final location being loaded.

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

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

As mentioned in the above caution, this example might not work if, e.g. AC 10 or AC 16 were used to supply the source and destination addresses. 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
```

A convenient way to clear a block in memory is to clear the first location and then use a BLT to transfer the zero successively from one location to the next. Suppose the block starts at A and contains *B* words.

```
MOVE    AC,[A,,A+1]
SETZM   A
BLT     AC,A+B–1
```

For a reverse BLT procedure (highest addresses first), refer to the POP instruction (§2.10).

XBLT Extended Block Transfer

1 2 3	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	0 2 0	0 0	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

E1 is not used.[3]

Move a block of words from one area of memory to another. The block size and the locations of the source and destination areas are defined by the contents of a block of three accumulators.

AC	NUMBER OF WORDS IN BLOCK	
AC+1	0 0	LOCATION OF SOURCE BLOCK
AC+2	0 0	LOCATION OF DESTINATION BLOCK
	0 5	6 35

If this instruction is given in section 0, execute it as an MUUO. Otherwise perform a forward or backward block transfer as follows.

> If AC contains a positive number *N*, move a block of *N* words from a source area beginning at the location specified by AC+1, to a destination area beginning at the location specified by AC+2, and extending through increasing addresses. At the end AC is clear, and AC+1 and AC+2 respectively contain addresses one greater than those of the final source and destination locations referenced.
>
> If AC contains a negative number –*N*, move a block of *N* words from a source area beginning at a location one less than that specified by AC+1, to a destination area beginning at a location one less than that specified by AC+2, and extending through decreasing addresses. At the end AC is clear, and AC+1 and AC+2 respectively contain the addresses of the final source and destination locations referenced.

CAUTION

This instruction uses three accumulators, and under no circumstances should any of these three be part of either the source or destination block. Because of the possibility of an interrupt or page failure, the contents of these accumulators even as a source cannot be guaranteed. And in any event, use of XBLT for moving an AC block is quite unnecessary, as a simple BLT can move fast memory to any section.

[3] *I*, *X* and *Y* are reserved and should be zero.

2.2 Fixed Point Arithmetic

For fixed point arithmetic the PDP–10 has instructions for performing addition, subtraction, multiplication and division of numbers in single and double precision fixed point format (§1.4), although double precision is not available in the KI10 or KA10. The processor can also do arithmetic shifting — which is essentially multiplication by a power of 2 — but those instructions are discussed with logical shifting and rotating (§2.5). For single precision the add and subtract instructions involve only single length numbers, whereas multiply supplies a double length product, and divide uses a double length dividend. 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, bytes, and character codes. For double precision the add and subtract instructions involve only double length numbers, whereas multiply supplies a quadruple length product, and divide uses a quadruple length dividend. In all cases the position of the binary point is arbitrary, and the programmer may adopt any point convention. Even the integer multiply and divide instructions can be used for small fractions 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 (§2.1), and the arithmetic test instructions that increment or decrement the test word (§2.6). In these instructions an incorrect result is indicated — and the Overflow flag set — if the carries are different, i.e. if there is a carry into the sign but not out of it, or vice versa. Overflow is determined directly from the carries, not from the carry flags, as their states may reflect events in previous instructions. 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 (§2.3), and loss of significant bits in left arithmetic shifting. Any condition that sets Overflow also sets the Trap 1 flag (§2.9).

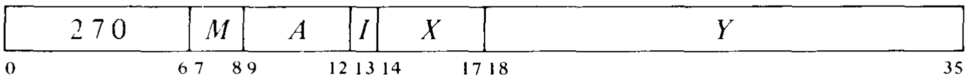
These flags can be read and controlled by certain program control instructions (§§2.9, 2.16), but overflow is usually handled by trapping through the setting of Trap 1 (§2.9). The KA10 lacks the trapping feature, so its program must make direct use of the Overflow flag, which is available as a processor condition (via an in-out instruction) that can request a priority interrupt if enabled (§5.6). In any event, user overflow is handled by the Monitor according to instructions from the user, as described in Chapter 3 of the appropriate Monitor Calls manual. 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.

Single Precision Instructions

As noted above the numbers manipulated by these instructions are single length except for double length products and dividends. Such double length fixed point numbers are in AC,AC+1, where the magnitude is the 70-bit string in bits 1–35 of the two words, the sign is in bit 0 of the high order word, and bit 0 of the low order word contains a copy of the sign. All six instructions have four modes that determine the source of the non-AC operand and the destination of the result.

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

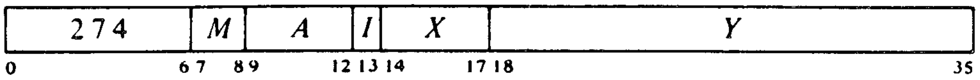
ADD Add



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

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

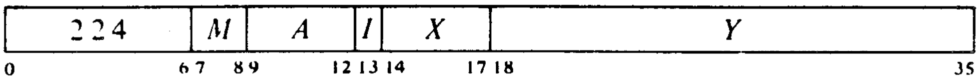
SUB Subtract



Subtract the operand specified by *M* from AC and place the result in the specified destination. If the difference is ⩾ 2^{35} set Trap 1, 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 Trap 1, Overflow and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the difference plus 2^{35}. Set both carry flags if the signs of the operands are the same and AC is the greater or the two are equal, or the signs of the operands differ and AC is negative.

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

MUL Multiply



Multiply AC by the operand specified by *M*, and place the high order word of the double length result in the specified destination. If *M* specifies AC as a destination, place the low order word in AC + 1. If both operands are -2^{35} set Trap 1 and 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

CAUTION

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

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 Trap 1 and Overflow if the product is $\geqslant 2^{35}$ or $< -2^{35}$ (i.e. 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 high order word of the magnitude of the double length number in AC,AC + 1 is greater than or equal to the magnitude of the operand specified by *M*, set Trap 1, 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 AC,AC + 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 AC + 1.

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

Notes. The magnitude restriction is required since the quotient developed would exceed 36 bits.

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, or AC contains -2^{35} and the operand specified by *M* is -1 (except in the KS10), set Trap 1, 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 AC + 1.

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

CAUTION

> In the KS10, dividing -2^{35} by -1 gives -2^{35} with no error indication. In the KA10, KI10, and a KL10 with microcode version before 271 (which includes all single-section KL10s), the overflow action is also triggered by attempting to divide -2^{35} by $+1$.

Double Precision Instructions[4]

There are just four instructions for the four basic operations, and they have no modes. All use AC and memory operands and place the result in the accumulators. Memory operands are double length in location *E*,*E* + 1. Most AC operands are double length in AC,AC + 1, but products and dividends are quadruple length in AC,AC + 1,AC + 2,AC + 3, and the double length remainder in division is placed in AC + 2,AC + 3. Double length numbers have the same format as the products and dividends of single precision instructions discussed above. In quadruple length numbers AC contains the highest order word; the magnitude is the 140-bit string in bits 1–35 of the four words, the sign is in bit 0 of the highest order word, and copies of the sign are kept in bit 0 of the other three words.

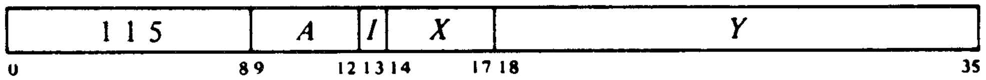
DADD **Double Add**

1 1 4	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Add the operand in location *E*,*E* + 1 to AC,AC + 1 and place the result in AC,AC + 1. If the sum is ⩾ 2^{70}, set Trap 1, Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the sum less 2^{70}. If the sum is $< -2^{70}$, set Trap 1, Overflow and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the sum plus 2^{70}. Set both flags if both summands are negative, or their signs differ and their magnitudes are equal or the positive one is the greater in magnitude.

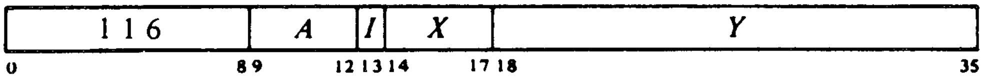
[4] In the KI10 and KA10 these instructions are trapped as unassigned codes.

DSUB Double Subtract



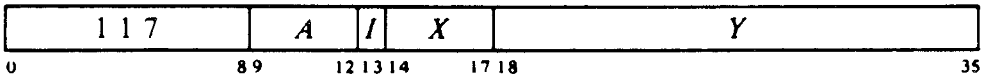
Subtract the operand in location $E,E+1$ from AC,AC + 1 and place the result in AC,AC + 1. If the difference is $\geqslant 2^{70}$, set Trap 1, Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{70}. If the difference is $< -2^{70}$,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

DMUL Double Multiply



Multiply AC,AC + 1 by the operand in location $E,E+1$ and place the result in AC–AC + 3. If both operands are -2^{70}, set Trap 1 and Overflow; the quadruple length result stored is -2^{140}.

DDIV Double Divide



If the high order doubleword of the magnitude of the quadruple length number in AC–AC + 3 is greater than or equal to the magnitude of the operand in location $E,E+1$, set Trap 1, 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 quadruple length number contained in the accumulators by the operand in location $E,E+1$, calculating a quotient of 70 magnitude bits including leading zeros. Place the unrounded quotient in AC,AC + 1, and the double length remainder, with the same sign as the dividend, in AC + 2,AC + 3.

2.3 Floating Point Arithmetic[5]

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), performing addition, subtraction, multiplication and division of numbers in single and double precision floating point formats, converting between different range floating formats, and converting numbers from fixed format to floating and vice versa. Except for conversion operations, instructions treated here interpret all operands as floating point numbers in the formats given in §1.4, and generate results in those formats. The reader is strongly advised to reread §1.4 if he does not remember the formats in detail.

For the four standard arithmetic operations in single precision, the program has a choice of modes, determining mostly the destination of the result, and can select whether or not the result shall be rounded. Rounding produces the greatest consistent precision using only single length operands. Instructions without rounding 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). In any event, the significance of the result depends on the relative values of the operands. For example, 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. 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.

Among the remaining floating point instructions, those that convert between number types in standard range operate only on single words. Instructions that convert to floating point assume the operand is an integer and always normalize and round 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 calculates a two-word rounded quotient that is already normalized if the original operands are normalized. 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 normalization and then for rounding.

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

[5] In a KA10 without floating point hardware, all of the instructions presented in this section are trapped as unassigned codes (§2.16).

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 also sets the Trap 1 flag. These flags can be read and controlled by certain program control instructions (§§2.9, 2.16), but overflow is usually handled by trapping through the setting of Trap 1. The KA10 lacks the trapping feature, so its program must make direct use of Overflow and Floating Overflow, which are available as processor conditions (via an in-out instruction) that can request a priority interrupt if enabled (§5.6). 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.

The floating point hardware functions at its best if given operands that are either normalized or zero, and it normalizes a nonzero result.[6] An operand with a zero fraction and a nonzero exponent can give wild answers in additive operations because of extreme loss of significance; e.g. adding $\frac{1}{2} \times 2^2$ and 0×2^{69} gives a zero result, as the first operand (having a smaller exponent) looks smaller to the processor and is shifted to oblivion. A number with a 1 in bit 0 and 0s in bits 9–35 is not simply an incorrect representation of zero, but rather an unnormalized "fraction" with value –1. This unnormalized number can produce an incorrect answer in any operation. But note that such malformed numbers must be created deliberately by the programmer — the processor never produces them.

[6] 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.

Single Precision with Rounding

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.[7]

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 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 Trap 1, Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < –128, set Trap 1, Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

FADR Floating Add and Round

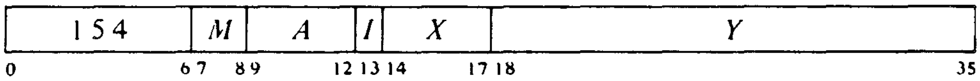
144	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

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

[7] 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 < ½ LSB. Moreover an extra single-step renormalization occurs if the rounded word is no longer normalized.

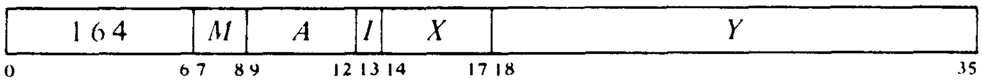
FSBR Floating Subtract and Round



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

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

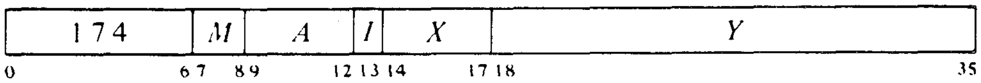
FMPR Floating Multiply and Round



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

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

FDVR Floating Divide and Round



If the magnitude of the fraction in AC is greater than or equal to twice that of the fraction in the operand specified by *M*, set Trap 1, Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

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

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

Single Precision without Rounding

Instructions that do not round are faster for processing floating point numbers with fractions containing fewer than 27 significant bits. They perform the four standard arithmetic operations with normalization but without rounding. All use AC and the contents of location *E* as operands and have three modes. They lack an immediate mode, but are otherwise analogous to the single precision instructions with rounding.

Mode	*Suffix*	*Effect*
Basic		High order word of result stored in AC.
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 Trap 1, Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < –128, set Trap 1, 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

$M \neq 1.$

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*. 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.[8]

FAD	Floating Add	140
FADM	Floating Add to Memory	142
FADB	Floating Add to Both	143

FSB Floating Subtract

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

$M \neq 1$.

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*. 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.[9]

FSB	Floating Subtract	150
FSBM	Floating Subtract to Memory	152
FSBB	Floating Subtract to Both	153

FMP Floating Multiply

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

$M \neq 1$.

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*. 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.

[8] *Caution:* In single precision addition the term with the smaller exponent is right shifted in a double length register, specifically a register with 54 magnitude bits. Now if the difference in the exponents is < 54, there is at least one significant bit after the shift (assuming normalized operands); and if the difference is > 72 (64 in the KI10), the hardware throws the term away by substituting zero. But when the exponent difference lies in the range 54 to 72 (64), the procedure disposes of all significant bits without actually substituting zero. This means that if the shifted term is positive it appears in the addition as all 0s, but if negative it appears as all 1s. The latter case gives an answer that is less by one LSB.

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

FMP	Floating Multiply	160
FMPM	Floating Multiply to Memory	162
FMPB	Floating Multiply to Both	163

FDV Floating Divide

170	*M*	*A*	*I*	*X*	*Y*	$M \neq 1$.
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 the magnitude of the fraction in location *E*, set Trap 1, Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

If division can be performed, floating divide AC by the contents of location *E*. Calculate a quotient fraction of 27 bits. If the fraction is zero, clear the destination specified by *M*. 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 in the specified destination.

NOTE

> In the KL10 and KS10, a negative quotient is represented by a twos complement only when the remainder is zero — otherwise it is a ones complement. In the KI10 and KA10, a twos complement is used for a negative quotient regardless of the value of the remainder.

FDV	Floating Divide	170
FDVM	Floating Divide to Memory	172
FDVB	Floating Divide to Both	173

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

Standard Range Double Precision[10]

There are four instructions for the four basic operations, and they have no modes. All use AC and memory operands and place the result in the accumulators. Memory operands are double length in location $E,E+1$; AC operands and results are double length in AC,AC+1. All operands are interpreted as double precision floating point numbers. All results are normalized regardless of the status of the original operands, except that in KI10 multiplication and division the result is guaranteed to be normalized only when the original operands are normalized. Except in KI10 division, the result is rounded. The rounding function is the same as that used in

[10] In the KA10 these instructions are trapped as unassigned codes.

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).

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 Trap 1, Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < –128, set Trap 1, Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

DFAD Double Floating Add

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

Floating add the operand in location *E*,*E* + 1 to AC,AC + 1. If the fraction in the sum is zero, clear AC,AC + 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 AC,AC + 1. Note: The KI10 zero test inspects only the high order 70 bits in the fraction.

DFSB Double Floating Subtract

111	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Floating subtract the operand in location *E*,*E* + 1 from AC,AC + 1. If the fraction in the difference is zero, clear AC,AC + 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 AC,AC + 1. Note: The KI10 zero test inspects only the high order 70 bits in the fraction.

DFMP Double Floating Multiply

112	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

KL10 and KS10: Floating multiply AC,AC + 1 by the operand in location *E*,*E* + 1. If the product is zero, clear AC,AC + 1. Otherwise normalize the product, round the high order double length part, test for exponent overflow and underflow as described above, and place the result in AC,AC + 1.

KI10: Floating multiply AC,AC + 1 by the operand in location *E*,*E* + 1.

If the high order 70 bits of the fraction in the product are zero, clear AC,AC + 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 AC,AC + 1. The 35-bit shift can be done only if the original operands are unnormalized. The single normalization shift produces a normalized result for normalized operands.

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

If the division can be performed, floating divide the AC operand by the memory operand, calculating a quotient fraction of 63 bits including one for rounding (62 in the KI10). If the fraction is zero, clear AC,AC + 1. Otherwise in the KL10 normalize the quotient and round it using the extra bit calculated. Test for exponent overflow or underflow as described above, and place the quotient in AC,AC + 1. The remainder is lost. Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

Notes. In the KI10 the quotient is normalized if the original operands are normalized.

Expanded Range Double Precision[11]

There are four instructions for the four basic operations in G format, and they have no modes. All use AC and memory operands and place the result in the accumulators. Memory operands are double length in location *E*,*E* + 1; AC operands and results are double length in AC,AC + 1. All operands are interpreted as G format double precision floating point numbers. All results are normalized and rounded regardless of the status of the original operands. 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).

[11] These instructions are trapped as unassigned codes except in a KL10 that runs microcode version 271 or greater.

In each of these instructions, the exponent that results from normalization and rounding is tested for overflow or underflow. If the exponent is > 1023, set Trap 1, Overflow and Floating Overflow; the result stored has an exponent 2048 less than the correct one. If < –1024, set Trap 1, Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 2048 greater than the correct one.

GFAD G Format Floating Add

102	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Interpreting all numbers in G format, floating add the operand in location *E*,*E* + 1 to AC,AC + 1. If the fraction in the sum is zero, clear AC,AC + 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 AC,AC + 1.

GFSB G Format Floating Subtract

103	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Interpreting all numbers in G format, floating subtract the operand in location *E*,*E* + 1 from AC,AC + 1. If the fraction in the difference is zero, clear AC,AC + 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 AC,AC + 1.

GFMP G Format Floating Multiply

106	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Interpreting all numbers in G format, floating multiply AC,AC + 1 by the operand in location *E*,*E* + 1. If the product is zero, clear AC,AC + 1. Otherwise normalize the product, round the high order double length part, test for exponent overflow and underflow as described above, and place the result in AC,AC + 1.

GFDV G Format Floating Divide

107	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

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

If the division can be performed, floating divide in G format the AC operand by the memory operand, calculating a quotient fraction of 60 bits including one for rounding. If the fraction is zero, clear AC,AC + 1. Otherwise normalize the quotient and round it using the extra bit calculated. Test for exponent overflow or underflow as described above, and place the quotient in AC,AC + 1. The remainder is lost. Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

Number Conversion[12]

Besides the groups of instructions for performing standard arithmetic operations with fixed and floating point numbers in the various formats, there is also a group for translating numbers from one format to another. This group includes several that are strictly single precision between floating and fixed, and over twice as many that handle conversion between G format and single or double precision fixed point as well as single precision floating point. In the following presentation these instructions are grouped in the way they would most likely be associated in use; but there are common characteristics that cross over these categories, in particular having to do with the rounding of the result.

If the result of a conversion is a floating point number in any format, the rounding function is the same as that used by the standard floating point arithmetic instructions described above. A fixed point result on the other hand may be rounded or simply truncated, which corresponds to whether the instruction mnemonic ends in "FIXR" or just "FIX".

> Truncation produces the integer of largest magnitude less than or equal to the magnitude of the original number. For example, a number > +1 but < +2 becomes +1; a number < –1 but > –2 becomes –1. This truncation function is that used in Fortran ("fixation"). For it, the processor drops the fractional part in a positive number, but adds one to the integral part (as required by twos complement format) if any bits of significance are shifted out in a negative number.

[12] In the KA10 all of these instructions are trapped as unassigned codes. The first three are available in all other processors, but the remaining eight are available only in a KL10 with microcode version 271 or greater. However the four instructions that convert from G format to fixed point are not implemented in microcode: they are instead simulated by the Monitor.

Rounding is in the positive direction: the magnitude of the integral part is increased by one if the fractional part is ⩾ ½ in a positive number but > ½ in a negative number. For example, +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 rounding function is the Algol standard for real to integer conversion. For it the processor adds one to the integral part if the fractional part is ⩾ ½ in a positive number or (as required by twos complement format) is ⩽ ½ in a negative number.

The first three of the following instructions convert between fixed and floating in single precision only; the next six are for converting between single and double precision integers and G floating point numbers; the final pair converts between G format and the standard range single precision floating point that is available in all machines. In all cases the operand is taken from location *E* or *E*,*E* + 1, and the converted result is placed in AC or AC,AC + 1.

FIX Fix

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

If the exponent of the single precision 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.

Notes. The overflow test checks for a value ⩾ 2^{35} assuming the operand is normalized.

FIXR Fix and Round

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

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

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

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

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 excess 128 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.

Since the largest single precision 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 single precision 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.

GFIX G Format Fix

	123	*A*	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35
E0	024	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

If the exponent of the G format floating point number in location $E,E+1$ is > 35, set Overflow and Trap 1, and go immediately to the next instruction without affecting AC or the contents of $E,E+1$ 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 double length fraction $N = X - 24$ 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 bits from the low order word 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. Then truncate to a single length integer and place the result in AC.

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

GFIXR G Format Fix and Round

123	A	I	X	Y
0 8	9 12	13	14 17	18 35

E0	026	00	I	X	Y	Bits 9–12 = 0.
	0 8	9 12	13	14 17	18 35	

If the exponent of the G format floating point number in location E,$E+1$ is > 35, set Overflow and Trap 1, and go immediately to the next instruction without affecting AC or the contents of E,$E+1$ 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 double length fraction $N = X - 24$ 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 bits from the low order word 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. Then round the integral part and place the result in AC. If rounding produces the number 2^{35}, set Overflow and Trap 1; the result stored is actually -2^{35}.

Notes. The initial overflow test checks for a value $\geqslant 2^{35}$ assuming the operand is normalized. Rounding can overflow only if the original operand has exponent 35 and fraction $\geqslant 1 - 2^{-36}$ (in other words the fraction is positive and begins with a string of thirty-six 1s).

GFLTR G Format Float and Round

123	A	I	X	Y
0 8	9 12	13	14 17	18 35

E0	030	00	I	X	Y	Bits 9–12 = 0.
	0 8	9 12	13	14 17	18 35	

Shift the magnitude part of the fixed point integer from location E right eleven places in a double length register with 0 bits at the right, insert the exponent 35 (in excess 1024 form) into bits 1–11 to move the shifted binary point to the left of bit 12 ($35 = 24 + 11$), and normalize the now double length fraction bringing 0s into bit 71. Place the G format result in AC,AC + 1.

Notes. No rounding can occur as the fraction contains fewer than fifty-nine significant bits.

GDFIX G Format Double Fix

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0

023	00	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

If the exponent of the G format floating point number in location *E*,*E* + 1 is > 70, set Overflow and Trap 1, and go immediately to the next instruction without affecting the ACs or the contents of *E*,*E* + 1 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 double length fraction *N* = *X* − 59 places to the correct position for its order of magnitude with the binary point at the right of bit 71. For positive *N*, shift left bringing 0s into bit 71 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 a double length integer. Place the result in AC,AC + 1.

Notes. The overflow test checks for a value ≥ 2^{70} assuming the operand is normalized.

GDFIXR G Format Double Fix and Round

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0

025	00	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

If the exponent of the G format floating point number in location *E*,*E* + 1 is > 70, set Overflow and Trap 1, and go immediately to the next instruction without affecting the ACs or the contents of *E*,*E* + 1 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 double length fraction *N* = *X* − 59 places to the correct position for its order of magnitude with the binary point at the right of bit 71. For positive *N*, shift left bringing 0s into bit 71 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 double length integral part. Place the result in AC,AC + 1.

Notes. The overflow test checks for a value ≥ 2^{70} assuming the operand is normalized.

DGFLTR Double G Format Float and Round

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	027	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

Shift the magnitude part of the double length fixed point integer from location *E*,*E* + 1 right eleven places, insert the exponent 70 (in excess 1024 form) into bits 1–11 to move the shifted binary point to the left of bit 12 (70 = 59 + 11), and normalize the fraction bringing first the bits originally shifted out and then 0s into bit positions vacated at the right. If fewer than eleven bits (left shifts) are needed to normalize, use the next bit to round the double length fraction. Place the result in AC,AC + 1.

GSNGL G Format to Single Precision

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	021	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

If the exponent of the G format floating point number in location *E*,*E* + 1 is > 127 or < –128, set Overflow, Floating Overflow and Trap 1 (and Floating Underflow if < –128), and go immediately to the next instruction without affecting the ACs or the contents of *E*,*E* + 1 in any way.

Otherwise convert the excess 1024 exponent in the doubleword from *E*,*E* + 1 to excess 128 form, and shift the fraction left three places to the correct position for single precision format. Round the high order word (reducing the fraction from 59 bits to 27), and place the resulting single precision number in AC. If rounding produces an exponent > 127, set Overflow, Floating Overflow and Trap 1; the result stored has an exponent 256 greater than the correct one.

Notes. Rounding can overflow only if the original operand has exponent 127 and fractional magnitude $\geqslant 1 - 2^{-28}$.

GDBLE Single Precision to G Format

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	022	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

Shift the fraction of the single precision floating point number from location *E* right three places in a double length register, with 0s at the right, to the correct position for G format, and convert the excess 128 exponent to excess 1024 form. Place the resulting G format number in AC,AC + 1.

Scaling

Two floating point instructions are in a category by themselves: they change the exponent of a number without changing the significance of the fraction. In other words they multiply the number by a power of 2, and are 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 or 2^{11} in magnitude respectively for single precision or G format operations. In other words the effective scale factor *E* is the number composed of bit 18 (which is the sign) and bits 28–35 or 25–35 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 or –1048 to +1023.

FSC Floating Scale

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

If the single precision fractional part of AC is zero, clear AC. Otherwise add the 8-bit signed 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. 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 Trap 1, Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < –128, set Trap 1, Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.[12A]

The above instruction can be used to float a fixed number with twenty-seven 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 ($233_8 = 155_{10} = 128 + 27$). Of course this is useful only in the KA10, which lacks the conversion instructions.

[12A] *Caution*: In the KI10 and KA10 only, extreme overflows are not detected properly in this instruction. An exponent > 255 sets Floating Underflow, and an exponent < –256 fails to set it.

GFSC G Format Floating Scale[12B]

1 2 3	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	0 3 1	00	*I*	*X*	*Y*	Bits 9–12 = 0.
	0 8	9 12	13	14 17	18 35	

If the G format fractional part of AC is zero, clear AC,AC + 1. Otherwise add the 11-bit signed scale factor given by *E* to the exponent part of AC,AC + 1 (thus multiplying AC,AC + 1 by 2^E), normalize the resulting doubleword bringing 0s into bit positions vacated at the right, and place the result back in AC,AC + 1. 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 > 1023, set Trap 1, Overflow and Floating Overflow; the result stored has an exponent 2048 less than the correct one. If < –1024, set Trap 1, Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 2048 greater than the correct one.

KA10 Software Double Precision

These instructions are regarded as obsolete — they are solely for assisting in the KA10 software implementation of double precision floating point arithmetic. Hence they exist only in the KA10, the KI10, and in those KL10s whose microcode implements them specifically for compatibility with KA10 usage. A programmer who employs these instructions must be aware that the double length format for KA10 software double precision is not the same as the standard double precision format given in §1.4. A double length number in KA10 software double precision format 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. For example, 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

[12B] This instruction is trapped as an unassigned code except in a KL10 that runs microcode version 271 or greater.

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

Routines for performing software double precision arithmetic are made possible by the six instructions described here. Four of these do the basic operations with normalization; the double length number in software format is used as a dividend or appears as the result in addition, subtraction or multiplication. The remaining two instructions do not normalize: one negates a software double length number, the other performs a special unnormalized addition for manipulating low order parts of numbers without shifting them from their proper positions. In the instructions for the basic operations, the exponent that results from normalization is tested for overflow or underflow. If the exponent is > 127, set Trap 1, Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < –128, set Trap 1, Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

NOTE

> These instructions are solely for assiting in KA10 software double precision floating point arithmetic. In any processor that does not implement them, their codes are unassigned, and they therefore execute as MUUOs rather than performing the operations given in the following descriptions.

DFN Double Floating Negate

1 3 1	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Negate the software 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.

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

DFN AC,AC is undefined.

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.[13] If the double length fraction in the sum is zero, clear AC+1. Otherwise normalize the sum only if the magnitude of its fractional part is ⩾ 1, and place the high order part of the result in AC+1. The original contents of AC and *E* are unaffected.

If the exponent of the sum following the one-step normalization is > 127, set Trap 1, Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one.

Notes. 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.

FADL Floating Add Long

141	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Floating add the contents of location *E* to AC[13]. If the double length fraction in the sum is zero, clear AC,AC+1. 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 AC. If the exponent of the sum is < –101 (–128 + 27) or the low order half of the fraction is zero, clear AC+1. Otherwise place a low order word for a double length result in AC+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.

FSBL Floating Subtract Long

151	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Floating subtract the contents of location *E* from AC[14]. If the double length fraction in the difference is zero, clear AC,AC+1. 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 AC. If the exponent of the difference is < –101 (–128 + 27) or the low order half of the fraction is zero, clear AC+1.

[13] The caution given in footnote 10 for FAD applies to this instruction as well.

[14] The caution given in footnote 11 for FSB applies to this instruction as well.

Otherwise place a low order word for a double length result in AC+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.

FMPL Floating Multiply Long

1 6 1	*A*	*I*	*X*	*Y*
0 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 AC,AC+1. 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 AC. 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+1. Otherwise place a low order word for a double length result in AC+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.

FDVL Floating Divide Long

1 7 1	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the magnitude of the software format double length fraction in AC,AC+1 is greater than or equal to twice the magnitude of the fraction in location *E*, set Trap 1, Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

If the division can be performed, floating divide the software format operand in AC,AC+1 by the contents of location *E*. Calculate a quotient fraction of 27 bits. If the fraction is zero, clear AC. 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 AC.

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

NOTE

> In the KL10, a negative quotient is represented by a twos complement only when the remainder is zero — otherwise it is a ones complement. In the KI10 and KA10, a twos complement is used for a negative quotient regardless of the value of the remainder.

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

A nonzero unnormalized dividend whose entire high order fraction is zero produces a zero quotient. In this case AC+1 is cleared in the KI10 but may receive rubbish in other processors.

2.4 Boolean Functions

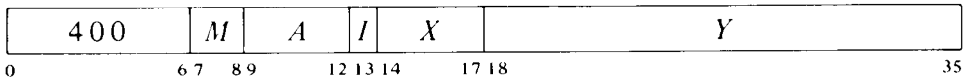
For logical operations the PDP–10 has instructions for shifting and rotating (§2.5) as well as for performing the complete set of sixteen Boolean functions of two variables (including those in which the result depends on only one or neither variable). The Boolean functions operate bitwise on full words, 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 at the end of the section 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

* In section 0 the immediate source is 0,E in all cases. But in a nonzero section, setting AC to immediate memory instead uses the entire extended effective address E as the source, including the section number (the left part of E).

SETZ Set to Zeros

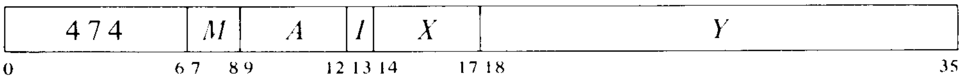


Change the contents of the destination specified by M to all 0s.

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

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

SETO Set to Ones

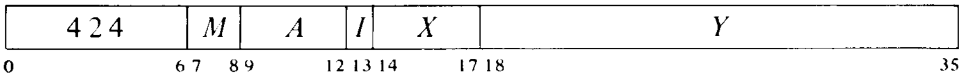


Change the contents of the destination specified by *M* to all 1s.

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

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

SETA Set to AC



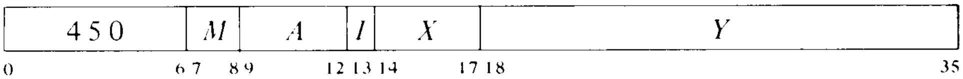
Make the contents of the destination specified by *M* equal to AC.

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

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

SETAM and SETAB are both equivalent to MOVEM, which is the preferred instruction (all move AC to location *E*).

SETCA Set to Complement of AC

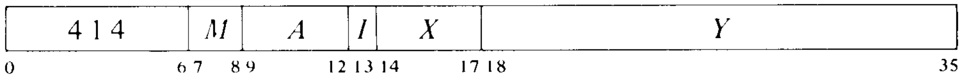


Change the contents of the destination specified by *M* to the complement of AC.

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

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

SETM Set to Memory



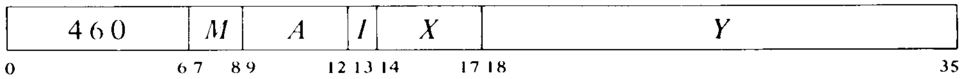
Make the contents of the destination specified by *M* equal to the specified operand.

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

If the program is running in a nonzero section, the instruction SETMI is called XMOVEI (§2.1), which performs an analogous function with an extended immediate operand (effective address).

Notes. SETM is equivalent to MOVE. In section 0 SETMI moves the word 0,*E* to AC and is thus equivalent to MOVEI. SETMM is a no-op that writes in memory. With nonzero *A*, SETMB is equivalent to MOVES. In all cases the move instruction is preferred.

SETCM Set to Complement of Memory



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

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

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

AND And with AC

404	*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 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

410	*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 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

420	*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 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

Notes. 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, i.e. 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, i.e. by taking the equivalence function of the remaining operand and the result.

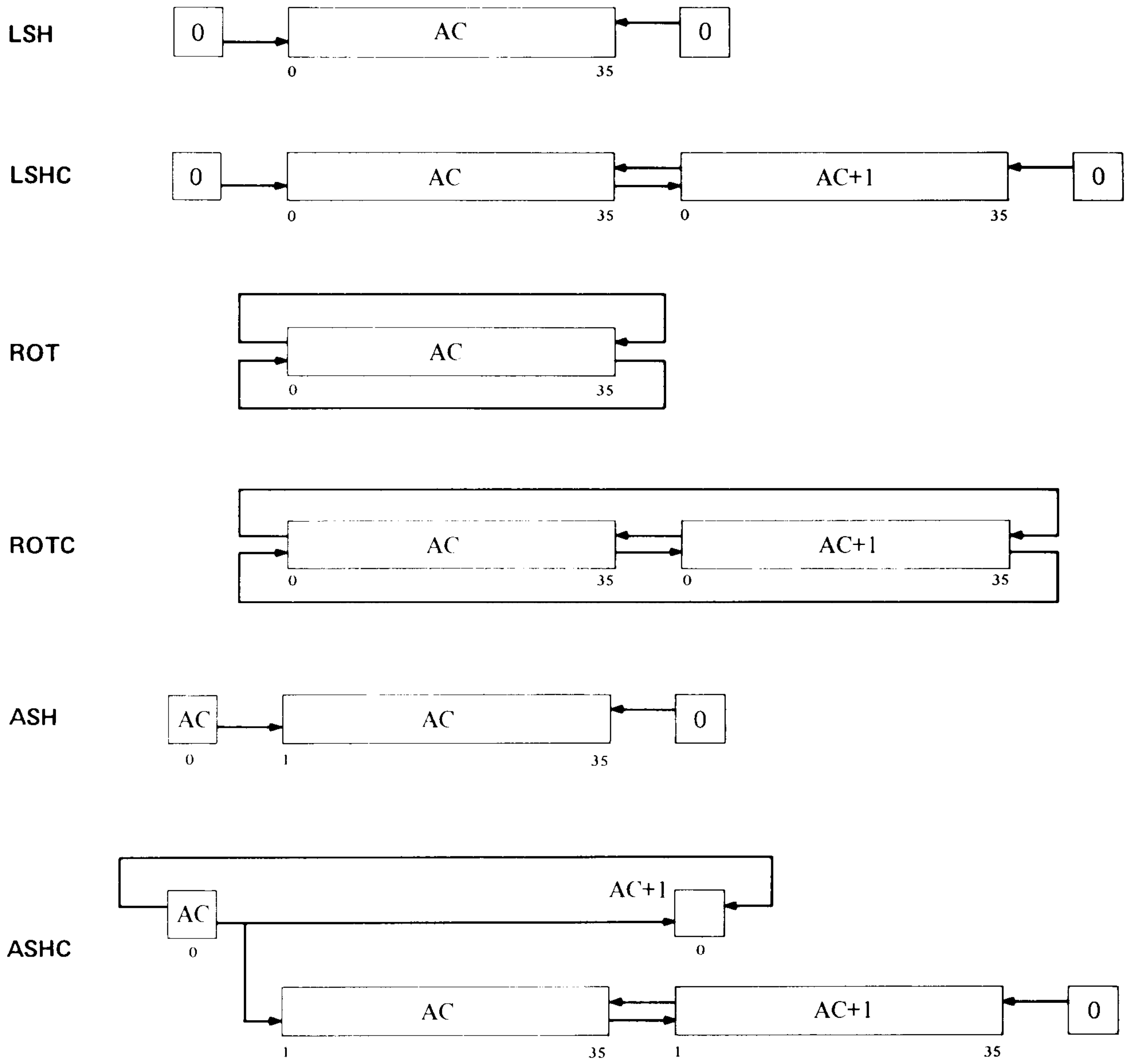
For the four possible bit configurations of the two operands, the above sixteen instructions produce the following results. In each case the result as listed is equal to bits 3–6 of the instruction word.

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

2.5 Shift and Rotate

These instructions shift or rotate right or left the contents of AC or the contents of AC,AC + 1 concatenated into a 72-bit register with AC on the left. Shifting is the movement of information bit-to-bit in a register. A logical shift involves the entire word or doubleword with no distinction among its bits, whereas an arithmetic shift involves only the magnitude, bypassing the sign. Figure 2.1 shows the movement of information these instructions produce in the accumulators. A logical shift moves the bits with 0s brought in at the end being vacated; information shifted out at the other end is lost. Rotation is a cyclic logical shift where information shifted out at one end is put back in at the other. An arithmetic shift does not affect the sign, but in a double length number, where it operates on the 70-bit string made up of the magnitude parts of the two words, it makes bit 0 of the low order word equal to the sign. 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, with truncation (see footnote 15).

Figure 2.1: Accumulator Bit Flow in Shift and Rotate Instructions



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. *E* is thus the power of 2 by which the number is multiplied.

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

Shift AC,AC + 1 the number of places specified by *E*. If *E* is positive, shift left bringing 0s into bit 71 (bit 35 of AC + 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

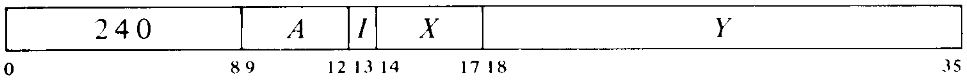
Rotate AC the number of places specified by *E*. If *E* is positive, rotate left; bit 0 is rotated into bit 35. If *E* is negative, rotate right; bit 35 is rotated into bit 0.

ROTC Rotate Combined

245	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

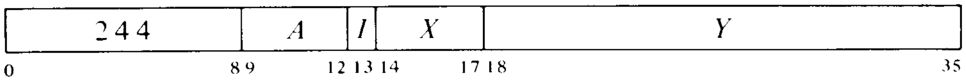
Rotate AC,AC + 1 the number of places specified by *E*. If *E* is positive, rotate left; bit 0 is rotated into bit 71 (bit 35 of AC + 1) and bit 36 into bit 35. If *E* is negative, rotate right; bit 35 is rotated into bit 36 and bit 71 into bit 0.

ASH Arithmetic Shift



Shift AC arithmetically the number of places specified by *E*. Do not shift bit 0. If *E* is positive, shift left bringing 0s into bit 35; data shifted out of bit 1 is lost; set Trap 1 and 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.[15]

ASHC Arithmetic Shift Combined



Shift AC,AC + 1 arithmetically the number of places specified by *E*. Do not shift bit 0 of AC or AC + 1, but make bit 0 of AC + 1 equal to AC bit 0 if at least one shift occurs (i.e. if *E* is nonzero). If *E* is positive, shift left bringing 0s into bit 71 (bit 35 of AC + 1); bit 37 (bit 1 of AC + 1) is shifted into bit 35; data shifted out of bit 1 is lost; set Trap 1 and 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.[15]

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

2.6 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.

[15] 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. Hence shifting –1 (all 1s) gives –1 as a result.

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
```

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[16] and place the result back in AC. If the result is greater than or equal to zero (i.e. 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.

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[16] and place the result back in AC. If the result is less than zero (i.e. 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.

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
```

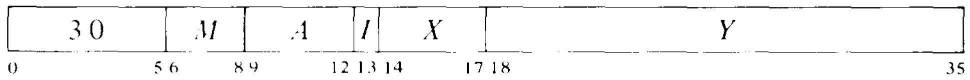
Note that even with extended addressing, AOBJN and AOBJP can be used for this sort of local indexing, as the left half being negative or zero satisfies the criterion for a local index.

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

[16] In the KA10, incrementing both halves of AC together is effected by adding 1000001_8. A count of −2 in AC left is therefore increased to zero if $2^{18} - 1$ is incremented in AC right.

CAI Compare AC Immediate and Skip if Condition Satisfied

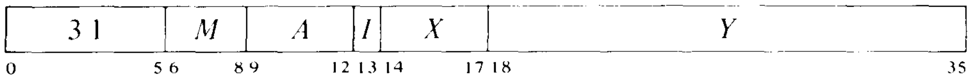


Compare AC with *E* (i.e. with the word 0,*E*) and skip the next instruction in sequence if the condition specified by *M* is satisfied.

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

Notes. CAI is a no-op in which *I*, *X* and *Y* are available for software use.

CAM Compare AC with Memory and Skip if Condition Satisfied

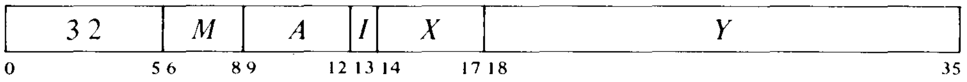


Compare AC with the contents of location *E* and skip the next instruction in sequence if the condition specified by *M* is satisfied. The pair of numbers compared may be either both fixed or both normalized floating point.

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

Notes. CAM is a no-op that references memory.

JUMP Jump if AC Condition Satisfied



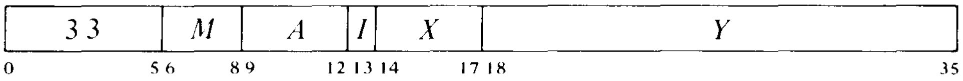
Compare AC (fixed or floating) with zero, and if the condition specified by *M* is satisfied, take the next instruction from location *E* and continue sequential operation from there.

JUMP	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

Notes. JUMP is a no-op (instruction code 320 has this mnemonic for symmetry). In it, *I*, *X* and *Y* are available for software use.

As an unconditional transfer, JRST is preferred to JUMPA.

SKIP Skip if Memory Condition Satisfied

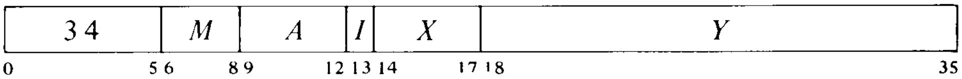


Compare the contents (fixed or floating) of location *E* with zero, and skip the next instruction in sequence if the condition specified by *M* is satisfied. If *A* is nonzero also place the contents of location *E* in AC.

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

Notes. If *A* is zero, SKIP is a no-op; otherwise it is equivalent to MOVE. (Instruction code 330 has mnemonic SKIP for symmetry.) SKIPA is a convenient way to load an accumulator and skip over an instruction upon entering a loop.

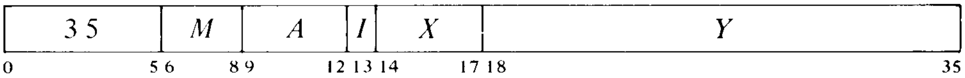
AOJ Add One to AC and Jump if Condition Satisfied



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

AOJ	Add One to AC but Do Not Jump	340
AOJL	Add One to AC and Jump if Less than Zero	341
AOJE	Add One to AC and Jump if Equal to Zero	342
AOJLE	Add One to AC and Jump if Less than or Equal to Zero	343
AOJA	Add One to AC and Jump Always	344
AOJGE	Add One to AC and Jump if Greater than or Equal to Zero	345
AOJN	Add One to AC and Jump if Not Equal to Zero	346
AOJG	Add One to AC and Jump if Greater than Zero	347

AOS Add One to Memory and Skip if Condition Satisfied



Increment the contents of location *E* by one and place the result back in *E*. Compare the result with zero, and skip the next instruction in sequence if the condition specified by *M* is satisfied. If location *E* originally contained $2^{35} - 1$, set Trap 1, Overflow and Carry 1; 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 Trap 1, Overflow and Carry 0; 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 Trap 1, Overflow and Carry 0; 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 multiprogramming environment. Suppose memory contains a routine that must be available to two processes but cannot be used by both at once. When one process finishes the routine it sets location LOCK to –1. Either process 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.

```
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 a long time to count to 2^{36}, it is alright to keep testing the lock.

2.7 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.

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)

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. 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	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

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

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
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

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

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

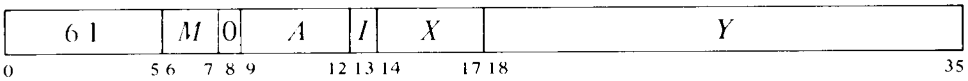
TLO Test Left, Ones, and Skip if Condition Satisfied

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

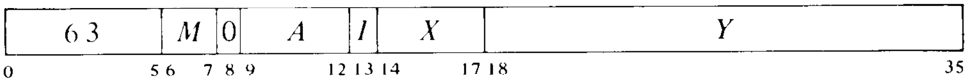


If the bits in AC corresponding to 1s in the contents of location E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

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

Notes. TDN is a no-op that references memory.

TDZ Test Direct, Zeros, and Skip if Condition Satisfied



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

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

TDC Test Direct, Complement, and Skip if Condition Satisfied

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, Complment, 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	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

Notes. TSN is a no-op that references memory.

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

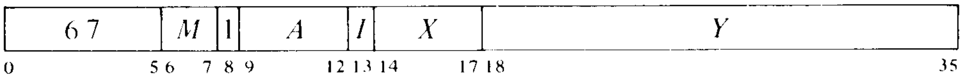
TSC Test Swapped, Complement, and Skip if Condition Satisfied

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



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 states
...               ;Do this if not both on
...               ;Skip to here if both on
```

2.8 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, but in a nonzero section the immediate mode of one of them acts in a special way, and is treated as a separate instruction. The sixteen forms are distinguished 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

* In section 0 the immediate source is 0,*E* in all cases, and selecting the left half of the source merely clears the selected half of the destination. But in a nonzero section the basic left-to-left transfer (XHLLI) instead uses the entire extended effective address *E* as the source, and it thus transfers the section number (the left part of *E*).

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

If the program is running in a nonzero section, the instruction HLLI is called XHLLI (see below), which performs an analogous function with an extended immediate operand (effective address).

Notes. In section 0 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

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

HLLO Half Word Left to Left, Ones

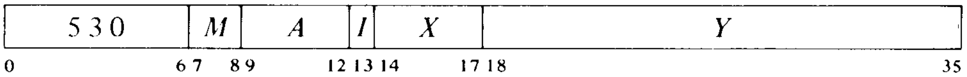
520	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

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

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

HLLE Half Word Left to Left, Extend

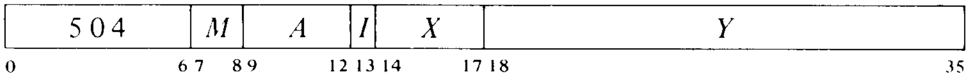


Move the left half of the source word specified by *M* to the left half of the specified destination, and make all bits in the destination right half equal to bit 0 of the source. The source is unaffected, the original contents of the destination are lost.

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

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

HRL Half Word Right to Left



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

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

HRLZ Half Word Right to Left, Zeros

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

Notes. HRLZI loads the word *E*,0 into AC and is thus equivalent to MOVSI.

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

Notes. 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

Notes. HRRZI loads the word 0,*E* into AC and is thus equivalent to MOVEI and SETMI. If *A* is zero, HRRZS merely clears the left half of location *E*.

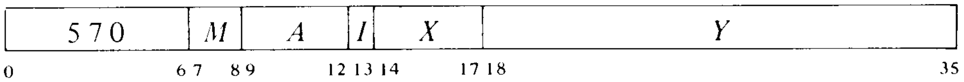
HRRO Half Word Right to Right, Ones

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

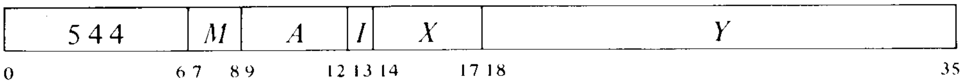
HRRE Half Word Right to Right, Extend



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

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

HLR Half Word Left to Right

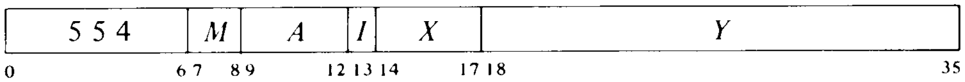


Move the left half of the source word specified by *M* to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

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

Notes. HLRI merely clears AC right.

HLRZ Half Word Left to Right, Zeros



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

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

Notes. HLRZI merely clears AC and is thus equivalent to HLLZI.

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

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

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

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

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

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. For example, this pair of instructions loads the 18-bit numbers *M* and *N* into the left and right halves respectively of accumulator XR.

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

It is not necessary to clear the other half of XR when loading the first half word. But any memory 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.)

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
```

The following instruction uses a half word transfer for inserting the section number that results from an effective address calculation into the left half of an accumulator.

XHLLI Extended Half Word Left to Left

501	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the program is running in a nonzero section, clear AC bits 0–5 and place the section number (the left part) of the effective address *E* in AC bits 6–17. If *E* is a local AC address, the section number is 1. AC right is unaffected; the original contents of AC left are lost.

If the program is running in section 0, this instruction is called HLLI, which performs an analogous function for section 0 (it moves a zero section number).

Notes. The section number given for a local AC address is that of a global AC address. Giving XHLLI with an address 20 or greater without indexing or indirection places the current PC section number in AC left, and it can thus be used to determine what section the program is in.

2.9 Program Control

A program control instruction is one that in some way affects the sequence in which instructions in the program are performed. Most such instructions are actually described in some other category, such as the arithmetic and logical testing instructions above, or the yet-to-be discussed stack instructions, UUOs, string compare instructions, and the condition IO instructions that test device flags. The present section treats the program flags, overflow trapping, and all program control instructions that do not belong to some other class. Most of these are specifically for handling subroutines. All but one 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. All but two of the jumps are unconditional; one exception tests several program flags, the other tests an accumulator.

When an instruction makes the processor leave the normal program sequence to jump to a subroutine or call the Monitor, it must save information sufficient to allow a later return to the original program. Such instructions generally save the states of the program flags and the location at which the disruption in the normal sequence occurred. Saving the program position is referred to as "saving PC," although the quantity actually saved may be the value currently contained in PC or an address one greater than that, depending on the circumstances. For example, the same instruction may be used to call a subroutine in a program or to call a service routine in an interrupt. When a return is later made using the saved address in the subroutine case, the instruction that saved PC should not be repeated — the return should be made instead to the instruction following it in normal sequence, i.e. the instruction at the address one greater than that originally in PC. In the interrupt case, on the other hand, a subsequent return has nothing to do with the instruction that saved PC — the return should be made to the interrupted instruction, the one PC pointed at when the interrupt occurred. Both cases are covered in the instruction descriptions by the phrase "save PC," and it is to be assumed that the address saved is the one appropriate to the situation in which the instruction is given.

Sometimes regarded as program control, in a somewhat trivial sense, are those instructions that do nothing. The most commonly used no-op is JFCL, which is described here. Other no-ops are among the testing and Boolean instructions discussed previously: SETA, SETAI, SETMM, CAI, CAM, JUMP, TRN, TLN, TDN, TSN.[17] Of these, SETA, SETAI, CAI, JUMP, TRN and TLN are preferred because they do not use the calculated effective address to reference memory.

The Execute Instruction

This instruction allows the programmer to execute the contents of any memory location as an instruction without altering the normal program counting sequence to do it.

XCT Execute

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

If *A* is zero or the processor is in user mode or is a KA10, execute the contents of location *E* as an instruction.[18] 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).

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

[18] *Caution:* In a private program (concealed or kernel mode) on the KI10, never give an XCT that executes an instruction in a public page. It does not work.

In executive mode this instruction performs as stated only when *A* is zero.[19] Nonzero *A* results in a so called "previous context XCT" or PXCT, whose ramifications are far more widespread than indicated here. PXCT is a very special instruction for the exclusive use of the Monitor, and it is described in the section on memory management in the system operations chapter for each processor.

Conditional Jumps

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+1 and go on to the next instruction in sequence.

If AC is not zero, count the number of leading 0s in it (0s at the left of the leftmost 1), and place the count in AC+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+1 are lost.

Notes. When AC is negative, the second accumulator is cleared, just as it would be if AC were zero.

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

To select one or a combination of these flags 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).

[19] The KA10 lacks previous-context capability. On that processor and in user mode on any processor, *A* is ignored, but it is reserved and should be zero.

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 Cary 1	25510
JCRY	JFCL 6,	Jump on Carry 0 or 1	25530
JFOV	JFCL 1,	Jump on Floating Overflow	25504

To left-normalize a positive integer in AC use

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

The flags tested by JFCL are described in detail below. 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 preferred no-op as it neither fetches nor stores an operand, and bits 18–35 of the instruction word can be used to store information.

JFCL is the only jump that can test any of the flags. But it can test only four of them, 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. For example, it may test the carry flags simply to take appropriate action in a multiple precision fixed point routine.

Program Flags

When an instruction saves the program flags, it loads their states into bits 0–12 of a word as shown here,

OVERFLOW	CARRY 0	CARRY 1	FLOATING OVERFLOW	FIRST PART DONE	USER	USER IN-OUT	PUBLIC	ADDRESS FAILURE INHIBIT	TRAP 2	TRAP 1	FLOATING UNDERFLOW	NO DIVIDE
PREVIOUS CONTEXT PUBLIC						PREVIOUS CONTEXT USER						
0	1	2	3	4	5	6	7	8	9	10	11	12

where the upper part of a double box indicates the flag saved in user mode, and the lower part indicates that saved in KL10 and KI10 executive mode. The flag listed in the lower part for bit 6 also applies to KS10 executive mode, but since the KS10 has no public mode, bit 0 always receives the state of the Overflow flag and bit 7 is not used. The KS10 also lacks a flag for bit 8. (In KA10 executive mode bits 0 and 6 receive the Overflow state and the (meaningless) User In-out state, and bits 7–10 are not used as their flags do not exist.)

Where the flags are saved (in an accumulator or memory location) and what other information is saved with them depends on the instruction and the circumstances of its execution. But whenever the flags are saved, their states are always stored in bits 0–12 of a word in the configuration shown. Some instructions when executed in section 0 save the flags and the in-section part of PC in a so-called "PC word" like this.

FLAGS	00	IN-SECTION PC
0 12	13 17	18 35

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. When such instructions are performed in a nonzero section, they generally save only the extended PC without flags. Other instructions, executable only in the KS10 and the extended KL10, combine the flags and the full PC in what is referred to as a "flag-PC doubleword" with this format,

<table>
<tr><td colspan="2">FLAGS</td><td>00</td><td>PROCESSOR-DEPENDENT INFORMATION</td></tr>
<tr><td>00</td><td colspan="3">PC</td></tr>
<tr><td>0 5</td><td>6 12</td><td>13 17</td><td>18 35</td></tr>
</table>

where still other information may be saved in the rest of the flag word. In a manner analogous to the PC word, nothing is ever stored in bits 13–17 of the first word or bits 0–5 of the second. Hence when the second word is addressed indirectly, it is interpreted as global and can produce neither indexing nor further indirect addressing. Note however that if it is used from an index register, it is taken as global or local depending on whether or not bits 6–17 are zero. Nothing is saved in the right half of the flag word.

Certain instructions can use bits 0–12 of a word to set up the program flags to restore them to their original states following an interruption or to control specific situations. Restoration of course assumes the flags are being restored from a word in which they were previously saved. When the flags are saved, the flag bits reflect the states and flags appropriate to the current situation. At a transition from one mode to another, the flags saved are those of the mode the processor is leaving, and the flags restored are those for the mode the processor is entering. For example, when the user calls the Monitor, bit 5 of the flag word is set; and the User flag must be cleared, either automatically or by a 0 in bit 5 of a restoring flag word. Moreover Overflow and User In-out are saved, but the flag bits used for restoration are adjusted to produce the correct states for the previous context flags. No conflict can result concerning bit 6, as User In-out exists only in user mode, and Previous Context User exists only in executive mode. On the other hand, although only one flag is ever saved in bit 0, at restoration bit 0 conditions the states of both Overflow and Previous Context Public (if present). The latter is irrelevant in user mode, but the executive programmer must be aware that if he wishes to use Overflow or give a JFCL to test it, its initial state is that assigned to Previous Context Public rather than that resulting from any arithmetic operation. When a return is made to an interrupted executive program via a flag-PC doubleword in an extended processor, the previous context section for that program is also restored from bits 24–35 of the flag word.

By manipulating the bits used to restore the flags, the programmer can set them up in any desired way, except that the hardware contains interlocks so 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 a trap flag immediately causes a trap.

The following lists the meaning of the information contained in bits 0–12 of a flag word at the time the flags are saved. Bits 0 and 6 are given only for user mode, as the special executive flags are relevant only to the previous context XCT instruction and are left for the discussion of system operations. Remember (§2.2) that overflow is determined directly from the carries, not the carry flags, which give useful information only if no more than one instruction that can set them occurs between clearing and reading them. The explanations assume the flags reflect normal circumstances — not arbitrary rigging. An *x* in a mnemonic indicates any letter (or none) that may appear in the given position to specify the mode, e.g. ADD*x* comprises ADD, ADDI, ADDM, ADDB.

Bit *Meaning of a 1 in the Bit*

0 Overflow — any of the following has occurred:

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

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

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

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

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

An FIX, FIXR, GFIX or GFIXR has fetched an operand with exponent > 35.

A GDFIX or GDFIXR has fetched an operand with exponent > 70.

A GFIXR has fixed a number with exponent 35 and fraction $\geqslant 1 - 2^{-36}$.

Floating Overflow has been set (bit 3).

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:

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

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

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

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

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

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

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

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

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

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

An MOVN*x* has negated zero.

A DMOVN or DMOVNM has negated zero (this condition does not affect the flags in the KI10).

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 $\geqslant 2^{35}$.

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

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

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

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

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

A DMOVN or DMOVNM has negated -2^{70} (this condition does not affect the flags in the KI10).

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 standard range floating point instruction other than FLTR or DFN, the exponent of the result was or would have been (GSNGL) > 127.

In a G format floating point instruction other than GFLTR, DGFLTR or GDBLE, the exponent of the result was > 1023.

Floating Underflow (bit 11) has been set.

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

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. For example, 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. Thus when the processor restarts the instruction, it must retrieve the pointer but *not* increment it. Note however that this flag is solely for use by the hardware: it is saved and restored by the Monitor, and the user should never touch it. On the other hand, if a trap handler (which may be supplied by the user) does any byte operations, the state of this flag must be taken into account; for details refer to the discussion of "special considerations" at the end of each of the sections on the interrupt.

5 User — the processor is in user mode.

6 User In-out — even with the processor in user mode, the program can use in-out instructions.

7 Public[20] — the last instruction performed was fetched from a public area of memory, i.e. the processor is in user mode public or executive mode supervisor.

8 Address Failure Inhibit[21] — an address failure cannot occur during the next instruction.

9 Trap 2[21] — if bit 10 is not also set, stack overflow has occurred. Unless the pager is disabled, the setting of this flag immediately causes a trap as explained at the end of this section. At present, bits 9 and 10 cannot be set together by any hardware condition.

10 Trap 1[21] — if bit 9 is not also set, arithmetic overflow has occurred. Unless the pager is disabled, the setting of this flag immediately causes a trap as explained at the end of this section. At present, bits 9 and 10 cannot be set together by any hardware condition.

11 Floating Underflow — either of the following has set Overflow and Floating Overflow:

> In a standard range floating point instruction other than FLTR or DFN, the exponent of the result was or would have been (GSNGL) < –128.
>
> In a G format floating point instruction other than GFLTR, DGFLTR or GDBLE, the exponent of the result was < –1024.

[20] Not available in the KA10 or KS10.

[21] Not available in the KA10.

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

In a DIV*x* or DDIV the high order half of the dividend was greater than or equal to the divisor.

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

In an FDV*x*, FDVR*x*, DFDV or GFDV 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.

In an ADJBP the number of bytes per word was zero.

The JRST Instruction

The basic use of this instruction is as a straightforward jump — it is the fastest jump and is the preferred instruction for such use. However it also allows the programmer to select individual functions by means of bits 9–12 of the instruction word. All KI10 and KA10 functions are included in the KL10–KS10 set, but the method of decoding is so different that the instruction is described twice, first for the KL10 and KS10, then for the earlier processors. Most of the functions are illegal in some circumstances on at least some processors; when a function is illegal, the instruction executes as an MUUO (§2.16) instead of performing the given function. The instruction descriptions explain what each function does when it is legal. Between the two descriptions is a table that indicates which of the functions are legal in which pıocessors under what circumstances.

JRST Jump and Restore (KL10–KS10)

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

Perform the function specified by *F* if it is legal. At present only ten functions are defined, and for all but one of these MACRO recognizes individual mnemonics for generating the combined 13-bit instruction codes (including bits 9–12). The defined functions, with their function codes, mnemonics, and combined instruction codes are as follows.

F	*Mnemonic and Instruction Code*	*Function*
00	JRST 25400	Jump to location *E*.
01	PORTAL 25404	If the instruction has been taken from a nonpublic area, clear Public; then jump to location *E*. A location containing a PORTAL is the only valid entry to a nonpublic area, and the instruction places the processor in concealed or kernel mode. Note that this function is equivalent to function 0 except when the instruction is taken from a private area by a public program, an event that cannot occur in a KS10 as it has no public mode.
02	JRSTF 25410	Restore the program flags from bits 0–12 of the final word used in the effective address calculation (indirect or index word), and jump to location *E*. CAUTION Restoring the flags requires that the instruction use indexing or indirect addressing. Without indexing or indirection the result is indeterminate.

Restoration of all but the user and Public flags is directly according to the contents of the corresponding bits in the flag word: 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, i.e. if User is clear. A 1 in bit 7 sets Public, but a 0 clears it only if the JRST is being performed in executive mode with a 1 in bit 5 (i.e. 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.

Notes. The flag bits are assumed to be in a previously stored PC word. If the PC word was stored in AC (as in a JSP), a common procedure is to use AC to index a zero address (e.g. 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.

04 HALT 25420

Load *E* into PC and halt the processor. While the KL10 is halted the microcode runs in the halt loop, in which it will handle interrupts on level 0 and will respond to console and diagnostic functions from the front end. The KS10 microcode performs the halt sequence discussed in §4.7, and then runs in the halt loop in which it responds only to commands from the console.

NOTE

The halt occurs of course only when the function is legal. But for debugging purposes the function is often used when illegal (and it executes as an MUUO).

05 XJRSTF 25424

Restore the program flags and PC (and the previous context section, if appropriate) from a flag-PC doubleword in location *E*,*E* + 1, and continue performing instructions in normal sequence beginning at the location then addressed by PC. Restrictions on the manipulation of the flags by the flag bits are the same as those for JRSTF given above.

06 XJEN 25430

Restore the level on which the highest priority interrupt is currently being held (dismiss the interrupt (§§3.1, 4.1)), and then perform an XJRSTF (function 5).

Notes. This instruction can be used in any section, and it is the only way to dismiss an interrupt routine or restore an interrupted program in a nonzero section.

07 XPCW 25434

Save the program flags and PC (and the previous context section, if relevant) in a flag-PC doubleword in location *E*,*E* + 1. Then restore the flags and PC from a flag-PC doubleword in location *E* + 2,*E* + 3, and continue performing instructions in normal sequence beginning at the location then addressed by PC. Restrictions on the manipulation of the flags by the flag bits are the same as those for JRSTF given above.

		Notes. This instruction can be used only for calling an interrupt routine in a KS10 or an extended processor. In the latter case it is the recommended instruction. When it is so used, the four-word block at location *E* must be in section 0, as that is the default section for instructions executed in interrupt locations. The return from the routine would typically be made by an XJEN that addresses the same block (i.e. that uses the first doubleword in the block).
10	25440	Restore the level on which the highest priority interrupt is currently being held (dismiss the interrupt (§§3.1, 4.1)).
12	JEN 25450	Restore the level on which the highest priority interrupt is currently being held (dismiss the interrupt (§§3.1, 4.1)), and then perform a JRSTF (function 2).
14	SFM 25460	Save the program flags in bits 0–12 of memory location *E* (clear bits 13–23). If the instruction is given in executive mode in an extended processor, save the previous context section in bits 24–35 (otherwise clear these bits as well).

The remaining undefined functions execute as MUUOs, as does any defined function when it is illegal.

One can program a function by giving JRST with the equivalent of an AC address that specifies the function code. For the sixteen forms of the instruction, the following table lists the individual mnemonic if any, and indicates where that form of the instruction is legal in each of the five processors. The meanings of the symbols used to define the legal domains of the functions are as follows.

Yes	Legal everywhere
Z	Legal only in section 0
NZ	Legal only in a nonzero section
IO	Legal wherever IO instructions are legal, i.e. in user IO mode (User and User In-out both set) and in kernel mode (executive mode in the KS10 and KA10)
K	Legal only in kernel mode (in the KS10, executive mode is kernel mode)
No	Legal nowhere (always executes as an MUUO)
–H	Legal where indicated by first symbol but causes a halt

			Single-			
		Extended	*section*			
		KL10	*KL10*	*KS10*	*KI10*	*KA10*
JRST 0,	JRST	Yes	Yes	Yes	Yes	Yes
JRST 1,	PORTAL	Yes	Yes	Yes	Yes	Yes
JRST 2,	JRSTF	Z	Yes	Yes	Yes	Yes
JRST 3,		No	No	No	Yes	Yes
JRST 4,	HALT	K–H	K–H	K–H	K–H	IO–H
JRST 5,	XJRSTF	Yes	No	Yes	K–H	IO–H
JRST 6,	XJEN	IO	No	K	K–H	IO–H
JRST 7,	XPCW	IO	No	K	K–H	IO–H
JRST 10,		IO	IO	IO	K	IO
JRST 11,		No	No	No	K	IO
JRST 12,	JEN	Z ∧ IO*	IO	IO	K	IO
JRST 13,		No	No	No	K	IO
JRST 14,	SFM	NZ ∨ IO*	No	K	K–H	IO–H
JRST 15,		No	No	No	K–H	IO–H
JRST 16,		No	No	No	K–H	IO–H
JRST 17,		No	No	No	K–H	IO–H

* JEN is legal only where IO is legal in section 0; SFM is legal anywhere in a nonzero section and also where IO is legal in section 0.

JRST Jump and Restore (KI10–KA10)

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

Perform the functions specified by *F* if they are legal; then if the function was performed and the processor is not halted, 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 if Legal*
9	Restore the level on which the highest priority interrupt is currently being held (dismiss the interrupt (§§5.2, 5.5)).
10	Halt the processor. When it stops, the AR lights on the KI10 and the MA lights on the KA10 display an address one greater than that of the location containing the instruction that caused the halt, and PC displays the jump address (the location from which the next instruction will be taken if the operator causes the processor to resume operation without changing PC). AR or MA actually displays the address of the location that would have been executed next had the JRST been replaced by a no-

op. So except for a JRST in an interrupt, the lights point to the location one beyond that containing the instruction that caused the halt. This instruction is ordinarily the JRST or perhaps an XCT, but could even be a UUO.

11 Restore the program flags from bits 0–12 of the final word used in the effective address calculation. Hence to restore flags requires that the instruction use indexing or indirect addressing. Restrictions on the manipulation of the flags by the flag bits are the same as those for the KL10 JRSTF given above. (The notes on addressing given there also apply.)

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 the only valid entry to a nonpublic area, and the instruction places the processor in concealed or kernel mode.

While the KA10 is in user mode, if this instruction is executed as an interrupt instruction or by an MUUO, the processor leaves user mode.

Notes. 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 Level	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 level and restoring the flags for the interrupted program. It is a combination of JRST 10, and JRSTF.

CAUTION

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

Subroutine Calling

Currently the stack instructions PUSHJ and POPJ, described in the next section, are employed almost universally for handling subroutines. Described here are four traditional subroutine-handling instructions, the first two of which still enjoy some popularity.

JSR Jump to Subroutine

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

A is not used.[22]

In section 0 save the program flags and PC in a PC word in location *E*; in a nonzero section save PC in bits 6–35 of location *E* (clear bits 0–5). In either case jump to location $E+1$. 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), the processor leaves user mode, clearing Public. (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

In section 0 save the program flags and PC in a PC word in AC; in a nonzero section save PC in AC bits 6–35 (clear bits 0–5). In either case jump to location *E*. The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared.

While the KI10 or KA10 is in user mode, if this instruction is executed as an interrupt instruction (or by a KA10 MUUO), the processor leaves user mode, clearing Public. (An interrupt that is not dismissed automatically returns control to kernel mode.)

When a subroutine is called in section 0 by a JSR M, the typical method of returning from it is to give a JRSTF @ M, which not only returns to the original program but also restores the original states of the program flags using the PC word saved by the JSR. In a nonzero section there is an analogous procedure using a flag-PC doubleword. The subroutine is called by

```
        SFM     M
        JSR     M+1
```

and the return is made by XJRSTF M. A similar analogy holds for JSP. The following discussion of subroutine calling is geared to section 0, but its extension to nonzero sections is straightforward, by such substitutions as a flag-PC doubleword for a PC word, XJRSTF for JRSTF, and so forth.

[22] The *A* portion of this instruction is reserved and should be zero.

JSR and JSP 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
JSR     OVRFLO      ;Jump over this if Overflow clear
  .
  .
  .
```

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:

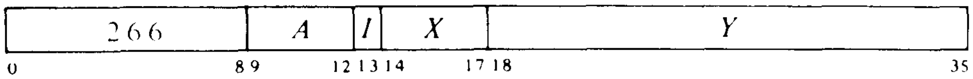
```
JSP     T,PRINT     ;Put PC word in accumulator T
 .                  ;Text inserted here by ASCIZ
 .                  ;pseudo-instruction, which
 .                  ;automatically places a zero (null)
                    ;character at the end
...                 ;Next instruction here
```

The subroutine can use T as a byte pointer (§2.11) that 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 for
                             ;size 7, position 36
        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
```

The next two instructions have no capacity for handling extended addresses. Hence their usefulness is limited to making intrasection subroutine calls. However most programmers regard them as obsolete anyway, as they have been supplanted entirely by the stack instructions.

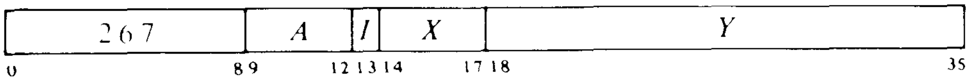
JSA Jump and Save AC



Save AC in location E, the in-section part of E in AC left, and the in-section part of PC in AC right. Then jump to location $E+1$. 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, the processor leaves user mode.

JRA Jump and Restore AC



Place the contents of the location addressed by AC left into AC, and jump to location E.

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.

Overflow Trapping[23]

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

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

An instruction in which an arithmetic overflow condition occurs sets Overflow and Trap 1, and an instruction in which a stack overflow occurs sets Trap 2. Note that it is the overflow condition that sets Trap 1 — not the state of the Overflow flag. Hence an overflow is trapped even if Overflow is already set. Note also that the trap flags have no effect at all when paging is disabled. Otherwise 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.

Trap Flags Set	*Trap Type*	*Trap Number*	*Location*
Trap 1 only	Arithmetic overflow	1	421
Trap 2 only	Stack overflow	2	422
Trap 1 and 2	Not used by hardware[24]	3	423

A trap instruction is executed in the same address space and section 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, e.g. 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.

[23] This feature is not available in the KA10. That processor is limited to the use of internal conditions that can act through the interrupt system (§5.5).

[24] A trap can be produced artifically simply by setting up the trap flags from bits in a flag word. In this way the program can also use trap number 3, which at present cannot result from any hardware-detected condition and is reserved.

The location of the instruction that caused the overflow can be determined from PC unless the instruction jumped, in which case its location can be determined only for a PUSHJ, from the stack entry. The trap instruction (the final instruction in an XCT and/or LUUO string) clears the trap flags, so the processor returns to the interrupted program unless the trap instruction changes PC. Thus the trap instruction can be a no-op (which ignores the trap), a skip, a jump, or anything else. However, should the trap instruction itself set a trap flag (not necessarily the same one), a second trap occurs. An arithmetic instruction that overflows on every iteration produces an infinite loop if used as a trap instruction for arithmetic overflow. A stack instruction in a stack overflow trap can overflow only once. (The memory allocated to a stack should have at least one extra location to handle this case — two extras if the program and the trap both use the same pointer.)

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

2.10 Stack Operations

A stack, or 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 stack. We can deposit data in *a*, *b*, *c*, *d*, then read *d*, then write in *d* and *e*, then read *e*, *d*, *c*, etc. Adding an item to the stack is referred to as "pushing" or "pushing down"; removing an item is "popping." The stack is used in two ways: for handling data, and for saving and restoring PC, as in calling and returning from a subroutine.

The mechanism for keeping track of the list is a stack pointer, which specifies the position of the last item in it. This pointer is always kept in an accumulator. In section 0 the pointer has two parts: the right half contains the address of the last item, and the left half can contain a control count. An instruction that pushes an item onto the list increments both parts of the pointer by one, and then places the item in the newly specified location; an instruction that pops an item takes it from the currently specified final location, and then decrements both parts of the pointer by one so it points to what has become the final item. To help prevent mismanagement of the stack, the control count in the left half is monitored for overflow. The overflow condition, which sets the Trap 2 flag, is a change in the count from negative to zero on a push or from zero to negative on a pop. The KA10 lacks the trapping feature, so in it overflow sets the Pushdown Overflow flag, which requests an interrupt on the level assigned to the processor (§5.6).

By 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. The common practice is to limit the size of the list. If only jump addresses are kept in the stack, the size limitation restricts the depth of nesting. A technique to catch extra popping of jump addresses is to put the address of an error routine at the bottom of the stack.

In a nonzero section there are two pointer formats, local and global. A local pointer is just like the one used in section 0, with the same manipulation in pushing and popping, except that the left half must be negative or zero (like a local index register). Restriction to a negative control count means it can be used only to limit the size of the list, as the only meaningful overflow condition is the change to zero on a push. AC right contains a local address that is interpreted as being in the same section as the instruction. Note that a local stack wraps around in the local section.

A global stack pointer is one in which bits 6–35 contain a global address, and since bits 0–5 must be zero, it is identified by the left half being nonzero positive. Manipulation of a global pointer by pushing and popping is simply incrementing and decrementing the 30-bit address by one, and a global stack can therefore cross section boundaries. There is no control count, but the program can limit stack size by making the pages at either end inaccessible. Note that pushing on a local stack whose pointer has already overflowed (whose control count has gone to zero) changes the pointer to the global format, and it then addresses a location in section 1. Similarly, adjusting a global stack pointer into the "section" beyond 7777 changes it to the local format. (A pointer with a 0 in bit 0 and any arbitrary configuration in bits 1–5 is interpreted as local or global depending on whether or not bits 6–17 are zero.)

The processor implements program use of the stack by providing five instructions: one for making arbitrary adjustments of the pointer, and two pairs for pushing and popping. One pair handles data; the instructions in the other are jumps that use the stack for handling subroutines.

PUSH Push

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

If the program is running in section 0 or AC left is negative (or AC bits 6–17 are zero), 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 Trap 2.[25] If the program is running in

[25] In the KA10, incrementing and decrementing both halves of AC together is effected by adding and subtracting 1000001_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. Also in the KA10 there are no trap flags, so Pushdown Overflow is set instead.

a nonzero section with a 0 in AC bit 0 and AC bits 6–17 nonzero, add one to AC, then move the contents of location *E* to the location now addressed by AC bits 6–35. The contents of *E* are unaffected, the original contents of the location added to the stack are lost.

Notes. Do not allow the pointer to address AC, as the result of the instruction is then indeterminate.

POP Pop

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

If the program is running in section 0 or AC left is negative (or AC bits 6–17 are zero), 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 Trap 2.[25] If the program is running in a nonzero section with a 0 in AC bit 0 and AC bits 6–17 nonzero, move the contents of the location addressed by AC bits 6–35 to location *E*, then subtract one from AC. The original contents of location *E* are lost.

Notes. Do not use the instruction POP AC,AC as its result is indeterminate. To decrement the pointer by one position (in other words to throw away the last item in the stack), give a POP AC,(AC) or ADJSP AC,–1.

Example. In section 0 a POP can be used to implement a reverse BLT, i.e. to transfer a block of words from one area of memory to another, starting at the largest addresses and proceeding to the smallest. To move a block of *N* words from a source area to a destination area whose maximum addresses are *S* and *D* respectively, the program must first set up a stack pointer in accumulator T, where T left contains *N* – 1 + 400000 and T right contains *S*. The transfer is then effected by this pair of instructions

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

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

PUSHJ Push and Jump

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

Take one of these three courses of action.

If the program is running in section 0, add one to each half of AC, then save the program flags and PC in a PC word in the location now addressed by AC right. If the addition causes the count in AC left to reach zero, set Trap 2.[25]

If the program is running in a nonzero section but AC left is negative (or AC bits 6–17 are zero), add one to each half of AC, then save PC in bits 6–35 of the location now addressed by AC right (clear bits 0–5). If the addition causes the count in AC left to reach zero, set Trap 2.[25]

If the program is running in a nonzero section with a 0 in AC bit 0 and AC bits 6–17 nonzero, add one to AC, then save PC in bits 6–35 of the location now addressed by AC (clear bits 0–5).

Then jump to location *E*.

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

While the KI10 or KA10 is in user mode, if this instruction is executed as an interrupt instruction (or by a KA10 MUUO), the processor leaves user mode, clearing Public. (An interrupt that is not dismissed automatically returns control to kernel mode.)

POPJ Pop and Jump

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

E is not used.[26]

Take one of these three courses of action.

If the program is running in section 0, subtract one from each half of AC. If the subtraction causes the count in AC left to reach –1, set Trap 2.[25] Then jump to the location addressed by the right half of the location that was addressed by AC right *prior* to the decrementing.

If the program is running in a nonzero section but AC left is negative (or AC bits 6–17 are zero), subtract one from each half of AC. If the subtraction causes the count in AC left to reach –1, set Trap 2.[25] Then jump to the location addressed by bits 6–35 of the location that was addressed by AC right *prior* to the decrementing.

[26] *I*, *X* and *Y* are reserved and should be zero.

If the program is running in a nonzero section with a 0 in AC bit 0 and AC bits 6–17 nonzero, subtract one from AC, and jump to the location addressed by bits 6–35 of the location that was addressed by AC bits 6–35 *prior* to the decrementing.

CAUTION

> The jump is completed before the processor responds to overflow. Hence it is impossible to determine the location of the POPJ that caused the overflow.

ADJSP Adjust Stack Pointer[27]

105	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the program is running in section 0 or AC left is negative (or AC bits 6–17 are zero), add the in-section part of *E* algebraically (bit 18 is the sign) to each half of AC. If a negative E_R changes the count in AC left from positive or zero to negative, or a positive E_R changes the count from negative to positive or zero, set Trap 2. If the program is running in a nonzero section with a 0 in AC bit 0 and AC bits 6–17 nonzero, add the in-section part of *E* algebraically to AC.

Notes. When an ADJSP changes the control count in a local pointer in a nonzero section from negative to positive, the result will appear to be a global pointer. Similarly an overflow to negative can occur only from zero, as otherwise the original would have been taken as global (excluding the irrelevant case of AC left being greater than zero only because of bits 1–5 being nonzero).

A stack 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 stack operations. To initialize a local 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
```

[27] In the KI10 and KA10 this instruction is trapped as an unassigned code.

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.

Since the stack is independent of the subroutine called, PUSHJ-POPJ can be used for multiple entries. Moreover, ordering by level is inherent in the structure of a stack, 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 easily programmed.

The stack instructions do 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 most operations. The programmer must keep track of whether a given entry in the list is data or a saved PC; in other words, generally every item inserted by a PUSH should be removed by a POP or ADJSP, and every PUSHJ should be matched by a POPJ. If flag restoration is desired in section 0, 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.

The stack is kept in a random access memory, so the restrictions on order of entry and removal of items actually apply only to the standard addressing by the pointer in stack instructions — other addressing methods can reference any item at any time. The most convenient way to do this is to use the address part of the pointer as an index. 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)
```

appends the third to last item to the end of the list (remember that *E* is calculated before the contents of P are changed).

```
          POP      P,–2(P)
```

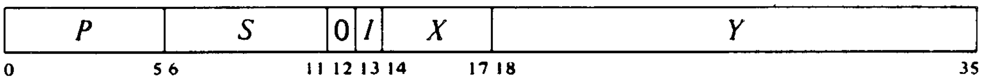
removes the last item and inserts it in place of the next to last item in the shortened list.

An ADJSP can delete an entire block from a stack, and in combination with a BLT it can be used to add a whole block.

2.11 Byte Manipulation[28]

This set of six 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. A pointer restricted only to local addressing is always one word and has the format

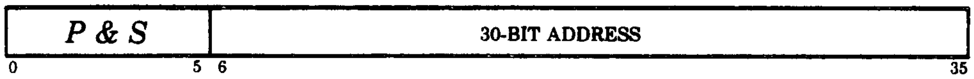


where *S* is the size of the byte as a number of bits (with zero *S* specifying a null byte), and *P* is its position as the number of bits remaining at the right of the byte in the word (e.g. if *P* is 3 the rightmost bit of the byte is bit 32 of the word). The rest of the pointer is interpreted in 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; the address calculation begins in the section containing the pointer.

In section 0 the pointer is always of the above type — local and one word — and *P* must be ≤ 36. In a nonzero section the pointer can be local in the above format, but it can also be global in either a one or two-word

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

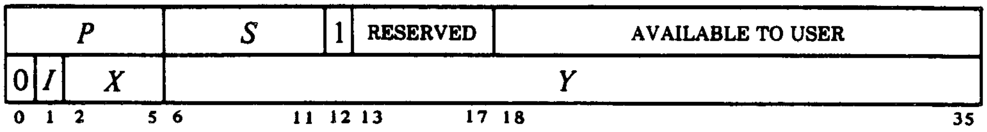
format. The one-word global pointer is available only with TOPS–20 microcode version 271 or greater, cannot use indirection, and provides for only the most common byte sizes via this format:



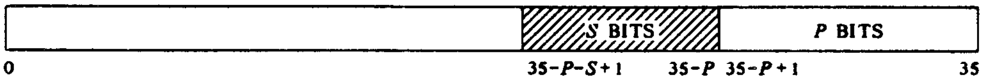
where the address can point to any section, and the left six bits specify both byte position and size by a number > 36 as follows.

P&S	*P*	*S*	*P&S*	*P*	*S*
37	36	6	49	36	7
38	30	6	50	29	7
39	24	6	51	22	7
40	28	6	52	15	7
41	12	6	53	8	7
42	6	6	54	1	7
43	0	6			
			55	36	9
44	36	8	56	27	9
45	28	8	57	18	9
46	20	8	58	9	9
47	12	8	59	0	9
48	4	8			
			60	36	18
			61	18	18
			62	0	18

For unrestricted use in a nonzero section, the pointer can be a doubleword in location *E*,*E* + 1 with this format:



which allows unlimited pointing, as *P* and *S* are independent, and the second word can be local or global, direct or indirect (see the discussion of indirect words in §1.6). The processor determines the number of words in a pointer with independent *P* and *S* by the state of bit 12 in the first word (in section 0 bit 12 is ignored and should be 0). Any type of pointer aims at a word whose format is



where the shaded area is the byte.

Bytes are always contiguous within a word, and the forward order is left to right in words and from low to high addresses. The position of the byte area in a word is called the "byte alignment." Let *P* be the position of a specified byte; 36 − *P* is then the number of bits in the left part of the word including the given byte and all byte positions at the left of it. Dividing

36 − P by S gives the number of byte positions in this left part, and the remainder is those extra bits at the left end that are not in any byte position. This number of extra bits is the byte alignment.

A block of 8-bit bytes might look like this.

Y	19 BITS	BYTE 0	BYTE 1	1	
$Y+1$	BYTE 2	BYTE 3	BYTE 4	BYTE 5	4 BITS
$Y+2$	BYTE 6	BYTE 7	BYTE 8	BYTE 9	4 BITS
•					
•					
•					

In the first word, the first byte can occupy any position, but as many bytes as will fit are packed into the rest of the word at the right. In the second and all succeeding words, the byte alignment is zero no matter where the bytes may start in the first word, and as many as will fit are packed into every word, although the last may run short. In our example the byte alignment in the first word is 3, even though two byte positions are not used: the alignment is always less than S and is the number mod S of bits at the left of the first byte. Bytes are assumed to be handled consecutively in the forward direction only, and for this type of processing the pointer is "incremented. Since bytes are contiguous and are processed from left to right, 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[29] 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.

To facilitate processing a series of bytes, two of the byte handling instructions increment the pointer before handling the byte. A typical procedure for using these instructions is to set up the pointer initially to point at the byte position preceding the first byte.

The pointer is referred to as being "at location E," which means that it is either a single word in location E or a doubleword in location E,$E+1$. Local and global pointers, and the operations associated with them as described above, are also utilized in handling byte strings, which are discussed in the three sections following this one.

CAUTION

Giving a pointer with P or S greater than 36 produces an indeterminate result in any instruction that uses it. A P of 36 should be used only for initial incrementing by an ILDB or IDPB (its effect on an LDB or DPB is indeterminate).

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.

Giving a one-word global pointer with a P&S number of 63 causes an instruction that uses it to be trapped as an MUUO.

[29] *Caution:* In the KA10 do not allow Y to reach maximum value. The whole pointer is incremented, 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 an interrupt should occur before the calculation is complete, the incremented X is used when the instruction is repeated.

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 at 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 at 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	00	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Bits 9–12 = 0.

Increment the byte pointer at location *E*, setting the byte alignment to zero if the incrementing crosses a word boundary, as explained above.

Notes. Giving this instruction code with bits 9–12 nonzero produces the ADJBP instruction described at the end of the section. In the KI10 and KA10, only the IBP form is available and bits 9–12 are ignored (but should be zero).

ILDB Increment Pointer and Load Byte

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

Increment the byte pointer at location *E*, setting the byte alignment to zero if the incrementing crosses a word boundary, 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 at location E, setting the byte alignment to zero if the incrementing crosses a word boundary, 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. For the convenience of the programmer, MACRO has a pseudoinstruction for setting up such a pointer: in assembly

```
POINT     S,Y
```

is replaced by a pointer that points to a byte of size S at position 36 in location Y. 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).

ADJBP Adjust Byte Pointer

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

$A \neq 0$.

Take one of these three courses of action depending on the value of S in the pointer at location E.

If S is 0, place an unmodified copy of the pointer in AC or AC,AC + 1.

If S is greater than 36 minus the byte alignment given by the pointer — so not even one byte will fit in a word — set Trap 1, Overflow and No Divide, and go on to the next instruction without affecting the ACs or memory.

If S is greater than 0 but less than 36 minus the byte alignment, make a copy of the pointer from location E or E,E + 1, and "adjust" the copy, forward or back, by the number of byte positions specified by AC, *preserving the byte alignment across word boundaries*: if AC contains a positive number N, adjust the copy by N bytes forward; if AC contains a negative number *–N*, adjust the copy by *N* bytes back. Place the revised pointer copy in AC or AC,AC + 1 as appropriate. The original pointer is unaffected; the original contents of AC or AC,AC + 1 are lost.

Notes. The adjustment always produces a pointer that specifies an actual byte; e.g. adjusting a pointer with a *P* of 36 by zero bytes results in a pointer that specifies the rightmost byte in the preceding word. Note that if the pointer specifies a byte alignment of zero, there is no difference between "adjusting" it by *N* and "incrementing" it *N* times (except that the latter actually modifies the pointer). Since the result goes to AC, it is not generally useful to adjust a local pointer that is in a different section from the instruction.

Giving this instruction code with a zero *A* field or in a KI10 or KA10 produces the IBP instruction described above. Note that if *S* = 0, this instruction is equivalent to MOVE.

This last instruction facilitates selection of individual bytes at arbitrary positions in an array whose format differs from the linear format used by the incrementing instructions in that the adjustment preserves the byte alignment across word boundaries. As an example of this format, let us again use 8-bit bytes where the pointer specifies one in the same position as byte 0 in our linear example at the beginning of the section. Such an array would look like this.

⋮

Y - 2	3	BYTE -10	BYTE -9	BYTE -8	BYTE -7	1
Y - 1	3	BYTE -6	BYTE -5	BYTE -4	BYTE -3	1
Y	3	BYTE -2	BYTE -1	BYTE 0	BYTE 1	1
Y + 1	3	BYTE 2	BYTE 3	BYTE 4	BYTE 5	1
Y + 2	3	BYTE 6	BYTE 7	BYTE 8	BYTE 9	1

⋮

Here the bytes are ordered in either direction from the zero position, and the byte alignment determined by the pointer is preserved throughout all words in the block. Bytes are packed as many as will fit in all words (except perhaps at either end of the block), but within the restriction that the alignment be preserved. For example, with 10-bit bytes there are always three per interior word in the linear format, but in the array format with an alignment of 8, there are only two, occupying bits 8–17 and 18–27. Specification of an arbitrary byte anywhere in the array is accomplished by using an ADJBP. The microcode makes the adjustment by changing *Y* to the location containing the byte and then setting up a new *P* for the specific byte.

Suppose we have bytes packed five to a word, a pointer at location E now points to the third byte in a given location, and we wish to retrieve the thirty-first (the fourth byte from the sixth location) beyond that. This routine loads the desired byte into AC.

```
MOVEI   AC,37
ADJBP   AC,E        ;Adjust by 31₁₀
LDB     AC,AC
```

2.12 String Manipulation[30]

This and the next two sections treat the instructions that handle strings. All string instructions are in the extended instruction set, and all therefore have a two-word format, the first word being EXTEND. The second instruction word, whose own effective address is *E1*, is at location *E0*, which is the effective address of the EXTEND. An instruction that "offsets" uses *E1* as a signed offset, in which bit 18 is the sign. An instruction that "translates" or "edits" makes use of a translation table that begins at *E1*.

A string is a sequence of bytes as specified by successive states of a standard byte pointer of the type described in the preceding section, the first page or so of which the reader should reread if he does not remember in detail the format of the pointer, the way it is incremented, and the way bytes are organized in consecutive words (specifically with zero byte alignment). The program defines a string by giving its length in number of bytes and an initial value for the pointer. Initially the pointer must point to the byte position preceding the first byte in the string, as every string instruction acts in a manner similar to a series of ILDBs or IDPBs, or in some cases both. Hence all string operations are from left to right due to the way byte pointers are incremented. A string byte pointer and length may define a string of bytes or define a string space that will receive bytes. In an instruction that moves a string, the actual string moved is referred to as the source string, and the receiving space is referred to as the destination string, even though initially the latter is a string of positions rather than bytes. Note that source and destination strings need not be the same length. When the source string is longer, only part of it will fit in the destination space. Conversely when the source is shorter, it can be inserted into part of the destination space, either starting at the left (left justified) or placed so that its final byte is in the last destination position (right justified).

Bytes may be of any size from zero bits to thirty-six, but in a given string all are the same size as indicated by the pointer. The relationship between source and destination byte sizes is a function of the way the programmer uses his data and the meaning he assigns to it. Depending on circumstances it may be desirable to spread out a source string into a destination space whose positions are larger than the source bytes (data is always right justified in a given byte position); or source bytes may be truncated to fit into smaller destination positions (the truncation being always from the left).

Most string operations make some use of bytes other than those in the strings themselves. Such bytes may be special characters found in locations $E0+1$ and $E0+2$, or substitutions supplied by a translation table. A byte

[30] In the KI10 and KA10 these instructions are trapped as unassigned codes (§2.16).

from any location not in a string defined by the pointers and lengths associated with the instruction is always from the right end of the word or half word containing it and has the same number of bits as the bytes in the string in which it will be used.

The interior of a string space is all of those bits in the words containing the string that lie between the first byte in the first word and the last byte in the last word. Since byte alignment is zero, the string is packed solid (with no unused interior bits) if 36 is an integral multiple of the byte size. For sizes that do not pack solid, there will be unused interior bits except in the last word, and they will lie at the right of the bytes in the words. If all unused interior bits are 0s initially in the string spaces (whether one or two) specified by a string instruction, they are guaranteed to be 0s at the completion of the instruction. If such bits are not all 0s initially, the subsequent states of unused interior destination bits are indeterminate (source strings are unaffected by the instructions).

Bytes in a string may represent anything — digits, letters, special characters. This section discusses the basic operations: those that compare two strings, or move a string to a new position with optional offsetting or translating of its bytes. The next section covers special operations for converting between binary and decimal, where a decimal number is a string of bytes representing decimal digits. §2.14 considers an instruction that is effectively a whole routine for complex editing of a text string.

All string instructions skip the next instruction in the PC sequence if all operations are carried out as expected, or a compare condition is satisfied, etc. Failure of a compare condition to be satisfied or something being amiss (such as loss of bytes because the source string will not fit in the destination space) usually causes the processor to perform the next instruction. Note that the "next instruction" is relative to the EXTEND (or an XCT that executes it) — in other words relative to the actual instruction PC points to. The location of the second instruction word, which is actually the operand of the EXTEND, does not affect the PC value.

Every string instruction uses a block of accumulators, which contain one or two byte pointers. A pointer may be one word or two, local or global, as explained at the beginning of §2.11. In the illustrations of the AC block format for the extend instructions, pointers are always shown as a pair of words in AC + N,AC + N + 1, where the actual byte pointer used may be in the first accumulator or in both. However the reader should note that when a global pointer is given as one word, the instruction always converts it to two.

CAUTION

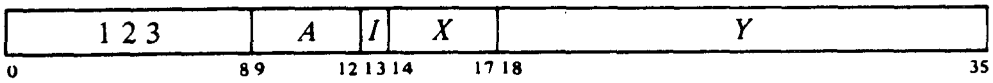
For the instructions described in this and the next two sections, the format illustrations show various parts of the accumulators and instruction words as being zero. These parts are reserved and *must be zero*. Failure to comply with this requirement will cause an extend instruction to give an indeterminate result.

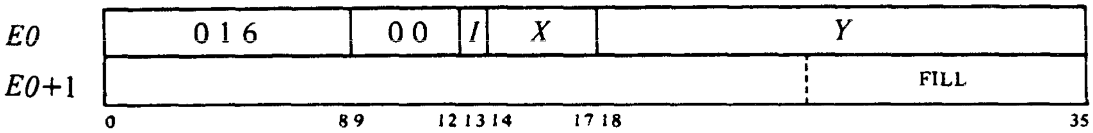
Moreover there can be no overlapping of the various quantities used in any extend instruction. The source and destination spaces must never overlap; under no circumstances should any string overlap anything else used by the instruction, such as the AC block, a translation table, an edit pattern, special character locations following *E0*, or even the instruction words themselves; and unused ACs in the specified block (such as that following a one-word byte pointer) cannot be used for any other purpose (such as an index register). Any such overlapping will cause the result of the instruction to be indeterminate.

This caution applies not only to the basic instructions discussed here, but also to those of the two sections that follow.

There are four move instructions. One right justifies the source string in the destination space, without otherwise modifying it. The others move the source string directly (i.e. left justified), with the bytes unmodified, or all offset by a constant, or translated where every byte of a given value is replaced by a corresponding substitution. The six compare instructions do not affect the specified strings; instead they are compared according to a collating sequence based on the algebraic relationships of their bytes taken as unsigned binary numbers. All of these are two-word instructions, where the first has the EXTEND code 123, and all use a block of six accumulators.

MOVSLJ Move String Left Justified



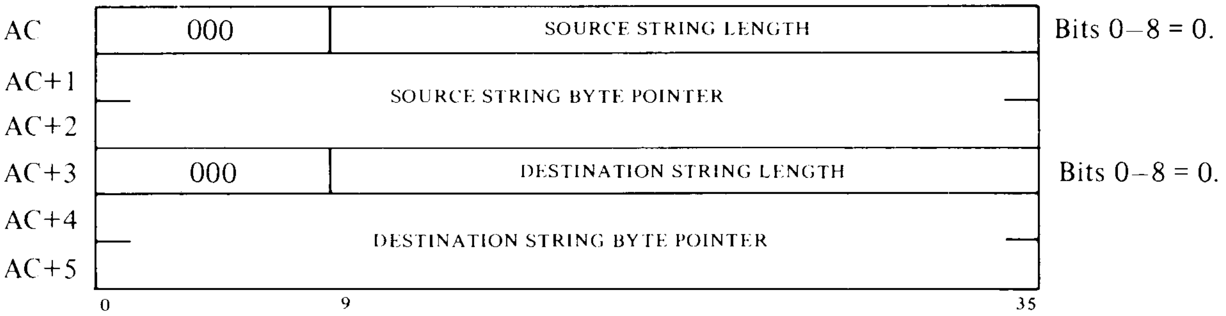


Bits 9–12 = 0
E1 is not used.[31]

Move the source string left justified into the destination string space.

[31] *I*, *X* and *Y* are reserved and should be zero.

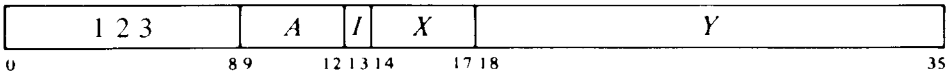
Source and destination are defined by the contents of a block of six accumulators.

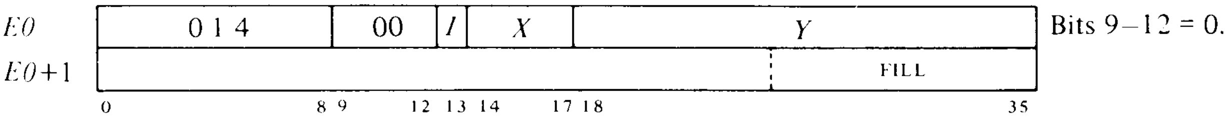

AC
AC+1
AC+2
AC+3
AC+4
AC+5
000
SOURCE STRING LENGTH
SOURCE STRING BYTE POINTER
000
DESTINATION STRING LENGTH
DESTINATION STRING BYTE POINTER
0
9
35
Bits 0–8 = 0.
Bits 0–8 = 0.

Beginning at the left, copy as many bytes from the source string as will fit into the destination string space. If any source bytes are left over (i.e. if the source string is longer than the destination string), go to the next instruction. Otherwise place the fill character from *E0* + 1 in the remaining destination byte positions (if any) and skip the next instruction.

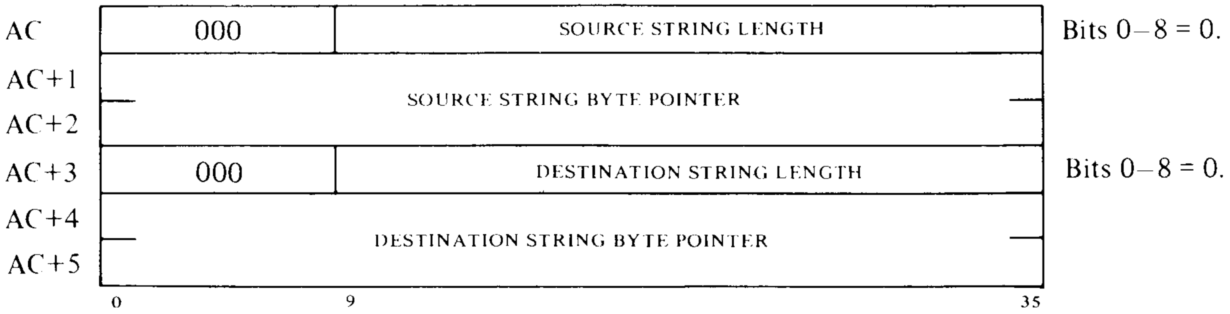
At the end the byte pointers point to the last positions referenced in source and destination, AC + 3 contains zero, and AC bits 9–35 contain the number of source bytes not copied (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise unused interior destination bits are indeterminate. The source string is unaffected.

MOVSO Move String Offset





Move the source string, with each byte offset by *E1*, left justified into the destination string space. Source and destination are defined by the contents of a block of six accumulators.


AC
AC+1
AC+2
AC+3
AC+4
AC+5
000
SOURCE STRING LENGTH
SOURCE STRING BYTE POINTER
000
DESTINATION STRING LENGTH
DESTINATION STRING BYTE POINTER
0
9
35
Bits 0–8 = 0.
Bits 0–8 = 0.

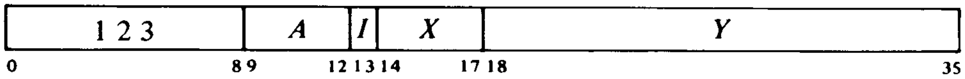
Beginning at the left, read each byte from the source string, add *E1* to it algebraically (bit 18 is the sign), and place the offset byte in the corresponding position in the destination string space provided it is not larger than the specified byte size (i.e. there are no 1s outside the area containing the offset byte in the register holding it). Continue in this fashion for each source byte until an oversize offset byte is encountered, or either the source string or the destination space is exhausted, whichever occurs first. Then if there are any source bytes not moved (because an offset byte is oversize or the source string is too long), go to the next instruction. Otherwise place the fill character from *E0* + 1 in the remaining destination byte positions (if any) and skip the next instruction.

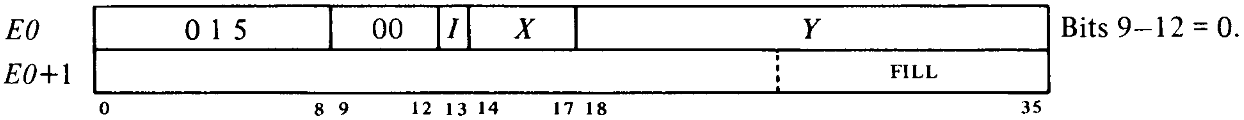
At the end the byte pointers point to the last positions referenced in source and destination, AC bits 9–35 contain the number of source bytes not moved (if any), and AC + 3 bits 9–35 contain the number of destination byte positions not used (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise unused interior destination bits are indeterminate. The source string is unaffected.

NOTE

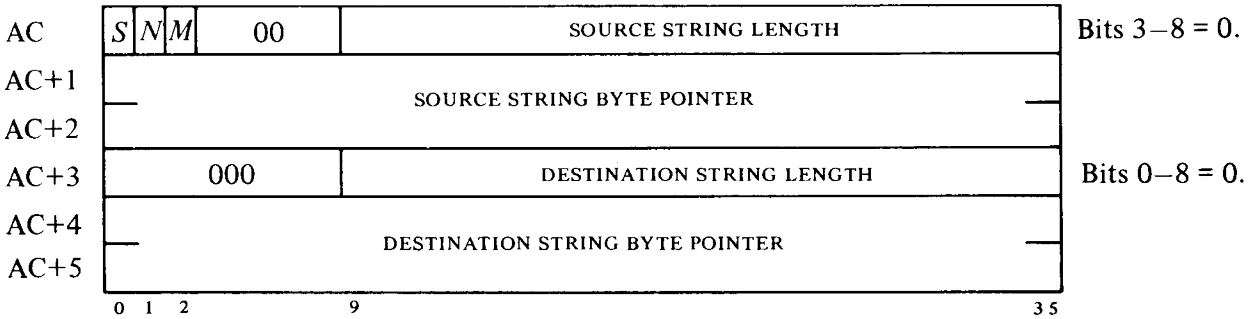
> MOVSO with a zero offset is equivalent to MOVSLJ, but the latter is faster and should always be used instead.

MOVST Move String Translated





Move the significant part of the source string, with its bytes replaced by bytes from a translation table at *E1*, left justified into the destination string space. Source and destination are defined by the contents of a block of six accumulators. *S* is the significance bit: setting it signals the start of that part of the source string that has significance, and bytes read while it is on are regarded as significant.



Beginning at the left, read each byte from the source string, and carry out the corresponding translation function given in the appropriate half word at location *E1* + *B*/2 in the translation table, where *B* is the value of the source byte. Each word in the table has this format.

020 000

TRANSLATION FUNCTION FOR EVEN *B*			TRANSLATION FUNCTION FOR ODD *B*			
OP CODE	0	SUBSTITUTE FOR BYTE (MAXIMUM 12 BITS)	OP CODE	0	SUBSTITUTE FOR BYTE (MAXIMUM 12 BITS)	Location $E1 + B/2$
0 2	6	17	18 20	24	35	

Perform the function specified by the op code in the half word corresponding to the source byte as follows.

0 If S is 1 take the substitute in place of the source byte.

1 Terminate translation.

2 If S is 1 take the substitute in place of the source byte. (Also clear M.)

3 If S is 1 take the substitute in place of the source byte. (Also set M.)

4 Set S and take the substitute in place of the source byte. (Also set N.)

5 Terminate translation. (Also set N.)

6 Set S and take the substitute in place of the source byte. (Also set N and clear M.)

7 Set S and take the substitute in place of the source byte. (Also set N and M.)

Then take one of these three courses of action:

> If the function makes no substitution and does not terminate, read the next byte from the source string and continue as described above.
>
> If the function makes a substitution, place the substituted byte in the next position in the destination string space, read the next byte from the source string, and continue as described above.
>
> If the function terminates the translation, go on to the next instruction.

Unless the translation is terminated by a translation function, continue the above procedure until either all source bytes are processed or the destination string is filled, whichever occurs first. Then if any source bytes are left over, go to the next instruction. Otherwise place the fill character from $E0+1$ in the remaining destination byte positions (if any) and skip the next instruction.

At the end the byte pointers point to the last positions referenced in source and destination, AC bits 9–35 contain the number of unprocessed bytes in the source string (if any), and AC + 3 bits 9–35 contain the number of destination byte positions not used (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise unused interior destination bits are indeterminate. The source string is unaffected.

Notes. The translation table starts at location $E1$, and since there are two functions per word, it contains 2^{n-1} locations, where n is the number of bits in a byte. The address is generated by adding the left $n - 1$ bits of a byte to $E1$.

Of the three flags in AC bits 0–2, only S is relevant to this instruction; the translation functions do manipulate M and N, but their states have no effect on the result. S being set means the translation has started. The programmer can make the translation start at the beginning of a string by having S already set when the instruction is given; or he can skip any number of initial bytes in the source string, and have the translation started by the first occurrence of some byte whose associated function sets *S*. Hence by the use of *S* and terminating functions, the programmer can have an MOVST translate any contiguous subset of the source string.

MOVSRJ Move String Right Justified

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	017	00	*I*	*X*	*Y*	Bits 9–12 = 0.
E0+1					FILL	*E1* is not used.[31]
	0 8	9 12	13	14 17	18 35	

Move the source string right justified into the destination string space. Source and destination are defined by the contents of a block of six accumulators.

AC	000	SOURCE STRING LENGTH	Bits 0–8 = 0.
AC+1 AC+2	SOURCE STRING BYTE POINTER		
AC+3	000	DESTINATION STRING LENGTH	Bits 0–8 = 0.
AC+4 AC+5	DESTINATION STRING BYTE POINTER		
	0	9 35	

Check the relation between the source and destination lengths to select one of the following three courses of action.

If the source and destination strings are the same length, move the source string into the destination space.

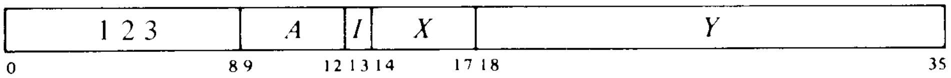
If the source string is shorter, place the fill character from *E0*+1 in destination byte positions beginning at the left until there are just enough places remaining in the destination space to accept the source string. Move the source string into the remaining destination positions at the right.

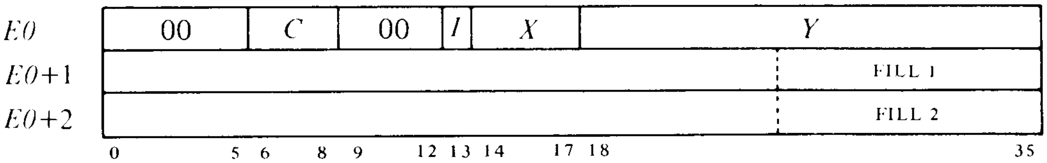
If the source string is longer, skip over enough source bytes at the left so the remaining source substring will fit in the destination space. Move the remaining source bytes into the destination space.

After completing the selected course of action, skip the next instruction.

At the end the byte pointers point to the last positions referenced in source and destination, AC+3 contains zero, and AC bits 9–35 contain the number of source bytes skipped over (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise unused interior destination bits are indeterminate. The source string is unaffected.

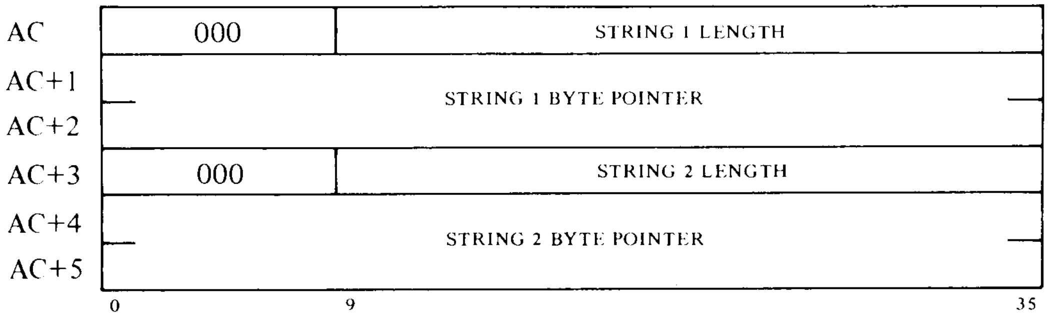
CMPS- Compare Strings





$C \neq 0, 4$.
Bits 9–12 = 0.
E1 is not used.[31]

Compare two strings and skip the next instruction if the condition specified by *C* is satisfied. The two strings are defined by the contents of a block of six accumulators.



Bits 0–8 = 0.

Bits 0–8 = 0.

Beginning at the left, compare string 1 with string 2, byte by byte, until a pair of bytes that are not identical is encountered. If a string runs out before an inequality is found, continue the comparison using a byte from *E0*+1 in lieu of bytes from string 1 or a byte from *E0*+2 in lieu of bytes from string 2, whichever is shorter.

Upon either encountering an inequality between corresponding bytes of the two strings or reaching the end of the longer string, stop the comparison and skip the next instruction if condition *C* is satisfied. The various values of *C* select different conditions and therefore specific forms of this instruction as follows.

CMPSL	Compare Strings and Skip if String 1 Less than String 2	001
CMPSE	Compare Strings and Skip if String 1 Equal to String 2	002
CMPSLE	Compare Strings and Skip if String 1 Less than or Equal to String 2	003
CMPSGE	Compare Strings and Skip if String 1 Greater than or Equal to String 2	005
CMPSN	Compare Strings and Skip if String 1 Not Equal to String 2	006
CMPSG	Compare Strings and Skip if String 1 Greater than String 2	007

At the end the byte pointers point to the last positions referenced in the strings, and bits 9–35 of AC and AC + 3 contain the number of bytes left in the strings beyond the unequal pair. The strings themselves are not affected. Note that except in the case where the inequality occurs at the last byte, the comparison continues to the end of the strings only if they are equal; and in both of these cases the final states of the pointers and lengths are the same.

If an interrupt or page failure occurs during execution of a string move or compare, the accumulators are adjusted for what has already been done. Afterwards the instruction resumes as though starting at the beginning, but manipulates substrings that are simply those parts of the original strings left from where the instruction was interrupted.

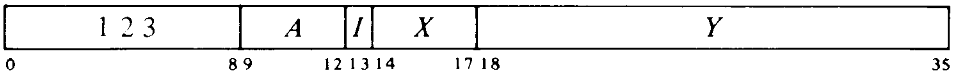
Offset can be used to change a string of capitals to lower case by adding 40 octal to every byte. Text in upper and lower case can be converted to all upper case by an MOVST with a translation table that substitutes capitals for both. Compare is useful for such applications as alphabetizing strings that represent words.

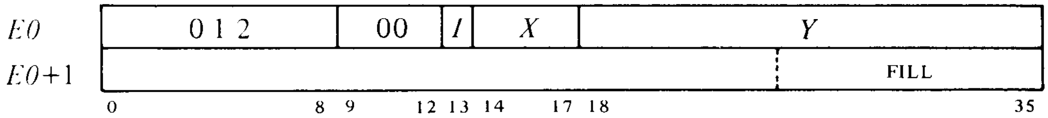
2.13 Decimal Conversion[30]

Included among the string instructions are four for converting between binary and decimal. The binary is always a twos complement, double length binary integer in the format given in §1.4: the magnitude is the 70-bit string in bits 1–35 of the two words, bit 0 of the high order word is the sign, and bit 0 of the low order word is a copy of the sign but is never used in any operation. The decimal is a string of bytes representing decimal digits (*the reader should be familiar with the general information and cautions about strings presented at the beginning of the previous section*). To be capable of conversion to double length binary, a decimal string can have a maximum of twenty-two significant digits, although the string may be longer because of the presence of leading zeros or nonnumeric characters. The decimal value corresponding to the binary maximum of 2^{70} is 1 180 591 620 717 411 303 424.

The four instructions are for converting with offset or translation in the two directions. All are two-word instructions, where the first has the EXTEND code 123, and all use a block of accumulators. Decimal to binary uses five accumulators, and binary to decimal requires a block of six, but one within the block is not used.

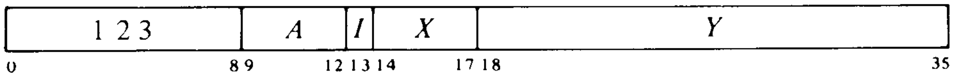
CVTBDO Convert Binary to Decimal Offset

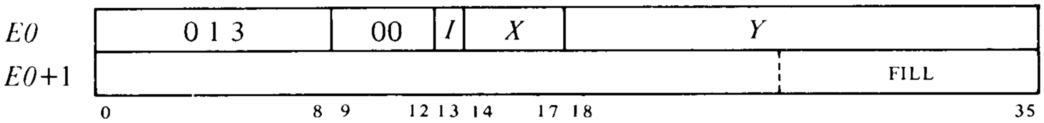




Bits 9–12 = 0.

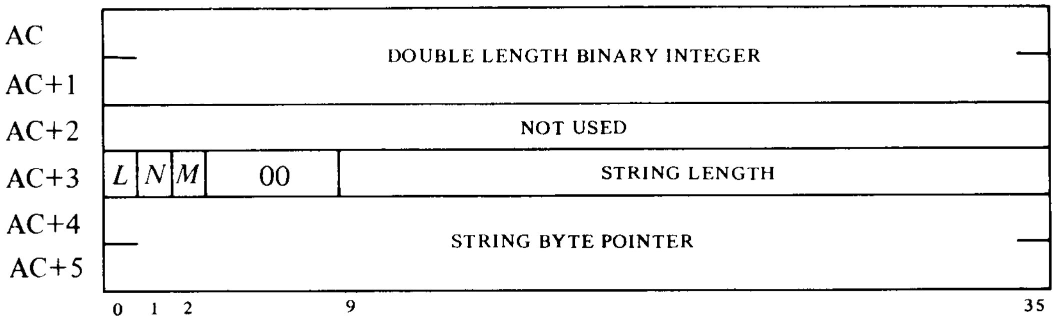
CVTBDT Convert Binary to Decimal Translated





Bits 9–12 = 0.

Convert the magnitude of a double length binary integer into a decimal digit string, offset or translated. The integer is given and the string space defined by the contents of a block of six accumulators.



Bits 3–8 = 0.

Determine the number of decimal digits required to convert the binary integer, and if this number is greater than the string length given by AC+3 bits 9–35, go on to the next instruction without affecting the string space or the accumulators in any way.[32] Note that the string length must specify a minimum of one digit byte even if the binary number is zero, for to represent zero in decimal requires at least the digit "0" (a string with no bytes cannot represent anything — not even zero). If the converted integer will fit in the defined string space, continue as follows.

If the binary integer in AC,AC+1 is not zero, set *N*; if it is less than zero, set *M* (minus). If the number of digits required is less than the given string length and *L* is 1, place the leading fill character from *E0*+1 in the excess positions at the left in the string space. This action causes the result to be right justified. Clear AC+3 bits 9–35.

[32] *Caution:* In the KL10 the *N* and *M* flags are set up first and may therefore be affected even by an instruction that is aborted because the binary integer is too large.

Compute each decimal digit for a positive representation of the magnitude of the binary integer (highest order first), and for each do one or the other of the following two operations depending on which instruction is being performed.

> If the instruction is CVTBDO, add *E1* to the computed digit algebraically (bit 18 is the sign).
>
> If the instruction is CVTBDT, for the digit substitute a byte from the right half of location *E1+D* in the translation table, where *D* is the value of the digit, unless this is the last digit in the conversion, in which case make the substitution from the right half of the location if *M* is 0, but from the left half if *M* is 1.

Place each offset or translated byte in the next position in the string space, compute the next digit, and continue as described above. When the conversion is complete — all digits computed, offset or translated, and deposited — clear AC and AC+1, and skip the next instruction.

At the end the byte pointer points to the last byte deposited in the string space, and AC, AC+1, and AC+3 bits 9–35 all contain zero. If unused interior bits in the string are clear initially, they are left clear; otherwise unused interior destination bits are indeterminate. The source string is unaffected.

Notes. The translation table, which starts at *E1*, contains ten locations for the decimal digits, each with substitute bytes in both half words, but the left half is only for the final digit. This allows the program to use a different final byte for a decimal string converted from a negative number. Note that setting *N* is just to indicate that the number converted is not zero; the state of the flag has no effect on the execution of the instruction.

CVTDBO Convert Decimal to Binary Offset

1 2 3	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

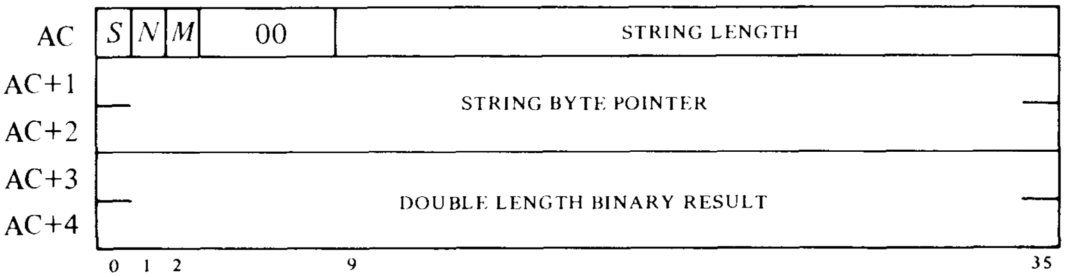
E0	0 1 0	00	*I*	*X*	*Y*	Bits 9–12 = 0.
	0 8	9 12	13	14 17	18 35	

CVTDBT Convert Decimal to Binary Translated

1 2 3	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	0 1 1	00	*I*	*X*	*Y*	Bits 9–12 = 0.
	0 8	9 12	13	14 17	18 35	

Convert a decimal string, offset or translated, to a double length binary integer. A block of five accumulators is used for defining the decimal string and receiving the binary result.



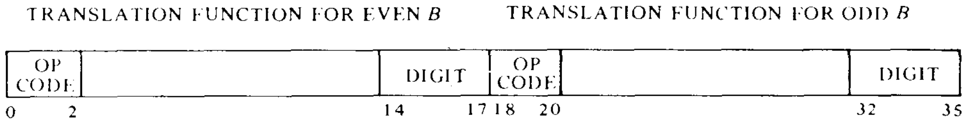
Bits 3--8 = 0.

If S is 1 initially there is already a binary number of significance in AC+3,AC+4: use it as a base for further accumulation of the digits derived from the decimal string. Otherwise begin with a zero base.

If the instruction is CVTDBO, set S to indicate the conversion has started.

Beginning at the left, read each byte from the string, and for each do one or the other of the following two operations depending on which instruction is being performed.

> If the instruction is CVTDBO, add $E1$ to the byte algebraically (bit 18 is the sign).
>
> If the instruction is CVTDBT, carry out the corresponding translation function given in the appropriate half word at location $E1 + B/2$ in the translation table, where B is the value of the byte. Each word in the table has this format.


TRANSLATION FUNCTION FOR EVEN B
TRANSLATION FUNCTION FOR ODD B
OP CODE
DIGIT
OP CODE
DIGIT
0
2
14
17
18
20
32
35

Location $E1 + B/2$

Perform the function specified by the op code in the half word corresponding to the byte as follows (setting S signals the start of significant digits in the decimal string).

0	If S is 1 substitute the table digit for the byte. If S is 0 ignore this byte and go on to the next.
1	Terminate the conversion.
2	Clear M, and if S is 1 substitute the table digit for the byte. If S is 0 ignore this byte and go on to the next.
3	Set M, and if S is 1 substitute the table digit for the byte. If S is 0 ignore this byte and go on to the next.
4	Set S and N, and substitute the table digit for the byte.
5	Set N and terminate the conversion.
6	Set S and N, clear M, and substitute the table digit for the byte.
7	Set S, N and M, and substitute the table digit for the byte.

If the translation function terminates the conversion, or the offset or translated digit is greater than 9, put the number of bytes remaining in the string in AC bits 9–35, put the partial binary result accumulated so far in AC+3,AC+4, and go on to the next instruction. Otherwise multiply the current binary value by 10 decimal, add in the current digit, and read the next byte from the string to continue as described above until the conversion is finished.

CAUTION

> It is up to the programmer to keep track of the size of the decimal number — the hardware runs no test on the string. If there are too many significant digits, the most significant part of the binary is lost, and the processor gives no indication of it.

The conversion is regarded as complete only when all bytes of the decimal string have been processed without causing a termination or generating a digit outside the range 0–9. Upon completion negate the accumulated binary if M is 1, place the result (negated or not) in AC + 3,AC + 4, and skip the next instruction.

At the end the byte pointer points to the last byte read from the decimal string, and AC bits 9–35 contain the number of unprocessed bytes left in the decimal string (if any). The string itself is unaffected.

The translation table starts at location *E1*, and since there are two functions per word, it contains 2^{n-1} locations, where n is the number of bits in a byte. The address is generated by adding the left $n - 1$ bits of a byte to *E1*.

Notes. CVTDBO always sets S immediately, but in CVTDBT its setting is controlled by the translation functions. Hence an instruction can skip over leading fill characters or nonnumeric characters preceding the decimal part of a string. If an interrupt or page failure occurs during this instruction, the number of bytes yet to be processed is put in AC bits 9–35, and the partial binary accumulated so far is placed in AC + 3,AC + 4. Thus when the instruction resumes after an interruption with S set, it simply continues where the conversion left off, adding the next digit to ten times the binary previously saved. If the programmer wishes to preset S to add the decimal string to a significant binary base already in AC + 3,AC + 4, he must be aware that the base is multiplied by ten before the first digit is added.

For a decimal string *abcde*, the evaluation procedure is

$$(((a \times 10 + b) \times 10 + c) \times 10 + d) \times 10 + e$$

which is equivalent to

$$\begin{array}{r} e \times 1 \\ + \; d \times 10 \\ + \; c \times 100 \\ + \; b \times 1000 \\ + \; a \times 10000 \end{array}$$

Of course the operations are all done in binary arithmetic.

Translation functions manipulate M, but the program can set it prior to either instruction to indicate the decimal string represents a negative number. N can also be preset or manipulated through the translation table, but its state has no effect on the execution of the instruction.

For decimal strings with 4-bit digits, conversion can be done by CVTBDO or CVTDBO with a zero offset. But note that decimal bytes need not be four bits: they can be larger using any decimal code provided only that on conversion to binary they are in the range 0–9 (0–1001 binary) after offset or translation.

In ASCII numeric strings, the bytes representing the digits are 60–71 octal. Conversion to ASCII decimal would be by CVTBDO with offset 60 (48 decimal), and CVTDBO with offset –60 would convert in the opposite direction. Consider an ASCII string containing decimal numbers of various unknown lengths separated by semicolons (ASCII code 73). The program could convert all of these numbers to binary by specifying a constant, but suitably large, string length while giving a sequence of CVTDBOs with offset –60. Each conversion would terminate (nonskip) upon encountering a semicolon, as its offset value is 11 decimal. Between conversions the program would have to store away the result and clear *S* by a sequence like this.

```
EXTEND      AC,[CVTDBO 0,–60]      ;Convert
DMOVEM      AC+3,VALUE1            ;Store result
TLZ         AC,700000              ;Reset SNM
EXTEND      AC,[CVTDBO 0,–60]
DMOVEM      AC+3,VALUE2
TLZ         AC,700000
 .
 .
 .
```

If there were very many numbers, the program would naturally use only one of the above sets of three instructions in a loop, along with some mechanism to change the storage address and test whether to reiterate. The procedure cannot of course provide a negative result. If the same situation were handled by translation, the table would not actually start at *E1* — it would run from *E1* + 30 to *E1* + 35.

2.14 String Editing[30]

The edit instruction implements more complex operations on strings than merely moving or translating, and before investigating it the reader should be familiar with the general characteristics of strings (and cautions about them) as presented at the beginning of §2.12. Edit provides the facilities needed, particularly in COBOL and PL/I, to create formatted character strings for output. Typical features are the ability to suppress leading zeros, insert special symbols such as decimal points or currency symbols, and recognize different types of numbers for operations like adding "CR" or "DB" after them. When numbers appear in running text, leading zeros are usually deleted; when they are lined up in columns (such as in a financial statement), the practice is to substitute spaces.

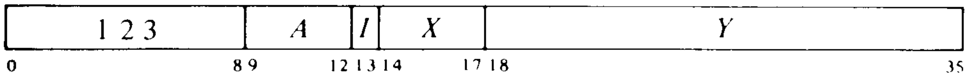
Edit uses the usual source and destination byte pointers, but no string lengths are given. Instead the source bytes are processed by commands in a pattern command string, whose structure is determined by the expected length of the source. The pattern commands are 9–bit bytes packed four to a word. They are executed according to a pattern pointer, which supplies the address of a memory location and a 2-bit byte number, wherein the numbers 0–3 identify the bytes from left to right in the word. The destination string space is assumed to be large enough for whatever string edit creates.

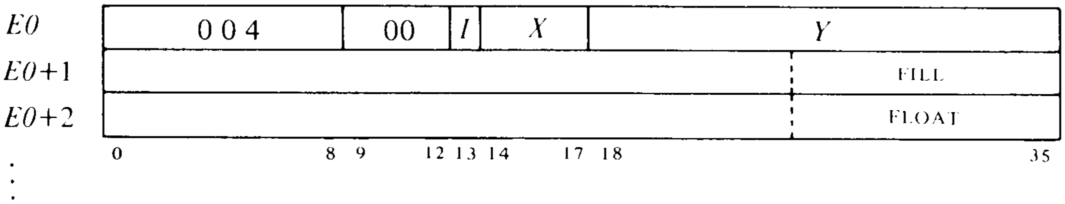
Available to the procedure are a translation table at *E1* like that of MOVST, and a message insertion table following *E0*. $E0+1$ contains the fill character — typically a space — for suppression of leading zeros; but if the whole word containing the fill character is zero, the fill is not inserted in the destination space, thus deleting leading zeros. $E0+2$ contains a float character — typically a currency symbol or plus sign — which, if the word containing it is nonzero, is inserted before the first significant byte. The table can extend to $E0+100$, thus supplying an additional sixty-two characters for insertion in the string being generated. Insert characters are typically decimal point, comma, "C" and "R", and so forth.

For signaling significance AC has an *S* bit, which can be set from the translation table when significance starts. At this point the destination string position is marked by storing the current value of the destination pointer at a location specified by a mark address. This provides a record of where significance started, so the instruction can go back to make revisions if need be after receiving more information from the source.

EDIT is a two-word instruction, where the first has the EXTEND code 123, and it uses a block of six accumulators. The description is accompanied by a flowchart.

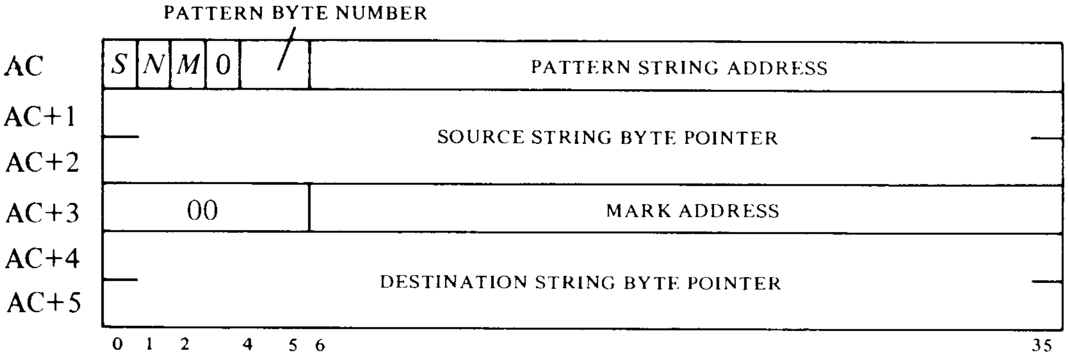
EDIT Edit String





Bits 9 12 = 0.

Execute the commands in the pattern string to edit a source string, employing byte substitutions from a translation table at *E1* and inserting characters from a message insertion table at *E0*+1, and place the result in the destination string space. Source, destination and pattern are defined by the contents of a block of six accumulators.



Bit 3 = 0.

Bits 0–5 = 0.

Definitions: Initially the pattern pointer, which comprises the pattern string address and byte number, points to the first pattern command. Pattern byte counting is effected by incrementing the byte number unless it is 3, in which case it is changed to 0 and the address is incremented. The address is limited to bits 18–35 if the program is running in section 0. The mark address is simply the address of the first in a pair of locations for receiving the destination byte pointer as a mark. Of course if the destination pointer is local, only one location is used to store it. Furthermore if the program is running in section 0, the mark address is limited to bits 18–35 and always points to a single location. In the following any reference to reading a source byte shall be taken to mean that the source string byte pointer is incremented first, and any reference to placing a character in the next position in the destination string space shall be taken to mean the destination byte pointer is incremented first.

Execute the pattern command specified by the pattern pointer. At the completion of any pattern command, unless the edit has been ended by a STOP command or a terminating translation function, increment the pattern pointer and execute the pattern command then specified by it. There are ten such commands as follows (all other command bytes are reserved and must not be used).

SELECT	0 0 1	*Select Next Source Byte*
	0 8	

Read the next byte from the source string, and carry out the corresponding translation function given in the appropriate half word at location $E1 + B/2$ in the translation table, where B is the value of the source byte. Each word in the table has this format.

TRANSLATION FUNCTION FOR EVEN *B*			TRANSLATION FUNCTION FOR ODD *B*			
OP CODE	0	SUBSTITUTE FOR BYTE (MAXIMUM 12 BITS)	OP CODE	0	SUBSTITUTE FOR BYTE (MAXIMUM 12 BITS)	Location $E1 + B/2$
0 2		6 17	18 20		24 35	

Perform the function specified by the op code in the half word corresponding to the source byte as follows.

0 If S is 1 place the substitute in the next position in the destination string space. Otherwise if location $E0+1$ is nonzero, place the fill character from it in the next destination position.

1 Increment the pattern pointer, and go on to the next instruction.

2 Clear M and then perform function 0.

3 Set M and then perform function 0.

4 Set N. If S is 1 place the substitute in the next position in the destination string space. Otherwise do the following: set S; put the current value of the destination byte pointer at the location specified by the mark address; if location $E0+2$ is nonzero, put the float character from it in the next destination position; then place the substitute in the next destination position after that.

5 Set N, increment the pattern pointer, and go on to the next instruction.

6 Clear M and then perform function 4.

7 Set M and then perform function 4.

Notes. The translation table starts at location $E1$, and since there are two functions per word, it contains 2^{n-1} locations, where n is the number of bits in a byte. The address is generated by adding the left $n - 1$ bits of a byte to $E1$.

SIGST	0 0 2	*Start Significance*
	0 8	

If S is 0 do the following: set S; put the current value of the destination pointer at the location specified by the mark address; and if location $E0+2$ is nonzero, put the float character from it in the next destination position.

Notes. A typical use of this command might be before a final character to guarantee that zero is represented by one "0." Or if the number of cents is 00004, to put in a decimal point and generate a result of .04.

MESSAG + n *Insert Message Character*

1	n
0 2	3 8

If S is 1 place the character from $E0 + n + 1$ in the next destination position. Otherwise if location $E0 + 1$ is nonzero, place the fill character from it in the next destination position.

FLDSEP *Separate Fields*

0 0 3
0 8

Clear S, M and N.

Notes. Essentially this instruction causes the procedure to start over on a new substring. A typical use would be in handling a series of numbers (separated by some character), where one would want to suppress leading zeros in all of them.

EXCHMD *Exchange Mark and Destination Pointers*

0 0 4
0 8

Interchange the destination pointer presently held in AC+4,AC+5 with the mark pointer at the location specified by the mark address.

Notes. This makes it possible to go back to where significance began in order to revise the destination string in light of further processing of the source, but at the same time saving the present position. A return forward can be made simply by repeating the instruction.

Note that it is unlikely to be very useful for the programmer to set up an initial mark pointer. In any normal procedure a mark is created from the destination pointer and is simply a particular state of it. Hence the destination and mark pointers have the same number of words. The result is indeterminate if the programmer deliberately sets up mark and destination pointers of different types.

SKPM + n *Skip on M*

5	n
0 2	3 8

If M is 1 skip over the next $n + 1$ pattern commands by incrementing the pattern pointer $n + 1$ times.

Notes. M is generally used as a minus sign, i.e. to indicate a string is negative, but the programmer can use it for any purpose. A typical use would be to determine whether "CR" or "DB" should be inserted after a number.

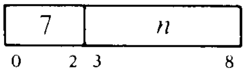
SKPN + n *Skip on N*

6	n
0 2	3 8

If N is 1 skip over the next $n + 1$ pattern commands by incrementing the pattern pointer $n + 1$ times.

Notes. *N* is generally set to mean the string is nonzero, but the programmer can use it for any purpose. Suppose we wish to output a blank on zero, but use of SIGST to handle cents-only quantities has produced ".00". We could use SKPN after the last source byte, so that if the output is nonzero we would skip over commands that would otherwise go back and blank the output.

SKPA + *n*



Skip Always

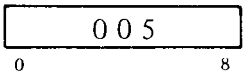
Skip over the next *n* + 1 pattern commands by incrementing the pattern pointer *n* + 1 times.

Notes. This command is used mostly to reverse the meaning of the other skips. For example, the sequence "SKPN,*X*" skips command *X* if *N* is 1, but the sequence "SKPN,SKPA,*X*" executes it if *N* is 1. SKPA can also be used to extend a conditional skip beyond sixty-four commands, as in

SKPN + 77,...*63 bytes*...,SKPA,SKPA + 3,...*4 bytes*...,*X*

in which *N* being 1 causes a skip over sixty-seven significant commands to get to *X*.

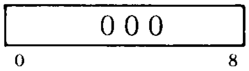
NOP



No-op

Do nothing.

STOP



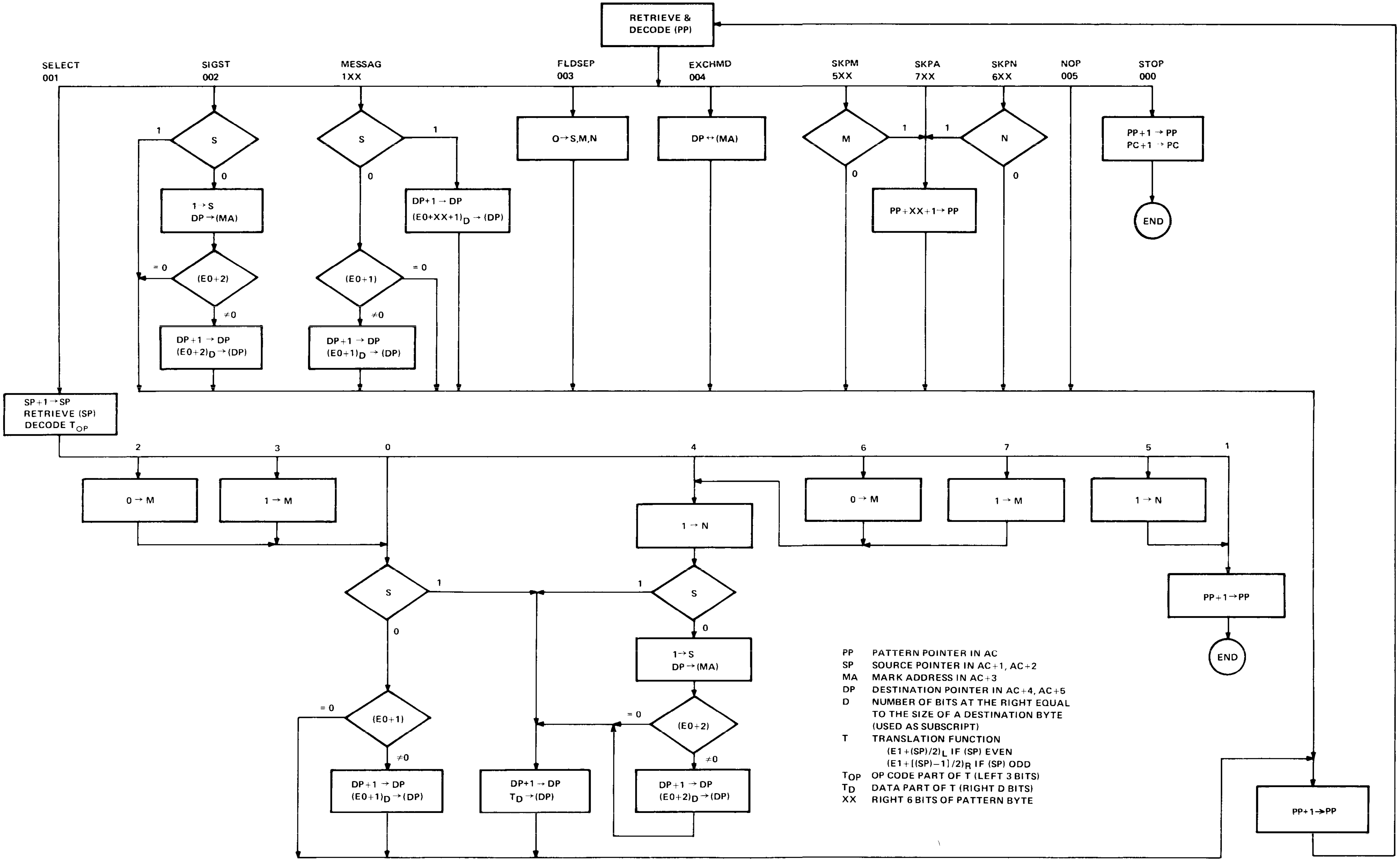
Stop Edit

Increment the pattern pointer, end the edit, and skip the next instruction.

At the end the byte pointers point to the last positions referenced in source and destination, and the pattern pointer points to the command byte following the last one executed. Note l owever that if the pattern gives an EXCHMD after the final byte is placed 'n the destination string, the "destination pointer" is actually at the mark location rather than in AC + 4,AC + 5. If unused interior bits n both strings are clear initially, they are left clear; otherwise unused in erior destination bits are indeterminate. The source string is unaffected.

Notes. If an interrupt or page failur occurs during EDIT, the accumulators are adjusted for restarting at the beginning of the current pattern command.

Figure 2.2: Edit Instruction



PP	PATTERN POINTER IN AC
SP	SOURCE POINTER IN AC+1, AC+2
MA	MARK ADDRESS IN AC+3
DP	DESTINATION POINTER IN AC+4, AC+5
D	NUMBER OF BITS AT THE RIGHT EQUAL TO THE SIZE OF A DESTINATION BYTE (USED AS SUBSCRIPT)
T	TRANSLATION FUNCTION $(E1+(SP)/2)_L$ IF (SP) EVEN $(E1+[(SP)-1]/2)_R$ IF (SP) ODD
T_{OP}	OP CODE PART OF T (LEFT 3 BITS)
T_D	DATA PART OF T (RIGHT D BITS)
XX	RIGHT 6 BITS OF PATTERN BYTE

EDIT

MR-0630

Example. The following program uses binary-to-decimal conversion and editing to translate a binary number into a message of seventeen characters containing a decimal string with appropriate nomenclature for commercial billing purposes. A positive result has the form

$12,345.46 DUE US

whereas a negative result has the form

$12,345.46 CREDIT

but if the number is zero, the entire field is blank (all spaces). The maximum number the routine can handle is $99,999.99.

This program employs seven accumulators, of which P is for the stack pointer, and a block of six, labeled AC1–AC6, is for the extend instructions. In the block however, AC3 and AC6 are never actually used as the program is entirely local, employing only one-word stack and byte pointers. Beginning at TEMP and FIELD are blocks of eight locations set aside for the edit source and destination strings. The routine is called by a PUSHJ to PNTFLD with the amount as a binary number of cents in AC1,AC2. It returns the result beginning at the left in FIELD.

```
PNTFLD:  MOVE     AC4,[400000,,7]  ;Convert up to 7 digits with leading fill
         MOVE     AC5,[POINT 7,TEMP]    ;Store decimal in edit source area
         EXTEND   AC1,[CVTBDO 60 ;Convert to decimal with leading zeros
                      60]
           JRST   ERROR    ;Here if need too many digits (binary too large)

         MOVEI    AC1,PATTRN        ;Set pattern pointer to first command
         TLNE     AC4,100000        ;Copy M flag from AC4 (CVTBDO result)
         TLO      AC1,100000        ;  to AC1
         MOVE     AC2,[POINT 7,TEMP]    ;Pointer for source string
                                        ;  (CVTBDO result)
         MOVEI    AC4,MARK              ;Address of mark pointer

         MOVE     AC5,[POINT 7,FIELD]  ;Pointer for destination string
         EXTEND   AC1,EDTINS        ;Edit the item

            HALT  .                 ;Should never get here
         POPJ     P,0               ;Return

;Here is the edit instruction
EDTINS:  EDIT      TABLE–30     ;Need only digit part of translation table
         " "                    ;Fill character is space
         "$"                    ;Float character is dollar sign
         ","                    ;Message 2 is comma
         "."                    ;Message 3 is decimal point
         "D"
         "U"
         "E"
         "S"
         "C"
         "R"
         "I"
         "T"
```

```
;Here is the translation table. Digits 1–9 set S and N flags; 0 does not affect the flags
TABLE:   60,,400061
         400062,,400063
         400064,,400065
         400066,,400067
         400070,,400071

;Here is the pattern
PATTRN:  001001,,102001     ;SELECT SELECT MESSAG+2 SELECT
                            ;  2 digits, comma, digit
         001001,,002103     ;SELECT SELECT SIGST MESSAG+3
                            ;  2 more digits, then start significance and insert
                            ;  a decimal point
         001001,,100506     ;SELECT SELECT MESSAG+0 SKPM+6
                            ;  2 more digits (cents) and a space, then skip
                            ;  next 7 commands if number was negative
         104105,,106100     ;Append the message "DUE US"
         105107,,705110     ;  Then skip 6 pattern commands
         111106,,104112     ;Append the message "CREDIT"
         113613,,004100     ;If number is nonzero skip 12 commands
         100100,,100100     ;  Else exchange mark and destination pointers
         100100,,100100     ;  and blank out result
         100100,,0          ;Then stop

MARK:    BLOCK 1
TEMP:    BLOCK 10
FIELD:   BLOCK 10
```

2.15 Programming Examples

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

Processor Identification

The instruction repertoires of all PDP–10 processors 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 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 (carry
BLT      AC,0       ;between halves)
JUMPE    AC,KI10    ;KI10 if AC = 0
```

```
          MOVEI   AC,1            ;KL10 or KS10 if AC = 1,,1
          SETZ    AC+1,           ;Big binary integer
          MOVEI   AC+3,1          ;One digit byte
          EXTEND  AC,[CVTBDO]     ;Convert will abort
          TLNE    AC+3,200000     ;Test effect on N
          JRST    KL10            ;KL10 if N set
          JRST    KS10            ;KS10 if N unaffected
```

Parity

Parity procedures are used regularly to check the accuracy of stored information. Parity generation and checking is generally handled automatically by memory and high speed, block-oriented peripheral devices, but must be handled by the program for character-oriented devices. Consider 8-bit characters, for which the program carries out two procedures: for output it generates a parity bit from seven data bits to produce an 8-bit character with parity; following input it checks the parity of the eight bits received. In either case however, the program can simply find the parity of an 8-bit character, by regarding the seven output data bits as eight including an irrelevant extra bit. The two procedures then differ only in the final action. In the first case the program uses the result to adjust the eighth bit for correct parity, whereas in the second it checks the result for an indication of error.

Assuming the character is right justified in accumulator A and the rest of A is clear, as it would be were the character placed in A by a load-byte instruction or a DATAI, the simplest and quickest procedure would be to use A to index an XCT into a table, each of whose locations contains an instruction that adjusts the parity for output or jumps to a routine for erroneous input. This procedure would normally be unacceptable because of the very large memory requirements. However the table can be reduced to sixteen entries without excessive loss in speed, by exclusive oring the left and right halves of the character and indexing on the result (parity is invariant under the exclusive OR function, which essentially disposes of pairs of 1s). This example, which uses a second accumulator T for character manipulation, requires six memory references to generate odd parity. (Numbers of memory references and locations given do not include those for the POPJ, which we will regard as subroutine overhead. Similarly every example also requires that the program give a PUSHJ to get to the subroutine.)

```
PARITY:   MOVEI   T,(A)         ;Copy character in T
          LSH     T,-4          ;Line up halves
          XORI    T,(A)         ;Reduce paritywise to 4 bits
          ANDI    T,17          ;Wipe out unwanted bits
          XCT     PARTAB(T)     ;Execute indicated table item
          POPJ    P,
PARTAB: XORI     A,200      ;0 — change high bit
        JFCL                ;1 — no-op
        JFCL                ;2
        XORI     A,200      ;3
        JFCL                ;4
        XORI     A,200      ;5
        XORI     A,200      ;6
        JFCL                ;7
        JFCL                ;10
        XORI     A,200      ;11
        XORI     A,200      ;12
        JFCL                ;13
        XORI     A,200      ;14
        JFCL                ;15
        JFCL                ;16
        XORI     A,200      ;17
```

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

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

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

Note that this example does not require the rest of A to be clear. For even parity change the address in the MOVEI from 200 to 0.

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

abc def gha bcd efg hab cde fgh

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

001 001 001 001 001 001 001 001

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

cfadgbeh

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

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

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

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

```
PARITY: MOVEI   T,(A)       ;Copy in T
        IMULI   T,200401    ;Duplicate twice
        AND     T,ONES      ;Pick LSBs
        IMUL    T,ONES      ;Generate product
        TLNN    T,10        ;Is bit 14 odd?
        XORI    A,200       ;No, change parity
        POPJ    P,
          .
          .
          .
ONES:   11111111
```

This procedure uses a minimum of both memory references and memory space, but takes considerably more time because the instructions themselves are slow.

The following table shows the trade-off of memory references against memory space for the above four procedures. The time is proportional to the number of references except in case 4.

	References	*Locations*
1.	2	257
2.	6	21
3.	7½	9
4.	7½	7

Reversing Order of Digits

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

$$a \quad bcd \quad efa \quad bcd \quad efa \quad bcd \quad efa \quad bcd \quad ef0$$

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

$$1 \quad 000 \quad 100 \quad 100 \quad 010 \quad 010 \quad 000 \quad 001 \quad 000$$

gives

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

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

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

The remainder in a division is equal to the sum, modulo the divisor, of the remainders left by dividing each of the dividend summands by the same divisor. And the sum of the terms in the lower row is obviously *fedcba*. The above procedure is implemented by this sequence (due to Schroeppel[34]) where the character is right justified in accumulator A (with the rest of A clear), and its reverse appears right justified in accumulator A + 1.

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

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

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

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

Counting Ones

Suppose we wish to count the number of 1s in a word. We could of course check every bit in the word. But there is a quicker way if we remember that in any word and its twos complement the rightmost 1 is in the same position, both words are all 0s to the right of this 1, and no corresponding bits are the same to the left (the parts of both words at the left of the rightmost 1 are complements). Hence using the negative of a word as a mask for the word in a test instruction selects only the rightmost 1 for modification. The example uses three accumulators: the word being tested (which is lost) is in T, the count is kept in CNT, and the mask created in each step is stored in TEMP.

```
MOVEI   CNT,0      ;Clear CNT
MOVN    TEMP,T     ;Make mask to select rightmost 1
TDZE    T,TEMP     ;Clear rightmost 1 in T
AOJA    CNT,.-2    ;Increase count and jump back
...                ;Skip to here if no 1s left in T
```

CNT is increased by one every time a 1 is deleted from T. After all 1s have been removed, the TDZE skips.

The preceding example uses little memory, but contains a loop so the time it takes is proportional to the number of 1s. The next example takes more memory but is constant; hence it is slower than the above when there are few 1s (less than eight), but is much faster when there are many. The word, which is lost, is in accumulator A, and the answer appears in accumulator $A+1$ (for convenience we let $B = A+1$). The routine (due to Gosper, Mann and Leonard[35]) has three distinct parts and is based on the fact that in a binary word, counting 1s is equivalent to calculating the sum of the digits. The first part, of seven instructions, manipulates the octal digits of the word so as to replace each digit by the number of 1s in it. Taking D as an octal digit and $[x]$ as the largest integer contained in x, the algorithm used to make the substitution is

$$D - [D/2] - [D/4]$$

Of course the computer always acts in binary terms regardless of programmer interpretation. In this case the procedure carried out on each 3-bit piece abc is

$$abc - ab - a$$

The instructions effect this algorithm by shifting a copy of the word right one place, masking out the LSB of each shifted octal digit to prevent it from interfering with the next digit at the right (i.e. to isolate the digits), and subtracting the shifted word from the original. The same process is then repeated, this time masking out what was originally the middle bit in each digit. That this algorithm gives the correct substitution is evident from the following table, in which it is seen that the bottom number in a given column is the sum of the bits in the octal digit given at the top of the column.

[35] Ibid, item 169, page 79.

Original digit	0	1	2	3	4	5	6	7
Subtract	0	0	1	1	2	2	3	3
	0	1	1	2	2	3	3	4
Subtract	0	0	0	0	1	1	1	1
Number of 1s	0	1	1	2	1	2	2	3

We have now replaced the original word with a set of twelve numbers, whose sum is equal to the number of 1s in the original. The next three instructions add together pairs of adjacent numbers so as to replace the twelve by six whose sum is still the same. Since these new numbers are isolated in 6-bit pieces of the word, we shall revise our point of view, and regard them as digits in a number in base 64. Now any number is simply the sum of the values of its digits, i.e., of its digits each multiplied by an appropriate power of the base. Dividing each such summand by 1 less than the base gives the digit itself as remainder. Hence the third part of the routine just divides our 6-digit number by 63, producing in B a remainder that is the sum of the remainders from the individual digits, i.e., that is the sum of the digits.[36]

```
MOVE   B,A                   ;Copy in B
LSH    B,-1                  ;Right one
AND    B,[333333,,333333]      ;Mask out LSBs
SUB    A,B                   ;D - [D/2]
LSH    B,-1                  ;Right one again
AND    B,[333333,,333333]      ;Mask out middle bits
SUBB   A,B                   ;D - [D/2] - [D/4]; two copies
LSH    B,-3                  ;Shift right one octal digit
ADD    A,B                   ;Add numbers in digit pairs
AND    A,[070707,,070707]      ;Throw out extra pair sums

IDIVI  A,77                  ;Divide by 63, sum in B
```

If it is known that the 1s in the word are entirely contained within bits 22–35 (the right fourteen bits), we can use the following somewhat shorter routine, which is faster than the loop for more than seven 1s. It first treats the number in quaternary, replacing each digit with the number of 1s in it, and then converts from quaternary to hexadecimal.

```
MOVEI  B,(A)
LSH    B,-1
ANDI   B,12525            ;Mask out LSBs
SUBB   A,B                ;D - [D/2]; two copies
```

[36] In general terms this is the statement that the sum S of the digits in any number N in base b mod $(b - 1)$ — *provided* b is deliberately chosen such that $S < b - 1$. The condition holds here of course as the number of 1s in a PDP–10 word is at most 36. And it is in fact to make this condition hold that the routine converts from base 8 to base 64.

```
        LSH     B,-2        ;Right one quaternary digit
        ANDI    A,31463     ;Mask out some of digits in A
        ANDI    B,31463     ;The rest in B
        ADDI    A,(B)       ;Now combine digit pairs

        IDIVI   A,17        ;Divide by 15, sum in B
```

Note that the pair of ANDIs gets rid of one out of each set of two identical bit pairs before adding. This is done because there can be digit overflow, i.e. a resulting hexadecimal digit can have more than two significant bits.

Number Conversion

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

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

The number in base b is then $r_n ... r_3 r_2 r_1$. For example, the octal equivalent of 61 decimal is 75:

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

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 stack 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.

In section 0 space can be saved in the stack by storing the computed digits in the left halves of the locations that contain the jump addresses. This is accomplished in the decimal print routine by changing

PUSH P,T+1   to   HRLM T+1,(P)

and

```
                    POP P,T   to   HLRZ T,(P)
```

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

```
        LSHC    T,-^D35     ;Shift right 35 bits into T+1
        LSH     T+1,-1      ;Vacate the T+1 sign bit
        DIVI    T,12        ;Divide double length integer by 10
```

Table Searching

Many data processing situations involve searching for information in tables and lists of all kinds. Suppose we wish to find a particular item in a table beginning at location TAB and containing *N* items. Accumulator T contains the item. The right half of A is used to index through the table, while the left half keeps a control count to signal when a search is unsuccessful.

```
        MOVSI   A,-N        ;Put -N,,0 in A
        CAMN    T,TAB(A)    ;Skip if current item not the one
        JRST    FOUND       ;Item found
        AOBJN   A,.-2       ;Try next item until left count = 0
        ...                 ;Item not in list
```

The location of the item (if found) is indicated by the number in the right half of A (its address is that quantity plus TAB). A slightly different procedure would be

```
        MOVSI   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,-1        ;Skip if AC right = 0; -1 = 777777
        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
```

Extended Addressing

For simplicity the preceding examples have employed only local addressing, as this is mostly what a typical program would use even when running in a nonzero section. Here we give a number of straightforward examples to show the differences between local and extended addressing, with and without indexing and indirection. In all cases the program is assumed to be running in section 22.

Local reference without indexing or indirection.

```
          MOVE      T,1000
```

loads accumulator T with the contents of location 1000 in section 22.

Local indexing.

```
          MOVEI     X,100
          MOVE      T,1000(X)
```

loads T with the contents of location 1100 in section 22. This would typically be the hundredth entry in an array starting at 1000 in the current section.

```
          MOVNI     X,100
LOOP:     ADD       T,1000(X)
          AOJL      X,LOOP
```

adds together the contents of locations 700–777 in section 22. (We assume that either T is cleared first or the array is added to whatever is in it initially.)

```
          MOVSI     X,–LENGTH
LOOP:     ADD       T,1000(X)
          AOBJN     X,LOOP
```

adds together the contents of all locations in an array of length LENGTH starting at location 1000 in section 22. Note that since local indexing is used, the array cannot cross over into section 23. If LENGTH is greater than 776777 the array wraps around, first into the AC block, and then continuing from location 20 in the current section.

Global indexing.

```
          MOVE      X,[30,,1000]
          ADD       T,100(X)
```

adds the contents of location 1100 in section 30 to T. Note that if the literal were "22,,1000" the ADD would address location 1100 in the current section even though the indexing is global.

```
          MOVE      X,[30,,1000]
          ADD       T,–100(X)
```

adds the contents of location 700 in section 30. Were the address part of the ADD instruction –1000, it would reference storage location 0 in section 30 (not a fast memory location). Furthermore were the address part –2000, it would address location 777000 in section 27, as global indexing can cross the section boundary.

Local indirection.

```
        MOVEI   T1,100
        MOVEM   T1,1000
        ADD     T,@1000
```

adds the contents of location 100 in section 22 to T.

Global indirection.

```
        MOVE    T,@[30,,1000]
```

loads T with the contents of location 1000 in section 30. If location 1000 in section 30 contained

```
        MOVE    T,2000
```

then in the current section (22) the instruction

```
        XCT     @[30,,1000]
```

would load T with the contents of location 2000 in section 30, as the instruction is executed in that section rather than in 22. On the other hand, were location 1000 in section 30 to contain

```
        JSR     SUBR
```

then an

```
        XCT     @[30,,1000]
```

given in location 100 in section 22 would transfer control to SUBR+1 in section 30, but the PC saved in 30,,SUBR would be 22,,101 as the XCT itself is performed in the current PC section, which is 22.

Global indirection with indexing.

```
        MOVEI   X,100
        MOVE    T,@[GIW 30,1000(X)]
```

loads T with the contents of location 1100 in section 30. The made-up pseudoinstruction GIW would create a global indirect word by causing the assembler to place the number X in bits 2–5 of the word in which it places 30,,1000 in bits 6–35. There is no such operation, but the programmer could define a macro for this purpose.

```
        MOVE    X,[2000000–1]           ;2 sections worth
LOOP:   ADD     T,@[GIW 30,1000(X)]
        SOJGE   X,LOOP
```

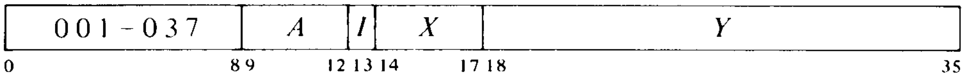
adds up the 512K array from location 777 in section 32 down to 1000 in section 30. Note that even if the array contained fewer than 2^{17} words and did not cross a section boundary, it would still not be possible to use AOBJN for the loop, as global indexing uses the entire index register. The following gets the same result with negative indexing.

```
        MOVE    X,[-2000000+1]
LOOP:   ADD     T,@[GIW 32,777(X)]
        AOJLE   X,LOOP
```

2.16 Unimplemented Operations

Codes not assigned as specific instructions act as unimplemented operations, wherein the word given as an instruction is trapped, either because it should not be given or because it must be interpreted by a routine included for this purpose by the programmer. Those that are available for interpretive use are unimplemented user operations, or UUOs (the several mnemonics mentioned in this discussion are for convenience and mean nothing to the assembler). Codes in the range 001–037 are for the local use (LUUOs) of the user anywhere or the Monitor in section 0. Various other codes are set aside specifically for user communication with the Monitor or for communication between one level of the Monitor and another; in either case these MUUOs are interpreted by the executive. Basic codes (except 000) that are not used for instructions or UUOs, and extended codes not used by EXTEND, are regarded as the "unassigned codes"; 000 is not regarded as a legal code at all. Let us consider first how an LUUO works.

Local Unimplemented User Operation



If the program is running in section 0, 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 in section 0 uses some pair of locations numbered 40 and 41, but which such pair depends upon the circumstances. An LUUO in a user program uses virtual locations 40 and 41 and is thus entirely a part of and under control of the user program. The locations used in executive mode depend on the processor:

KL10,KS10	40 and 41 in executive virtual space
KI10	40 and 41 in the executive process table
KA10	Unrelocated 40 and 41[37]

[37] 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.

If the program is running in a nonzero section, take one of these two courses of action.

In executive mode perform an MUUO — not because the code is illegal, but because it is actually unassigned rather than an LUUO.

In user mode perform the following operations using a block of four locations beginning at that specified by bits 6–35 of location 420 in the user process table. In the first two locations save the program flags and PC in a flag-PC doubleword; in the rest of the flag word clear bits 13–17 and 31–35, and store the instruction code and *A* in bits 18–26 and 27–30. In the third location store *E* in bits 6–35 (clear bits 0–5).

	0 5	6 12	13 17	18 26	27 30	31 35
0	FLAGS		00	CODE	*A*	00
1	00	PC				
2	00	*E*				
3	00	NEW PC				

Then load bits 6–35 of the fourth location into PC, and continue performing instructions in normal sequence beginning at the location then addressed by PC. If *E* is a local AC address, store it in global form (i.e. with a section number of 1).

MUUOs

The actions of MUUOs depend to a considerable degree on the processor, and also on which Monitor is in use. These are the MUUO codes.

TOPS–20	104;040–051, 055–077 in section 0
TOPS–10 except KA10	040–051, 055–077
KA10	040–051, 055–100

MUUOs have considerable flexibility in the way they can alter the operating characteristics of the machine (mode, section). But the information that governs the alterations is contained in the user process table, and is therefore assumed to be under sole control of the kernel program.

The unassigned codes, which are listed in Appendix E, are not MUUOs, but the processor reacts to them in the same way in order to turn control over to the Monitor. (In the KA10 there are minor differences as explained below.) The processor also takes the same action if the program gives a JRST with an undefined function, an instruction that is illegal because of the context in which it is given, an extended instruction with incorrectly formatted accumulators, or code 000. The last is so that control returns to the Monitor should a user program wipe itself out or inadvertently attempt to execute a location that has been cleared.

The rest of this section is devoted to the different ways in which MUUOs are performed. Except in the KA10, all types use locations in the user process table to store similar information. Figure 2–3 shows what information is stored in which locations for each processor type.

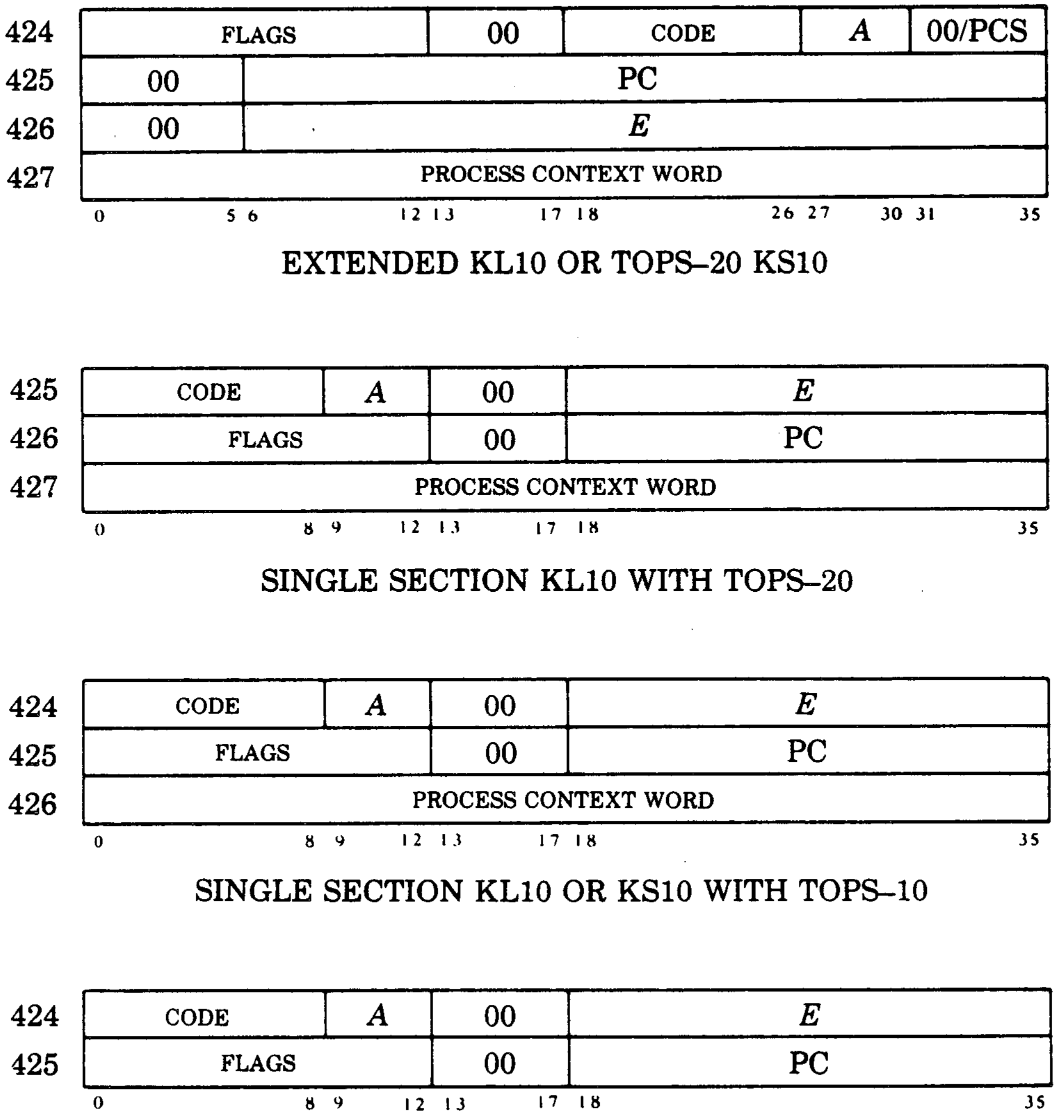
Extended KL10 MUUOs. In locations 424–426 of the user process table, store the same information (as specified above) that is stored in the first three locations of an LUUO block by an LUUO given in a nonzero section, except that when the MUUO is given in executive mode, also save the previous context section in bits 31–35 of location 424. Store the "process context word" in location 427; this word saves information that partially defines the context in which the MUUO is given, and is exactly the information read by a DATAI PAG, (§3.5). Complete the specification of the MUUO context by setting up the previous context flags, and clear the rest of the flags to place the processor in kernel mode. Then load PC from bits 6–35 of the appropriate location in a PC list, and continue performing instructions in normal sequence beginning at the location then addressed by PC. (Note that the MUUO can change PC from any section to any other.) The new PC 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 an overflow trap.

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

Single-section KL10 MUUOs. With either the TOPS-20 or TOPS-10 Monitor, MUUOs store the same information and take the same action, but they use a different set of three locations in the user process table. In the first location store the instruction code, *A* and the effective address *E* in bits 0–8, 9–12 and 18–35 respectively, and clear bits 13–17 (this is the same information as that stored by an LUUO given in section 0); save the flags and PC in a PC word in the second location; and save the process context word in the third location. Then set up the flags and PC according to the contents of the appropriate location in a PC word list, and continue performing instructions in normal sequence beginning at the location then addressed by PC. The PC word list occupies the same area as the PC list for an extended KL10, and it is organized and used (with respect to mode and trap) in the same way.

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.

Figure 2.3: User Process Table MUUO Configurations



EXTENDED KL10 OR TOPS–20 KS10

SINGLE SECTION KL10 WITH TOPS–20

SINGLE SECTION KL10 OR KS10 WITH TOPS–10

KI10

KS10 MUUOs. For the KS10 the PC or PC-word list contains only four entries for executive and user modes, in the locations corresponding to the kernel and concealed modes as given above — the supervisor and public locations are not used. The process context word for the KS10 is that read by an RDUBR (§4.5). Otherwise, with TOPS–20 an MUUO is performed in the same way as in an extended KL10, and with TOPS–10 it is performed in the same way as in a single-section KL10 running under TOPS–10.

KI10 MUUOs. An MUUO is performed in exactly the same way as on a single-section KL10 with the TOPS-10 Monitor, except that it does not store a process context word (only two words of information are stored in locations 424 and 425). Note that the trap locations in the PC-word table are used for either overflow or a page failure.

KA10 MUUOs. MUUOs and unassigned codes,[38] 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). 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, i.e. 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.

2.17 KS10 Input-Output Instructions

Unlike earlier processors, the KS10 has no special format for IO instructions. Instead they are simply those instructions that handle the peripheral equipment, the console and memory status — although for consistency, they do have 1s in the left three bits. KS10 IO instructions are oriented toward Unibus-type devices, as all peripheral equipment in a DECSYSTEM–2020 is handled through Unibus adapters. There are twelve of these instructions, six each for manipulating full words and bytes, described here in terms of their general effects for handling external devices. Information about external devices — individual instruction descriptions, IO addresses, etc. — is given in the device documentation (however memory status is defined in §4.8).

NOTE

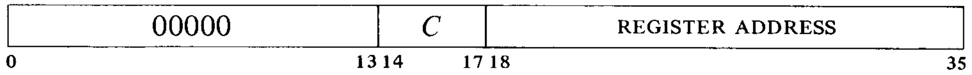
Ordinarily the user has no use whatever for the instructions described in this section. In almost all cases, input and output is handled by the Monitor in response to user requests employing MUUOs and various software formats. For information on user procedures vis-a-vis Monitor handling of user IO requirements, the reader should refer to the appropriate Monitor Calls manual.

Programmers who do handle their own input-output should note that the instructions described here are in-out instructions, which are affected by the timeshare instruction restrictions. Namely an instruction of this type cannot be performed by a user program unless User In-out is set. Any in-out instruction that violates this restriction does not perform the functions given for it in the instruction description. Instead it executes as an MUUO.

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

> The system instructions discussed in Chapters 3 and 5 for the other processors are also IO instructions. System instructions for the KS10 are not IO, but for consistency and convenience they are subject to the same restriction as IO instructions (determination of their legality is done by the same microcode test). This restriction will not be mentioned in the instruction descriptions, as it applies to *all* instructions from this point on.

As in all instructions the processor does an effective address calculation, but for the IO instructions it ignores the result and recomputes an effective IO address beginning with the *I*, *X* and *Y* parts of the instruction word. The IO address specifies an IO register in some Unibus device or in the console or memory controller, and for convenience we shall refer to this effective IO address also as *E*. An IO address is analogous to an extended virtual address in that it has a fundamental length of thirty bits, but not all of its bits are implemented in a given processor. In a KS10 IO address the right eighteen bits are the register address, and the left twelve are the controller number, of which only four bits are implemented. An IO address thus has this format,



where *C* is the controller number and bits 0–13 must be zero. Of the sixteen possible controller numbers only three are used at present: 0 addresses the console and the memory controller; 1 addresses Unibus adapter 1; 3 addresses Unibus adapter 3. Presently allowed IO addresses are these, and no others can be used.

Controller	*Register Address*	*Specifies*
0	100000	Memory status
0	200000	Console (microcode only)
1	400000–777777	Adapter 1 Unibus registers
3	400000–777777	Adapter 3 Unibus registers

The IO address calculation is like an effective address calculation in which the result can be "global", i.e. can have more than eighteen bits. When the result is an 18-bit "local" register address, it is automatically interpreted as specifying controller 0. The calculation is limited to one level of indirection or indexing or both, and any intermediate result that is used as a memory address must be local (since the KS10 is confined to section 0).

If there is no indexing or indirection, the IO address is simply *Y*.

If there is indexing only and the left half of XR is negative, the IO address is the local sum of *Y* and XR right.

If there is indexing only and XR is positive, the IO address is the global sum of *Y* and XR (but remember that bits 0–13 must be zero).

If there is indirection only, the IO address is the contents of location *Y*.

If there is both indexing and indirection, the IO address is the contents of the location specified by the sum of *Y* and XR right.

Note that an index register can supply the entire IO address, but it can also be used to supply only the controller number when *Y* is the register address. This latter technique is useful for employing common code for both adapters.

BSIO **Bit Set IO**

714	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the word read from IO register *E*, set bits corresponding to 1s in AC, and write the result back in register *E*.

BCIO **Bit Clear IO**

715	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the word read from IO register *E*, clear bits corresponding to 1s in AC, and write the result back in register *E*.

RDIO **Read IO**

712	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of IO register *E* into AC.

WRIO **Write IO**

713	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Write the contents of AC into IO register *E*.

TIOE Test IO Equal

710	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If all bits of IO register *E* corresponding to 1s in AC are zero, skip the next instruction in sequence.

TION Test IO Not Equal

711	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If not all bits of IO register *E* corresponding to 1s in AC are zero, skip the next instruction in sequence.

BSIOB Bit Set IO Byte

724	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the byte read from IO register *E*, clear bits corresponding to 1s in AC bits 28–35, and write the result back in register *E*.

BCIOB Bit Clear IO Byte

725	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the byte read from IO register *E*, clear bits corresponding to 1s in AC bits 28–35, and write the result back in register *E*.

RDIOB Read IO Byte

722	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of IO register *E* into AC bits 28–35. Clear AC bits 0–27.

WRIOB Write IO Byte

723	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Write the contents of AC bits 28–35 into IO register *E*.

TIOEB Test IO Equal, Byte

720	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If all bits of IO register *E* corresponding to 1s in AC bits 28–35 are zero, skip the next instruction in sequence.

TIONB Test IO Not Equal, Byte

721	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If not all bits of IO register *E* corresponding to 1s in AC bits 28–35 are zero, skip the next instruction in sequence.

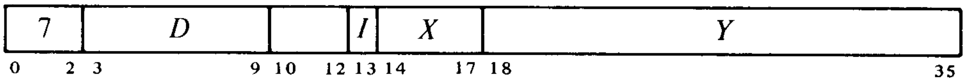
Unibus devices generally have data registers and control/status registers. Frequently a single IO address specifies two registers, one for reading and one for writing. A control register and a status register in a device usually have the same address and also have bits in common, i.e. information loaded into some of the control bits can be read as status. Ordinarily a device is set up by loading or adjusting individual bits of its control register. Data can then be read or written, and the state of the device can be determined by reading status or testing individual status bits. Complete information about the characteristics of each device is given in the device documentation.

Giving an IO address for a register that does not exist produces a page fail trap (§§4.3, 4.4).

2.18 Pre-KS10 Input-Output Instructions

In the KL10 and earlier processors, the input-output instructions control the movement of information to and from the peripheral equipment and perform many system-oriented operations within the processor, i.e. management of the internal devices, which in the KL10 are connected to the E bus.

An instruction in the in-out class is designated by 111 in bits 0–2, i.e. its left octal digit is 7. In this section these instructions are shown like this,



where bits 10–12 are given as a 2-digit octal number to select one of eight IO instructions, which are described here in terms of their general effects for handling external devices, and *D* addresses the device that is to respond to the instruction. The format thus allows for 128 device codes, of which the KL10 uses the first six (000–024) for internal devices (the KI10 uses the first three, the KA10 the first two). In instruction descriptions for individual devices, the instruction and device codes are combined into a single 5-digit code for bits 0–12. Codes for the internal devices are included in the tables in Appendix A, but all information about external devices — device codes, individual instruction descriptions, etc. — is given in the device documentation.[39] 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.

NOTE

Ordinarily the user has no use whatever for the instructions described in this section. In almost all cases, input and output is handled by the Monitor in response to user requests employing MUUOs and various software formats. For information on user procedures vis-a-vis Monitor handling of user IO requirements, the reader should refer to the appropriate Monitor Calls manual.

Programmers who do use these instructions should note that unless otherwise specified, all instructions described in the remainder of this manual are in-out instructions, which are affected by the timeshare instruction restrictions. Except in the KA10, in-out instructions using device codes 740 and above are not restricted. But an instruction using a device code under 740 (or given in a KA10) cannot be performed by a user 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 executes as an MUUO.

These restrictions will not be mentioned in the instruction descriptions, as they apply to *all* instructions from this point on.

[39] Electrical and logical specifications of the IO bus are given in the interface manual.

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*.[40] The number of condition bits in *E* that are actually used depends on the device.

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.

[40] *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.

CONSZ Conditions In and Skip if Zero

7	*D*	3 0	*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.[41]

CONSO Conditions In and Skip if One

7	*D*	3 4	*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.[41]

BLKO Block Out

7	*D*	1 0	*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 [42] 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.

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.

[41] Condition bits in the left half word can be tested by reading them with a CONI and then using a test instruction (§2.7).

[42] In the KA10 incrementing both halves of the pointer is effected by adding 1000001_8 to the entire register (and a carry can therefore go from the right half into the left).

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 level. Otherwise dismiss the interrupt and return to the interrupted program.

It is not expected that block instructions will be of any use in a DECSYSTEM–20. For compatibility however, the address supplied by the pointer is taken to be in the local section.

Notes. 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 external device can be given in terms of these four instructions, two of which are for input and two for output.[43] The four exhaust the types of information transfer that occur in the IO system.

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 may have no direct connection with output conditions.

Data is moved in and out in bytes of various sizes or in full 36-bit words. Each program 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. A DATAI that addresses an output-only device simply clears location *E*. On the other hand a DATAO that addresses an input-only de-

[43] 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.

vice 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*.

The general effects of the IO instructions are as given above, but a single instruction varies in its individual effects from one external device to another. For KI10 and KA10 internal devices, the instructions still have the same general effects and have the same relation to one another; but again they vary in individual effects that are documented in the descriptions in Chapter 5. The situation is quite different however with respect to KL10 internal devices. For example, a DATAI PAG, is really a DATAI — it reads information from the pager; but a DATAI CCA, is not a DATAI — it sweeps through the cache invalidating all pages, and it has its own mnemonic, SWPIA. The instruction BLKI PI, has no connection whatever with DATAI PI, because it is not a block instruction at all — it is actually the instruction RDERA, which reads the error address register. In other words, although some of the IO instructions for KL10 internal devices are equivalent in general terms to the same instructions for external peripherals, many of them are uniquely defined operations that bear none of the standard relationships to the typical case or to other instructions using the same device code (in some cases even when for the same device). When a unique mnemonic is assigned for an instruction, the form using a device mnemonic is given at the right end of the top line in the description.

2.19 User Programming

The preceding sections define the machine language characteristics of the system from a user point of view. But efficient and effective use of the system is affected greatly by the software; the user should therefore consult the appropriate Monitor manual, especially for the employment of the Monitor for input-output. For convenience we list here those rules that the user must observe and that are the result of KL10 and KS10 hardware characteristics.

- If an area of memory is write-protected, e.g. 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 only in the manner prescribed in the Monitor manual. Unless they are prescribed for special circumstances, the unassigned codes should not be used. Code 000 should never be used under any circumstances.
- Do not use HALT (JRST 4), unless you want your program stopped.
- Always be aware of the context in which the program is running, and make sure to use only operations appropriate to that context. In particular be familiar with which forms of the JRST instruction are legal in which circumstances, as explained in §2.9. JRST functions for handling interrupts are legal when IO is legal.
- Unless User In-out is set do not give any IO instruction with device code less than 740 (any at all in the KS10). The program can determine if User In-out is set by examining bit 6 of the saved flags.

- 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, i.e. a location containing a PORTAL (JRST 1,).
- In an extended processor, do not use XBLT or SFM in section 0 or JEN in a nonzero section. Also be aware of the differences between running in section 0 and in other sections. Differences appear both in the execution of instructions, such as JSR and JSP, and in the format and handling of such quantities as index registers, indirect address words, and stack and byte pointers.
- Make sure to format the accumulators correctly in string instructions (§2.12).

The user can give a JRSTF or XJRSTF but a 0 in bit 5 of the PC word or flag 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. Similiarly a 1 in bit 7 sets Public, but a 0 does not clear it, so a public program cannot enter concealed mode this way.

Many hardware characteristics however 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 or very large (even larger than available physical memory), the actual constraints will be dictated by the particular Monitor and the system manager. Most TOPS–10 Monitors 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 space than necessary or cause excessive page swapping.

The basic idea is to localize everything as much as possible. Do not spread parts of the program out through the address space leaving gaps. Put together whatever will be used together: divide a large program into smaller segments, and with each group of instructions put whatever pointers, data locations and the like that will be used with it. Group together subroutines that are called by the same programs. If a package is to be used at all frequently, take advantage of the various features, e.g. a core map, provided by the Digital software to determine just how the package was assembled, and if necessary revise it to reduce the working set of pages.

The rules given above apply generally to all systems, but there are minor differences from one to another, and a user who wishes to write programs to run on more than one type of processor must be aware of whatever incompatibilities exist. For example, the interrupt handling JRST functions are legal in user IO mode except on the KI10, where they are restricted to kernel mode. Because of the more restricting JRST decoding in the earlier processors, the KL10 and KS10 have more functions, and they produce quite different effects when given in a KI10 or KA10 program. The matter of unassigned codes works both ways with respect to different processor models: instructions added in a later machine use codes unassigned in earlier machines, but the codes for the software double precision floating point instructions are unassigned in later machines. Unassigned codes that correspond to implemented instructions in other machines should be used only if the software includes interpretive routines for them, but wherever possible they should be avoided because of the severe time penalty.

Chapter 3
KL10 System Operations

The information presented in this chapter is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own IO, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

WARNING

> KL10 functions are implemented in microcode, which can be changed much more easily than hardware. Although user operations are deliberately kept as compatible as possible from one machine to the next, Digital will change the KL10 system microcode whenever such change will result in greater speed, efficiency or effectiveness. Therefore anyone writing system software should make sure to use the most recently updated version of this documentation, and before embarking on any project as enormous and critical as an operating system, to check with Large Systems Engineering for any changes not yet documented.

Programming for the system as a whole is programming in executive mode. Only the kernel program is without instruction restrictions, and only it can, if needed, access physical memory unpaged. The supervisor program labors under the same instruction restrictions as the user and has no way of bypassing them, although it can read but not alter concealed pages (the kernel program can supply data tables to the supervisor program, and the latter cannot affect them).

The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working sets, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of a user. The kernel program accomplishes these objectives by handling all in–out for the system, setting up page maps, trap locations, interrupt locations and the like for both itself and the users, handling user accounts, communicating with the front end, and so forth. In other words, except for handling in-out, the activities of an operating system are the topics covered in this chapter. Of course the system programmer must also be quite familiar with all of the material presented in the preceding chapters. In particular he must fully understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the JRST instruction, MUUOs, and IO instructions (§§2.9, 2.16, 2.18).

System information for other processors is given in Chapters 4 and 5. The present chapter is devoted solely to the KL10, but contains two sections on paging, only one of which is applicable to a given system. §3.3 describes the paging used with the TOPS–10 Monitor in a Single-section KL10; this paging is similar to that of the KI10. §3.4 treats the paging associated with the TOPS–20 Monitor and the TOPS–10 Monitor in an Extended KL10. Both kinds of paging employ essentially the same hardware — the difference lies principally in the microcode.

Much of the material presented here is related to the DTE20s, the channels, and the DIA20. Although the chapter does describe all activities of the microcode undertaken for these devices (e.g. the front end functions in §3.7), the descriptions of the devices themselves are not included.

CAUTION

All IO instructions in this chapter are for internal devices (E bus functions). An address given by such an instruction for storing a result is always interpreted as global in the section containing the instruction. Hence data or conditions in cannot be stored in an AC unless the instruction is in section 0 or 1.

3.1 Priority Interrrupt

The DECSYSTEM–20 is essentially a system of processors clustered around the E bus. The various controllers and interfaces are subsidiary to the PDP–10, but maintain a considerable degree of independence from it. Each RH20 Massbus controller operates from its own command list in memory and handles all data transfers via the channels; but it must reach the Ten program to start a new list or if something should go wrong. Each PDP–11 is a whole computer with its own internal program; but for handling IO equipment or acting as the system console, it must communicate with Ten memory via the E bus (to which it is interfaced by a DTE20), and the peripheral computer must reach the Ten program for setting up mutual operations. Basically the priority interrupt system allows the other processors to interrupt the central processor at various levels of priority, so that all can operate simultaneously. The hardware also allows conditions internal to the PDP–10 to signal its own program by requesting an interrupt.

In a DECsystem–10, the PDP–11 is limited to use as a system console and diagnostic facility, and the unit-record peripheral equipment is organized around a KI10-type IO bus connected to the E bus via a DIA20 IO bus interface. If the system lacks internal channels, Massbus controllers must

be of the RH10 type, which the program controls via the IO bus. For data purposes an RH10 is connected to external memory by a separate memory bus. It is recommended that those who program a DECsystem–10 read both this section and the first few pages of the discussion of the KI10 interrupt[1] (§5.2).

Interrupt Requests

Interrupt requests are handled on eight levels arranged in a priority sequence. Levels are numbered 0–7, with 0 having highest priority. Level 0 is quite unlike the others, however, in that it is available only to the front end processors for simulating console functions and handling byte transfers. Moreover level 0 is always active — it cannot be turned off even by inactivating the interrupt system. The program does control the enabling of level 0 in the DTE20s, but the master front end can even override that. Assignment of devices[2] to the remaining levels is entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35); a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be placed on the same level.

When a device requires service, it sends an interrupt request signal on its assigned level over the bus to the processor. A request is recognized by the processor if the level is active — meaning that both the interrupt system and the individual level[3] have been turned on. But the processor can accept no requests while it is processing a request or starting an interrupt at any level, or holding an interrupt on the same level or on a level with higher priority than those on which requests have been recognized (in other words, if the current program is a higher priority interrupt routine). The request signal remains on the bus however until turned off by an appropriate response from the processor: either given by the program (CONO, DATAO, or DATAI, depending on the device), or generated automatically by the hardware. Thus if a request is not recognized or accepted when made, it will be when the necessary conditions are satisfied. A single level will even 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.

[1] On the Ten side of the DIA20, the interrupt works as described here. But on the other side it acts more like the KI10 interrupt, with seven programmable levels, second-order priority determined by proximity to the DIA20, etc. Of course the processor activities and interrupt functions available are those of the KL10.

[2] As explained in §2.18, the program treats all E bus controllers, internal subsystems, and IO bus peripherals as IO devices. In other words, it monitors and controls them by means of IO instructions using appropriate device codes. For a PDP–11, the device is the DTE20.

[3] Remember that level 0 is always active, even when the interrupt system is off. In other respects this discussion applies to all levels.

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 level 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. For example, 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.

Processing a Request. The processor handles only one request at a time. When it is ready, it accepts the highest priority request currently recognized, provided that request is on a level higher than the current program (all levels are higher than a noninterrupt program). To process a request the hardware sends an interrupt service demand to the devices on the E bus to determine which ones are currently requesting an interrupt on the accepted level. Note that at this point the processor is accepting not an individual request, but rather a class of requests: namely all those being made on the same level. Should the bus be busy, the demand is sent as soon as it becomes available, taking precedence over any IO instruction that may also be waiting (note that in this situation the program actually stops). From among the devices that respond to the demand on the accepted level, the processor selects the one of highest priority[4] according to this schedule:

Devices in Order of Decreasing Priority	*Physical Device Numbers*[5]
Interval counter	
Other internal requests — processor error flags, program initiated request	
Channels 0–7	0–7
DTE20s 0–3	10–13
DIA20 — i.e. any device on the IO bus	17

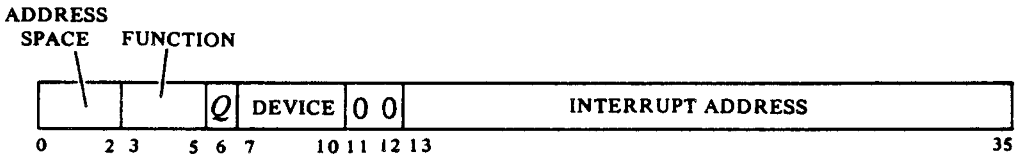
[4] There are therefore two orders of priority associated with an interrupt: first the level, and then for all devices requesting interrupts simultaneously on the same level, physical device number. These physical numbers are not the device codes used in the IO instructions; they are just for interrupt priority purposes and depend on position on the backplane (the RH20s are ordered opposite from the slot numbers).

[5] Physical numbers 14–16 are not used.

If the device selected is internal, no further processing of the request is required. Otherwise the hardware sends a function demand to the selected device (by specifying its physical number along with the interrupt level), and the device responds by returning an interrupt function word. In either case, once all necessary information about the request has been gathered, the interrupt system waits for the interrupt to start. The microcode checks frequently for a waiting request, and upon discovering one departs from its normal routine to start an interrupt. At such time PC points to the interrupted instruction, so a correct return can later be made to the interrupted program.

Interrupt Functions and Instructions

The action taken by the microcode to start an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the executive process table are associated with each level, locations 40 + 2*N* and 41 + 2*N*, where *N* is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 57. The processor starts a "standard" interrupt for level *N* by executing the instruction in the first interrupt location for the level, i.e. location 40 + 2*N*. This type of interrupt is performed for a processor error or program-initiated request, for an external device whose function word specifies a standard interrupt, and also for an IO bus device that returns no function word. The fixed locations however need not be used. The interrupt function word sent by the device may specify an equivalent interrupt using a pair of locations selected by the function word, or some other interrupt function entirely. The function word (which is saved in AC 3, block 7) has this format.



The microcode acts from a function word whether there is one or not; its absence is taken as a zero function. The DIA20 returns the word supplied over the IO bus or simulates a zero word. Bits 7–10 identify the device by its physical number, but this is supplied by the interrupt hardware, not the device. The meanings of the other bits in the word are as follows.

0–2 In unrestricted examine and deposit functions, codes given in these bits select the space in which the address supplied in bits 13–35 is interpreted.

0 Executive process table
1 Executive virtual address space
4 Physical address space

Remaining codes are reserved.

3–6 Interrupt function (bits 3–5), sometimes qualified by *Q* (bit 6). When unspecified, *Q* is irrelevant. The microcode handles functions 4–6 even when it is in the halt loop.

0 Internal device or zero word: for the interval counter perform a vector interrupt (see function 2); otherwise perform a standard interrupt (see function 1).

1 Standard interrupt — execute the instruction in location 40 + 2*N* of the executive process table.

2 Vector interrupt — action depends on device type as follows:

Interval counter — execute the instruction in location 514 of the executive process table.

DTE20 — execute the instruction in location 2 of the corresponding DTE20 control block.[6]

Channel — execute the instruction in the executive process table location specified by bits 27–35.

DIA20 — dispatch interrupt: execute the instruction in the executive virtual location specified by bits 13–35.

3 Increment — depending on whether *Q* is 0 or 1, add 1 to or subtract 1 from the contents of the executive virtual location specified by bits 13–35.

4 Examine — send the contents of the specified location to the selected DTE20. If *Q* is 0, select the location according to bits 0–2 and 13–35. If *Q* is 1, use bits 14–35 as a physical address and restrict the function to the communication area defined in the DTE20 control block.[6] The examine is effected by performing a DATAO to the DTE20.

5 Deposit — load the word supplied by the selected DTE20 into the specified location. If *Q* is 0, select the location according to bits 0–2 and 13–35. If *Q* is 1, use bits 14–35 as a physical address and restrict the function to the communication area defined in the DTE20 control block.[6] The deposit is effected by performing a DATAI to the DTE20.

6 Byte transfer — increment the byte pointer for the direction specified by *Q* (0 out, 1 in) from the control block for the selected DTE20, and then move a byte between Ten memory and the DTE20 according to the altered pointer.[6]

7 Reserved (result indeterminate).

CAUTION

Because of the special cycle in which it is executed, an interrupt function that uses virtual addressing cannot employ indirect pointers in its paging procedure (§3.4).

[6] For further information on front end interrupt functions, refer to §3.7.

13–35 The bits among these that supply the address when the function requires one depend on the address space.

Executive process table	27–35
Executive extended virtual address space	13–35
Executive unextended virtual address space	18–35
Physical address space	14–35

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode, and are therefore in executive virtual address space unless the particular function selects some other form of addressing. 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 level assigned to the processor (§3.8). These considerations of course do not apply to a service routine called by an interrupt instruction.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being 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 level, in direct response by the hardware (rather than by the program) to a request on that level. These locations may be the fixed ones for a standard interrupt or those given by the function word for a vector interrupt. §2.17 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. There are two types of interrupt instructions executed in a standard or dispatch interrupt; the effects of all other instructions are undefined.

BLKI, BLKO. If the pointer count is not zero, the processor dismisses the interrupt and returns immediately to the interrupted program (i.e. it returns control to the unchanged PC). If the count is zero, the processor executes the instruction contained in the second interrupt location.

XPCW, JSR. The processor holds an interrupt on the level, 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 flag word of the XPCW. Hence the instruction is usually a jump to a service routine handled by the Monitor. XPCW is the preferred instruction on the extended KL10.

The most important point of which the programmer must be aware is that even while User is set, the interrupt instructions are not part of the user program. They are executed in kernel mode and are therefore subject only to kernel mode restrictions. Regardless of the current PC section, the address part of an interrupt instruction is interpreted as referencing section 0, except in a dispatch interrupt, where it references the section specified by the interrupt function word. As an interrupt instruction, JSR automatically clears both User and Public to jump to a kernel mode service routine. An XPCW should be set up to produce the same result. The XPCW control block must be in section 0 unless the interrupt is a dispatch.

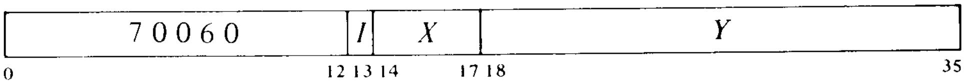
CAUTION

> Because of the special cycle in which an interrupt instruction is executed, the paging procedure for it cannot employ indirect pointers (§3.4).

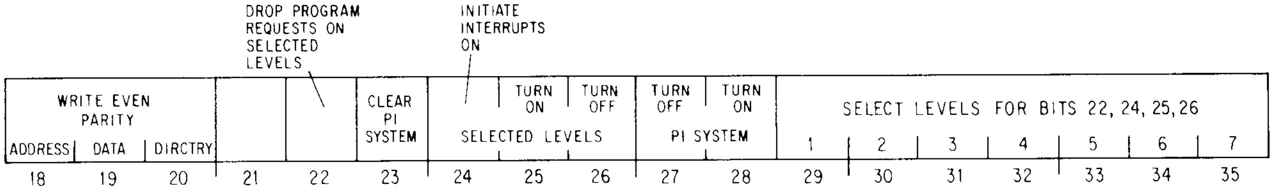
Interrupt Programming

The program can control the priority interrupt system by means of condition IO instructions. The device code is 004, mnemonic PI.[7]

CONO PI, **Conditions Out, Priority Interrupt**



Perform the functions specified by the effective conditions *E* as shown[8] (a 1 in a bit produces the indicated function, a 0 has no effect).



22 On levels selected by 1s in bits 29–35, turn off any interrupt requests made previously by the program (via bit 24).

23 Turn off the priority interrupt system, turn off all levels, drop all program-set requests, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by 1s in bits 29–35, and force the processor to recognize them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a CONO that selects the same channel and has a 1 in bit 22.

[7] Data instructions with device code PI are unassigned and execute as MUUOs. The block instructions are used for error and diagnostic purposes (§3.8).

[8] Bits 18–20 are for test purposes only. They are used to force errors and are discussed in §3.8.

Remember that the processor allows the program to continue while it processes a request. Thus when this bit forces recognition of a request, many additional program instructions may be performed before the interrupt, even on the highest priority level. Moreover if the request is allowed to remain, additional instructions may be performed between successive interrupts. For other than the highest priority level, the greater the number of higher levels active, the greater the amount of program time available both initially and between successive interrupts. If the program forces an interrupt on the lowest level when all are active, there can be a very long time between CONO PI, and its interrupt.

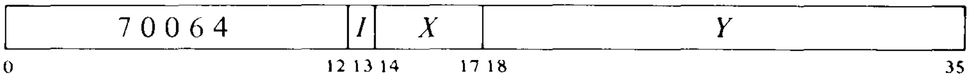
25 Turn on the levels selected by 1s in bits 29–35 so interrupt requests can be recognized on them.

26 Turn off the levels by 1s in bits 29–35, so interrupt requests cannot be recognized on them unless made by a CONO PI, with a 1 in bit 24.

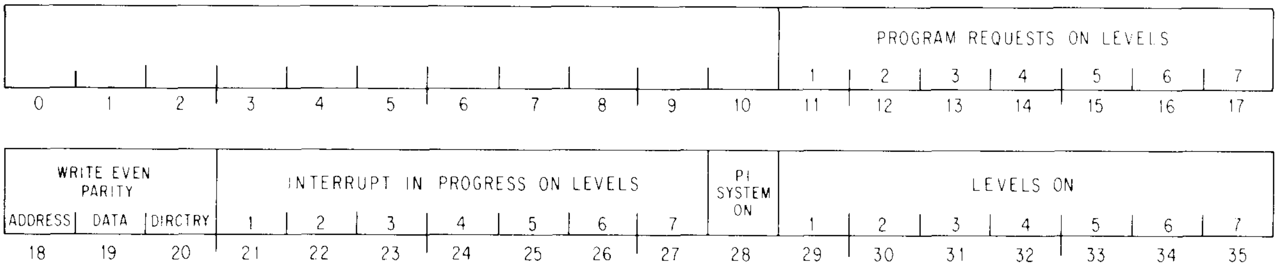
27 Turn off the interrupt system so no requests can be recognized.

28 Turn on the interrupt system so the hardware can process requests.

CONI PI, Conditions In, Priority Interrupt



Read the status of the priority interrupt (and several diagnostic bits) into location *E* as shown.



Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; and 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on, and 1s in bits 29–35 therefore indicate active levels.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18–20 reflect several diagnostic functions discussed in §3.8.

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 level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt request can be accepted on that level or any of lower priority.

A routine dismisses the interrupt by using an instruction that restores the level on which the interrupt is being held at the same time it returns to the interrupted program. The proper instruction is XJEN (JRST 7,) in an extended KL10, otherwise JEN (JRST 12,). Once the level is restored, the hardware can again accept requests and start interrupts on it and lower priority levels. These instructions also restore the flags: XJEN from the flag-PC doubleword if the routine was called by an XPCW; JEN from the left half of the PC word if the routine was called by a JSR in section 0. XJEN also restores the previous context section if the return is being made to an executive program.

CAUTION

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

Timing. The maximum time a device may wait for an interrupt to start depends on how many active devices are of higher priority and how long their service routines are. When a given request is of highest priority, its device need never wait longer than 10 μs.

Special Considerations. When an interrupt occurs, PC points to the interrupted instruction (or to an XCT that executed it), unless the interrupt occurred in an overflow trap instruction, in which case PC points to the instruction that overflowed. After taking care of the interrupt, the processor can always return to the interrupted instruction. Either *a*) the instruction did not change anything; *b*) the interrupt 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; or *c*) the interrupt occurred at some point in a multipart instruction where the microcode rigged the various pointers and other quantities so the processor actually restarts the instruction where it stopped, rather than from the beginning. However, in a BLT and in byte manipulation, the very mechanism that facilitates the return results in special properties of which the programmer must be aware.

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 flag 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 IDLB or IDPB would skip a byte. And if the routine restored the flag, the interrupted IDLB 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 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.

- Use interrupt instructions in a manner consistent with the special effects and conditions applicable to such instructions as described above.
- No request can be accepted, not even on higher priority levels, while a request is being processed or an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- To prevent a device from hanging up a level, the programmer must be aware of — and satisfy — whatever requirements the device has for dropping the request.
- The interrupt instruction that calls the routine should be an XPCW on an extended KL10, otherwise a JSR. In either case the paging for the instruction *must not* use indirect page pointers.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. Computations and any other time-consuming activities that can possibly be performed outside the routine should not be included within it.
- Never turn off the interrupt system in a routine unless it is absolutely necessary, and then always turn it back on again as soon as possible. If one or more levels can be turned off in place of the entire system, always do that instead.
- If the routine uses a UUO it must first save the contents of the locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type (§2.16).
- The routine must dismiss the interrupt (with an XJEN or JEN) when returning to the interrupted program. Flags and UUO locations should be restored.

3.2 Cache Management

For the user, the cache is transparent: any program simply gets information from memory and stores information in memory. But use of a cache as part of the memory subsystem reduces program time, since the cache is faster than the storage modules, and also reduces storage use by the program, making a larger percentage of total storage cycles available to other parts of the system. As explained in §1.7, transfers between processor and memory are in four-word groups: storage references are to four locations at a time.[9] The cache contains representations of a selection of such location groups. One may view the cache as 2048 general purpose registers, organized in sets of four, which substitute temporarily for the most frequently referenced physical storage location groups. The cache serves this function regularly for the program, and where considered appropriate, for microcode references as well. The way the hardware handles the cache depends upon whether the initial processor reference to a location in a particular group is read or write.

When the first processor reference to a group is to read the contents of one of its locations, memory control retrieves the entire four-word group containing the referenced location. The single word requested is supplied to the program, but all four are placed in the cache and are validated, i.e. they are tagged as words that do represent the true contents of memory. Subsequent references, read or write, to the same group are made to the cache, not to storage. If the processor modifies the contents of a location in the group, the new word supplied is substituted for the one in the cache location, which is tagged as written. Thus the cache word is different from storage but still valid — i.e. it represents what the storage location should contain.

When the first reference to a group is for writing, there is no call to storage at all. Instead the hardware sets aside a location group in the cache, with the one word in it tagged as both valid and written. Further reads or writes of the same location are handled solely with the cache, and subsequent writes to other locations in the same group are handled just like the first. But a read to a location that has not been written produces a storage reference. The requested word is given to the processor, and all words in the group that do not already have written representations in the cache are inserted into the group entry.

When storage is being updated or a group entry that is not in use is replaced by another, words just valid can be thrown away. But written words must eventually be sent to a storage module.

Cache Structure. The 2048 locations in the cache are contained in 128 lines of sixteen each. The lines are identified by the possible group numbers in a single page, 0–177. Each line contains four group entries for the given number. Each group entry in turn comprises the number of the physical page[10] containing the storage group corresponding to the entry and representations of the four locations in the group, each with valid, written and parity bits.

[9] Of course memory control does not blindly request four storage cycles for every group even when it is known that some are unnecessary. Fewer references are made when some locations in a group already have valid representations in the cache, or the first or last transfer in a channel block is for part of a group.

[10] The list of all page numbers makes up the cache "directory." For many hardware functions the cache is organized in four quadrants. A quadrant contains 128 group entries, one from each line.

The hardware also includes a mechanism for keeping track of the use of the various group entries. Whenever the processor references a group whose corresponding line in the cache already contains valid entries from four other pages, the hardware puts the new group representation in place of the least recently used entry in the line. But in doing so it also updates from any representations tagged as written in the displaced group entry.

Internal Channels. The channels are expected in general to deal with the storage modules, but if the cache contains any valid words for a page being handled through the channels, the hardware acts as follows:

> In an output operation, any valid representations at locations addressed by a channel are taken from the cache instead of storage.
>
> In an input operation, all data is sent to storage. However any entries that are in the cache for locations addressed by the channel are invalidated.

The reasons for this behavior are apparent. For output any valid words left in the cache might as well be taken since that is faster than going to storage. Furthermore some valid entries may have been written, and it is assumed that storage will certainly not be more up to date than the cache. Anything brought in via a channel is assumed to be the correct copy, and it should therefore go to storage as the page cannot be in use at the same time it is being loaded. Any valid entries left over in the cache must be from some previous operation, and they should therefore be invalidated, so any future references to those locations will go to storage for the correct copy. Should any of the valid leftovers be tagged as written, it is assumed the Monitor would have swapped out the modified page before bringing in the new. Of course a page used as temporary storage, or to hold counters and control words, albeit modified, can just be thrown away.

Cache Programming

The operations the program can perform on or for the cache are three: to invalidate, to validate, and to unload. Any of these operations may be carried out for all entries in the cache or for all entries of a single page. To invalidate a location is simply to clear its valid and written bits so it no longer represents anything. To validate or unload means to update storage, i.e. to write a cached word into storage if it is tagged as written, and to clear the written bit. Otherwise validating storage leaves the validity of the cache entries unchanged, whereas unloading invalidates all entries, written or not, in the groups being processed (all those in a single page or the entire cache).

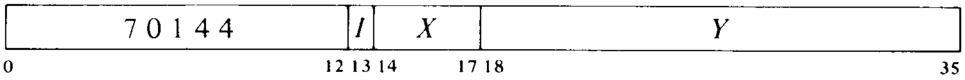
Following power turnon in any system, the cache use tables must be initialized and the cache invalidated, as its initial state is indeterminate. Beyond this, a system with a single central processor and internal channels requires no cache programming, as everything is handled adequately by the hardware. However if a system contains facilities that bypass the processor to deal directly with external memory, whether such facility be an external channel or another central processor, then the Monitor must actually manage the relationship between storage modules and cache.

As an example of such management and to illustrate the difference in use between validation and unloading, consider the situation in which a program is through with the data in a particular (modified) page and it is to be swapped via an external channel with new data brought into the same physical page for later use. The page must be unloaded into storage so that subsequently the program will go there for the new data. On the other hand suppose a program has created some code in a page, and the system is both to go ahead and execute it immediately and place it in a library. Now validation is the proper procedure: while the storage copy is being filed, the program can continue execution from the cache.

For initialization and management, there is one instruction that initializes the use tables and six that sweep the cache to perform the above three operations for a single page or all pages. Note that a sweep of the entire cache is always necessary, even for handling a single page, as there is no prior way of knowing whether any given line contains a group from any given page. Sweeping for a single page does however take less time than sweeping for all pages. In the latter case the sweeper must check all 512 group entries, whereas the former requires checking only every line to see if it contains an entry for the specified page, and there can be at most one such entry. Moreover sweeping for all pages can usually be expected to require more storage references than sweeping for a single page. In this light it should be noted that the sweep instructions simply initiate operations which are then carried forward by the cache sweeper. The program can continue while the sweep is going on, but this can be expected to slow down the sweep as the cache and program would then compete for storage references. That a sweep is in progress is indicated by the Sweep Busy flag being on, and at completion the sweeper clears Busy and sets Sweep Done. The program can check both of these flags among what are otherwise the processor error conditions, and it can enable the latter to request an interrupt on the level assigned to the processor (§3.8).

These are IO instructions wherein the cache sweeper has device code 014, mnemonic CCA. But the instructions have their own mnemonics since they bear no relation to the standard IO operations. Six of the eight are used: the BLKI and CONO also sweep, doing nothing but wasting cache cycle time. The single instruction that initializes the use tables is discussed at the end of the section.

SWPIA **Sweep Cache, Invalidate All Pages** (DATAI CCA,)



E is not used.[11]

Set Sweep Busy, and clear the valid and written bits in all cache entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

[11] *I*, *X* and *Y* are reserved and should be zero.

SWPIO Sweep Cache, Invalidate One Page (CONI CCA,)

70164	*I*	*X*	*Y*
0 — 12	13	14 — 17	18 — 35

Set Sweep Busy, and clear the valid and written bits in all cache entries for the physical page specified by bits 23–35 of *E*. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPVA Sweep Cache, Validate All Pages (BLKO CCA,)

70150	*I*	*X*	*Y*
0 — 12	13	14 — 17	18 — 35

E is not used.[11]

Set Sweep Busy, and write into storage all cached words whose written bits are set. Clear all written bits but do not change the validity of any entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPVO Sweep Cache, Validate One Page (CONSZ CCA,)

70170	*I*	*X*	*Y*
0 — 12	13	14 — 17	18 — 35

Set Sweep Busy, and write into storage all cached words whose written bits are set and which are found in entries for the physical page specified by bits 23–35 of *E*. Clear the written bits associated with those words sent to storage, but do not change the validity of any entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPUA Sweep Cache, Unload All Pages (DATAO CCA,)

70154	*I*	*X*	*Y*
0 — 12	13	14 — 17	18 — 35

E is not used.[11]

Set Sweep Busy, and write into storage all cached words whose written bits are set. Invalidate the entire cache, i.e. clear all valid and written bits. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPUO Sweep Cache, Unload One Page (CONSO CCA,)

70174	*I*	*X*	*Y*
0 12	13	14 17	18 35

Set Sweep Busy, and write into storage all cached words whose written bits are set and which are found in entries for the physical page specified by bits 23–35 of *E*. Invalidate all entries for the specified page, i.e. clear both their valid and written bits. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

Management of the cache is relatively straightforward. With external channels the program must simply be sure always to update storage pages before having them sent out, and to invalidate the cache entries for pages being brought in so processor references will go to storage for the new data.

The same procedures are used for a multiprocessor system, but here a problem arises when different processors are allowed to reference the same page at the same time, if either is allowed also to modify the page. Without modification the cache copies in both processors will remain valid; but if a processor modifies the page, the other cannot expect to get up-to-date data from cached words. To handle this situation, the pager includes mechanisms for bypassing the cache. Each page mapping[12] contains a cache bit for determining whether cache use is allowed for the given page. This cache bit applies only to an individual page, and has no effect at all unless cache use is enabled by the cache look bit. Analogous to the mapping cache bit is a load bit that applies to all unpaged references (such as pager references to the process tables). The look and load bits are among the conditions the Monitor provides to the pager. The way these "cache strategy" conditions govern cache use is as follows.

Look

0 The cache is disabled — go to storage for all references.

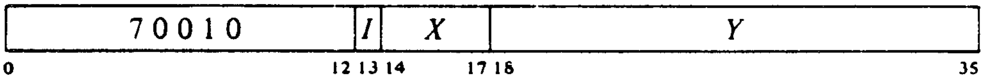
1 Look in the cache for all references. This means always use the cache (reading or writing) for any locations that already have valid representations. Furthermore when there is no valid representation for a reference, load the cache (reading or writing) if either the reference is unpaged and the load bit is 1, or the reference is paged and the cache bit in the mapping for the page is 1.

[12] For information on page mapping refer to §3.3 or §3.4 depending on whether the system uses respectively the TOPS–10 or TOPS–20 Monitor. Instructions for handling the pager are discussed in §3.5.

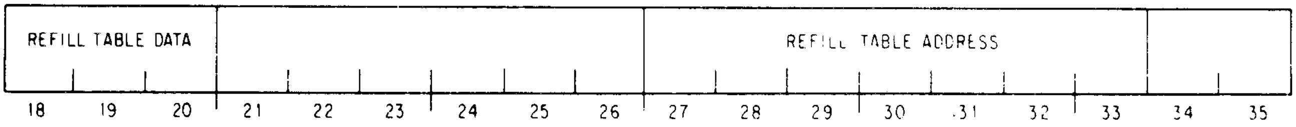
Timing. Simple invalidation takes little time, and it interferes minimally with the program since it requires no storage references. Otherwise an average sweep requires on the order of several hundred microseconds, but varies widely depending on the number of references required. Allowing the program to run simultaneously slows down the sweep because of competition for storage cycles, but program time is saved nonetheless.

Initializing the Cache. The use logic contains two tables each with 128 entries. Each entry in the use table identifies the use history — from most to least recently used — of the group entries in the corresponding cache line. With each reference, the use entry for the line must be updated. But instead of containing complex computational logic, the hardware has a refill table that supplies new use entries as a function of the previous use history of a given line and the group entry currently being accessed in the line. Following power up the program must initialize the use logic by giving this instruction 128 times to load every 3-bit location in the refill table.

WRFIL **Write Refill Table** (BLKO APR,)



Load the refill data given by bits 18–20 of *E* into the refill table location specified by bits 27–33.[13]



After filling the refill table by stepping through locations 0–177 (values of *E* that are multiples of 4 from 0 to 774), the program should give an SWPIA to invalidate the indeterminate initial contents of the cache. During the sweep the normal monitoring of cache access by the use logic initializes the use table from the refill table. The way the use table gets set up depends on the data pattern — the "refill algorithm" — loaded into the refill table, and the pattern selected depends on the use strategy desired for the cache. To limit cache use to a single quadrant, simply load the quadrant number (0–3) into the entire refill table. The usual use strategy is to allow equal use of all quadrants and to start with a presumed use history of most to least recently used corresponding to the numerical order of the quadrants. To implement this strategy,[14] load the following data pattern.

[13] The refill locations are selected by bits 27–33 to make use of the same lines that supply group numbers to address entries in the use table.

[14] For information on refill algorithms for other use strategies, refer to the writeup of MAINDEC10–DDQDA–L–D(SUBRTN).

	0	1	2	3	4	5	6	7
000	0	1	2	3	4	5	6	7
010	3	1	2	3	2	1	2	3
020	7	1	2	7	1	1	2	7
030	6	5	6	7	5	5	6	7
040	0	3	2	3	0	2	2	3
050	0	1	2	3	4	5	6	7
060	0	7	7	7	0	0	0	7
070	4	6	6	6	4	4	6	4
100	3	1	3	3	1	1	1	3
110	0	7	7	7	0	0	0	7
120	0	1	2	3	4	5	6	7
130	4	5	5	7	4	5	4	7
140	0	1	2	2	0	1	2	1
150	0	5	6	6	0	5	6	0
160	4	5	6	5	4	5	6	4
170	0	1	2	3	4	5	6	7

3.3 TOPS–10 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.3. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses TOPS–10 paging, i.e. a Single-section KL10 running a TOPS–10 Monitor (microcode version earlier than 271). The next section presents equivalent information for TOPS–20 paging. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–10 or TOPS–20, and are described in §3.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt functions and instructions reference executive virtual address space except in special cases where a function specifically calls for physical references. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the pager-microcode to carry out the mapping procedure, and also microcode references to retrieve interrupt instructions, handle traps and UUOs, and service the meters and front end.

Paging

All of memory both virtual and physical is divided into pages of 512 words each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18-bit addresses, where the left nine bits (18–26) specify the page number and the right nine (27–35) the location within the page. Physical memory can contain 8192 pages and requires 22-bit addresses, where the left thirteen bits (14–26) specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18-bit addresses into 22-bit addresses.[15] 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 pager 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 pager 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. For example, 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).

The executive virtual address space is also 256K, but the page map for it is in three parts. The map for the first 112K (pages 0–337) is in executive process table locations 600–757. 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. This means the Monitor can assign a different set of thirty-two physical pages (the per-process area) for its own use relative to each user. Hence when switching from one user to another, the Monitor need change only the user process table, this single substitution making whatever change is necessary in the executive address space for a particular user.

[15] For paging purposes 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 storage module locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to storage.

Figure 3.1: TOPS–10 Virtual Address Space and Process Table Layout

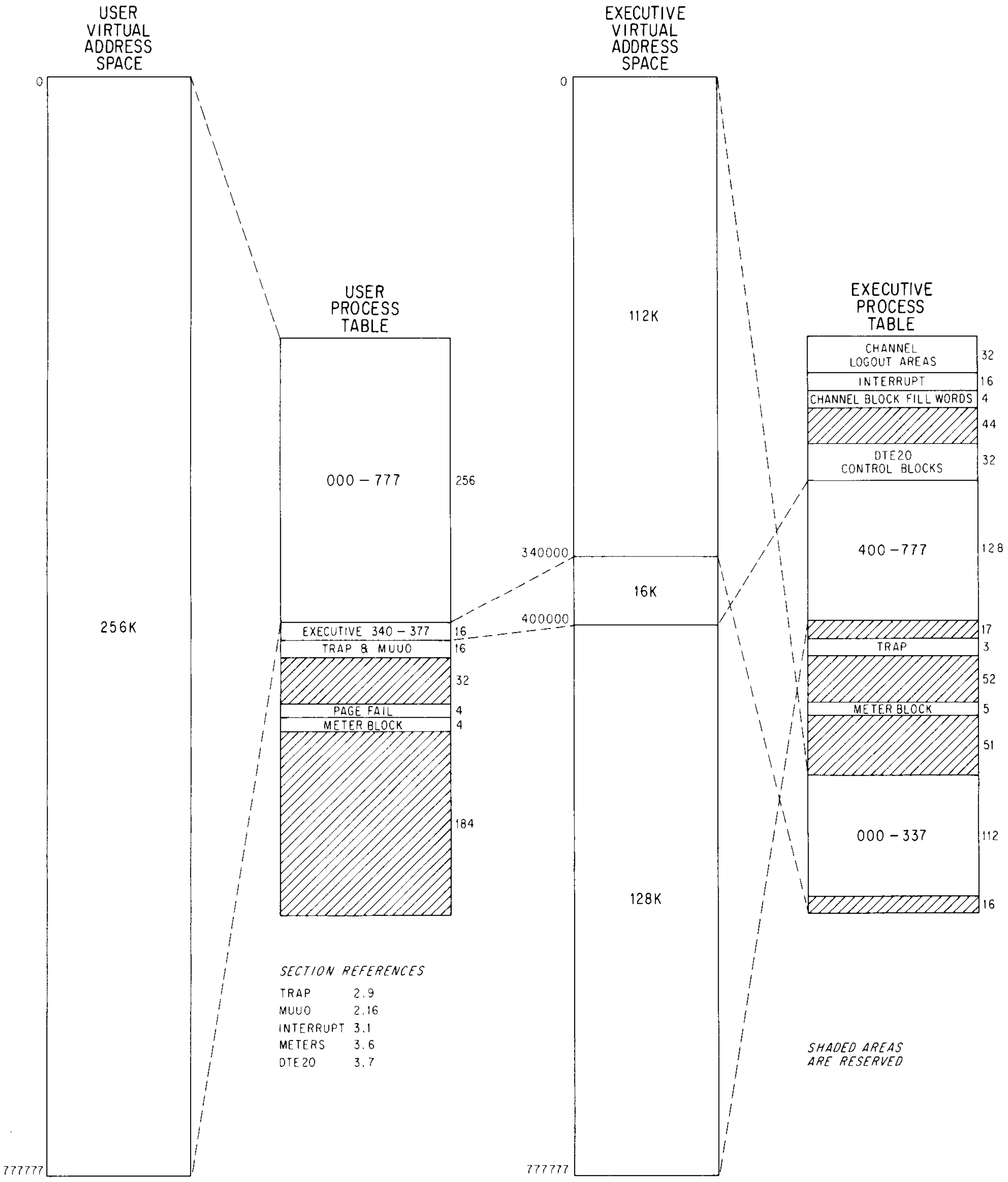


Figure 3.2: TOPS–10 Process Table Configuration

USER PROCESS TABLE

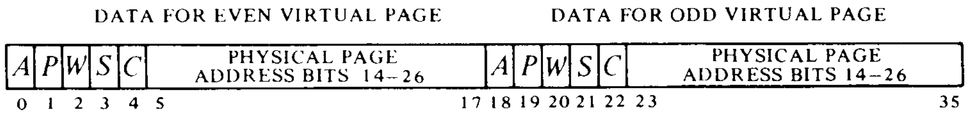
0	USER PAGE 0	USER PAGE 1
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	RESERVED	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	MUUO OLD PC WORD	
426	MUUO PROCESS CONTEXT WORD	
427	RESERVED	
430	KERNEL NO TRAP MUUO NEW PC WORD	
431	KERNEL TRAP MUUO NEW PC WORD	
432	SUPERVISOR NO TRAP MUUO NEW PC WORD	
433	SUPERVISOR TRAP MUUO NEW PC WORD	
434	CONCEALED NO TRAP MUUO NEW PC WORD	
435	CONCEALED TRAP MUUO NEW PC WORD	
436	PUBLIC NO TRAP MUUO NEW PC WORD	
437	PUBLIC TRAP MUUO NEW PC WORD	
440 477	RESERVED	
500	PAGE FAIL WORD	
501	PAGE FAIL OLD PC WORD	
502	PAGE FAIL NEW PC WORD	
503	RESERVED	
504 505	USER PROCESS EXECUTION TIME	
506 507	USER MEMORY REFERENCE COUNT	
510 777	RESERVED	

EXECUTIVE PROCESS TABLE

0 37	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED	
40 41	RESERVED	
42 57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
60 63	FOUR CHANNEL BLOCK FILL WORDS	
64 137	RESERVED	
140 177	FOUR DTE20 CONTROL BLOCKS	
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400 420	RESERVED	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424 507	RESERVED	
510 511	TIME BASE	
512 513	PERFORMANCE ANALYSIS COUNT	
514	INTERVAL COUNTER INTERRUPT INSTRUCTION	
515 577	RESERVED	
600	EXECUTIVE PAGE 0	EXECUTIVE PAGE 1
757	EXECUTIVE PAGE 336	EXECUTIVE PAGE 337
760 777	RESERVED	

Figures 3.1 and 3.2 show the organization of the virtual address spaces, the process tables and the maps for both user and executive. The first illustration gives the correspondence between the various parts of the address spaces and the corresponding parts of the page maps. The second illustration lists the detailed configuration of the process tables as determined by the hardware. Any table locations not used are reserved for future use by the hardware or for use 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.[16] The Monitor also specifies whether each page is public or not, writable or not, and cacheable or not. The cache bit has an effect only if cache use is enabled as the current cache strategy (§3.2); in this case a 1 in the cache bit allows loading the cache for the physical page when referenced as this particular virtual page, whereas a 0 limits cache use to look but do not load. Each word in the page map has this format to supply the necessary information for two virtual pages.


DATA FOR EVEN VIRTUAL PAGE
DATA FOR ODD VIRTUAL PAGE
A
P
W
S
C
PHYSICAL PAGE ADDRESS BITS 14–26
A
P
W
S
C
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 "page use" bits are as follows.

Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Public
W	Writable (not write-protected)
S	Software (not interpreted by the hardware)
C	Cacheable

Page Table. 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

[16] 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.

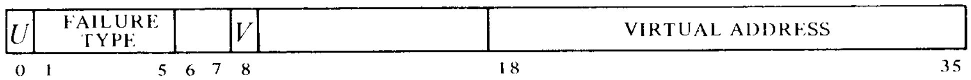
pager contains a page table, in which it keeps a large assortment of mappings for both the executive and the current user. In a manner analogous to the way the cache is organized to handle word groups of four, the pager handles mappings in sets of eight. A page set is eight consecutively numbered pages beginning with one whose number is a multiple of 10_8. Each page set consists of those pages whose mappings are contained in a single word group in the page map. The 512 locations in the page table are contained in sixty-four lines, each of eight locations holding the mappings for the eight pages of a set. The lines are identified by the possible page-set numbers in an address space, 0–77, and the individual locations are accessed by means of the virtual page numbers, 0–777. Each location has a parity bit and the complete mapping (i.e. map half word) for the virtual page that identifies it, including the physical page number and the five page use bits. Associated with each line are a bit that indicates whether the specified page set is in the user or executive address space, and a bit that indicates whether the set of mappings is valid or not (it is not suitable to clear a line as zero is a perfectly valid mapping, albeit for an inaccessible page). The user and validity bits for all lines collectively constitute the page table directory.

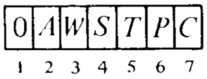
When the program references a page contained in a page set whose mapping entry is tagged as valid and in the program address space, the 13-bit physical number from the mapping location for the virtual page is used as the left thirteen bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *A*, *P* and *W* bits). If however the mapping set is invalid or is not for the correct address space, the pager makes a memory reference (referred to as a "page refill cycle") to get the word group containing the mapping for the specified virtual page from the page map. Even when there is no cache, all eight mappings from the word group are entered into the page table, filling and validating the line for the page set. This means the mappings will also be in the table for subsequent references to pages in the same set, although some may require a trap to the Monitor to make them accessible.

Note that all the mappings in an entire line of the page table are for a single space, user or executive. Since most programs are written beginning at page 0 (and often page 400 for a pure part), a mechanism is built into the table to avoid excessive refills due to switching between user and executive. In the numbers actually used to select lines in the table, the value of address bit 19 is inverted in user address space. For a given page number, this causes a difference of 200 in the line selection number for user space as against executive space. Suppose the executive uses pages 0–37 and 400–437, and also uses the per-process area, pages 340–377. Then if the user is limited to pages 0–137, 240–577 and 640–777, no conflict will ever occur between them in the page table.

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the pager terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap.[17] The trap operation makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current PC word in location 501, and sets up the flags and PC according to a new PC word in location 502. The processor then resumes operation in the new state at the location now addressed by PC. The page fail word supplies this information.





IF BIT 1 IS 0, BITS 1–7 HAVE THIS FORMAT

Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a virtual address was given for the reference. If bit 1 is 1, bits 6 and 7 are indeterminate, and the number in bits 1–5 (≥ 20) indicates the type of "hard" failure as follows.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page, or a public program has attempted to fetch an instruction from a concealed page at an illegal entry point (one not containing a PORTAL). The failure for an illegal entry (which forces bit 8 to 0) occurs at the next reference, after the instruction is decoded, so the fail address is meaningless.

22 Page refill failure — this is a hardware malfunction. The pager found no mapping for the virtual page in the page table, so it refilled the line from the page map but still could not find it.

23 Address failure — this is caused by the satisfaction of an address condition selected by the program. It is used for debugging purposes, such as to find an instruction that is maliciously wiping out a memory location, and is explained in §3.5 with the description of the DATAO APR, instruction that sets it up. Bit 8 is forced to 0 by this failure.

25 Page table parity error — the pager has encountered a page table mapping with incorrect parity.

36 AR parity error — the processor has detected incorrect parity in a word read into AR from a storage module, the cache, or the E bus, and has saved the word with correct parity in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

[17] A page failure that occurs during an interrupt instruction does not act this way. Instead it places a page fail word in AC 2, block 7, and sets the In-out Page Failure flag (CONI APR, bit 26), requesting an interrupt on the level assigned to the processor.

37 ARX parity error — the processor has detected incorrect parity in a word read into ARX from a storage module or the cache, and has saved the word with correct parity in AC 1, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

If the failure is not one of these, then bits 1–7 have the format shown above, where *A*, *W*, *S*, *P* and *C* are simply the corresponding bits taken from the mapping for the page specified by bits 18–26, and *T* indicates the type of reference in which the failure occurred — 0 for a read-only reference, 1 for any reference involving writing. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course *T* being 1 in conjunction with *W* being 0 certainly implies the cause of failure.

For a page fail trap, the new PC word is set up 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 or from the stopping position in a multipart instruction. Even a two-part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §3.1.

Note that a soft failure[18] seldom implies that anything is "wrong" — unless a program has attempted to write in a truly write-protected area. Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in core; 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 its mapping as inaccessible), the Monitor would respond to the page failure by bringing in the needed page from the disk, either adding to the user space or swapping out a page the user no longer needs.

The same situation exists for writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered, distinguishing those that must be revised on the disk at the end from those that can be thrown away by setting the software bit. Then in response to a write failure, the Monitor makes the page writable and sets the software bit to indicate to itself that that page has in fact been altered and must be saved. When the user is done, the Monitor need write back onto the disk only those pages for which both *W* and *S* are set.

[18] In a soft page failure or page table parity error, the line containing the mapping for the page is invalidated on the assumption the Monitor will change it. When the instruction is restarted, the pager must go to the page map to get new information for the table.

The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory, e.g. to set up a channel command list. For such purposes the processor has this instruction, which unlike all other instructions described in this chapter, is not an IO instruction even though it is subject to the same restrictions.

MAP **Map an Address**

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

If the pager is on and the processor is in kernel or user IO mode, map the page number of the virtual effective address *E* and place the resulting physical address and other map data in AC. The information loaded into AC for a true mapping is of the form

U	0	*A*	*W*	*S*	0	*P*	*C*	1	00	PHYSICAL ADDRESS
0	1	2	3	4	5	6	7	8	9 13	14 35

where bits 14–26 are the physical page number the pager supplies for *E*, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and *A*, *W*, *S*, *P* and *C* are the page use bits from the mapping as explained above. If however there is a parity error in the page table entry, or the paging is done in user mode public but the page, while accessible, is private, AC receives

U	FAILURE TYPE	*P*	*C*	1	00	PHYSICAL ADDRESS
0	1 5	6	7	8	9 13	14 35

The failure code can be only 21 or 25 for a proprietary or parity error, where in the latter case those bits supplied by the mapping, 6, 7 and 14–35, are meaningless.

This instruction cannot be performed in a user program unless User Inout is set, nor in a supervisor program. Instead of mapping the address, it executes as an MUUO. If the pager is off, the result is undefined.

Notes. The instruction itself cannot fail because it does not actually reference memory: it just translates the address and gets other mapping data. However the effective address calculation could fail, and getting the mapping may require a refill, in which a hard failure could occur.

3.4 TOPS–20 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.3. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses TOPS–20 paging,[19] i.e. any KL10 running the TOPS–20 Monitor, or an Extended KL10 running the TOPS–10 Monitor (microcode version 271 or greater). The previous section presents equivalent information for TOPS–10 paging. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–20 or TOPS–10, and are described in §3.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt functions and instructions reference executive virtual address space except in special cases where a function specifically calls for physical references. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the pager-microcode to carry out the mapping procedure, and also microcode references to retrieve interrupt instructions, handle traps and UUOs, and service the meters and front end.

NOTE

> Hardware paging operations are inextricably intertwined with the activities of the Monitor. The reader must be familiar with both to be able to understand either fully.

Paging

All of memory both physical and virtual is divided into pages of 512 words each. Physical memory can contain 8192 pages; its locations are specified by 22-bit addresses, where the left thirteen bits (14–26) specify the page and the right nine (27–35) the location within the page. The virtual memory space addressable by a program is 16,384 pages and requires 23-bit addresses, where the left fourteen bits (13–26) are the extended page number. However the virtual space is usually regarded as composed of thirty-two sections, each of 512 pages. With this view, the extended page number has two parts: the left five bits (13–17) specify the section, and the right nine (18–26) specify the page.[20] Thus within each virtual section, locations

[19] For additional information on this kind of paging, refer to "Storage organization and management in TENEX", by Daniel L. Murphy, AFIPS — Conference Proceedings, Vol. 41, page 23, AFIPS Press, Montvale, NJ.

[20] The reasons for holding to the section-page view are two. First, the page mapping procedures are actually set up that way. Second, although large data structures can arbitrarily cross section boundaries, the program cannot. For the program to get from one section to another requires an explicit transfer of program control. PC has twenty-three bits, but it counts in only the right eighteen: when going beyond the end of a section, it simply wraps around to the beginning of the same section (from location 777777 to 0).

are specified by 18-bit addresses, where the left nine bits (18–26) are the page number. The hardware maps each section of the virtual address space into a part of the physical address space by transforming the 18-bit addresses into 22-bit addresses.[21] In this transformation 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 translation maps a virtual page into a physical page by substituting a 13-bit physical page number for the 9-bit virtual page number. The mappings are different for each section by virtue of each section having a separate page map. The procedure is carried out automatically by the pager, but the maps that supply the necessary substitutions are set up by the kernel program.

Pointers to the page maps for the various user and executive virtual sections are contained in section tables that begin at location 540 in the user and executive process tables. The pager contains two 13-bit registers that the Monitor loads to specify the physical page numbers of these tables. To retrieve a section pointer from a process table, the pager uses the appropriate base page number as the left thirteen bits of the physical address and 540 plus the virtual section number as the right nine bits.[22] The section pointer must identify — either directly or indirectly — a physical page that contains the page map for the section. Every pointer and mapping takes one word, and since there are 512 pages in a section and 512 words in a page, a page map for a section requires exactly one page.

Figures 3.3, 3.4 and 3.5 show the organization of the virtual address spaces, the process tables and the section tables for both user and executive. The first illustration gives the general layout of the process tables and shows the relation between the virtual address spaces and section tables. The second and third illustrations list the detailed configuration of the process tables for the extended and single-section versions of the processor respectively. Any table locations not used are reserved for future use by the hardware or use by the Monitor for software functions.

Although the virtual space is always thirty-two sections of 256K by virtue of the addressing capability of the instruction and indirect word formats, the Monitor usually limits the actual address space for a given program by defining only certain sections or pages as accessible. There is no requirement that the accessible space be continuous — it can be scattered pages. The Monitor also specifies whether each section or page is public or not, writable or not, and cacheable or not. To determine the mapping for a given virtual page, the microcode carries out a pointer evaluation procedure that starts at the appropriate entry in the section table. If it is discovered during this procedure that the section or page is inaccessible, the page map or the referenced page is not in memory, or the program is attempting to write in a write-protected page, the microcode traps to the Monitor, which must handle the situation. A trap to the Monitor for a

[21] The mapping procedure is of course applied only to storage module references, whether cached or not. AC references, which can be made by any program, even when virtual page 0 is accessible, are made directly to fast memory and require no mapping.

[22] In a single-section KL10 paging procedures are still as given here, but all addresses have zero section numbers.

Figure 3.3: TOPS–20 Virtual Address Space and Process Table Layout

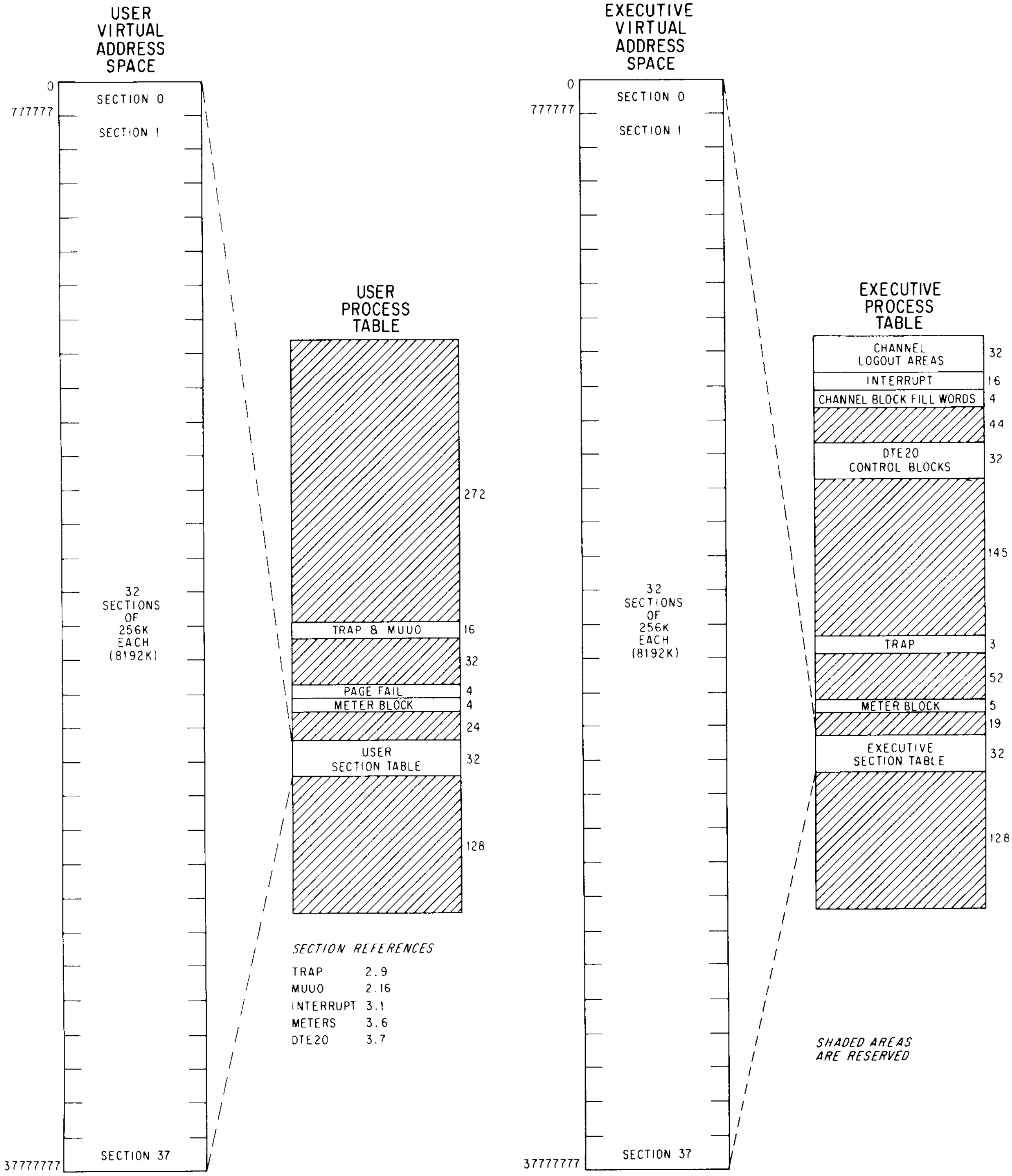


Figure 3.4: Extended TOPS–20 Process Table Configuration

USER PROCESS TABLE

0	RESERVED	
417		
420	ADDRESS OF LUUO BLOCK	*
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO FLAGS | MUUO OP CODE, A	*
425	MUUO OLD PC	*
426	E OF MUUO	*
427	MUUO PROCESS CONTEXT WORD	
430	KERNEL NO TRAP MUUO NEW PC	*
431	KERNEL TRAP MUUO NEW PC	*
432	SUPERVISOR NO TRAP MUUO NEW PC	*
433	SUPERVISOR TRAP MUUO NEW PC	*
434	CONCEALED NO TRAP MUUO NEW PC	*
435	CONCEALED TRAP MUUO NEW PC	*
436	PUBLIC NO TRAP MUUO NEW PC	*
437	PUBLIC TRAP MUUO NEW PC	*
440	RESERVED	
477		
500	PAGE FAIL WORD	*
501	PAGE FAIL FLAGS	*
502	PAGE FAIL OLD PC	*
503	PAGE FAIL NEW PC	*
504	USER PROCESS EXECUTION TIME	
505		
506	USER MEMORY REFERENCE COUNT	
507		
510	RESERVED	
537		
540	USER SECTION 0	
577	USER SECTION 37	
600	RESERVED	
777		

NOTE:
ASTERISKS INDICATE LOCATIONS WHOSE USE DIFFERS FROM THOSE IN THE SINGLE-SECTION PROCESS TABLE LISTED ON THE NEXT PAGE.

EXECUTIVE PROCESS TABLE

0	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED
37	
40	RESERVED
41	
42	STANDARD PRIORITY INTERRUPT INSTRUCTIONS
57	
60	FOUR CHANNEL BLOCK FILL WORDS
63	
64	RESERVED
137	
140	FOUR DTE20 CONTROL BLOCKS
177	
200	RESERVED
420	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION
424	RESERVED
507	
510	TIME BASE
511	
512	PERFORMANCE ANALYSIS COUNT
513	
514	INTERVAL COUNTER INTERRUPT INSTRUCTION
515	RESERVED
537	
540	EXECUTIVE SECTION 0
577	EXECUTIVE SECTION 37
600	RESERVED
777	

Figure 3.5: Single-section TOPS–20 Process Table Configuration

USER PROCESS TABLE

NOTE:
ASTERISKS INDICATE LOCATIONS WHOSE USE DIFFERS FROM THOSE IN THE EXTENDED PROCESS TABLE LISTED ON THE PRECEDING PAGE.

0 420	RESERVED	*
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	RESERVED	*
425	MUUO STORED HERE	*
426	MUUO OLD PC WORD	*
427	MUUO PROCESS CONTEXT WORD	
430	KERNEL NO TRAP MUUO NEW PC WORD	*
431	KERNEL TRAP MUUO NEW PC WORD	*
432	SUPERVISOR NO TRAP MUUO NEW PC WORD	*
433	SUPERVISOR TRAP MUUO NEW PC WORD	*
434	CONCEALED NO TRAP MUUO NEW PC WORD	*
435	CONCEALED TRAP MUUO NEW PC WORD	*
436	PUBLIC NO TRAP MUUO NEW PC WORD	*
437	PUBLIC TRAP MUUO NEW PC WORD	*
440 500	RESERVED	*
501	PAGE FAIL WORD	*
502	PAGE FAIL OLD PC WORD	*
503	PAGE FAIL NEW PC WORD	*
504 505	USER PROCESS EXECUTION TIME	
506 507	USER MEMORY REFERENCE COUNT	
510 537	RESERVED	
540 577	USER SECTION 0 USER SECTION 37	
600 777	RESERVED	

EXECUTIVE PROCESS TABLE

0 37	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED
40 41	RESERVED
42 57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS
60 63	FOUR CHANNEL BLOCK FILL WORDS
64 137	RESERVED
140 177	FOUR DTE20 CONTROL BLOCKS
200 420	RESERVED
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION
424 507	RESERVED
510 511	TIME BASE
512 513	PERFORMANCE ANALYSIS COUNT
514	INTERVAL COUNTER INTERRUPT INSTRUCTION
515 537	RESERVED
540 577	EXECUTIVE SECTION 0 EXECUTIVE SECTION 37
600 777	RESERVED

reason of this sort is produced by generating a "soft page failure." But if nothing is amiss, the procedure is carried out entirely by the microcode — with no need to call the software — and it generates the mapping for the specified virtual page. The procedure requires access to both the section table and page map, to a memory status table in which the microcode keeps track of the use made of the page map and the program-referenced page, and perhaps to other predefined or software-defined tables as well. If the complete procedure were carried out in every instance, the processor would require at least five memory references for every one by the program. To avoid this, each mapping generated by the procedure is placed in a page table, and the pager makes its virtual-to-physical translations from the mappings held in the table. Hence it is necessary to go through the evaluation procedure only when the mapping is not available in the page table. Since the objective of the procedure is to place a mapping in the table, it is referred to as a "page refill."

Page Table. A location in the page table contains a mapping entry in this format.[23]

A	*P*	*M*	*W*	*C*	PHYSICAL PAGE ADDRESS BITS 14–26

Each entry is identified as providing the physical page number for the translation for a particular virtual page in a particular section and address space (user or executive). A 1 in the *A* bit means the location contains a valid mapping, and the page is therefore immediately accessible without reguiring further action by the pager. Otherwise the rest of the entry is meaningless,[24] as *A* being 0 does not necessarily mean the page is inaccessible — only that a refill is required to determine its accessibility. The properties represented by 1s in the remaining "page use" bits are as follows.

Bit *Meaning of a 1 in the Bit*

P Public. A 0 means the page is private.

M Modified — and therefore writable without further ado. A refill produces a 1 in this bit if the page has already been modified or the reference that caused the refill is for write and the page is writable. A 0 does not imply that the page is write-protected, but simply that if a write reference occurs, the pager must find out if it can be written. Throughout this discussion, "write reference" means any reference involving writing; "read reference" means read only.

W Writable. A refill sets this bit if the page is writable (i.e. not write-protected).

[23] In the engineering drawings and even in some Monitor documents, the *M* bit is labeled "writeable" and the *W* bit is labeled "software", which names are consistent with their use in TOPS–10 paging.

[24] The microcode invalidates a mapping entry by clearing it, but clearing would not be sufficient were there no access bit, as zero is a legitimate mapping.

C Cacheable. This bit has an effect only if cache use is enabled as the current cache strategy (§3.2). In this case a 1 in the cache bit allows loading of the cache for the physical page when referenced as this particular virtual page, whereas a 0 limits cache use to look but do not load.

The page table is organized for page groups in a manner somewhat analogous to the way the cache handles word groups. A page group is four consecutively numbered pages beginning with one whose number is a multiple of 4. Each page group consists of those pages whose mappings are contained in a single word group in the page map. The 512 locations in the page table are contained in 128 lines, each of four locations for holding the mappings for the four pages of a group. The lines are identified by the possible page group numbers in a section, 0–177, and the individual locations are accessed by means of the virtual page numbers, 0–177. Each location has a parity bit and the complete mapping resulting from a refill, including the physical page number and the five page use bits. Associated with each line is a bit that indicates whether or not the line is valid, a bit that indicates whether the specified page group is in user or executive address space, and five bits that identify the section containing the page group.[25]

When the program references a page, the 13-bit physical number from the mapping for that page is used as the left thirteen bits in the physical address for the reference provided all necessary conditions are satisified. When the directory indicates the appropriate line is invalid or contains mappings for a different section or address space, the pager changes and validates the directory entry to match the desired reference but invalidates the four locations in the line by clearing their access bits. It then executes a refill to get the needed mapping into the table and tries the reference again. If there is already an appropriate directory entry, but the individual mapping is invalid or the reference is for writing and *M* is 0, the pager does a refill to get a valid mapping or checks whether it can be revised to allow the desired reference.

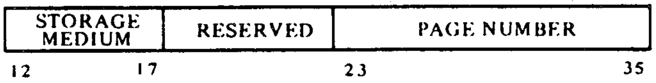
Note that all the mappings in a line of the page table are for a single space, user or executive, and for a single section. Since most programs are written beginning at page 0, a mechanism is built into the table to avoid excessive refills due to switching between user and executive and among sections. In the numbers actually used to select lines in the table, the value of address bit 19 is inverted in user address space, and the value of address bit 20 is inverted in an odd numbered section. For a given page number this causes a difference of 200 in the line selection number for user space as against executive space, and a difference of 100 for an odd section as against an even one. Suppose the executive uses pages 0–77 and 400–744 in section 1. Then if the user is limited to pages 0–277 and 400–677 in any even section, no conflict will ever occur between them in the page table. In

[25] The user bits, validity bits, and section numbers for all lines collectively constitute the page table directory. The Monitor invalidates the contents of the entire table by setting all the validity bits in the directory.

general a program should be organized so that it runs in a single section or in nonconflicting parts of different sections for some significant amount of time. Considerable yet unavoidable switching among sections can occur however in handling large data structures, as when it is necessary to handle the elements of a very large array in a number of different orders.

Page Refill

The refill of a mapping into the page table is accomplished by evaluating various types of pointers found in several kinds of tables. At some point in the procedure the microcode must encounter a "page address" that identifies the page map for the section, and it must end with a page address that identifies the physical page corresponding to the referenced virtual page. A page address has this format.



If bits 12–17 are zero, the storage medium is memory: i.e. bits 23–35 supply the number of a page that is in memory. If bits 12–17 are nonzero, the page exists but is stored on some other medium — perhaps the disk — and the microcode traps to the Monitor. A page address may be contained in a pointer, in which case some of the bits at its left have defined uses. But when the page address stands alone, bits 0–11 of the word containing it can be used arbitrarily by the software.

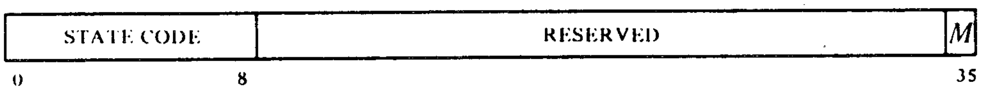
Special Tables. Besides the section tables in the process tables, a refill makes use of two predefined tables: the special page-address table (SPT) and the (core) memory status table (CST). These are software-determined tables in memory, but their base addresses are held in reserved fast memory locations, rather than in hardware registers like those of the process tables.[26]

The special page-address table contains page addresses that specify shared pages or special pages (e.g. those used as page maps or other software-defined tables). The microcode accesses specific entries in the SPT by indexing on a physical base address (bits 14–35) contained in AC 3, block 6. The pointer format provides for an index of eighteen bits, so the SPT can actually be as large as 256K (and it need not start on a page boundary).

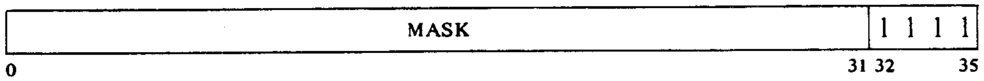
Information about the use made by programs of the various physical pages is kept in the memory status table. In every refill, unless the base address is zero the microcode updates CST entries for both the page containing the page map and the page referenced by the program. The entry for a page is a full word, and is accessed by adding the page number to a nonzero base address contained in AC 2, block 6. If memory is fully implemented at 8192 pages, the CST occupies sixteen of them, but need not begin on a page boundary. Note that the microcode does not manipulate CST entries for the process tables, the SPT, nor the CST itself, unless they are actually referenced by the program — in other words, unless the refill is being performed for a program reference to one of the tables.

[26] Remember that all memory tables defined by the pager are in physical address space, i.e. they have physical base addresses. Of course, to load or access a table, the Monitor must use paged virtual addresses. Note that if the base address is limited to a page number (bits 14–26), the table must begin at a page boundary.

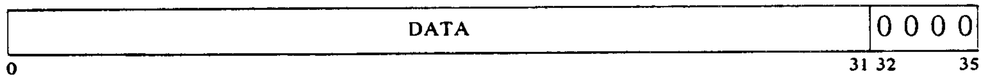
The status of a physical page in memory is indicated by a CST entry in this format.



The Monitor keeps a state code in bits 0–8 of the entry; within the code, bits 0–5 represent the page age, which must be nonzero for the page to be usable, whether it is the program-referenced page or the page map. Bits 0–5 being zero causes an age trap to the Monitor.[27] The microcode updates the entry by anding a CST mask word into it and oring a CST data word into that result. These two words are held respectively in AC 0 and AC 1, block 6. Bits 32–35 in them must be all 1s or all 0s as illustrated in order to preserve hardware information. A 1 in the *M* bit indicates the page has been modified since being brought into memory.[28] The microcode sets this bit in the entry for the referenced page — not that for the page map — if the reference is write and the page is writable.



CST MASK WORD



CST DATA WORD

Indirect pointers make use of tables whose locations are defined entirely by the Monitor. In a single refill, these may include one or more secondary section tables or page maps. Each such table or map is determined by a page address and a 9-bit index, and is therefore a single page. Memory status is kept only for the page maps.

Pointers. The microcode evaluates two kinds of pointers: section pointers and map pointers. The former are used in section tables and the latter in page maps. Members of these two classes are identical in form but differ enough in function so they must be treated separately. There are four types of section and map pointers distinguished by a type code in bits 0–2; of these, three are access pointers, i.e. they allow access to the given section or page. An access pointer has this format in its left seven bits.

[27] Zero age usually means the page is being swapped in and is not yet available for reference. The Monitor can use part of a CST entry to record which processes use the page.

[28] At the completion of a process, the Monitor checks the CST to determine which pages have been modified and must be rewritten on the disk.

TYPE	*P*	*W*		*C*
0	2	3	4	6

Every access pointer must have use bits for the section or page it represents. These bits, *P*, *W* and *C*, indicate whether the section or page is public, writable or cacheable. Throughout the evaluation procedure the microcode effectively ands these bits from one pointer to the next, so the final result requires that the given characteristics be specified at every step. In other words if *P* is 1 in the final pointer for the mapping, the page is public provided the entire section was also specified as public by the original section pointer, and "publicness" has been specified by every other pointer encountered along the way. Every access pointer must also either contain a page address or point to an SPT location that contains a page address.

Section Pointers. Entries in a section table are of these four types.[29]

No Access

0	AVAILABLE TO SOFTWARE
0 2	

The section is inaccessible.

Immediate

1	*P*	*W*		*C*	RESERVED	STORAGE MEDIUM	RESERVED	PAGE NUMBER OF PAGE MAP
0	2	3	4	6		12 17		23 35

If bits 12–17 are zero, the page map is in the page specified by bits 23–35. Otherwise the page map is not in memory.

An immediate pointer contains the page address of the page map.

Shared

2	*P*	*W*		*C*	RESERVED	INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF PAGE MAP
0	2	3	4	6		18 35

The page address of the page map is in the SPT at the location specified by bits 18–35.

This pointer is used for a page map shared by a number of processes. Switching to another map requires changing only the common SPT entry.

Indirect

3	*P*	*W*		*C*		SECTION TABLE INDEX	INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER SECTION TABLE
0	2	3	4	6		9 17	18 35

In the SPT location specified by bits 18–35 is the page address of a second-

[29] Type codes 4–7 are undefined.

ary section table. The next section pointer to be evaluated is in that table at the location specified by bits 9–17.

Indirect pointers are used for Monitor reference to per-job and per-process areas. The pointers remain while the second section table is swapped with the job or process, or the SPT entry is changed.

Map Pointers. Entries in a page map are of these four types.[29]

No Access

0	AVAILABLE TO SOFTWARE
0 2	

The page is inaccessible.

Immediate

1	*P*	*W*		*C*	RESERVED	STORAGE MEDIUM	RESERVED	PAGE NUMBER FOR MAPPING
0 2	3	4		6		12 17		23 35

If bits 12–17 are zero, the physical page specified by bits 23–35 corresponds to the referenced virtual page. Otherwise the referenced page is not in memory.

An immediate pointer contains the page address for the mapping.

Shared

2	*P*	*W*		*C*	RESERVED	INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS FOR MAPPING
0 2	3	4		6		18 35

The page address for the mapping for the referenced virtual page is in the SPT at the location specified by bits 18–35.

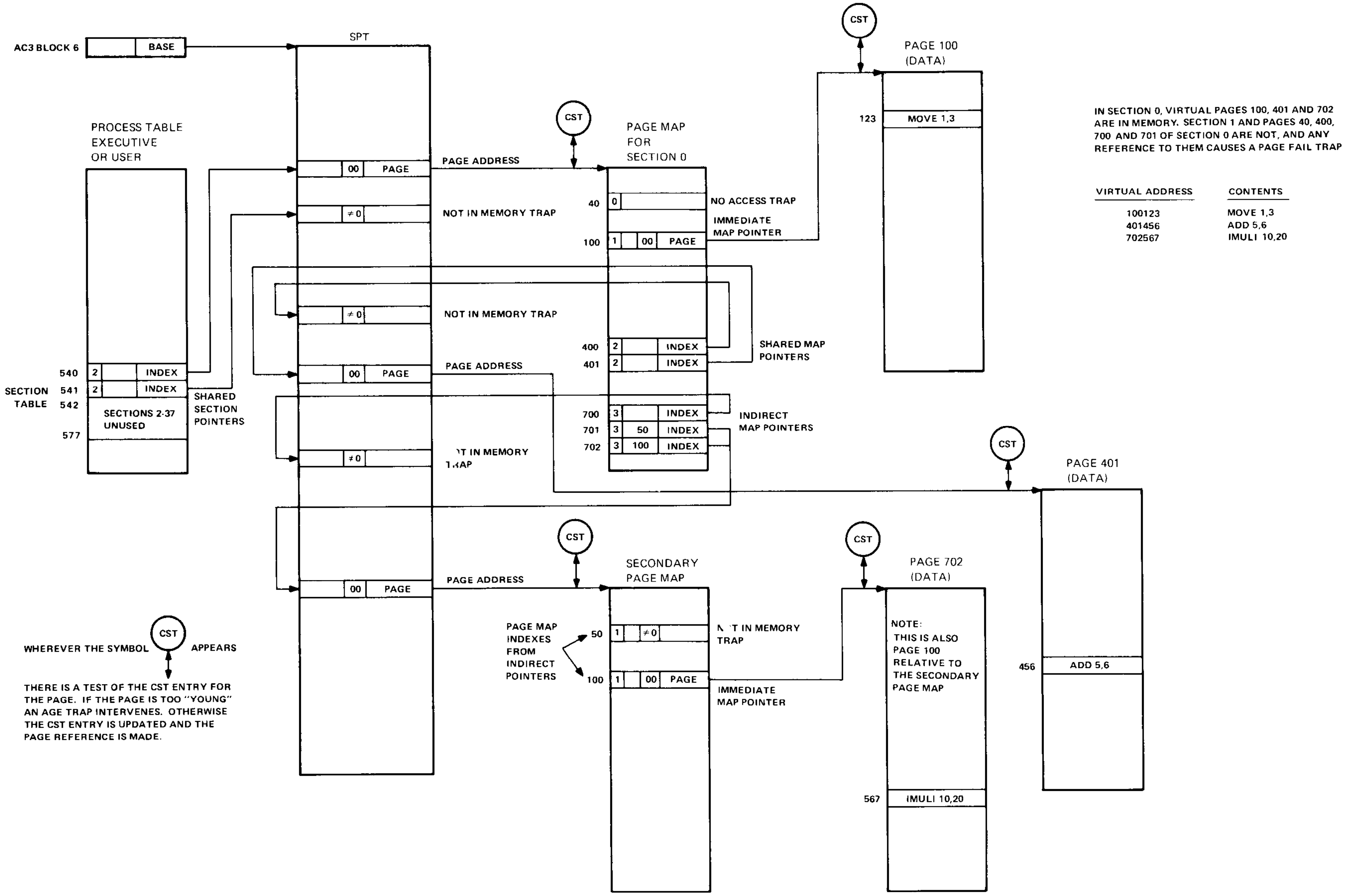
This pointer is used for a physical page referenced as different virtual pages by different programs. The Monitor can move the page simply by changing the SPT entry.

Indirect

3	*P*	*W*		*C*		PAGE MAP INDEX	INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER PAGE MAP
0 2	3	4		6		9 17	18 35

In the SPT location specified by bits 18–35 is the page address of a secondary page map. The next map pointer to be evaluated is in that map at the location specified by bits 9–17.

Figure 3.6: TOPS–20 Paging Pointer Evaluation



IN SECTION 0, VIRTUAL PAGES 100, 401 AND 702 ARE IN MEMORY. SECTION 1 AND PAGES 40, 400, 700 AND 701 OF SECTION 0 ARE NOT, AND ANY REFERENCE TO THEM CAUSES A PAGE FAIL TRAP

VIRTUAL ADDRESS	CONTENTS
100123	MOVE 1,3
401456	ADD 5,6
702567	IMULI 10,20

CAUTION

Indirect page pointers cannot be used for references made by interrupt instructions.

Refill Procedure. If the page table lacks a valid mapping for a reference, the pager must evaluate section and map pointers to get the desired mapping. The procedure begins with the pointer for the section from the process table, and the pager follows the trail laid by the various pointers, as illustrated in Figure 3.6. At any step the microcode traps to the Monitor if it encounters a no-access pointer or a page address that indicates the page is not in memory. The first part of the procedure, which may go to the SPT or indirectly through it to other section tables, evaluates section pointers to arrive at the page address of the page map. Using this physical page number as the left thirteen bits of an address and the number of the referenced virtual page as the right nine bits, the second part of the procedure retrieves a map pointer and evaluates it. This part may also go to the SPT or indirectly through it to other page maps to arrive at a page address for the mapping. Unless an age trap intervenes, or the CST base register is zero, memory status is updated along the way for any page maps used. If the reference can be made and there is no age trap for the referenced page, its status is updated including setting the *M* bit if the program is writing. The microcode then constructs the desired mapping, places it in the page table, and returns to the waiting reference.

The mapping data is constructed from the result of the pointer evaluation, including the running evaluation of the use bits, and has the format illustrated in the discussion of the page table. The microcode always places a 1 in the *A* bit to indicate that the virtual page is accessible and this is a valid mapping for it. *P* and *C* are simply the result of anding the *P* and *C* bits of the various pointers. *M* however is not. A refill sets up *M* and *W* according to the type of reference and the characteristics of the referenced page.

Circumstances[30]	*MW*	*Effect*
Read reference, page not writable.	00	An attempt to write will fail.
Read reference, page writable but not yet modified (according to CST).	01	An attempt to write will succeed, after the mapping is revised.
Page writable, write reference or page already modified.	11	Sets *M* in CST entry; an attempt to write will succeed.

Page Failure

When for any reason the pager is unable to make a desired memory reference, or an extended effective address calculation encounters an incorrectly formatted indirect word, an event known as a "page failure" occurs. For this the microcode terminates the instruction immediately, without dis-

[30] The missing circumstance produces a page failure.

turbing PC or storing any results in memory or the accumulators, and executes a page fail trap.[31] The trap operation makes use of certain locations in the user process table depending on whether the KL10 is extended.

Extended KL10

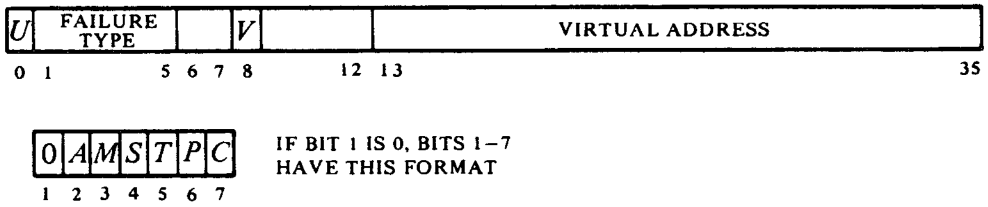
The trap places a page fail word in location 500, identifies the failed state of the processor by placing the current flag-PC doubleword in locations 501 and 502, sets up PC according to a new value in location 503, and clears the flags (placing the processor in kernel mode).

Single-section KL10

The trap places a page fail word in location 501, identifies the failed state of the processor by placing the current PC word in location 502, and sets up the flags and PC according to a new PC word in location 503.

The processor then resumes operation in the new state at the location now addressed by PC.

The page fail word supplies this information.



Whether the violation occurred in user or executive virtual address space is indicated, respectively, by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a virtual address was given for the reference. If bit 1 is 1, bits 6 and 7 are indeterminate, and the number in bits 1–5 (≥ 20) indicates the type of "hard" failure as follows.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page, or a public program has attempted to fetch an instruction from a concealed page at an illegal entry point (one not containing a PORTAL). The failure for an illegal entry (which forces bit 8 to 0) occurs at the next reference, after the instruction is decoded, so the fail address is meaningless.

23 Address failure — this is caused by the satisfaction of an address condition selected by the program. It is used for debugging purposes,

[31] A page failure that occurs during an interrupt instruction does not act this way. Instead it places a page fail word in AC 2, block 7, and sets the In-out Page Failure flag (CONI APR, bit 26), requesting an interrupt on the level assigned to the processor.

such as to find an instruction that is maliciously wiping out a memory location, and is explained in §3.5 with the description of the DATAO APR, instruction that sets it up. Bit 8 is forced to 0 by this failure.

24 Illegal indirect — an extended effective address calculation has encountered an indirect word with 11 in bits 0 and 1.

25 Page table parity error — the pager has encountered a page table mapping with incorrect parity.

27 Illegal address — a memory reference has supplied an address whose section number is greater than 37. Bit 8 is forced to 0 by this failure.

36 AR parity error — the processor has detected incorrect parity in a word read into AR from a storage module, the cache, or the E bus, and has saved the word with correct parity in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

37 ARX parity error — the processor has detected incorrect parity in a word read into ARX from a storage module or the cache, and has saved the word with correct parity in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

If the failure is not one of these, then bits 1–7 (if meaningful) have the format shown above, where *A*, *M*, *W*, *P* and *C* are simply the corresponding bits taken from the mapping for the page specified by bits 13–26, and *T* indicates the type of reference in which the failure occurred — 0 for a read-only reference, 1 for any reference involving writing. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Moreover the possible configurations for these bits are quite limited. A soft page failure can result only from actions taken in a refill or writability check. A valid page table mapping can require action by the pager only if *M* is 0 in a write reference. Hence in a soft failure resulting from a valid mapping, bits 0–8 of the page fail word are of the form

U	0	1	0	0	1	*P*	*C*	1
0	1	2	3	4	5	6	7	8

for a write failure. When no valid mapping is found, the page fail bits have the form

U	0	0	0	0	*T*	0	0	1
0	1	2	3	4	5	6	7	8

where for a write failure, T must be 1.

For a page fail trap, the extended KL10 automatically switches to kernel mode, and in the unextended version the Monitor should set up the new PC word for that action. After rectifying the situation, the Monitor eventually returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two-part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §3.1. Before returning to the failed instruction, the Monitor must invalidate the mapping for the page and revise the pointers for the new situation. Then when the instruction is restarted, the pager will do a refill to get the new, correct mapping.

A no-access pointer may well imply that the section or page simply does not exist. Otherwise a soft failure seldom implies that anything is "wrong." Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in memory. When the user attempts to gain access to a page that is not there (i.e. for which the refill encounters a not-in-memory page address), the Monitor would respond to the failure by bringing in the needed page from the disk, either adding to the user space, or swapping out a page the user no longer needs or has not used recently. Similarly a process using several sections may have only one in core at a time. While swapping is in progress, the Monitor runs some other user, returning to the interrupted job when the requested page is available.

The same situation exists for writability. Keeping track of modified pages is handled by the refill procedure using the memory status table. But a page may be write-protected because is it shared by a number of processes, wherein a change made by one might not be wanted by the others. Thus in response to a write failure, the Monitor might make a separate writable copy of the page for the sole use of the process that wishes to modify it.

The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory, e.g. to set up a channel command list. For such purposes the processor has this instruction, which unlike all other instructions described in this chapter, is not an IO instruction even though it is subject to the same restrictions.

MAP **Map an Address**

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

If the pager is on and the processor is in kernel or user IO mode, map the (extended) page number of the virtual effective address *E* and place the resulting physical address and other map data in AC. The information loaded into AC for a true mapping is of the form

U	0	1	*M*	*W*	0	*P*	*C*	1	00	PHYSICAL ADDRESS
0	1	2	3	4	5	6	7	8	9 13	14 35

where bits 14–26 are the physical page number the pager supplies for *E*, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and *M*, *W*, *P* and *C* are page use bits from the mapping as explained above. Failure of the instruction to generate a valid mapping is indicated by AC receiving

U	FAILURE TYPE		00	UNDEFINED
0	1 5	6 8	9 13	14 35

where bits 6–8 are undefined, and the failure code can be 21, 25, 27, 36 or 00 (refer to the preceding discussion of page failures). Of these, 25 and 36 represent what are effectively real failures: a parity error in the page table entry or in a word retrieved from memory in a refill. The others represent failures that would occur were the instruction actually to reference memory rather than simply requesting a mapping: 21, an attempt by a public program to reference a private page; 27, an illegal address; and 00, an age, no-access or not-in-memory trap in a refill.

This instruction cannot be performed in a user program unless User In-out is set, nor in a supervisor program. Instead of mapping the address, it executes as an MUUO. If the pager is off, the result is undefined.

Notes. The instruction cannot actually fail, because regardless of what happens, the refill or page fail microcode returns to it instead of trapping to the Monitor. The effective address calculation done for it could fail however.

3.5 Memory Management

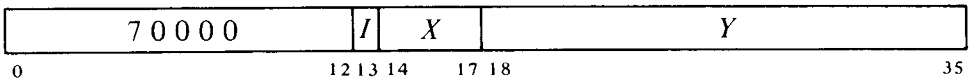
In order properly to manage memory, the kernel program must select the kind of paging and the cache strategy, set up process tables and page maps for itself and the various users, oversee the operation of the page table, and select the fast memory block to be used by each program (usually block 0 for itself). At any given time, accumulator, index register and fast memory references are made to that AC block that is assigned as "current." Given a particular processor mode (user or executive, public or private) and an appropriate process table and page map, the Monitor effectively defines the address space for a process (which may be itself) by specifying the base address for the process table and selecting the current AC block.

When a user program calls the Monitor it is usually to request some activity, which may often require the executive to gain access to the user address space. To facilitate the crossover from one address space to another, the same instruction through which the Monitor assigns its own current AC block also allows assignment of an AC block and section for the "previous context" — i.e. the context of the process that made the call. These quantities, together with flags that indicate the mode of the caller, allow execution of instructions in the previous context (more about this subject later). At any point in time, the previous context is essentially the circumstances in which the previous process was running. Note that the previous context need not be the user; the same techniques can be exploited following a call from one level of the Monitor to another.

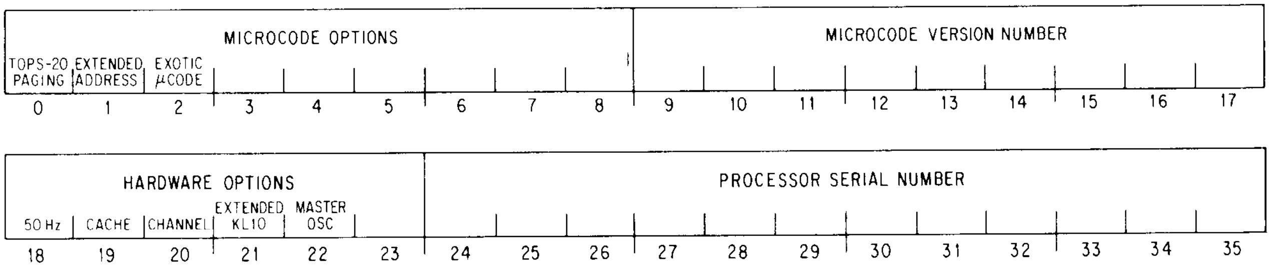
For initial setup, the kernel program must be cognizant of certain fundamental characteristics that can vary from one system to another. For this purpose the instructions for basic management include not only those that address the pager, but also one that addresses the processor to discover what those characteristics are.

The device code for the pager is 010, mnemonic PAG.[33]

APRID Arithmetic Procesor Identification



Read the microcode version number, the processor serial number, and a listing of the fundamental characteristics of the system into location E as shown.



0 The microcode implements paging for the TOPS–20 Monitor; 0 indicates TOPS–10 paging.

1 The microcode handles extended addresses.

2 The microcode differs in some way from the standard version.

18 Line power frequency is 50 Hz rather than the standard 60 Hz.

21 The processor is an extended KL10; 0 indicates a single-section KL10. The microcode options must of course be consistent with the processor type.

22 The system has a master oscillator, which is available as an external clock source. In a system containing MOS memory, the software must select this source (CPU clock source 2) from the PDP–11.

[33] BLKI PAG, is unassigned and executes as an MUUO.

CONO PAG, Conditions Out, Pager

70120	*I*	*X*	*Y*
0 12	13	14 17	18 35

Set up the system-oriented characteristics of the pager according to the effective conditions *E* as shown.

CACHE STRATEGY			TOPS-20 PAGING	ENABLE PAGER	EXECUTIVE BASE ADDRESS (PAGE NUMBER)
LOOK	LOAD				
18	19	20	21	22	23 24 25 26 27 28 29 30 31 32 33 34 35

Load bits 23–35 into the executive base register to select the executive process table. If bit 22 is 1 enable overflow trapping and enable the pager for the type of paging selected by bit 21: 1 TOPS–20, 0 TOPS–10. The paging selected *must* be the same as that implemented by the microcode as indicated by APRID bit 0. A 0 in bit 22 prevents traps and disables paging so all memory references are to physical locations unpaged.[34]

CAUTION

> Paging can be disabled only for executive mode. A user mode program will not run correctly unless the pager is turned on.

Select the cache strategy according to bits 0 and 1 as follows:

0*x* Disable the cache.

10 Look for all references, but do not load physical references; for virtual references act as directed by the cache bit in the mapping for the page.

11 Make complete use of the cache for physical references; for virtual references act as directed by the cache bit in the mapping for the page.

Invalidate the entire page table by setting the invalid bits in all lines.

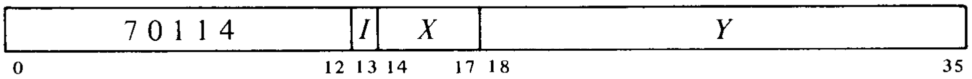
CONI PAG, Conditions In, Pager

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

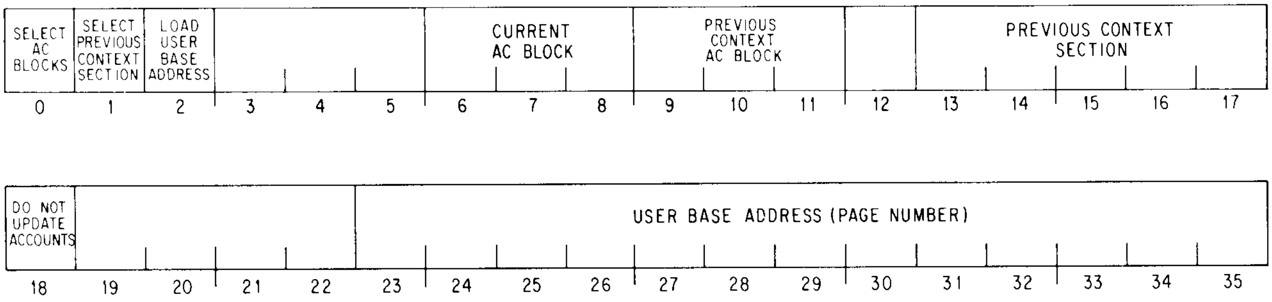
Read the system status of the pager into the right half of location *E*. The information read is the same as that supplied by a CONO.

[34] Note that disabling the pager does not mean there can be no page failures, as these can be caused by conditions having nothing to do with paging, i.e. with translating virtual to physical addresses.

DATAO PAG, Data Out, Pager



Set up the process-oriented elements of the pager according to the contents of location *E* as shown.



Bits 0–2 are change indicators for parts of the data word: when a bit is 0, the corresponding part of the word is ignored, and the equivalent value supplied by a previous DATAO remains in effect.

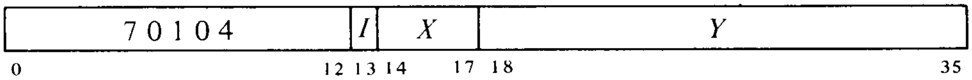
If bit 0 is 1, select as the current and previous context AC blocks those specified by bits 6–8 and 9–11, respectively. If bit 1 is 1, select as the previous context section that specified by bits 13–17 (which must be zero in a single section processor). If bit 2 is 1, perform these functions:

If bit 18 is 0, update the user accounts as explained in §3.6.

Load bits 23--35 into the user base register to select the user process table.

Invalidate the entire page table by setting the invalid bits in all lines.

DATAI PAG, Data In, Pager



Read the process status of the pager into location *E*. The information read is in the same format as that supplied by a DATAO (bits 0–2 are 1s and bit 18 is 0). Note however that only the AC block designations and user base address are necessarily the same information supplied by a previous DATAO. When an MUUO stores its own context as given by the DATAO that set up the process containing it, it changes the designation of the previous context section to that in which the program is currently running. Hence following a call by an MUUO, a DATAI PAG, in the called program will see as the previous context section that specified by PC at the time the MUUO was performed.

CLRPT Clear Page Table Entry (BLKO PAG,)

70110	*I*	*X*	*Y*
0 12	13	14 17	18 35

TOPS–20

Invalidate the page table mapping entry for the page referenced by *E*.

TOPS–10

Invalidate the page table line (eight entries) containing the mapping for the page referenced by *E*.

At power turnon the contents of the cache and page table are indeterminate, the processor is in kernel mode, paging is disabled, the cache is off, and the current AC block is 0 by default. After the front end loads the microcode, it then loads the initializing kernel program. This program, running unpaged in physical memory, should give an APRID to determine system characteristics and an SWPIA to invalidate the cache. The unpaged program ends with a CONO PAG, that selects the cache strategy, selects and enables paging, specifies the executive base address, and invalidates the page table. From this point the kernel program runs paged and must set up the first user or users, loading the user process tables and page maps, bringing in whatever parts of user programs and data that are consistent with good working-set management, and setting up the timing and accounting meters. Finally the Monitor gives a DATAO PAG, to assign the base address and current AC block for the first user, and then transfers control to the user program via an XJRSTF or JRSTF. The initial DATAO PAG, should have a 1 in bit 18 to inhibit updating accounts before any user has run.

On a call from the user via an MUUO, give a DATAI PAG, to determine the context of the user, i.e. his AC block and section. Then give a DATAO PAG, that assigns block 0 as current for the Monitor, assigns the user AC block and section as previous context for accessing user space, but leaves the base address alone so the right paging is still available for such access. To return to the same user, reassign the AC block without changing the base address. Leaving the base address alone also avoids unnecessary updating of user accounts. Note that on the transfer to a user program no previous context values need be given as the user cannot employ PXCTs. For switching from one user to another, give a DATAO PAG, that updates the first user's accounts in his process table, as specified by the old base address, and then loads a base address for the new user. The transfer to a user is done with a JRSTF or XJRSTF; the latter also restores the previous context section when used to return from a higher to a lower level within the executive.

The usual procedure for administering AC blocks is to assign block 1 to all users and assign two or three blocks for the sole use of interrupt routines. Suppose the assignments are: block 0 for the Monitor, block 1 for all users, block 2 for the highest priority interrupt level, block 3 for the second highest level, and block 4 for all other levels. Then in no circumstances is it necessary to determine which block to save, and interrupt routines on the highest, second highest and lowest levels need not save any. Moreover the Monitor need not even store block 1 when it takes control from a user temporarily. When switching from one user to another, the Monitor usually stores the first user's accumulators in his process table or 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 process table or shadow area, where they were stored after the last time the new user ran.

On a change from one process to another the entire page table must be invalidated, but this is done automatically by the instruction that assigns the new user base address. If the system uses shared or indirect pointers, or several virtual page numbers point to the same physical page, then the table must be invalidated whenever a page is removed from memory or a pointer is removed from a user section table or page map. On the other hand deletion of a page with a unique mapping requires only that a CLRPT be given to invalidate the line containing it. In multiprocessor operation all page tables must be cleared whenever one is. CST entries can be used to communicate paging information from one processor to another.

Previous Context Execute

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. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero *A* bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous context. A PXCT can be given only in executive mode, but the previous context may be the user, as following a call to the Monitor by the user. The previous context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. (Note: it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.)

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory and index register references are made. To work as a PXCT, and XCT must be given in executive mode, and the bits in its *A* field (9–12) must not all be 0 (in user mode *A* is ignored). But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction, all accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a fast memory location.) If the instruction makes no memory operand references, as in a shift or immediate mode instruction, and it has no indexing or indirection (i.e. the instruction word gives *E* directly), then its execution differs in no way from the normal case. The only difference is in memory and index register references.

The previous context is specified by four quantities. Following a call by an MUUO, the section in which the calling program was running (its PC section) and the fast memory block assigned to it appear as the previous context section and current context AC block in the word read by a DATAI PAG,. For the called program, these two quantities can then be assigned as the previous context by a DATAO PAG,. The current AC block of the calling program also appears in the process context word supplied by the MUUO. Various levels of the Monitor may all use fast memory block 0; or a separate block may be assigned to that part of the Monitor that uses PXCTs in handling MUUO calls from other parts of the Monitor.

Just as the current mode is indicated by the User and Public flags, the mode in which the calling program was running is indicated by Previous Context User and Previous Context Public.[36] At a call these flags may be set up automatically or they may be set up by a flag-PC doubleword or a PC word. Note that the restrictions on references made in the previous context are those of the previous context — not those of the context in which the PXCT is given — with the single exception that if the current program is running in section 0, the previous context is also limited to section 0. Suppose the executive executes an instruction that references the concealed user area. Such a reference would fail if Previous Context Public were set.

Which references in the executed instruction are made in the previous context is determined by 1s in the *A* portion of the PXCT instruction word as follows.

Bit *References Made in Previous Context if Bit is 1*

9 Effective address calculation of instruction, including both instruction words in EXTEND (index registers, address words by indirection); also EXTEND effective address calculation of source pointer if bit 11 is 1 and of destination pointer if bit 12 is 1

10 Memory operands specified by *E*, whether fetch or store (e.g. PUSH source, POP or BLT destination); byte pointer; second instruction word in EXTEND

11 Effective address calculation of byte pointer; source in EXTEND; effective address calculation of EXTEND source pointer if bit 9 is 1

[36] Previous Context User and Previous Context Public are in the same flag bits that are used for User In-out and Overflow in user mode. The former has no meaning in executive mode, and the latter is not really necessary as the executive program is not ordinarily interested in performing extensive mathematical procedures.

12 Byte data; stack in PUSH or POP; source in BLT; destination in EXTEND; effective address calculation of EXTEND destination pointer if bit 9 is 1

Previous context referencing is useful and reasonable in some instructions but inapplicable to others. There is no trap of any kind, and the effect of using the feature with an instruction to which it does not apply is simply undefined.

Applicable

Move, XMOVEI
EXCH, BLT, XBLT
Half word, XHLLI
Arithmetic
Boolean
Double move
CAI, CAM
SKIP, AOS, SOS
Logical test
PUSH, POP, ADJSP
Byte
MOVSLJ (extended KL10 only)
MAP

Inapplicable

LUUO, MUUO
AOBJN, AOBJP
JUMP, AOJ, SOJ
JSR, JSP, JSA, JRA, JRST
PUSHJ, POPJ
XCT, PXCT
Shift-rotate
String (except MOVSLJ)
IO

Note that no jumps can use previous context referencing. Even among the instructions to which such referencing is applicable, only a limited number of the sixteen possible bit combinations is useful or meaningful. Doing an effective address calculation in the previous context (selected by bit 9 or 11) makes sense only if the corresponding data access is also in the previous context (as selected by bit 10 or 12 except 11 or 12 in EXTEND). Only these combinations are permitted.

Instructions	*9*	*10*	*11*	*12*	*References in Previous Context*
General	0	1	0	0	Data
	1	1	0	0	*E*, Data
Immediate	1	–	0	0	*E* (no data access)
BLT	0	0	0	1	Source
	0	1	0	0	Destination
	0	1	0	1	Source, destination
	1	1	0	0	*E*, destination
	1	1	0	1	*E*, source, destination
XBLT	0	0	1	0	Source
	0	0	0	1	Destination
	0	0	1	1	Source, destination

Stack	0	0	0	1	Stack
	0	1	0	0	Memory data
	0	1	0	1	Memory data, stack
	1	1	0	0	*E*, memory data
	1	1	0	1	*E*, memory data, stack
Byte	0	0	0	1	Data
	0	0	1	1	Pointer *E*, data
	0	1	1	1	Pointer, pointer *E*, data
	1	1	1	1	*E*, pointer, pointer *E*, data
MOVSLJ (extended KL10 only)	0	0	0	1	Destination
	1	0	0	1	*E* (= *Y*), destination pointer, destination
	0	0	1	0	Source
	1	0	1	0	*E* (= *Y*), source pointer, source
	0	0	1	1	Source, destination
	1	0	1	1	*E* (= *Y*), pointers, source, destination

Execution of a BLT by a PXCT is limited to these three cases:

Where all operations, regardless of context, are in section 0.

Where the previous context fast memory block is being saved in or restored from the current context, provided all addresses are local and thus in the same section. (Remember that regardless of context a local address in the range 0–17 always refers to fast memory. Hence an AC block can never be saved in or restored from the first sixteen storage locations in any section.)

Where all operations are confined to a single section in the previous context, as would be the case when clearing a user page.

In all other circumstances XBLT must be used instead.

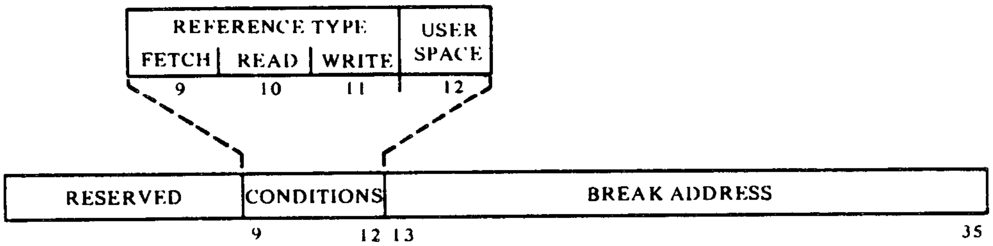
Address Debugging

The address failure, or address break, feature of the pager implements the traditional program debugging technique of catching a particular type of memory reference to a selected location (it does not catch fast memory references). It may be used to determine whether a given program is modifying a particular location, is executing a particular piece of code, or is simply using a particular block of data. This instruction uses the processor device code to specify the circumstances in which a break shall occur.

DATAO APR, Data Out, Arithmetic Processor

70014	*I*	*X*	*Y*
0 12	13	14 17	18 35

Select the break address and the break conditions according to bits 9–35 of location *E* as shown (a 1 in a condition bit selects the condition indicated, a 0 makes no reference selection or selects the opposite address space).



The break conditions selected by 1s in bits 9–12 are as follows.

9 A normal fetch of an instruction in the program under control of PC.

10 Any reference that reads except the normal fetch of an instruction. This includes retrieval of operands, address words in an effective address calculation, or an instruction to be executed by an XCT or user LUUO.

11 Any reference that writes.

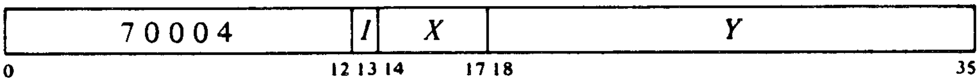
12 A reference made in user virtual address space (0 selects executive space). The break mechanism operates only for virtual address space. It does not catch microcode physical references, such as to the process tables.

Whenever the processor attempts one of the selected types of reference to the location specified by the break address in the selected virtual address space, a page failure results[37] unless the Address Failure Inhibit flag is set. This flag, which is bit 8 of the program flags and can be set only by an instruction that restores them, 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 is saved and cleared if the flags are saved with PC. 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 PC. Using the inhibit flag, the Monitor can return to a user instruction that caused an address failure and "get by it."

Since this feature is entirely under the control of the above IO instruction, it can be used quite flexibly for the executive to debug its own routines, or to debug a single user program without bothering either the executive or other users. The break conditions in effect at any time can be ascertained by giving this instruction.

[37] Executive conditions also catch virtual references in interrupt functions, but the page failure sets the In-out Page Failure flag instead of resulting in a trap for an address failure.

DATAI APR, Data In, Arithmetic Processor

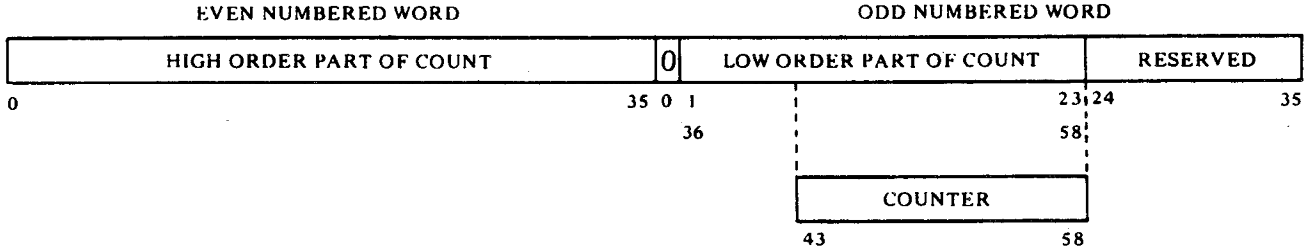


Read the current break conditions into bits 9–12 of location *E*. The information read is the same as that supplied by the last DATAO. (Note that the break address cannot be read.)

3.6 Timing and Accounting

The processor includes a subsystem with elements for keeping track of time, use of system facilities, and use of individual system features. One element is a standard 12-bit interval counter that is set up by the program to interrupt when the count reaches a preset value. The others are meters for keeping a 59-bit count, wherein only the low order sixteen bits are implemented in hardware. In each case the actual counting is done in a 16-bit hardware counter, while the overall count is kept in a doubleword in a process table. A count is updated from its counter by a procedure that is performed periodically by the microcode and whenever appropriate to an operation requested by the software. In the update procedure the contents of a counter are added into the corresponding count and the counter is cleared. Whenever the microcode checks for interrupt requests it updates any count whose counter is more than half full, i.e. whose MSB is 1. The current user accounts are generally updated when the Monitor switches to a new user.

A doubleword count is a 59-bit unsigned quantity whose format and relationship to the hardware counter are as shown here. The entire first word comprises the high order thirty-six bits, and the low order twenty-three are in bits 1–23 of the second word.[38] Reserving bits for expansion at the low order end guarantees format compatibility with future machines that may be much faster (and therefore require bits for counting smaller time units). Altogether there are four meters that use this counter-doubleword format. One is a straightforward time base that counts at 1 MHz. Two keep track of process execution time and number of memory references for purposes for user accounting. Last is a mechanism for analyzing system performance by investigating the use of individual system fea-



[38] Remember, it is a property of twos complement arithmetic that the sign can be used as an extra magnitude bit in an unsigned number. But since the hardware is set up for signed arithmetic, bit 0 of any lower order word must be skipped.

tures, either by counting the number of times particular events occur or measuring the duration of time particular procedures are in progress.

The program controls the various subsystem elements through two sets of IO instructions using device codes 20 and 24, mnemonics TIM and MTR.[39] In general the meter code is for handling the accounting meters and the timer code is for the other elements, but the MTR conditions are for both. Data instructions read updated doubleword counts, but affect neither the counts nor the counters. Condition bits (in a CONO) directly affect only the 16-bit hardware counters. Of course a counter being enabled does mean updating of the doubleword count will probably occur. But to reset a count, the program must not only clear the hardware counter but separately clear the corresponding pair of locations in the process table.

System Timing

For regular system use, the processor provides a time base and an interval counter: The time base is a doubleword count (of the type described above) kept in locations 510 and 511 of the executive process table. It counts elapsed time in microseconds (a rate of 1 MHz). Drift is guaranteed to be less than 5 seconds per day for at least the first six years of use. To maintain day-to-day accuracy, the Monitor can reset the time base once each day from the line frequency clock in the front end processor (although a line frequency clock has quite low resolution, it has very high long-term accuracy.)

The interval counter is a 12-bit hardware counter that counts in 10 us increments (100 kHz). It can therefore count, and signal completion of, any interval from 10 us to 40.95 ms; and it can also be read at any time to determine how long some particular operation or procedure has taken. The counter can be used for any purpose by the software, but it is employed principally to signal the Monitor should a user tie up the system too long. Associated with the counter are two flags, Interval Done and Interval Overflow. Done sets when the counter reaches the value the program specifies as its period or reaches its maximum (all 1s); in either event, the counter clears and starts counting over. Overflow sets only if the counter reaches its maximum without ever matching its period.[40] Setting Done requests an interrupt on the level assigned to the counter, and the processor responds by executing the instruction in location 514 of the executive process table.

WRTIME **Conditions Out, Meters** (CONO MTR,)

7 0 2 6 0	*I*	*X*	*Y*
0	12 13	14 17	18 35

Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 18–26 as shown.

[39] Unassigned instructions using these codes are DATAO TIM,, BLKO MTR, and DATAI MTR,. They execute as MUUOs.

[40] Overflow can occur only if at some time during the count, the program changes the period to a value less than the current counter value.

SET UP ACCOUNTS		ACCOUNTING			TIME BASE				PRIORITY INTERRUPT ASSIGNMENT
		EXECUTIVE PI ACCOUNT	EXECUTIVE NON-PI ACCOUNT	TURN ON	TURN OFF	TURN ON	CLEAR		
18	19 20	21	22	23	24	25	26	27 28 29 30 31 32	33 34 35

Only bits 24–26 and 33–35 are for the system timing features under discussion (time base, interval counter); bits 18–23 are for the accounting meters discussed in a later part of this section.

The interrupt level assignment is solely for the interval counter. Bits 24–26 control the hardware counter for the time base, wherein 1s clear it and turn it on or off (0s have no effect). The result of putting 1s in both bits 24 and 25 is indeterminate.

Bit 18 is a change bit for the accounting setup. If it is 0, bits 21–23 are ignored. But if it is 1, the way in which the meters are enabled is adjusted according to the configuration of those bits, where a 1 produces the indicated function and a 0 has the opposite effect. A 1 in bit 23 turns on the meters, and while on they automatically keep an account of user activity. In addition the meters are enabled during interrupt routines, during noninterrupt executive time, or both (i.e. all executive time) as selected by bits 21 and 22.

Caution: Although this instruction does not write, and *E* is not even an address, the value of *E* must be the address of a writable page.

Notes. The accounting bits affect only the circumstances in which the accounts are kept. Whenever the accounting meters are enabled, they automatically count both execution time and memory references.

CONI MTR, Conditions In, Meters

70264	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the accounting meters and time base, and the interrupt level assigned to the interval counter into the right half of location *E* as shown.

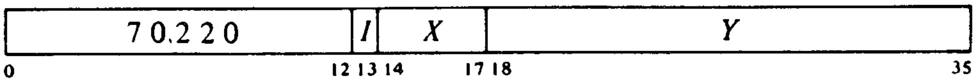
	ACCOUNTING				TIME BASE ON		PRIORITY INTERRUPT ASSIGNMENT
	EXECUTIVE PI ACCOUNT	EXECUTIVE NON-PI ACCOUNT	ON				
18 19 20	21	22	23	24	25	26 27 28 29 30 31 32	33 34 35

RDTIME Read Time Base (DATAI TIM,)

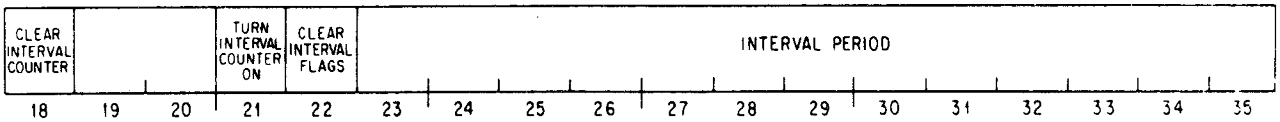
70204	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the time base doubleword count from locations 510 and 511 in the executive process table, add the current contents of the time base hardware counter to the doubleword read, and place the result in location *E*,*E* + 1.

CONO TIM, Conditions Out, Interval Counter

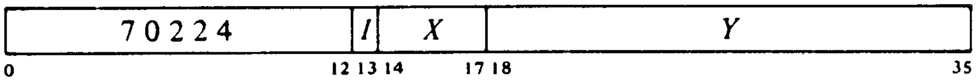


Set up the interval counter according to the effective conditions *E* as shown.

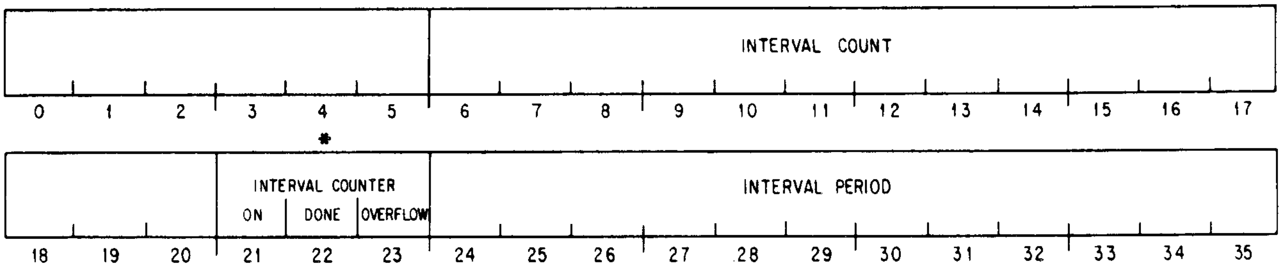


A 1 in bit 18 clears the counter, and can be given simultaneously with a 1 or 0 in bit 21 to turn the counter on or off. A 1 in bit 22 clears both Interval Done and Interval Overflow. If the counter is on, Interval Done will set when the count reaches the value specified by bits 24–35.

CONI TIM, Conditions In, Interval Counter



Read the status of the interval counter into location *E* as shown (the single bit that can cause an interrupt is indicated by an asterisk).



Bits 22 and 23 are the counter flags; note that Done can be set alone, but a 1 in bit 23 implies a 1 in bit 22 as well. Bits 24–35 are the period supplied by the CONO, and bits 6–17 are the current contents of the counter.

User Accounts

Two doubleword counts are kept for every user process. These are under the control of the accounting bits in a CONO MTR, as described above, and they always work together — i.e. the bits that select the circumstances for accounting do so for both of them. When the accounting meters are enabled, the execution meter counts at half the system clock rate while the processor is actually executing instruction operations, in other words except while waiting for memory (note that fast memory references are handled during execution — there is no wait). The memory meter counts memory references by or for instructions, not including fast memory references. Each individual instruction reference is regarded as a single reference even if it requires a page refill, and even if in one case memory control might handle four words whereas in the next three cases the references might be to the cache.

While the accounting meters are on, they are always enabled in user mode, except in certain special procedures discussed at the end of this paragraph. Additional enabling circumstances are selected by bits 21 and 22 of a CONO MTR,. Bit 21 enables while interrupts are actually being held, in other words during the execution of interrupt routines. Bit 22 enables in executive mode except while interrupts are being held. Programming 1s in both bits causes selection throughout executive mode. Note that interrupt routines executed in user mode are always included regardless of the selected circumstances by virtue of their being in user mode. Lastly there are two circumstances that automatically disable the meters regardless of any selection made and whatever mode the processor is in. These are the execution of interrupt functions (PI cycles) (§3.1) and special exempt microcode procedures: updating the meters, handling a page failure, and handling a TOPS–20 page refill.[41]

When a DATAO PAG, assigns a new user base address (§3.5), the accounts for the preceding user are updated in this process table unless such action is inhibited by a 1 in bit 18. The program can read the current user accounts by these two instructions.

RDEACT **Read Execution Account** (DATAI MTR,)

70244	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the process execution time doubleword count from locations 504 and 505 in the user process table, add the current contents of the execution time hardware counter to the doubleword read, and place the result in location $E,E+1$.

RDMACT **Read Memory Account** (BLKI MTR,)

70240	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the memory reference doubleword count from locations 506 and 507 in the user process table, add the current contents of the memory reference hardware counter to the doubleword read, and place the result on location $E,E+1$.

[41] A TOPS–10 page refill is excluded from accounting by virtue of being done by memory control while the execution meter is waiting.

The accounting meters provide an accurate and reproducible measure of the resources used by a given process. Even though one model processor may differ in speed from another, the execution time count should be the same for a given program run on either of them (the unit of time counted will of course be different). Billing of charges to a user can be based on the execution time and the memory reference count taken separately, or a time equivalent can be assigned to a memory reference and the two accounts combined in a single quantity.

Performance Analysis

The performance analysis meter is a tool for studying the performance of the hardware and software of the system. With it, the analysis software can find bottlenecks, such as overuse of a particular system facility. Information of this sort should help the system administrator decide what new equipment to add or how to expand the system, and should help Digital decide how to modify existing software or what new hardware or software to design.

The result of an analysis is a doubleword count kept in locations 512 and 513 of the executive process table. Available to the analyzer is a large set of logic signals representing various conditions in the system. Incrementing of the hardware counter is controlled by a subset of these conditions selected by the program. The conditions are treated as a Boolean expression, and are divided into six groups, each corresponding to a term in the expression. Counting is enabled when the expression is true, which requires that all six terms be true. Within each term the conditions are ored, so a given term is true when any chosen condition in it is true. In each term the program must select some condition, or the term will be false by default. Selection of conditions is by means of the bit configuration of a word supplied to the analyzer. The following table lists the categories of conditions for the terms, the bits in the word that make the selection, and the individual conditions available in each category.

Terms	*Bits*	*Conditions*
Mode	27–28	User, executive, ignore
Memory	12–16	Processor waiting (E box wait), cache miss, writeback for reference (cache writeback), writeback for sweep (sweep write), ignore
Interrupt	18–26	Interrupt on any level 0–7, no interrupt in progress
Channels	0–8	Any channel busy (0–7), ignore
Microcode	9	Microcode enable, ignore
Probe	10–11	Probe high or low, ignore

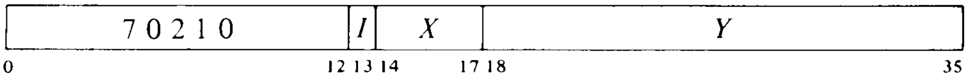
By setting bits 18–26 to select all available interrupt conditions — interrupts on all levels and no interrupt — the program effectively deletes the interrupt term from the expression. In other words it forces the term true so the state of the interrupt system has no effect on whether analysis counting is enabled. All other categories include a specific provision by which the program can force the term true and thus cause the selected conditions in it to be ignored in evaluating the expression. For example the mode choice is made by bit 27: 1 selects user mode, 0 selects executive. But a 1 in bit 28 causes the selection made by bit 27 to be ignored; thus enabling of the analyzer no longer depends on the mode and is purely a function of the conditions selected in other categories.

Besides selecting conditions for analysis, the program also chooses the counting method used by the analyzer. In the duration method the analyzer counts at half the system clock rate while the expression is true. In the event method the counter advances one step each time the expression changes from false to true. Selection of multiple conditions for the duration method produces a composite picture of performance. Suppose we select interrupts on levels 4 and 6 as our interrupt conditions. The analyzer will then give a count of the total time spent handling interrupts on those levels, and the nesting of an interrupt on level 4 within one on level 6 will not affect the result.

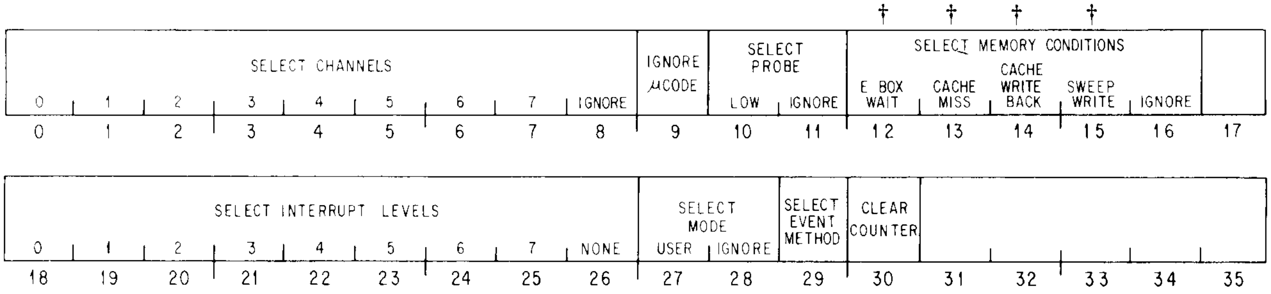
Event counting however can vary considerably depending upon the order in which events occur. If we choose only interrupts on level 6, each return to an interrupt routine at level 6 from some higher level that interrupted it will be counted as separate event; hence a single interrupt on the level of interest may be counted several times. On the other hand selecting interrupts on say levels 2 and 6 may mean that a level 6 interrupt plus half a dozen level 2 interrupts will be seen as only one event. This would happen if all of the level 2 interrupts occurred during the level 6 interrupt routine.

There are two instructions for the performance analyzer: one to set it up and one to read it.

WRPAE Write Performance Analysis Enables (BLKO TIM,)



Select the counting method and conditions for performance analysis according to the contents of location *E* as shown (a dagger indicates a bit in which a 0 makes the selection indicated).



Bit groups corresponding to the terms in the enabling expression and the individual conditions that constitute the groups are as follows.

0–8 Channel conditions. Bits 0–7 select channels 0–7 busy. A channel is busy when it is waiting for a device to respond or a transfer is in progress. A 1 in bit 8 deletes the term from the expression.

9 Microcode condition. A 1 in this bit deletes the term from the expression. If the bit is 0, the counter can run only when specifically enabled by the microcode, which is the standard case.

10–11 Probe conditions. The probe is simply an available input at pin CA1 on the meter board, so the program must generally give a 1 in bit 11 to delete this term from the expression. Should a signal under investigation be connected to the pin, then a 0 in bit 11 enables bit 10 to select the input level that satisfies the condition: 0 high, 1 low.

CAUTION

Connecting a signal line to the probe input may produce ringing in that line, which depending on its length, may seriously degrade signal quality and cause machine malfunction.

12–16 Memory conditions.[42] A 1 in bit 16 deletes this term from the expression. Otherwise 0s (not 1s) in bits 12–15 select enabling conditions as follows.

12 The E box is waiting for the M box in a memory reference. This is only for a reference made by the E box. Its duration may however encompass a writeback to free a cache group entry or a TOPS–10 page refill.

13 Because of an E box reference, the M box is fetching data from storage or filling the cache (a cache miss). This includes only a fetch and load stemming from an E box reference made because the cache does not contain the desired word or is not in use.

14 The M box is writing in storage because of an E box reference. This would usually be a writeback to free a cache entry.

15 The M box is performing a writeback for a cache sweep.

18–26 Interrupt conditions. Bits 18–35 select interrupts on levels 0–7. An interrupt condition includes both the execution of an interrupt function and the subsequent interrupt routine, if any; in other words it includes both PI cycles and an interrupt held for the level. A 1 in bit 26 selects the condition that no interrupt is currently in progress. If bits 18–26 all contain 1s, the interrupt term is always true and thus ignored. Similarly all 0s holds it false.

[42] Note: M box references initiated by the E box include those for instructions, operands, interrupts, and special microcode procedures (meter update, page failure, TOPS–20 page refill). References for writebacks, cache sweeping, TOPS–10 page refills, and the channels are initiated by the M box itself.

27–28 Mode conditions. A 1 or 0 in bit 27 enables the counter during user or executive mode respectively; a 1 in bit 28 deletes this term from the expression.

29 This bit selects the method of counting when the expression corresponding to the set of conditions selected by bits 0–28 is true. A 1 selects the event method wherein there is one count for each time the expression becomes true; and a 0 selects the duration method wherein the counter increments at half the system clock rate while the expression is true.

Notes. There is no specific provision for turning the counter on and off. It functions automatically whenever the selected expression is satisfied, but it can easily be stalled by selecting an impossible combination. In particular, giving a WRPAE [40] clears the counter and disables it.

RDPERF Read Performance Analysis Count (BLKI TIM,)

70200	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the process execution time doubleword count from locations 504 and 505 in the user process table, add the current contents of the execution time hardware counter to the doubleword read, and place the result in location *E*,*E*+1.

Applications. The event method allows software to collect counts of the number of times specific events occur over a period. Examples are calls to the executive, interrupts on a particular level or disjoint interrupts to all levels, cache misses, cache misses in user mode, traffic on the channels. There are also more esoteric analyses, such as counting the number of times a particular instruction or set of instructions is used (this would require modifying the microcode to enable) or how often a particular piece of software is called (this would require a patch in the Monitor). But the event method is subject to the limitations discussed above. A low priority interrupt routine could easily be recognized several times, and with the selection of multiple conditions, events can be lost due to overlap. The memory conditions especially overlap one another, and channel events are very likely to be lost if combined with memory or interrupt conditions.

These limitations do not affect the duration method. Suppose we wish to determine the total time spent doing interrupts and waiting for memory references. Overlap here is of no significance: the fact that sometimes the system is doing both does not matter. Typical uses are measuring the duration spent in user mode, or in executive mode, handling interrupts, handling interrupts at a particular level, doing DTE20 console functions or byte transfers (interrupt level 0), doing writebacks, and so forth. With an enable inserted in the microcode, one could measure the time spent manipulating strings.

3.7 Front End Functions

Every system contains one or more PDP–11 front end processors. But from the point of view of the KL10, a front end is a DTE20 interface — it is only the DTE20 that the KL10 hardware, microcode and program see on the E bus, and it is only the relationship between KL10 and DTE20 that concerns us here (there is nothing in this section about the PDP–11 per se). A DTE20 handles communication between the central processor and a front end processor by way of the KL10 interrupt system. The program can assign a level for standard or vector interrupts, but the interface can also perform special interrupt functions — examine, deposit, byte transfer — on level 0. In general all but one of the DTE20s are restricted: this means that a unit can request special interrupt functions only if interrupt level 0 is enabled in it, and examine and deposit are restricted to communication areas defined by the Monitor.

Among the DTE20s, one is master and is thus unrestricted. It gains this privileged status by means of a switch setting on the unit. The master can perform diagnostic operations[43] (included among these are the console functions start, stop, execute, and continue), can perform the special interrupt functions even when level 0 is disabled, and can override the restrictions on examine and deposit so as to gain access to all PDP–10 memory in either physical or executive virtual address space or the executive process table. Removal of the restrictions by placing a 0 in the *Q* bit of the interrupt function word must be done individually for each transfer.

For each DTE20 the executive process table contains an 8-word control block. These blocks contain the following information for byte transfer, vector, examine and deposit interrupt functions.

Locations in Executive Process Table				
Unit 0	*Unit 1*	*Unit 2*	*Unit 3*	*Contents*
140	150	160	170	Output byte pointer (to 11)
141	151	161	171	Input byte pointer (to 10)
142	152	162	172	Vector interrupt instruction
143	153	163	173	Reserved
144	154	164	174	Size of communication area for examine
145	155	165	175	Relocation address for examine area
146	156	166	176	Size of communication area for deposit
147	157	167	177	Relocation address for deposit area

A byte pointer is limited to a single word; it must therefore have a 0 in bit 12, and its address is interpreted in executive virtual address space, section 0. The programmer must also refrain from using any indexing or indirection (bits 13–17 must be zero). After the microcode increments the byte pointer selected by *Q* (0 out, 1 in) and calculates its effective address,

[43] Except for stopping, diagnostic operations should be performed only when the processor is halted or something has actually gone wrong. Otherwise they would interfere with normal traffic on the E bus.

an input byte is inserted at the appropriate position in a memory location, or an output byte from memory is sent to the DTE20 right-justified with the rest of the output word filled with 0s. An output byte transfer is essentially an ILDB-DATAO combination; input is a DATAI-IDPB. Output bytes larger than sixteen bits can produce spurious E bus parity errors in the DTE20.

In a DTE20 vector interrupt, the address part of the function word is ignored, and the microcode executes the instruction supplied by the control block. This should be a call to an interrupt routine.

Communication areas are defined separately for examine and deposit. Thus the Monitor might divide the overall communication area into separate parts for deposits by several units, but allow all of them to examine the entire area. The size of an area is given as a number of locations, and the relocation address is the physical address of the first location in the area. Suppose we wish to assign a deposit area of sixteen words beginning at location 22660 for DTE20 No. 2. In locations 166 and 167 of the executive process table we would put respectively 20 and 22660. In its deposit function words the DTE20 would then use addresses 0–17, and these would be relocated to 22660–22677.

3.8 Error and Diagnostic Instructions

The first part of this section explains the instructions through which the software handles the error flags and identifies the source of a hardware error. The second part discusses a special instruction the Monitor uses to set up the memory system and to get diagnostic and configuration information directly from individual memory controllers. The objective of this treatment is to complete the definition of all KL10 instructions and to give the programmer what he needs to identify sources of hardware error for purposes of software recovery. For information on diagnosing equipment ills, the reader must turn to maintenance documents. Note that this section does not touch on diagnostic functions the front end can execute in the KL10 without the KL10 microcode running; that subject is treated in the maintenance documentation.

Error Monitoring and Investigation

A few hardware errors — specifically a parity error in the page table or in a word brought into AR or ARX from memory — are detected by the pager and produce a page failure. Other hardware errors detected in the processor or on the S bus are indicated by flags that can request an interrupt on a level assigned to the processor. Several of these flags also lock information about the bad reference into the error address register ERA. The program can read this register, and it continues to hold the same information, even should subsequent errors occur, until the flag that locked it is cleared.

The error conditions are generally regarded as important enough to be assigned to the highest priority level. However for conditions that may be associated with user instructions (a parity error or unanswered memory reference), the common practice is for the error interrupt to switch over to the lowest priority level by means of a program-set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

Error flags are handled by two condition IO instructions that address the processor, which has device code 000, mnemonic APR.[44] These instructions also handle the sweep flags for the cache (§3.2). The instruction that reads ERA uses the interrupt device code.

CONO APR, Conditions Out, Processor Flags

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

Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 19–31 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

		ENABLE	DISABLE	CLEAR	SET	SELECT FLAGS FOR BITS 20-23									PRIORITY INTERRUPT ASSIGNMENT		
	CLEAR ALL IN-OUT DEVICES	SELECTED FLAGS				S BUS ERROR	NO MEMORY	IN-OUT PAGE FAILURE	MB PARITY	CACHE DIRCTRY	ADDRESS PARITY	POWER FAILURE	SWEEP DONE				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

A 1 in bit 19 generates the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects none of the internal devices, such as the pager or the processor flags).

Bits 20–23 select flag functions: 1s in these bits produce the indicated effects on the processor flags selected by 1s in bits 24–31. A 1 in bit 20 enables the setting of any selected flag to request an interrupt on the level assigned to the processor; a 1 in bit 21 disables the selected flags from requesting interrupts. Similarly a 1 in bit 22 or 23 clears or sets the selected flags. The result of putting 1s in both bits 20 and 21 or 22 and 23 is indeterminate.

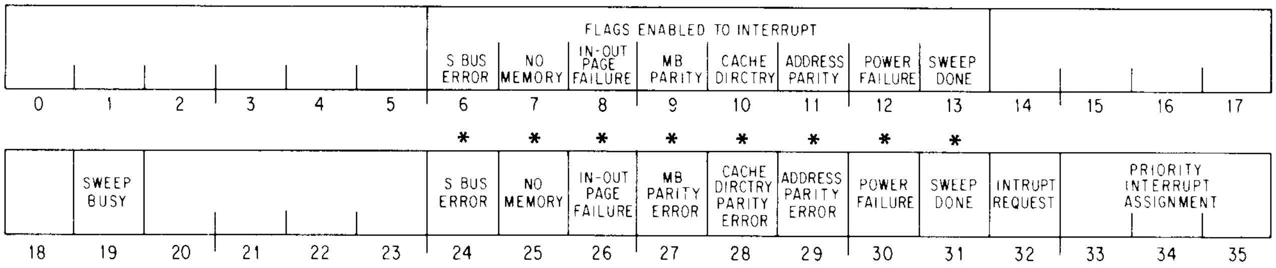
Notes. Setting flags has of course no relation to what the flags represent; the function is used only to check out the flag logic.

CONI APR, Conditions In, Processor Flags

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

Read the status of the processor error and sweep flags into location *E* as shown (asterisks indicate bits that can cause interrupts).

[44] The processor device code is also used in several instructions for the pager and the cache.



6–13 A 1 in any of these bits indicates that setting the listed flag will request an interrupt on the level assigned to the processor by bits 33–35 of the CONO.

19 The cache is currently undergoing a sweep.

24 A storage controller has signaled the processor that it has detected an error in its own operation or in information it has received over the S bus or from one of its storage modules. If the type of error is not identified by there also being a 1 in bit 25, 27 or 29, then the condition is either an incomplete cycle or a parity error in data sent to the memory (all data received by memory is written, even if bad). Controller flags for some of these conditions can be read by the diagnostic instruction discussed in the second part of this section.

25 The processor attempted to access a memory that did not respond within a preset time. This time is 68 μs on an extended KL10, 82 μs on a single-section KL10. The setting of this flag locks information about the attempted reference into ERA. Since a nonexistent memory supplies zero data, on read this error should be accompanied by a 1 in bit 27.

26 A page failure has occurred in an interrupt instruction, or a word with even parity has been received at AR from the E bus (the latter can be recognized only if the transmitting device generates a parity bit). An interrupt failure caused by an address break sets this flag instead of producing an address failure (§3.5).

NOTE

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

27 The buffer (MB) in memory control has received a word with even parity. The setting of this flag locks information about the reference into ERA.

28 A physical page number with even parity has been encountered in the cache directory. The setting of this bit turns off the cache, and it remains off until the flag is cleared by giving a CONO APR, with 1s in bits 22 and 28.

29 A storage controller has signaled that it has received an address with even parity from the processor. The parity check actually encompasses both the address and the control signals that accompany it on the S bus. The setting of this bit locks information about the attempted reference into ERA.

30 Ac power has failed. The program should save PC, the flags, mode information and fast memory in storage, update the accounting meters, validate the entire cache, and halt the processor. Note that PC may point to an interrupt routine rather than the main program. After power is restored the front end must reboot the system, and the Monitor must reestablish the operating environment (§3.5).

31 A cache sweep has been completed.

32 Some processor flag is currently requesting an interrupt, i.e. some flag in bits 24–31 is set and has been enabled to interrupt as indicated by a 1 in the corresponding position in bits 6–13.

RDERA Read Error Address Register (BLKI PI,)

70040	*I*	*X*	*Y*
0 … 12	13	14 … 17	18 … 35

Read the contents of the error address register into location *E*. If No Memory, MB Parity Error or Address Parity Error is set, ERA contains information about the reference corresponding to the first of those flags to be set as shown.

WORD NUMBER	REFERENCE IDENTIFICATION					INDETERMINATE	0	0	0	0	0	HIGH ORDER ADDRESS BITS
	SWEEP	CHANNEL	DATA	SOURCE	WRITE							
0 1	2	3	4	5	6	7 8	9	10	11	12	13	14 15 16 17

PHYSICAL ADDRESS OF FIRST WORD OF TRANSFER
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

Bits 0–1 and 14–35 identify the physical location of the reference in which the error occurred. Bits 14–35 are the address of the specific memory reference made by the program or whatever. If the reference required only a single transfer, that address is the error address. But if the reference triggered a group transfer, bits 14–35 are the address of the first reference chronologically in the group, and bits 0 and 1 give the number of the word on which the error actually occurred. Note that word numbers are in physical, not chronological, order.

Information given in bits 2–6 identifies the reference. A 1 in bit 2 or 3 respectively means the reference was made for a cache sweep or a channel transfer. Bit 6 indicates the memory function being performed for the reference, where the read and write parts of a read-pause-write are separately indicated by 0 and 1. Bits 4, 5 and 6 together identify the source of the data for the transfer or attempted transfer (on write the word is always going to storage).

Bits 4–5	*Source with 0 in bit 6*	*Source with 1 in bit 6*
00	Storage for any read or read-pause-write	Channel status
01		Channel data
10		AR
11	Cache for channel read or TOPS–10 page refill	Cache writeback

ERA retains the same information until the program clears the locking flags by giving a CONO APR,2260*P*. Of course only flags that are set actually need be cleared, and the routine that responds to errors should consider and clear all set flags. To facilitate diagnosis from the front end, the master reset does not clear ERA. Hence if need be, the front end can give diagnostic functions that reset the KL10 and then read ERA.

The processor includes provision for forcing bad parity to check the error detection logic. Bits 18–20 of a CONO PI, (§3.1) respectively cause even parity to be generated for an address sent to memory, a data word available from AR, and a page number entered into the cache directory. Where the data error shows up depends on where the word is sent from AR. Which errors are being forced can be seen by checking the flags in the same bits of a CONI PI,.

Programming Cautions. When handling parity error or nonexistent memory interrupts, the programmer should beware of the following.

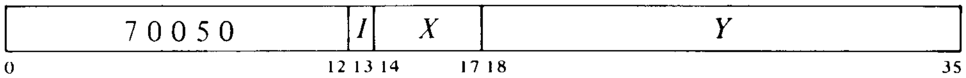
- An incorrect word from memory to AR or ARX can result in both a page failure and an interrupt. In general the page fail trap to the Monitor can be expected to occur slightly ahead of the interrupt.
- Should an error flag be set while another interrupt request is being processed, the system would handle the lower priority interrupt before getting to the processor interrupt. This means PC may be pointing to a lower level interrupt routine rather than the program level at which the error occurred. Remember that during request processing, the interrupt system is otherwise static and the program continues.
- Even without inadvertent interference from another level, it is quite likely the processor will perform one or perhaps two more instructions between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may well be pointing to the first or second instruction following the one in which the error occurred.
- A processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second processor interrupt before the error-handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another

— consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what went wrong.

S Bus Diagnostic Cycle

Ordinarily the S bus is used for the processor to reference memory. But the S bus also has a diagnostic cycle that allows the processor to communicate with the memory controllers rather than to access a particular location. The diagnostic cycle is initiated by the processor giving a special instruction that sends a function word to a controller and receives a word of error and diagnostic information back from it.

SBDIAG **S Bus Diagnostic Function** (BLKO PI,)



Send the contents of location *E* as a function word over the S bus to the controller specified by bits 0–4, and read the return word for the function from that controller into location *E*+1. Which function a word represents is indicated by its code in bits 31–35.

Chapter 4
KS10 System Operations

The information presented in this chapter is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own IO, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

WARNING

> KS10 functions are implemented in microcode, which can be changed much more easily than hardware. Although user operations are deliberately kept as compatible as possible from one machine to the next, Digital will change the KS10 system microcode whenever such change will result in greater speed, efficiency or effectiveness. Therefore anyone writing system software should make sure to use the most recently updated version of this documentation, and before embarking on any project as enormous and critical as an operating system, to check with Large Systems Engineering for any changes not yet documented.

Programming for the system as a whole is programming in executive mode. Only the executive program is without instruction restrictions, and only it can, if needed, access physical memory unpaged. The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working sets, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of a user. The executive program accomplishes these objectives by handling all in-out for the system, setting up page maps, trap locations, interrupt locations and the like for both itself and the users, handling user accounts, and so forth. In other words, except for handling in-out, the activities of an operating system are the topics covered in this chapter. Of course the system programmer must also be quite familiar with all of the material presented in Chapters 1 and 2. In particular he must fully understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the JRST instruction and MUUOs (§§2.9, 2.16).

System information for other processors is given in Chapters 3 and 5. The present chapter is devoted solely to the KS10, but contains two sections on paging, only one of which is applicable to a given system. §4.3 describes the paging used with the TOPS–10 Monitor; this paging is similar to that of the KI10. §4.4 treats the paging associated with the TOPS–20 Monitor. Both kinds of paging employ the same hardware — the difference lies in the microcode. All instructions discussed in this chapter are for system operations and are thus subject to the same restrictions as IO instructions: namely, they can be performed only when the processor is in executive mode or is in user mode with User In-out set.

Some of the material presented here is related to the Unibus adapters. The chapter describes only the activities of the microcode undertaken for the adapters; it does not describe the adapters themselves or their programming.

4.1 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, i.e. the use of interruptions in the current program sequence facilitates concurrent operatiᵒn of the main program and a number of peripheral devices through the Unibus adapters. The hardware also allows system flags (representing the console and conditions internal to the processor) to signal the program by requesting an interrupt. To avoid confusion with Unibus peripheral devices, let us regard the entities with which the interrupt system deals as "units". The system flags together constitute a unit.

Interrupt requests are handled through seven levels arranged in a priority chain, with assignment of units to levels entirely at the discretion of the programmer. To assign a unit to a level, the program sends the number of the level to the unit control register as part of its operating conditions. Levels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the unit from the interrupt levels altogether. Any number of units can be connected to a single level, and an adapter can be connected to two levels.

When a unit requires service it sends an interrupt request signal over the request line corresponding to its assigned level in the processor. The processor recognizes the request if the level is active (on). The request signal remains on the line until turned off by an appropriate response from the processor, either given by the program or generated automatically by the hardware. Thus if a request is not recognized or accepted when made, it will be when the appropriate conditions are satisfied. A single level 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.

In a Unibus system the IO devices receive and send information via the adapter, and they signal the adapter to indicate their needs. To transfer data for high speed devices, the adapter can make direct access to memory over the KS10 bus. But to transfer data for slower devices and to handle control situations for all devices, the adapter uses the KS10 interrupt. For individual devices to signal the adapter, the Unibus has its own interrupt system of four levels, BR4–BR7, with the last having highest priority. Requests for interrupts on BR6 and BR7 are translated into requests on the KS10 interrupt level specified by the so-called "high" assignment, and those on BR4 and BR5 are translated into KS10 requests on the "low" level. Of course complete control over the adapter and the Unibus devices, including assignment of levels for KS10 and Unibus interrupts, is entirely in the hands of the KS10 program.

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 level only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device. For example, 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.

Having recognized a request, the processor will do nothing further with it unless the priority interrupt system is on. But even with the system off, the processor will continue to recognize requests on other levels; and when the system is finally turned on, it will respond as though all requests had just been recognized, handling the highest priority one first.

Processing an Interrupt

The processor handles only one request at a time. When it is ready, it accepts the highest priority request currently recognized, provided that request is on a level higher than the current program (all levels are higher than a noninterrupt program). To process a request the microcode stops the program, turns off the interrupt system to prevent interference from other requests, and executes a "who are you?" cycle on the KS10 bus to determine which adapters are currently requesting interrupts on the accepted level. Note that at this point the processor is accepting not an individual request, but rather a class of requests: namely all those being made on the same level. In this cycle the microcode sends out the number of the level, and the individual adapters 0–3 indicate whether they are requesting interrupts on that level by placing 1s on bus lines 18–21 respectively. (Hence only lines 19 and 21 are used, for adapters 1 and 3.)

If no adapter responds, the request is assumed to be internal, originating either from the system flags or the program itself. In this case the microcode starts the interrupt by executing the instruction at location 40 + 2*N* in the executive process table, where *N* is the level number. Level 1 uses location 42, level 2 uses 44, and so on to level 7 which uses 56.

If the response on lines 18–21 is nonzero, the processor gives priority to the lowest-numbered adapter that has a request on the accepted level[1] by sending out the number of that adapter[2] in a vector request cycle on the bus. The vector address returned from a device is divided by 4, and the result[3] is used as an index into a table of interrupt instructions for that adapter. The table address is taken from executive process table location 100 + *N*, where *N* is the adapter number (i.e. locations 101 and 103 are used). The processor then starts the interrupt by executing the instruction contained in the location specified by the table address plus the vector address divided by 4. The table pointer must be nonzero — otherwise an illegal interrupt halt occurs (§4.7).

Interrupt Instructions. An interrupt instruction is one executed in the interrupt location for a level, in direct response by the hardware (rather than by the program) to a request on that level. An interrupt location is either executive process table location 40 + 2*N* specifically for level *N*; or the adapter table location derived from the interrupt vector and the table pointer corresponding to the adapter having priority among those on the accepted level. Only two instructions can be used as interrupt instructions: JSR and XPCW. For either, the processor holds an interrupt on the level, turns the interrupt system back on, and takes the next instruction from the location specified by the jump (as indicated by the newly changed PC). For a JSR the processor automatically enters executive mode. For an XPCW it enters the mode specified by the new flag word. Either instruction is a jump to a service routine handled by the Monitor. Use of any other instruction results in an illegal interrupt instruction halt (§4.7).

The most important point of which the programmer must be aware is that even while User is set, the interrupt instructions are not part of the user program. They are executed in executive mode and are therefore subject only to executive restrictions. As an interrupt instruction, JSR automatically clears User to jump to an executive service routine. An XPCW should be set up to produce the same result.

Interrupt Programming

The program can control the priority interrupt system by means of these two instructions.

[1] There are therefore two orders of priority associated with an interrupt: first the level, and then for all adapters requesting interrupts simultaneously on the same level, adapter number.

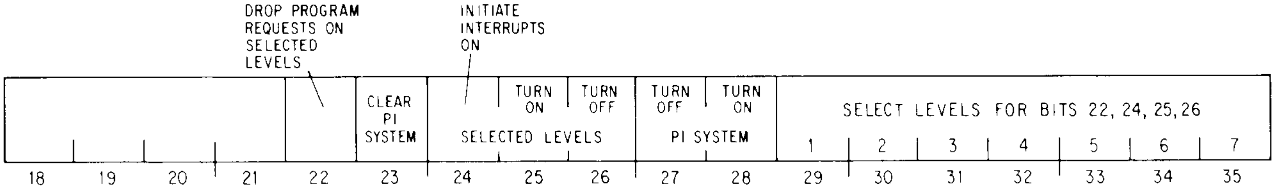
[2] Note that these are the adapter numbers (1 and 3), *not* the controller numbers used in IO addresses (0 and 1).

[3] A vector address is a multiple of 4 because it specifies a pair of word locations in the byte-oriented Unibus addressing scheme.

WRPI Write Priority Interrupt Conditions

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).



22 On levels selected by 1s in bits 29–35, turn off any interrupt requests made previously by the program (via bit 24).

23 Turn off the priority interrupt system, turn off all levels, drop all program-set requests, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by 1s in bits 29–35, and force the processor to recognize them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a WRPI that selects the same channel and has a 1 in bit 22.

When this bit forces recognition of a request on the highest priority level, at most one additional program instruction may be performed before the interrupt.

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

26 Turn off the levels by 1s in bits 29–35, so interrupt requests cannot be recognized on them unless made by a WRPI with a 1 in bit 24.

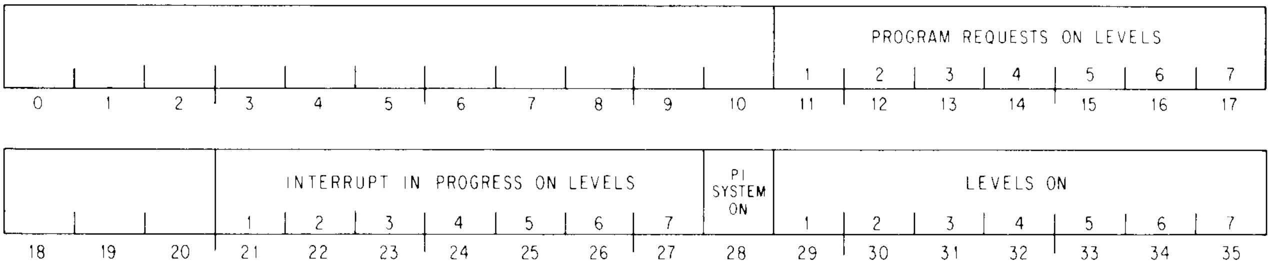
27 Turn off the interrupt system so no requests can be accepted.

28 Turn on the interrupt system so the hardware can process requests.

RDPI Read Priority Interrupt Status

70064	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the priority interrupt into location *E* as shown.



Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; and 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a WRPI with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on, and 1s in bits 29–35 therefore indicate active levels.

Dismissing an Interrupt. The processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt request can be accepted on that level or any of lower priority.

A routine dismisses the interrupt by using an instruction that restores the level on which the interrupt is being held at the same time it returns to the interrupted program. The proper instruction is XJEN (JRST 7,) or JEN (JRST 12,). Once the level is restored, the hardware can again accept requests and start interrupts on it and lower priority levels. These instructions also restore the flags: XJEN from the flag-PC doubleword if the routine was called by an XPCW; JEN from the left half of the PC word if the routine was called by a JSR.

CAUTION

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

Timing. The maximum time a device may wait for an interrupt to start depends on how many active devices are of higher priority and how long their service routines are. When a given request is of highest priority, its device need never wait longer than 40 μs.

Special Considerations. When an interrupt occurs, PC points to the interrupted instruction (or to an XCT that executed it), unless the interrupt occurred in an overflow trap instruction, in which case PC points to the instruction that overflowed. After taking care of the interrupt, the processor can always return to the interrupted instruction. Either *a*) the instruction did not change anything; *b*) the interrupt 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; or *c*) the interrupt occurred at some point in a multipart instruction where the microcode rigged the various pointers and other quantities so the processor actually restarts the instruction where it stopped, rather than from the beginning. However, in a BLT and in byte manipulation, the very mechanism that facilitates the return results in special properties of which the programmer must be aware.

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 flag 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 IDLB or IDPB would skip a byte. And if the routine restored the flag, the interrupted IDLB 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 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 request can be accepted, not even on higher priority levels, while a request is being processed or an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- To prevent a device from hanging up a level, the programmer must be aware of — and satisfy — whatever requirements the device has for dropping the request.
- The interrupt instruction that calls the routine must be an XPCW or a JSR.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. Computations and any other time-consuming activities that can possibly be performed outside the routine should not be included within it.
- Never turn off the interrupt system in a routine unless it is absolutely necessary, and then always turn it back on again as soon as possible. If one or more levels can be turned off in place of the entire system, always do that instead.
- If the routine uses a UUO it must first save the contents of the locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type (§2.16).

- The routine must dismiss the interrupt (with an XJEN or JEN) when returning to the interrupted program. Flags and UUO locations should be restored.

4.2 Cache

For the user, the cache is transparent: any program simply gets information from memory and stores information in memory. But use of a cache as part of the memory subsystem reduces program time, since the cache is faster than the storage modules, and also reduces storage use by the program, making a larger percentage of total storage cycles available to other parts of the system. The cache is essentially 512 registers that duplicate the contents of frequently referenced storage locations in the virtual address space. Its only use is for reading information from it instead of taking the time to go to storage, but this can result in a considerable saving for the program.

Each register in the cache corresponds to a unique position within a page. Associated with the cache is a directory that labels each register by the virtual page containing the word that the register duplicates. A directory entry also has a parity bit and other bits that identify certain characteristics of the reference that caused the word to be written in the cache. A cache hit can occur only when the circumstances of a read reference for a particular location match those of the last time the location was written. These requirements are a virtual reference[4] to the same page in the same address space (user or executive). Given a match, it is also required that paging be enabled by the Monitor, that the page map indicate the individual page is cacheable, and that the directory entry have correct parity. Moreover the cache can be disabled altogether from the console, and the microcode can inhibit its use in individual references.

There is no real programming for the cache except that the Monitor must decide, and so indicate in the page map, which pages are cacheable and which are not. Obviously the contents of the cache must be invalidated whenever there is any significant change in the virtual address environment, but the microcode handles this automatically. A sweep of the entire cache takes about 80 μs.

[4] The cache is also written on a physical reference, but the word cannot later be used as the directory entry is invalid (i.e. not virtual).

4.3 TOPS–10 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.3. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses the TOPS–10 Monitor. The next section presents equivalent information for the TOPS–20 Monitor. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–10 or TOPS–20, and are described in §4.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt instructions reference executive virtual address space. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the microcode to carry out the mapping procedure, retrieve interrupt instructions, and handle traps, halts and UUOs.

Paging

All of memory both virtual and physical is divided into pages of 512 words each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18-bit addresses, where the left nine bits (18–26) specify the page number and the right nine (27–35) the location within the page. Physical memory can contain 1024 pages and requires 19-bit addresses, where the left ten bits (17–26) specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18-bit addresses into 19-bit addresses.[5] 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 10-bit physical page number for the 9-bit virtual page number. The mapping procedure is carried out automatically by the pager, but the page map that supplies the necessary substitutions is set up by the executive 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.

Two locations in the register file are used by the Monitor to specify the physical page numbers of the user and executive process tables. To retrieve a map word from a process table, the pager uses the appropriate base page

[5] For paging purposes 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 storage module locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to storage.

number as the left ten bits of the physical address and some function of the virtual page number as the right nine bits. For example, 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).

The executive virtual address space is also 256K, but the page map for it is in three parts. The map for the first 112K (pages 0–337) is in executive process table locations 600–757. 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. This means the Monitor can assign a different set of thirty-two physical pages (the per-process area) for its own use relative to each user. Hence when switching from one user to another, the Monitor need change only the user process table, this single substitution making whatever change is necessary in the executive address space for a particular user.

Figures 4.1 and 4.2 show the organization of the virtual address spaces, the process tables and the maps for both user and executive. The first illustration gives the correspondence between the various parts of the address spaces and the corresponding parts of the page maps. The second illustration lists the detailed configuration of the process tables as determined by the hardware. Any table locations not used are reserved for future use by the hardware or for use 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.[6] The Monitor also specifies whether each page is writable or not and cacheable or not. Each word in the page map has this format to supply the necessary information for two virtual pages.

DATA FOR EVEN VIRTUAL PAGE							DATA FOR ODD VIRTUAL PAGE						
A	*P*	*W*	*S*	*C*		PHYSICAL PAGE ADDRESS BITS 17–26	*A*	*P*	*W*	*S*	*C*		PHYSICAL PAGE ADDRESS BITS 17–26
0	1	2	3	4		8 17	18	19	20	21	22		26 35

[6] 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.

Bits 8–17 and 26–35 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 "page use" bits are as follows.

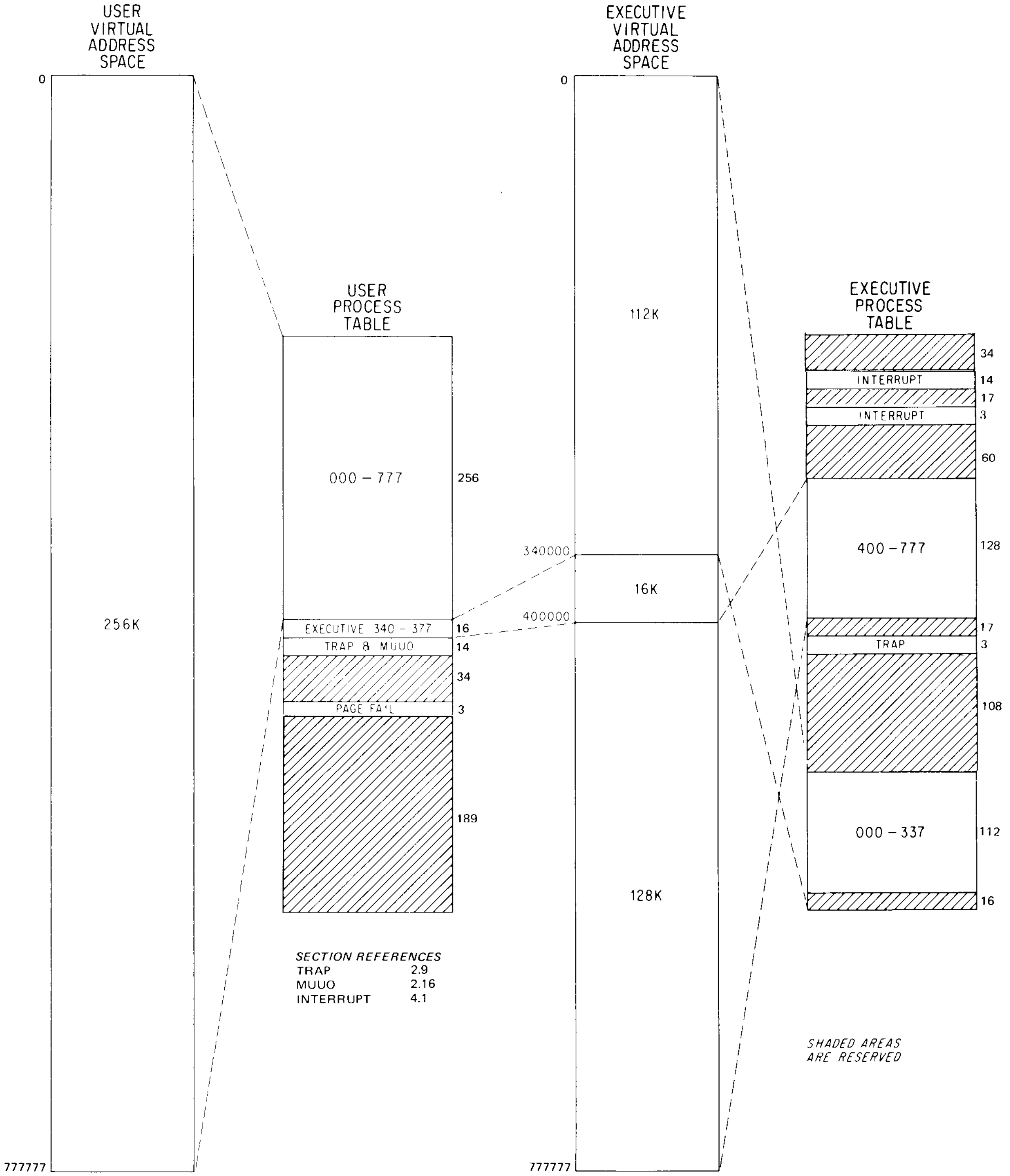
Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Not used (public in other processors)
W	Writable (not write-protected)
S	Software (not interpreted by the hardware)
C	Cacheable

Page Table. 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 pager contains a page table, in which it keeps a large assortment of mappings for both the executive and the current user. The table has 512 locations, one for each virtual page number. Each location contains a mapping (from a map half word) for the virtual page that identifies it, including the physical page number and the *W* and *C* bits. Each location also has a parity bit, a bit that indicates whether the mapping is for user or executive address space, and a bit that indicates whether the entry is valid. A zero mapping is perfectly valid, but a location is labeled as containing no valid mapping by clearing it, thus clearing the valid bit. It is not necessary to keep the access bit, as mappings for inaccessible pages are not entered into the table.

When the program references a page whose mapping entry is tagged as valid and in the program address space, the 10-bit physical number[7] from the mapping for the virtual page is used as the left ten bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *W* bit). If however the entry is invalid or is not for the correct address space, or the reference is for writing and *W* is 0, the pager makes a separate memory reference (referred to as a "page refill") to get the mapping for the specified virtual page from the page map. The mapping is placed in the table unless the reference fails because the page is inaccessible or the program is attempting to write in a protected page.

[7] Actually table locations have eleven bits for physical numbers, but the most significant is not used.

Figure 4.1: TOPS–10 Virtual Address Space and Process Table Layout



SECTION REFERENCES

TRAP	2.9
MUUO	2.16
INTERRUPT	4.1

SHADED AREAS ARE RESERVED

Figure 4.2: TOPS–10 Process Table Configuration

USER PROCESS TABLE

0	USER PAGE 0	USER PAGE 1
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	RESERVED	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	MUUO OLD PC WORD	
426	MUUO PROCESS CONTEXT WORD	
427	RESERVED	
430	EXECUTIVE NO TRAP MUUO NEW PC WORD	
431	EXECUTIVE TRAP MUUO NEW PC WORD	
432	RESERVED	
433	RESERVED	
434	USER NO TRAP MUUO NEW PC WORD	
435	USER TRAP MUUO NEW PC WORD	
436	RESERVED	
477		
500	PAGE FAIL WORD	
501	PAGE FAIL OLD PC WORD	
502	PAGE FAIL NEW PC WORD	
503	RESERVED	
777		

EXECUTIVE PROCESS TABLE

0	RESERVED	
41		
42	PRIORITY INTERRUPT INSTRUCTIONS	
57		
60	RESERVED	
100		
101	ADAPTER 1 INTERRUPT TABLE POINTER	
102	RESERVED	
103	ADAPTER 3 INTERRUPT TABLE POINTER	
104	RESERVED	
177		
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400	RESERVED	
420		
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424	RESERVED	
577		
600	EXECUTIVE PAGE 0	EXECUTIVE PAGE 1
757	EXECUTIVE PAGE 336	EXECUTIVE PAGE 337
760	RESERVED	
777		

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the pager terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap. The trap operation[8] makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current PC word in location 501, and sets up the flags and PC according to a new PC word in location 502. The processor then resumes operation in the new state at the location now addressed by PC. The same sequence of events occurs if the processor performs an IO instruction and the adapter fails to indicate the transfer was accomplished.

There are two kinds of page failures, hard and soft. A hard failure means that something really is amiss, whereas a soft failure generally means only that the program requires some kind of service from the Monitor. A hard failure is indicated by a 1 in bit 1 of the page fail word, and the particular failure is specified by a code (which is therefore ⩾ 20) in bits 1–5. There are three such failures of which two are true page failures, i.e. failures involving memory reference, and for these the page fail word has this format.

U	36 OR 37	0 0	*P*	000		ADDRESS
0	1 5		8		17	18 35

Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a physical address was given for the reference. The code names the particular failure as follows.

36 Uncorrectable memory error — in a processor reference the memory controller has read an incorrect word from storage and was unable to correct it. The processor has saved the word in AC 0 and AC 1, block 7, and has set the Bad Memory flag (RDAPR bit 28).

37 Nonexistent memory — the processor has called for a storage reference over the bus but the memory controller did not respond. This error also sets the No Memory flag (RDAPR bit 27).

If the failure code is 20, the fail word instead has this format

U	20	0 0	1	0	1	0 0	*B*	IO ADDRESS
0	1 5		8		10		13	14 35

and indicates a nonexistent IO register, i.e. an IO instruction gave an IO address to which there was no response. A 1 in bit 13 indicates a byte operation. (The 1s in bits 8 and 10 mean a physical reference and an IO function on the bus.) Note that this is not an IO page failure, which is a true (memory) page failure and causes a halt.

[8] A page failure that occurs during an interrupt instruction does not act this way. Instead the processor halts (§4.7).

A soft failure — of which there are two, an inaccessible page and an attempt to write in a write-protected page — is indicated by a 0 in bit 1. The fail word still contains the *U* bit and the virtual address, but now bits 1–8 have one of these formats,

INACCESSIBLE	1	2	3	4	5	6	7	8
	0	0	0	0	*T*	0	0	1

WRITE VIOLATION	1	2	3	4	5	6	7	8
	0	1	0	*S*	*T*	0	0	1

where *S* is simply the software bit taken from the mapping for the page specified by bits 18–26, bit 8 is the inverse of bit 8 in the hard case (1 means virtual), and *T* indicates the type of reference in which the failure occurred: 0 for a read-only reference, 1 for any reference involving writing. It is evident from inspection of the two configurations that bit 2 is actually the *A* bit from the mapping; and when the page is accessible, the 0 in bit 3 comes from the *W* bit. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course *T* and *A* both being 1 implies a write failure.

For a page fail trap, the new PC word is set up by the Monitor to transfer control to executive mode. After rectifying the situation, the Monitor returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two-part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §4.1.

Note that a soft failure seldom implies that anything is "wrong" — unless a program has attempted to write in a truly write-protected area. Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in core; 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 its mapping as inaccessible), the Monitor would respond to the page failure by bringing in the needed page from the disk, either adding to the user space or swapping out a page the user no longer needs.

The same situation exists for writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered, distinguishing those that must be revised on the disk at the end from those that can be thrown away by setting the software bit. Then in response to a write failure, the Monitor makes the page writable and sets the software bit to indicate to itself that that page has in fact been altered and must be saved. When the user is done, the Monitor need write back onto the disk only those pages for which both *W* and *S* are set.

The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory. For such purposes the processor has this instruction.

MAP **Map an Address**

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

If the pager is on, map the page number of the virtual effective address *E* and place the resulting physical address and other map data in AC. If the page is accessible, the information loaded into AC is of the form

U	0	1	*W*	*S*	0	0	*C*	1	000	PHYSICAL ADDRESS
0	1	2	3	4	5	6	7	8	9 16	17 35

where bits 17–26 are the physical page number the pager supplies for *E*, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and *W*, *S* and *C* are the page use bits from the mapping as explained above (the 1 in bit 2 represents *A*). If the page is inaccessible, AC receives the given virtual address in place of a physical address; the word also includes *U* and a 1 in bit 8, but the remaining bits are all zero.

However, should a memory error occur during access to the page map, AC receives a hard page fail word. If the pager is off, the result is undefined.

Notes. The instruction cannot actually fail, because regardless of what happens, the page fail microcode returns to it instead of trapping to the Monitor. The effective address calculation done for it could fail however.

4.4 TOPS–20 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.3. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses the TOPS–20 Monitor.[9] The previous section presents equivalent information for the TOPS–10 Monitor. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–20 or TOPS–10, and are described in §4.5.

[9] For additional information on the kind of paging employed in a TOPS–20 system, refer to "Storage organization and management in TENEX", by Daniel L. Murphy, AFIPS — Conference Proceedings, Vol. 41, page 23, AFIPS Press, Montvale, NJ.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt instructions reference executive virtual address space. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the microcode to carry out the mapping procedure, retrieve interrupt instructions, and handle traps, halts and UUOs.

NOTE

> Hardware paging operations are inextricably intertwined with the activities of the Monitor. The reader must be familiar with both to be able to understand either fully.

Paging

All of memory both physical and virtual is divided into pages of 512 words each. Physical memory can contain 1024 pages; its locations are specified by 19-bit addresses, where the left ten bits (17–26) specify the page and the right nine (27–35) the location within the page. The virtual memory space addressable by a program is 512 pages and uses 18-bit addresses, where the left nine bits (18–26) are the page number. However for compatibility with extended processors, the TOPS–20 paging system regards the virtual page as composed of sections, each of 512 pages, even though the KS10 has only one such section, and its virtual addresses have no section number. The hardware maps the one-section virtual address space into a part of the physical address space by transforming the 18-bit addresses into 19-bit addresses.[10] In this transformation 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 translation maps a virtual page into a physical page by substituting a 10-bit physical page number for the 9-bit virtual page number. The mapping procedure is carried out automatically by the pager, but the page map that supplies the necessary substitutions is set up by the executive program.

Pointers to the page maps for the user and executive virtual address spaces are contained in section tables that begin at location 540 in the user and executive process tables. But in the KS10 each section table has only one entry (for section 0) at location 540. Two locations in the register file are used by the Monitor to specify the physical page numbers of the process tables. To retrieve the section pointer from a process table, the pager uses the appropriate base page number as the left ten bits of the physical address and 540 as the right nine bits. The section pointer must identify — either directly or indirectly — a physical page that contains the page map. Every pointer and mapping takes one word, and since there are 512 pages and 512 words in a page, a page map requires exactly one page.

[10] The mapping procedure is of course applied only to storage module references, whether cached or not. AC references, which can be made by any program, even when virtual page 0 is accessible, are made directly to fast memory and require no mapping.

Figure 4.3: TOPS–20 Process Table Configuration

USER PROCESS TABLE

0 420	RESERVED	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO FLAGS	MUUO OP CODE, A
425	MUUO OLD PC	
426	E OF MUUO	
427	MUUO PROCESS CONTEXT WORD	
430	EXECUTIVE NO TRAP MUUO NEW PC	
431	EXECUTIVE TRAP MUUO NEW PC	
432	RESERVED	
433	RESERVED	
434	USER NO TRAP MUUO NEW PC	
435	USER TRAP MUUO NEW PC	
436 477	RESERVED	
500	PAGE FAIL WORD	
501	PAGE FAIL FLAGS	
502	PAGE FAIL OLD PC	
503	PAGE FAIL NEW PC	
504 537	RESERVED	
540	USER SECTION 0 POINTER	
541 777	RESERVED	

EXECUTIVE PROCESS TABLE

0 41	RESERVED
42 57	PRIORITY INTERRUPT INSTRUCTIONS
60 100	RESERVED
101	ADAPTER 1 INTERRUPT TABLE POINTER
102	RESERVED
103	ADAPTER 3 INTERRUPT TABLE POINTER
104 420	RESERVED
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION
424 537	RESERVED
540	EXECUTIVE SECTION 0 POINTER
541 777	RESERVED

Figure 4.3 shows the detailed organization of the process tables for both user and executive, as determined by the hardware. Any table locations not used are reserved for future use by the hardware or use by the Monitor for software functions.

Although the virtual space is always 256K by virtue of the addressing capability of the instruction and indirect word formats, the Monitor usually limits the actual address space for a given program by defining only certain pages as accessible. There is no requirement that the accessible space be continuous — it can be scattered pages. The Monitor also specifies whether each page is writable or not and cacheable or not.[11] To determine the mapping for a given virtual page, the microcode carries out a pointer evaluation procedure that starts with the section pointer. If it is discovered during this procedure that the page is inaccessible, the page map or the referenced page is not in memory, or the program is attempting to write in a write-protected page, the microcode traps to the Monitor, which must handle the situation. A trap to the Monitor for a reason of this sort is produced by generating a "soft page failure." But if nothing is amiss, the procedure is carried out entirely by the microcode — with no need to call the software — and it generates the mapping for the specified virtual page. The procedure requires access to the page map, to a memory status table in which the microcode keeps track of the use made of the page map and the program-referenced page, and perhaps to other predefined or software-defined tables as well. If the complete procedure were carried out in every instance, the processor would require at least two memory references for every one by the program. To avoid this, each mapping generated by the procedure is placed in a page table, and the pager makes its virtual-to-physical translations from the mappings held in the table.[12] Hence it is necessary to go through the evaluation procedure only when the reference cannot be made from the page table. Since the objective of the procedure is to place a mapping in the table, it is referred to as a "page refill."

Page Table A location in the page table contains a mapping entry in this format.[13]

M	*C*	PHYSICAL PAGE ADDRESS BITS 17–26

Each entry is identified as providing the physical page number for the translation for a particular virtual page. The properties represented by 1s in the two "page use" bits are as follows.

[11] Again for consistency with extended processors, the Monitor can make the section (i.e. the whole virtual space) inaccessible, unwritable or uncacheable, but is rather unlikely to do so.

[12] In the evaluations the microcode does carry out, it generally does not need to access a process table for a section pointer, as it keeps copies of the current pointers in the workspace.

[13] In the engineering drawings and even in some Monitor documents, the *M* bit is labeled "writeable", which name is consistent with its use with the TOPS–10 Monitor.

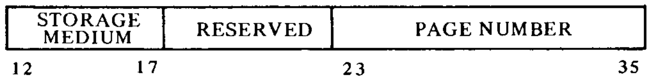
Bit	*Meaning of a 1 in the Bit*
M	Modified — and therefore writable without further ado. A refill produces a 1 in this bit if the page has already been modified or the reference that caused the refill is for write and the page is writable. A 0 does not imply that the page is write-protected, but simply that if a write reference occurs, the pager must find out if it can be written. Throughout this discussion, "write reference" means any reference involving writing; "read reference" means read only.
C	Cacheable.

The page table has 512 locations, one for each virtual page number. Besides a mapping for the virtual page that identifies it, each location has a parity bit, a bit that indicates whether the mapping is for user or executive address space, and a bit that indicates whether the entry is valid. A zero mapping is perfectly valid, but a location is labeled as containing no valid mapping by clearing it, thus clearing the valid bit.

When the program references a page whose mapping entry is tagged as valid and in the program address space, the 10-bit physical number[14] from the mapping for the virtual page is used as the left ten bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *M* bit). If however the entry is invalid or is not in the correct address space, or the reference is for writing and *M* is 0, the pager does a refill to get or revise the mapping for the specified virtual page from the page map. The result of the refill is placed in the table unless the reference fails because the page is inaccessible or the program is attempting to write in a protected page.

Page Refill

The refill of a mapping into the page table is accomplished by evaluating various types of pointers found in several kinds of tables. At some point in the procedure the microcode must encounter a "page address" that identifies the page map for the section, and it must end with a page address that identifies the physical page corresponding to the referenced virtual page. A page address has this format.



If bits 12–17 are zero, the storage medium is memory: i.e. bits 23–35 supply the number of a page[15] that is in memory. If bits 12–17 are nonzero, the

[14] Actually table locations have eleven bits for physical numbers, but the most significant is not used.

[15] All pointers have provision for 13-bit physical page numbers (as in the KL10), but the microcode uses only the right ten bits.

page exists but is stored on some other medium — perhaps the disk — and the microcode traps to the Monitor. A page address may be contained in a pointer, in which case some of the bits at its left have defined uses. But when the page address stands alone, bits 0–11 of the word containing it can be used arbitrarily by the software.

Special Tables. Besides the section tables in the process tables, a refill makes use of two predefined tables: the special page-address table (SPT) and the (core) memory status table (CST). These are software-determined tables in memory, but their base addresses are held in the workspace, rather than in the register file like those of the process tables.[16]

The special page-address table contains page addresses that specify shared pages or special pages (e.g. those used as page maps or other software-defined tables). The microcode accesses specific entries in the SPT by indexing on the physical base address (bits 17–35). The pointer format provides for an index of eighteen bits, so the SPT can actually be as large as 256K (and it need not start on a page boundary).

Information about the use made by programs of the various physical pages is kept in the memory status table. In every refill, the microcode updates CST entries for both the page containing the page map and the page referenced by the program. The entry for a page is a full word, and is accessed by adding the page number to the base address. If memory is fully implemented at 1024 pages, the CST occupies two of them, but need not begin on a page boundary. Note that the microcode does not manipulate CST entries for the process tables, the SPT, nor the CST itself, unless they are actually referenced by the program — in other words, unless the refill is being performed for a program reference to one of the tables.

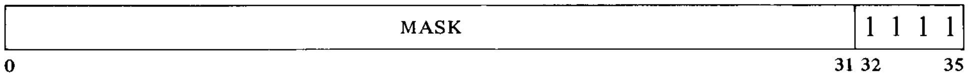
The status of a physical page in memory is indicated by a CST entry in this format.

STATE CODE	RESERVED	*M*
0 8		35

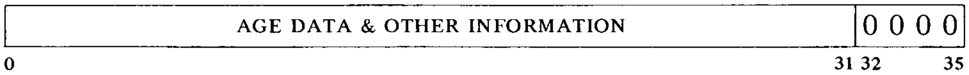
The Monitor keeps a state code in bits 0–8 of the entry; within the code, bits 0–5 represent the page age, which must be nonzero for the page to be usable, whether it is the program-referenced page or the page map. Bits 0–5 being zero causes an age trap to the Monitor.[17] The microcode updates the entry by anding a CST mask word into it and oring a process use word into that result. These two words are also held in the workspace. Bits 32–35 in them must be all 1s or all 0s as illustrated in order to preserve hardware information. A 1 in the *M* bit indicates the page has been modified since

[16] Remember that all memory tables defined by the pager are in physical address space, i.e. they have physical base addresses. Of course, to load or access a table, the Monitor must use paged virtual addresses. Note that if the base address is limited to a page number (bits 17–26), the table must begin at a page boundary.

[17] Zero age usually means the page is being swapped in and is not yet available for reference. The Monitor can use part of a CST entry to record which processes use the page.



CST MASK WORD

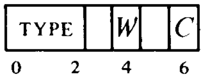


PROCESS USE WORD

being brought into memory.[18] The microcode sets this bit in the entry for the referenced page — not that for the page map — if the reference is write and the page is writable.

Indirect pointers make use of tables whose locations are defined entirely by the Monitor. In a single refill, these may include one or more secondary section tables or page maps. Each such table or map is determined by a page address and a 9-bit index, and is therefore a single page. Memory status is kept only for the page maps.

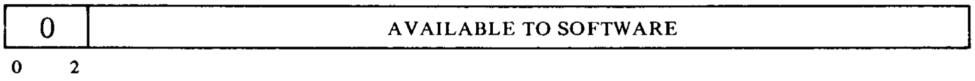
Pointers. The microcode evaluates two kinds of pointers: section pointers and map pointers. The former are used in section tables and the latter in page maps. Members of these two classes are identical in form but differ enough in function so they must be treated separately. There are four types of section and map pointers distinguished by a type code in bits 0–2; of these, three are access pointers, i.e. they allow access to the given section or page. An access pointer has this format in its left seven bits.



Every access pointer must have use bits for the section or page it represents. These bits, *W* and *C*, indicate whether the section or page is writable or cacheable. Throughout the evaluation procedure the microcode effectively ands these bits from one pointer to the next, so the final result requires that the given characteristics be specified at every step. In other words if *W* is 1 in the final pointer for the mapping, the page is writable provided the entire section was also specified as writable by the original section pointer, and "writability" has been specified by every other pointer encountered along the way. Every access pointer must also either contain a page address or point to an SPT location that contains a page address.

Section Pointers. Entries in a section table are of these four types.[19]

No Access



The section is inaccessible.

[18] At the completion of a process, the Monitor checks the CST to determine which pages have been modified and must be rewritten on the disk.

[19] Type codes 4–7 are undefined and result in a page failure.

Immediate

1		*W*		*C*	RESERVED	STORAGE MEDIUM	RESERVED	PAGE NUMBER OF PAGE MAP
0	2	4		6		12	17	23 35

If bits 12–17 are zero, the page map is in the page specified by bits 26–35. Otherwise the page map is not in memory.

An immediate pointer contains the page address of the page map.

Shared

2		*W*		*C*	RESERVED	INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF PAGE MAP
0	2	4		6		18 35

The page address of the page map is in the SPT at the location specified by bits 18–35.

This pointer is used for a page map shared by a number of processes. Switching to another map requires changing only the common SPT entry.

Indirect

3		*W*		*C*		SECTION TABLE INDEX	INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER SECTION TABLE
0	2	4		6		9 17	18 35

In the SPT location specified by bits 18–35 is the page address of a secondary section table. The next section pointer to be evaluated is in that table at the location specified by bits 9–17.

Indirect pointers are used for Monitor reference to per-job and per-process areas. The pointers remain while the second section table is swapped with the job or process, or the SPT entry is changed.

Map Pointers. Entries in a page map are of these four types.[19]

No Access

0	AVAILABLE TO SOFTWARE
0	2

The page is inaccessible.

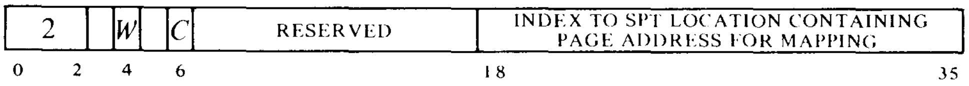
Immediate

1		*W*		*C*	RESERVED	STORAGE MEDIUM	RESERVED	PAGE NUMBER FOR MAPPING
0	2	4		6		12	17	23 35

If bits 12–17 are zero, the physical page specified by bits 26–35 corresponds to the referenced virtual page. Otherwise the referenced page is not in memory.

An immediate pointer contains the page address for the mapping.

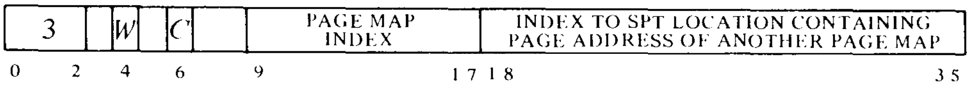
Shared



The page address for the mapping for the referenced virtual page is in the SPT at the location specified by bits 18–35.

This pointer is used for a physical page referenced as different virtual pages by different programs. The Monitor can move the page simply by changing the SPT entry.

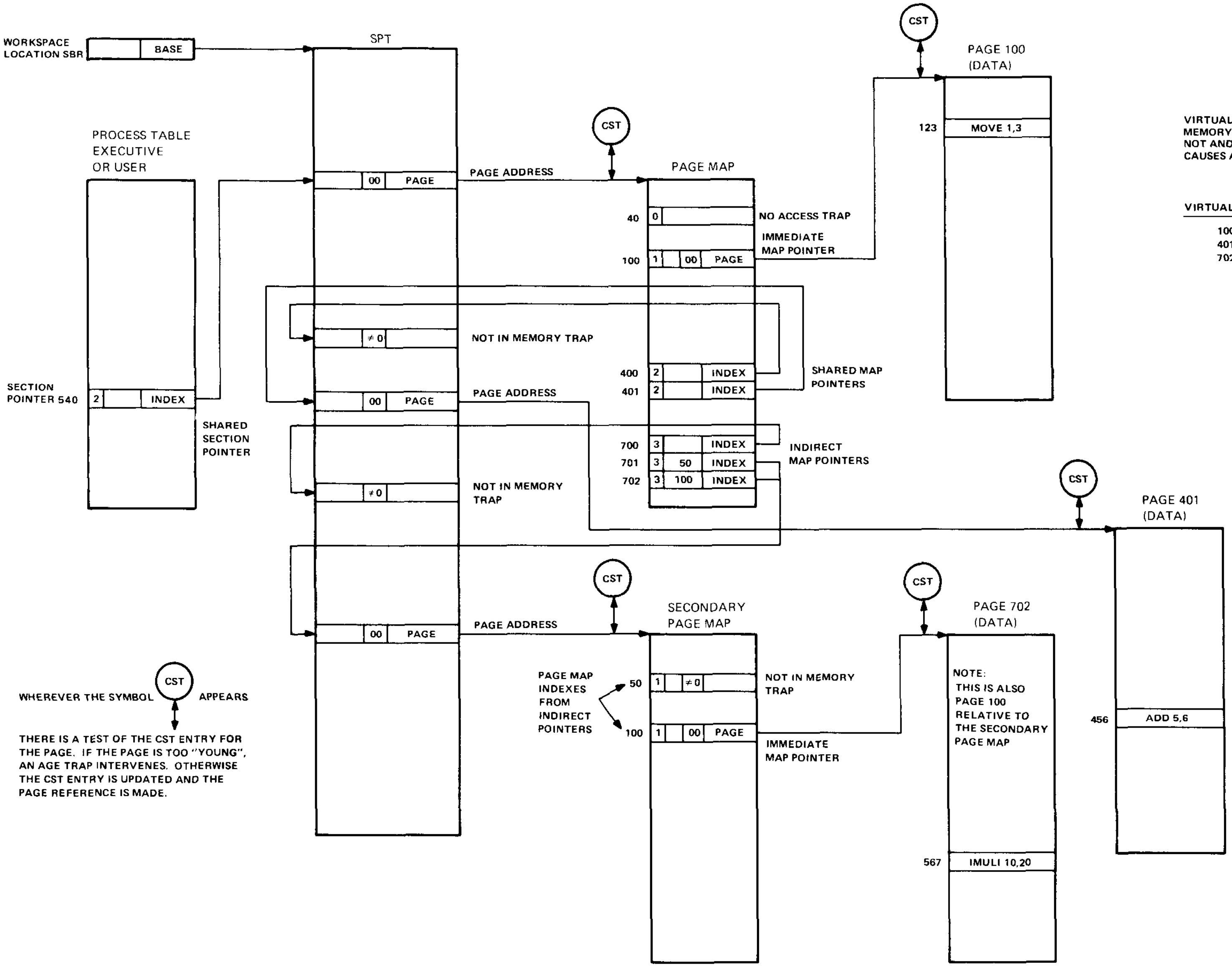
Indirect


3
W
C
PAGE MAP INDEX
INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER PAGE MAP
0
2
4
6
9
17
18
35

In the SPT location specified by bits 18–35 is the page address of a secondary page map. The next map pointer to be evaluated is in that map at the location specified by bits 9–17.

Refill Procedure. If the page table lacks a valid mapping for a reference, the pager must evaluate section and map pointers to get the desired mapping. The procedure begins with the pointer for the section from the process table, and the pager follows the trail laid by the various pointers, as illustrated in Figure 4.4. At any step the microcode traps to the Monitor if it encounters a no-access pointer or a page address that indicates the page is not in memory. The first part of the procedure, which may go to the SPT or indirectly through it to other section tables, evaluates section pointers to arrive at the page address of the page map. Using this physical page number as the left ten bits of an address and the number of the referenced virtual page as the right nine bits, the second part of the procedure retrieves a map pointer and evaluates it. This part may also go to the SPT or indirectly through it to other page maps to arrive at a page address for the mapping. Unless an age trap intervenes, memory status is updated along the way for any page maps used. If the reference can be made and there is no age trap for the referenced page, its status is updated including setting the *M* bit if the program is writing. The microcode then constructs the desired mapping, places it in the page table, and returns to the waiting reference.

Figure 4.4: TOPS–20 Paging Pointer Evaluation



NOTE: THIS IS ALSO PAGE 100 RELATIVE TO THE SECONDARY PAGE MAP

WHEREVER THE SYMBOL CST APPEARS THERE IS A TEST OF THE CST ENTRY FOR THE PAGE. IF THE PAGE IS TOO "YOUNG", AN AGE TRAP INTERVENES. OTHERWISE THE CST ENTRY IS UPDATED AND THE PAGE REFERENCE IS MADE.

VIRTUAL PAGES 100, 401 AND 702 ARE IN MEMORY. PAGES 40, 400, 700 AND 701 ARE NOT AND ANY REFERENCE TO THEM CAUSES A PAGE FAIL TRAP

VIRTUAL ADDRESS	CONTENTS
100123	MOVE 1,3
401456	ADD 5,6
702567	IMULI 10,20

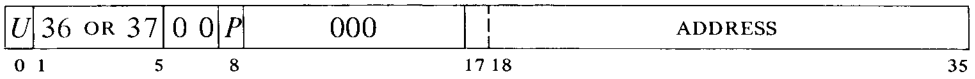
The mapping data is constructed from the result of the pointer evaluation, including the running evaluation of the use bits, and has the format illustrated in the discussion of the page table. The microcode always places a 1 in the valid bit to indicate that the virtual page is accessible and this is a valid mapping for it. *C* is simply the result of anding the *C* bits of the various pointers. *M* however is not. A refill sets up *M* according to the type of reference and the characteristics of the referenced page.

Circumstances[20]	*M*	*Effect*
Read reference, page not writable.	0	An attempt to write will fail.
Read reference, page writable but not yet modified (according to CST).	0	An attempt to write will succeed, after the mapping is revised.
Page writable, write reference or page already modified.	1	Sets *M* in CST entry; an attempt to write will succeed.

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the microcode terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap.[21] The trap operation makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current flag-PC doubleword in locations 501 and 502, sets up PC according to a new value in location 503, and clears the flags (placing the processor in executive mode). The processor then resumes operation in the new state at the location now addressed by PC. The same sequence of events occurs if the processor performs an IO instruction and the adapter fails to indicate the transfer was accomplished.

There are two kinds of page failures, hard and soft. A hard failure means that something really is amiss, whereas a soft failure generally means only that the program requires some kind of service from the Monitor. A hard failure is indicated by a 1 in bit 1 of the page fail word, and the particular failure is specified by a code (which is therefore ⩾ 20) in bits 1–5. There are three such failures of which two are true page failures, i.e. failures involving memory reference, and for these the page fail word has this format.



[20] The missing circumstance produces a page failure.

[21] A page failure that occurs during an interrupt instruction does not act this way. Instead the processor halts (§4.7).

Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a physical address was given for the reference. The code names the particular failure as follows.

36 Uncorrectable memory error — in a processor reference the memory controller has read an incorrect word from storage and was unable to correct it. The processor has saved the word in AC 0 and AC 1, block 7, and has set the Bad Memory flag (RDAPR bit 28).

37 Nonexistent memory — the processor has called for a storage reference over the bus but the memory controller did not respond. This error also sets the No Memory flag (RDAPR bit 27).

If the failure code is 20, the fail word instead has this format

U	20	0	0	1	0	1	0	0	*B*	IO ADDRESS
0	1 5			8		10			13	14 35

and indicates a nonexistent IO register, i.e. an IO instruction gave an IO address to which there was no response. A 1 in bit 13 indicates a byte operation. (The 1s in bits 8 and 10 mean a physical reference and an IO function on the bus.) Note that this is not an IO page failure, which is a true (memory) page failure and causes a halt.

A soft failure can result only from actions taken in a refill or writability check and is indicated by a 0 in bit 1. This means either an attempt to write in a write-protected page, or the evaluation procedure encountered some condition beyond which it could not go — a no-access pointer, an illegal pointer code, some page (not necessarily the program-referenced one) not in memory, or an age trap. The fail word still contains the *U* bit and the virtual address, but now bits 1–8 have one of these formats,

	1	2	3	4	5	6	7	8
WRITE VIOLATION	0	1	0	0	1	0	0	1
OTHER FAILURE	0	0	0	0	*T*	0	0	1

where bit 8 is the inverse of bit 8 in the hard case (1 means virtual), and *T* indicates the type of reference in which the failure occurred: 0 for a read-only reference, 1 for any reference involving writing. A 0 in bit 2 means the evaluation procedure was incomplete. In the write violation configuration, the 1 in bit 2 means the procedure was completed, and the 0 in bit 4 comes from anding the *W* bits in the string of pointers. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course *T* and bit 2 both being 1 implies a write failure.

For a page fail trap, the processor automatically switches to executive mode. After rectifying the situation, the Monitor eventually returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two-part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §4.1. Before returning to the failed instruction, the

Monitor must invalidate the mapping for the page and revise the pointers for the new situation. Then when the instruction is restarted, the pager will do a refill to get the new, correct mapping.

A no-access pointer may imply that the page simply does not exist. Otherwise a soft failure seldom implies that anything is "wrong." Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in memory. When the user attempts to gain access to a page that is not there (i.e. for which the refill encounters a not-in-memory page address), the Monitor would respond to the failure by bringing in the needed page from the disk, either adding to the user space, or swapping out a page the user no longer needs or has not used recently. Similarly a process using several sections may have only one in core at a time. While swapping is in progress, the Monitor runs some other user, returning to the interrupted job when the requested page is available.

The same situation exists for writability. Keeping track of modified pages is handled by the refill procedure using the memory status table. But a page may be write-protected because is it shared by a number of processes, wherein a change made by one might not be wanted by the others. Thus in response to a write failure, the Monitor might make a separate writable copy of the page for the sole use of the process that wishes to modify it.

The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory. For such purposes the processor has this instruction.

MAP Map an Address

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

If the pager is on, map the page number of the virtual effective address *E* and place the resulting physical address and other map data in AC. If the page is accessible, the information loaded into AC is of the form

U	0	1	*M*	*W*	0	0	*C*	1	000	PHYSICAL ADDRESS
0	1	2	3	4	5	6	7	8	9 16	17 35

where bits 17–26 are the physical page number the pager supplies for *E*, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and *M*, *W* and *C* are page use bits resulting from the pointer evaluation procedure as explained above. If the page is inaccessible, AC receives the given virtual address in place of a physical address; the word also includes *U* and a 1 in bit 8, but the remaining bits are all zero.

However, should a memory error occur during the refill, AC receives a hard page fail word. If the pager is off, the result is undefined.

Notes. The instruction cannot actually fail, because regardless of what happens, the page fail microcode returns to it instead of trapping to the Monitor. The effective address calculation done for it could fail however.

4.5 Memory Management

In order properly to manage memory, the executive program must select the kind of paging, set up process tables and page maps for itself and the various users, oversee the operation of the page table, and select the fast memory block to be used by each program (usually block 0 for itself). At any given time, accumulator, index register and fast memory references are made to that AC block that is assigned as "current." Given a particular processor mode and an appropriate process table and page map, the Monitor effectively defines the address space for a process (which may be itself) by specifying the base address for the process table and selecting the current AC block.

When a user program calls the Monitor it is usually to request some activity, which may often require the executive to gain access to the user address space. To facilitate the crossover from one address space to another, the same instruction through which the Monitor assigns its own current AC block also allows assignment of an AC block for the "previous context" — i.e. the context of the process that made the call. This, together with a flag that indicates the mode of the caller, allows execution of instructions in the previous context (more about this subject later). At any point in time, the previous context is essentially the circumstances in which the previous process was running. Note that the previous context need not be the user; the same techniques can be exploited following a call from one level of the Monitor to another.

For initial setup, the executive program must be cognizant of certain fundamental characteristics that can vary from one system to another. For this purpose the instructions for basic management include not only those that control the pager, but also one that addresses the processor to discover what those characteristics are. The first five of the following instructions are for either kind of paging; the remaining eight are solely for handling the special registers used in the TOPS–20 pointer evaluation.

APRID Arithmetic Procesor Identification

70000	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the microcode version number, the processor serial number, and a listing of the fundamental characteristics of the system into location *E* as shown. At present there are no microcode or hardware options.

MICROCODE OPTIONS	MICROCODE VERSION NUMBER
0 1 2 3 4 5 6 7 8	9 10 11 12 13 14 15 16 17

HARDWARE OPTIONS	PROCESSOR SERIAL NUMBER
18 19 20	21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

WREBR Write Executive Base Register

70120	*I*	*X*	*Y*
0 12	13	14 17	18 35

Set up the system-oriented characteristics of the pager according to the effective conditions *E* as shown.

	TOPS-20 PAGING	ENABLE PAGER		EXECUTIVE BASE ADDRESS (PAGE NUMBER)
18 19 20	21	22	23 24	25 26 27 28 29 30 31 32 33 34 35

Load bits 25–35 into bits 16–26 in the executive base register (EBR in the register file) to select the executive process table. If bit 22 is 1 enable overflow trapping and enable the pager for the type of paging selected by bit 21: 1 TOPS–20, 0 TOPS–10. A 0 in bit 22 prevents traps and disables paging so all memory references are to physical locations unpaged.[22]

CAUTION

> Paging can be disabled only for executive mode. A user mode program will not run correctly unless the pager is turned on.

Invalidate the entire cache and page table by clearing the valid bits in all entries.

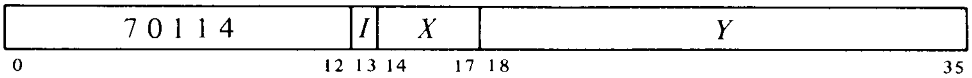
RDEBR Read Executive Base Register

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

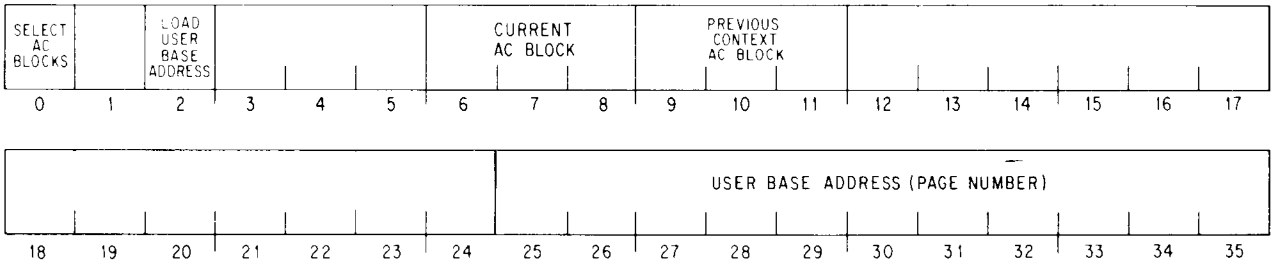
Read the system status of the pager into the right half of location *E*. The information read is the same as that supplied by WREBR.

[22] Note that disabling the pager does not mean there can be no page failures, as these can be caused by conditions having nothing to do with paging, i.e. with translating virtual to physical addresses.

WRUBR Write User Base Register



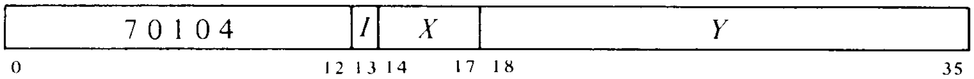
Set up the process-oriented elements of the pager according to the contents of location *E* as shown.



Bits 0 and 2 are change indicators for parts of the data word: when a bit is 0, the corresponding part of the word is ignored, and the equivalent value supplied by a previous WRUBR remains in effect.

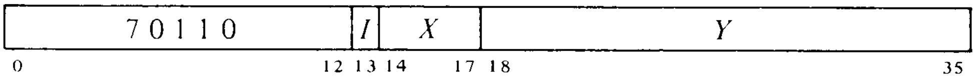
If bit 0 is 1, select as the current and previous context AC blocks those specified by bits 6–8 and 9–11, respectively. If bit 2 is 1, load bits 25–35 into bits 16–26 in the user base register (UBR in the register file) to select the user process table, and invalidate the entire cache and page table by clearing the valid bits in all entries.

RDUBR Read User Base Register



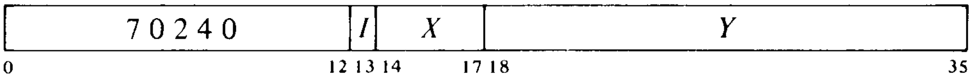
Read the process status of the pager into location *E*. The information read is the same as that supplied by a WRUBR (bits 0 and 2 are 1s).

CLRPT Clear Page Table Entry



Invalidate the page table mapping entry for the page referenced by *E*, and invalidate the entire cache.

WRSPB Write SPT Base Address



Load the contents of location *E* into the SPT base register in the workspace.

RDSPB Read SPT Base Address

70200	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the SPT base register into location *E*.

WRCSB Write CST Base Address

70244	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location *E* into the CST base register in the workspace.

RDCSB Read CST Base Address

70204	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the CST base register into location *E*.

WRCSTM Write CST Mask

70254	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location *E* into the CST mask register in the workspace for use as the mask in CST updating.

RDCSTM Read CST Mask

70250	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the CST mask register into location *E*.

WRPUR Write Process Use Register

70214	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location *E* into the process use register in the workspace for use as the process use word in CST updating.

RDPUR Read Process Use Register

70210	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the process use register into location *E*.

At power turnon the contents of the cache and page table are indeterminate, the processor is in executive mode, paging is disabled, and the current AC block is 0. After the console loads the microcode, it then loads the initializing executive program. This program, running unpaged in physical memory, should give an APRID to determine system characteristics. The unpaged program ends with a WREBR that selects and enables paging, specifies the executive base address, and invalidates the cache and page table. From this point the executive program runs paged and must set up the first user or users, loading the user process tables and page maps, and bringing in whatever parts of user programs and data that are consistent with good working-set management. Finally the Monitor gives a WRUBR to assign the base address and current AC block for the first user, and then transfers control to the user program via an XJRSTF or JRSTF.

On a call from the user via an MUUO, give an RDUBR to determine the context of the user, i.e. his AC block. Then give a WRUBR that assigns block 0 as current for the Monitor, assigns the user AC block as previous context for accessing user space, but leaves the base address alone so the right paging is still available for such access. To return to the same user, reassign the AC block without changing the base address. Note that on the transfer to a user program no previous context AC block need be given as the user cannot employ PXCTs.

The usual procedure for administering AC blocks is to assign block 1 to all users and assign two or three blocks for the sole use of interrupt routines. Suppose the assignments are: block 0 for the Monitor, block 1 for all users, block 2 for the highest priority interrupt level, block 3 for the second highest level, and block 4 for all other levels. Then in no circumstances is it necessary to determine which block to save, and interrupt routines on the highest, second highest and lowest levels need not save any. Moreover the Monitor need not even store block 1 when it takes control from a user temporarily. When switching from one user to another, the Monitor usually stores the first user's accumulators in his process table or 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 process table or shadow area, where they were stored after the last time the new user ran.

On a change from one process to another the entire page table must be invalidated, but this is done automatically by the instruction that assigns the new user base address. If the system uses shared or indirect pointers, or several virtual page numbers point to the same physical page, then the table must be invalidated whenever a page is removed from memory or a pointer is removed from a user page map. On the other hand deletion of a page with a unique mapping requires only that a CLRPT be given to invalidate the entry containing it.

Previous Context Execute

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. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero *A* bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous context. A PXCT can be given only in executive mode, but the previous context may be the user, as following a call to the Monitor by the user. The previous context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. (Note: it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.)

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory and index register references are made. To work as a PXCT, an XCT must be given in executive mode, and the bits in its *A* field (9–12) must not all be 0 (in user mode *A* is ignored). But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction, all accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a fast memory location.) If the instruction makes no memory operand references, as in a shift or immediate mode instruction, and it has no indexing or indirection (i.e. the instruction word gives *E* directly), then its execution differs in no way from the normal case. The only difference is in memory and index register references.

The previous context is specified by two quantities. Following a call by an MUUO, the fast memory block assigned to the calling program appears as the current context AC block in the word read by an RDUBR. For the called program, this value can then be assigned as the previous context by a WRUBR. The current AC block of the calling program also appears in the process context word supplied by the MUUO. Various levels of the Monitor may all use fast memory block 0; or a separate block may be assigned to that part of the Monitor that uses PXCTs in handling MUUO calls from other parts of the Monitor.

Just as the current mode is indicated by the User flag, the mode in which the calling program was running is indicated by Previous Context User.[24] At a call this flag may be set up automatically or it may be set up by a flag-PC doubleword or a PC word. Note that the restrictions on references made in the previous context are those of the previous context — not those of the context in which the PXCT is given. Suppose the executive executes an instruction that references an inaccessible user area. Such a reference would fail.

Which references in the executed instruction are made in the previous context is determined by 1s in the *A* portion of the PXCT instruction word as follows.

Bit *References Made in Previous Context if Bit is 1*

9 Effective address calculation of instruction, including both instruction words in EXTEND (index registers, address words by indirection); also EXTEND effective address calculation of source pointer if bit 11 is 1 and of destination pointer if bit 12 is 1

10 Memory operands specified by *E*, whether fetch or store (e.g. PUSH source, POP or BLT destination); byte pointer; second instruction word in EXTEND

11 Effective address calculation of byte pointer; source in EXTEND; effective address calculation of EXTEND source pointer if bit 9 is 1

12 Byte data; stack in PUSH or POP; source in BLT; destination in EXTEND; effective address calculation of EXTEND destination pointer if bit 9 is 1

Previous context referencing is useful and reasonable in some instructions but inapplicable to others. There is no trap of any kind, and the effect of using the feature with an instruction to which it does not apply is simply undefined.

Applicable

Move, XMOVEI
EXCH, BLT, XBLT
Half word, XHLLI
Arithmetic
Boolean
Double move
CAI, CAM
SKIP, AOS, SOS
Logical test
PUSH, POP, ADJSP
Byte
MAP

Inapplicable

LUUO, MUUO
AOBJN, AOBJP
JUMP, AOJ, SOJ
JSR, JSP, JSA, JRA, JRST
PUSHJ, POPJ
XCT, PXCT
Shift-rotate
String
IO
System (except MAP)

[24] Previous Context User is in the same flag bit that is used for User In-out, which has no meaning in executive mode.

Note that no jumps can use previous context referencing. Even among the instructions to which such referencing is applicable, only a limited number of the sixteen possible bit combinations is useful or meaningful. Doing an effective address calculation in the previous context (selected by bit 9 or 11) makes sense only if the corresponding data access is also in the previous context (as selected by bit 10 or 12 except 11 or 12 in EXTEND). Only these combinations are permitted.

Instructions	*9*	*10*	*11*	*12*	*References in Previous Context*
General	0	1	0	0	Data
	1	1	0	0	*E*, Data
Immediate	1	0	0	0	*E*
BLT	0	0	0	1	Source
	0	1	0	0	Destination
	0	1	0	1	Source, destination
	1	1	0	0	*E*, destination
	1	1	0	1	*E*, source, destination
XBLT	0	0	1	0	Source
	0	0	0	1	Destination
	0	0	1	1	Source, destination
Stack	0	0	0	1	Stack
	0	1	0	0	Memory data
	0	1	0	1	Memory data, stack
	1	1	0	0	*E*, memory data
	1	1	0	1	*E*, memory data, stack
Byte	0	0	0	1	Data
	0	0	1	1	Pointer *E*, data
	0	1	1	1	Pointer, pointer *E*, data
	1	1	1	1	*E*, pointer, pointer *E*, data

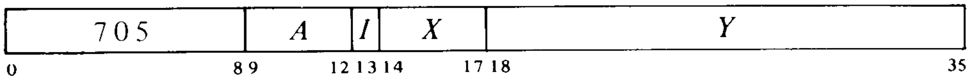
The most frequent use of previous context referencing is simply for the transfer of words between user and executive. For this reason the processor has these two convenient instructions.

UMOVE User Move

704	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the same function as PXCT 4,[MOVE *A*,*E*]. However, whereas a PXCT can be performed only in executive mode, UMOVE can also be done in user in-out mode.

UMOVEM User Move to Memory



Perform the same function as PXCT 4,[MOVEM *A*,*E*]. However, whereas a PXCT can be performed only in executive mode, UMOVEM can also be done in user in-out mode.

4.6 System Timing

The timer includes a 12-bit hardware millisecond counter, a doubleword time base kept from it, and an interval register for timed interrupts. The millisecond counter runs continuously at 4.1 MHz and represents an elapsed time of just under 1 ms at each overflow. Whenever the counter is read, its two least significant bits are ignored, so its contents effectively represent a count in microseconds (1/1025th ms).

The time base is a double length number kept in a pair of registers in the workspace. It is a 71-bit unsigned quantity in which the entire first word comprises the high order thirty-six bits, and the low order thirty-five are in bits 1–35 of the second word.[25] In this doubleword, the hardware counter corresponds to the right twelve bits of the low order word. The program can initialize the time base as a number of milliseconds (the low order twelve bits are ignored), and every time the counter overflows the microcode adds 2^{12} to the base.

The interval register (in the workspace) holds a period that is specified by the program and corresponds in magnitude to the low order word of the time base. This allows a maximum interval of 2^{23} ms, which is almost 140 minutes. At the end of each interval, the microcode sets Interval Done (RDAPR bit 30), requesting an interrupt on the level assigned to the system flags (§4.8). In a separate workspace register, the microcode starts with the given period, decrements it by 2^{12} every time the millisecond counter overflows, and sets the flag when the contents of this "time to go" register reach zero or less. Hence the countdown is by milliseconds, and any nonzero quantity in the low order twelve bits of the given period adds a whole millisecond to the count. (However, following specification of an interval by the program, the first downcount occurs at the first counter overflow regardless of when the register was loaded.)

The processor has these instructions for the program to handle the time base and the interrupt interval.

[25] Remember, it is a property of twos complement arithmetic that the sign can be used as an extra magnitude bit in an unsigned number. But since the hardware is set up for signed arithmetic, bit 0 of any lower order word must be skipped.

WRTIM Write Time Base

70260	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of location *E*,*E*+1, clear the right twelve bits of the low order word read (the part corresponding to the hardware millisecond counter), and place the result in the time base registers in the workspace.

RDTIM Read Time Base

70220	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the time base registers, add the current contents of the millisecond counter to the doubleword read, and place the result in location *E*,*E*+1.

WRINT Write Interval

70264	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location *E* into the interval register in the workspace.

RDINT Read Interval

70224	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the interval register into location *E*. The period read is the same as that supplied by WRINT.

4.7 Halt Status

Whenever the processor halts, the microcode places a halt code, giving the reason for the halt, in physical (i.e. storage) location 0, and places PC in physical location 1. Except at error-free powerup, it then saves the register file and VMA in a halt status block beginning at a physical location specified by the program, although the program can inhibit storing of halt status altogether. The registers saved in the status block are as follows.

Location	*Register*
0	MAG
1	PC
2	HR
3	AR
4	ARX
5	BR
6	BRX
7	ONE (1)
10	EBR
11	UBR
12	MASK
13	FLG (flags, page fail code)
14	PI
15	XWD1 (1,,1)
16	T0
17	T1
20	VMA (with flags)

Halt codes in the range 0–77 are used for "normal" halts. Codes in the ranges 100–777 and 1000 or greater respectively indicate software and microcode/hardware failures. Codes currently assigned are these.

Code	*Halt Condition*
0	Microcode just started; on this halt no status block is stored
1	Program gave a HALT (AR and PC contain *E*)
2	Console halted the processor
100	IO page failure
101	Illegal interrupt instruction If halt occurs on a vector interrupt, status block contains these quantities: T0 Vector as read from bus ARX EPT address + 100 + adapter number BR Address of illegal instruction BRX Vector masked and shifted
102	Zero table pointer for vector interrupt (for contents of T0 and ARX, see code 101)
1000	Error in BWRITE dispatch on dispatch ROM
1005	In powerup sequence, processor got wrong result when computing table of powers of 10 for use by string microcode (BR and ARX contain high and low words of incorrect 10^{21})

At powerup the microcode assigns an address of 376000 for storing halt status. The program can change the assignment at any time using these instructions.

WRHSB Write Halt Status Block Base Address

70270	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location E into the halt status block base register in the workspace. If bit 0 of the word in E is 0, bits 17–35 will be used as the physical address for storing halt status. But if bit 0 is 1, no status will be stored.

RDHSB Read Halt Status Block Base Address

70230	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the halt status block base register into location E.

4.8 System Conditions

This section discusses special logic through which the program controls and receives information about other parts of the system, specifically memory and the console. Any program also has considerable dealings with the peripheral equipment, but that is another subject.

System Flags

Four of these eight flags are set by memory hardware error conditions. Two others are used for communication between processor and console, and one is used by the microcode to signal completion of an interval count. The program can enable any flag to request an interrupt on a level assigned to them all. There are of course other error indications besides the flags. A parity error in the internal data paths of the processor causes the console to shut down the system by turning off the processor clock. Software errors in the handling of interrupts and some processor hardware failures cause the microcode to halt the processor as discussed in §4.7. And yet other conditions cause page failures.

The system flags are generally regarded as important enough to be assigned to the highest priority level. However for most conditions the common practice is for the interrupt to switch over to the lowest priority level by means of a program-set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

The flags are handled by these two instructions.

WRAPR Write System Flags

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

Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 20–31 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

		ENABLE	DISABLE	CLEAR	SET	SELECT FLAGS FOR BITS 20 – 23									PRIORITY INTERRUPT ASSIGNMENT		
		SELECTED FLAGS				FLAG 24	INTRUPT CONSOLE	POWER FAILURE	NO MEMORY	BAD MEMORY DATA	CORECTD MEMORY DATA	INTERVAL DONE	CONSOLE INTRUPT				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Then after 300 ns clear the Interrupt Console flag.

Bits 20–23 select flag functions: 1s in these bits produce the indicated effects on the system flags selected by 1s in bits 24–31. A 1 in bit 20 enables the setting of any selected flag to request an interrupt on the level assigned to the flags; a 1 in bit 21 disables the selected flags from requesting interrupts. Similarly a 1 in bit 22 or 23 clears or sets the selected flags. The result of putting 1s in both bits 20 and 21 or 22 and 23 is indeterminate.

The reason for clearing Interrupt Console is to provide a pulse on the signal line to the console in case the instruction has set the flag. Pulsing the line triggers an interrupt in the console microprogram.

Notes. Except for Flag 24 (which has no defined meaning) and Interrupt Console, the program setting a flag has no relation to what the flag represents — the function is used only to check out the flag logic.

RDAPR Read System Flags

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

Read the status of the system flags into location *E* as shown (asterisks indicate bits that can cause interrupts).

						FLAGS ENABLED TO INTERRUPT											
						FLAG 24	INTRUPT CONSOLE	POWER FAILURE	NO MEMORY	BAD MEMORY DATA	CORECTD MEMORY DATA	INTERVAL DONE	CONSOLE INTRUPT				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
						*	*	*	*	*	*	*	*				

						FLAG 24	0	POWER FAILURE	NO MEMORY	BAD MEMORY DATA	CORECTD MEMORY DATA	INTERVAL DONE	CONSOLE INTRUPT	INTRUPT REQUEST	PRIORITY INTERRUPT ASSIGNMENT		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

6–13 A 1 in any of these bits indicates that setting the listed flag will request an interrupt on the level assigned to the flags by bits 33–35 of the WRAPR.

24 Spare — available to the program for any purpose.

25 When read, this flag should always be 0, as any WRAPR that sets it also clears it to provide a pulse on the interrupt line to the console.

26 Ac power has failed. The program should execute some agreed-upon shutdown procedure and halt the processor. Note that PC may point to an interrupt routine rather than the main program. After power is restored the console must reboot the system, and the Monitor must reestablish the operating environment (§4.5).

27 The processor was granted the bus for access to memory, but the memory controller did not respond within two bus cycles. This is most likely because the memory subsystem contained no array board corresponding to the address given, or there has been a refresh error. Note that this condition also produces a page failure. Since a nonexistent memory supplies zero data, on read this error may be accompanied by a 1 in bit 28.

28 In a read reference by the processor, the word retrieved (and sent) was wrong and the memory circuits were unable to correct it. Note that this condition also produces a page failure.

29 In a read reference by the processor, the word retrieved was wrong but the memory circuits were able to correct it.

30 The microcode has completed a count of the interval specified by the program.

31 The console is requesting a processor interrupt.

32 Some system flag is currently requesting an interrupt, i.e. some flag in bits 24–31 is set and has been enabled to interrupt as indicated by a 1 in the corresponding position in bits 6–13.

Programming Cautions. When handling bad data or nonexistent memory interrupts, the programmer should beware of the following.

NOTE

> In general it is better not to use the interrupt for these conditions, as the page failure provides more information. Moreover if the interrupt is used, the processor interrupts out of the page failure, which occurs first.

- Should an error flag be set while another interrupt request is being processed, the system would handle the lower priority interrupt before getting to the flag interrupt. This means PC may be pointing to a lower level interrupt routine rather than the program level at which the error occurred.
- Even without inadvertent interference from another level, the processor may perform another instruction between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may be pointing to the instruction following the one in which the error occurred.
- An error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second flag interrupt before the error-handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another — consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what has gone wrong.

Memory Status

The memory controller reports information on error conditions by means of status that the program (or operator) can read or test using IO instructions that address the controller (IO address 0100000). Note that the errors reported may have nothing whatever to do with the program or processor: they may be the result of access by an adapter or the console. On every access the controller regularly loads the address and, if read, the error correction code into the status register. But if a read error (incorrect data read from the storage array) or refresh error occurs, the address and code are held — even through other errors — until the processor or console writes a status word that clears the holding flag.

The remainder of this section identifies the information read as status and the functions that can be performed by writing status. For advice on how to use the information for diagnosing memory problems, the reader should turn to the maintenance documentation.

Read Status

ERROR HOLD	UNCORRECTABLE ERROR HOLD	REFRESH ERROR	PARITY ERROR	ECC ON	ERROR CORRECTION CODE							POWER FAILED		HIGH ORDER ADDRESS BITS			
					CP	C40	C20	C10	C4	C2	C1						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

LAST ADDRESS OR FIRST ERROR ADDRESS																	
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

0 The memory controller has detected a read error or a refresh error (bit 3) and has held the error correction code in bits 5–11 and the address supplied over the bus in bits 14–35.

1 The code and address are being held for a read error in which the data read was uncorrectable.

2 A refresh cycle was still not finished 10.3 μs after the refresh logic requested it. The most likely cause is that the memory cycle logic was waiting for write data that failed to arrive. Setting this flag both clears and shuts down the cycle logic, so refreshing can continue but the memory is unavailable to the rest of the system until a write status clears the flag.

3 A parity error has been detected in information (command/address, data, status) received by the memory controller over the bus. This error indication is sent to the console, which may respond by turning off the processor clock.

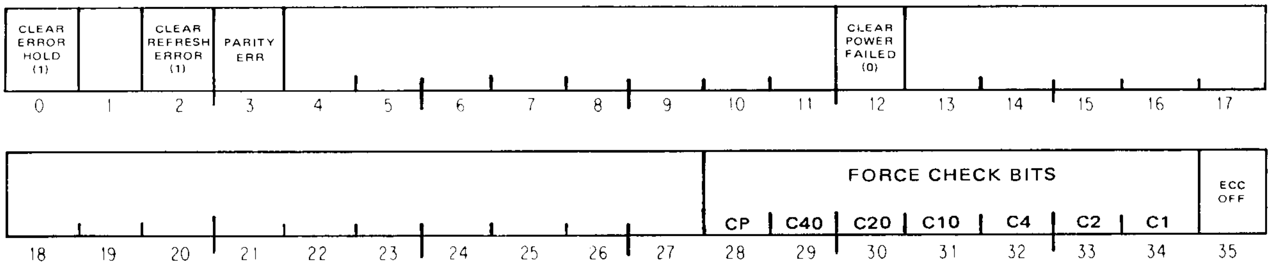
4 The error correcting circuits are active.

5–11 This is the error correction code for the last read data access, unless bit 0 is 1, in which case it is the code for the cycle on which a read error occurred or for the last read access before a refresh error.

12 Battery backup power (if present) is low, and will not be able to sustain memory refresh in the event of an ac power failure.

14–35 This is the address supplied in the last bus transaction with memory, unless bit 0 is 1, in which case it is the address used in the data access that caused the error hold (a read address on a read error, a write address on a refresh error).

Write Status



0 A 1 in this bit clears Error Hold, which in turn clears Uncorrectable Error Hold and drops the hold on the error code and address.

2 A 1 clears Refresh Error.

3 A 0 clears Parity Error, but a 1 sets it allowing checkout of the associated logic.

12 A 0 clears Power Failed.

28–34 A nonzero code forces the indication of errors where none exist, allowing checkout of the error detection and correction circuits.

35 A 1 disables the error correcting circuits. A 0 restores them to their normal, active state.

Chapter 5
KI10 and KA10 System Operations

The information presented in this chapter is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own IO, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

Programming for the system as a whole is programming in executive mode. In the KI10 executive mode is divided into kernel and supervisor modes. Only the kernel program is without instruction restrictions, and only it can access physical core unpaged. The supervisor program labors under the same instruction restrictions as the user and has no way of bypassing them, although it can read but not alter concealed pages (the kernel program can supply data tables to the supervisor program, and the latter cannot affect them). In the KA10 the executive program has no restrictions, and it manages protection and relocation hardware that is applicable only to the user.

The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working areas, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of the user. The KI10 kernel program accomplishes these objectives by handling all in-out for the system, setting up page maps, trap locations, interrupt locations and the like for both itself and the users, keeping job accounts, and so forth. The KA10 executive program also handles in-out, job accounts and interrupts, but it manages the user working space by setting up protection and relocation registers, and it takes care of arithmetic and stack overflow via the interrupt.

Except for handling in-out, the activities of an operating system are the topics covered in this chapter. The first section, on the console, is applicable to both processors. The basic system information is covered in three sections separately for each: §§5.2–5.4 for the KI10, §§5.5–5.7 for the KA10. The last section discusses the DK10 real time clock, which is used in both. Of course the system programmer must also be quite familiar with all of the material presented in Chapters 1 and 2. In particular he must fully understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the JRST instruction, MUUOs, and IO instructions (§§2.9, 2.16, 2.18).

In several of the CONI bit assignment drawings in this chapter, bits that can cause interrupts are indicated by asterisks.

5.1 Console

Most console operations are entirely manual, and these are described in Appendix F. However the program can communicate with the console in a limited way, and the programmer must be familiar with the format and execution of the readin function.

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 (paper tape, DECtape, and standard magnetic tape). 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 with executive paging disabled, so all access will be to the first 256K of physical memory unpaged. Following this the processor places the device in operation, brings the first word (the pointer) into location 0, and then reads the data block, placing the words in the locations specified by the pointer. Data can be placed anywhere in the first 256K of memory (including fast memory) except in location 0. The operation affects none of memory except location 0 and the block area.

Upon completing the block, the procesor leaves readin mode and begins normal operation. This is done in the KI10 by jumping to the location containing the last word in the block, in the KA10 by executing the last word as an instruction. In the KA10 the processor stops after executing the first instruction if the single instruction switch is on.

Console-Program Communication

Neither the processor nor the priority interrupt system require all four types of IO instructions, so the program can make use of their device codes for communicating with the console. Both processors have two instructions that transfer data between the 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 (§§5.2, 5.3).

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*.

Notes. MACRO also recognizes the mnemonic RSW (Read Switches) as equivalent to DATAI APR,.

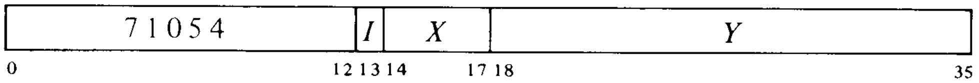
DATAO PI, Data Out, Console

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

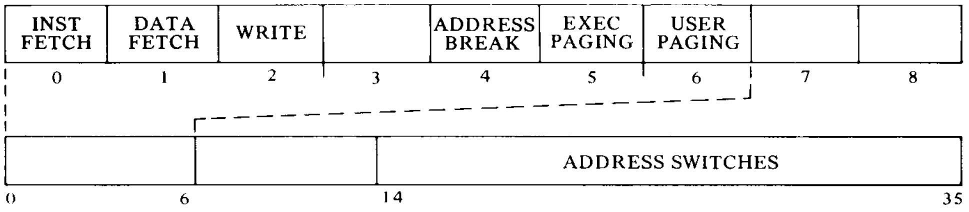
Unless the console MI program disable switch is on, display the contents of location *E* in the console memory indicators and turn on the triangular light beside the words PROGRAM DATA just above the indicators (turn off the light beside MEMORY DATA).

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

DATAO PTR, Operating Data Out, Console



Unless the MI program disable switch is on, set up the console address and address-condition switches according to the contents of location *E* as shown (a 1 in a bit turns on the switch, a 0 turns it off).



For complete information on the use of these switches, see Appendix F.1.

Notes. On the KI10 console, all switches are pushbutton flip-flop combinations; the instruction of course controls the flip-flops, not the buttons.

5.2 KI10 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, i.e. 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 levels arranged in a priority chain, with assignment of devices to levels entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35). Levels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be connected to a single level, and some can be connected to two levels (e.g. a device may signal that data is ready on one level, that an error has occurred on another).

When a device requires service it sends an interrupt request signal over the in-out bus to its assigned level in the processor. The processor accepts the request depending upon certain conditions, such as that the level must be active (on). The request signal remains on the bus until turned off by an appropriate response from the processor: either given by the program (CONO, DATAO or DATAI, depending on the device), or generated automatically by the hardware. Thus if a request is not recognized or accepted when made, it will be when conditions are satisfied. A single level 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 level 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 level 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. For example, 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.

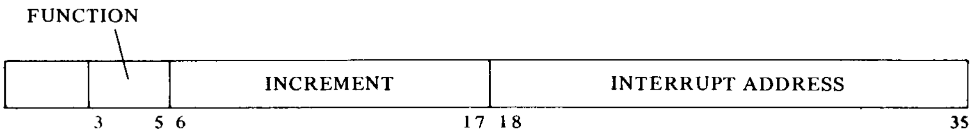
Having accepted a request, the processor will do nothing further with it unless the priority interrupt system is on. But even with the system off, the processor will continue to accept requests on other levels; and when the system is finally turned on, it will respond as though all requests had just been accepted, handling the highest priority one first.

Starting an Interrupt

A request made to an active level is accepted immediately unless some level is already waiting for an interrupt to start or an interrupt is starting for some level. Once a request is accepted with the system on, the level must wait for the interrupt to start. The processor however will delay any action on the request if it is already holding an interrupt for the same level or for a level with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). When a waiting level has priority higher than the current program, the processor sends an interrupt-granted signal for the waiting level that has highest priority. This action makes use of the IO bus. Should the bus be busy, the grant is sent as soon as the bus becomes available, taking precedence over any IO instruction that may also be waiting (note that in this situation the program actually stops). The grant signal goes out on the bus and is transmitted serially from one device to the next. Upon receiving the grant, a device that is not requesting an interrupt on the specified level sends the signal on to the next device. A device that is requesting an interrupt on the specified level terminates the signal path and sends an interrupt function word back to the processor. Note that there are therefore two orders of priority associated with an interrupt: first the level, and then for all devices requesting interrupts simultaneously on the same level, proximity to the processor on the bus. For priority purposes, all devices on the left bus are closer than those on the right bus.

Upon receipt of the function word, the processor stops the current program at the first allowable point to start an interrupt for the waiting level for which the grant was made. Allowable stopping points are at the completion of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), between transfers in a BLT, between steps in the calculation of the first part of the quotient in double floating division, and while an IO instruction is waiting for the bus. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

The action taken by the processor in starting an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the executive process table are associated with each level: locations 40 + 2*N* and 41 + 2*N*, where *N* is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 57. The processor starts a "standard" interrupt for level *N* by executing the instruction in the first interrupt location for the level, i.e. location 40 + 2*N*. The fixed locations however need not be used. The interrupt function word sent by the device may specify a standard interrupt using the fixed locations, or an equivalent interrupt using a pair of locations specified by the function word, or some other interrupt function entirely. The format of the function word and the operations the processor performs in response to the function selected by bits 3–5 of the word are as follows.



Bits 3–5 *Interrupt Function*

0 Processor waiting. If no response, perform a standard interrupt (see function 1).

A device designed originally for use with the KA10 will work when connected to the KI10 bus, where it always requests a standard interrupt by providing no response to the grant. Note that for simultaneous requests on a given level, all KI10 devices that return a function word have priority over all KA10 devices and over any KI10 devices that do not return a function word. The last group includes the reader, punch and console terminal, which are contained in the processor, as well as the processor itself acting as a device (see processor conditions, §5.3).

1 Standard interrupt — execute the instruction in location 40 + 2*N* of the executive process table.

2 Dispatch — execute the instruction in the location specified by bits 18–35.

3 Increment — add the contents of bits 6–17 to the contents of the location specified by bits 18–35. The increment is a fixed point number in twos complement notation, bit 6 being the sign, and bit 17 corresponding to bit 35 of the memory word.

4 DATAO — do a DATAO for this device using the contents of bit 18–35 as the effective address.

5 DATAI — do a DATAI for this device using the contents of bit 18–35 as the effective address.

6 Reserved (produces a standard interrupt).

7 Reserved (produces a standard interrupt).

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode. No interrupt operation can set Overflow or either of the trap flags; hence an overflow trap can never occur as a direct result of an interrupt. A page failure that occurs in an interrupt operation is never trapped; instead it sets the In-out Page Failure flag, which requests an interrupt on the level assigned to the processor (§5.3). These considerations of course do not apply to a service routine called by an interrupt instruction.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being 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 level, in direct response by the hardware (rather than by the program) to a request on that level. These locations may be the fixed ones for a standard interrupt or those given by the function word for a dispatch interrupt. §2.18 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. The interrupt instructions executed in a standard or dispatch interrupt fall into three categories.

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 (i.e. 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 §2.18, 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 level, 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 orginal 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 a PXCT (§5.4). The ultimate effect of the XCT depends of course on the instruction executed — and its effect is as described here for the various categories.

CAUTION

Neither an LUUO, a BLT, a DMOVEM, nor a DMOVNM will function in a reasonable manner as an interrupt instruction. Therefore do not use them.

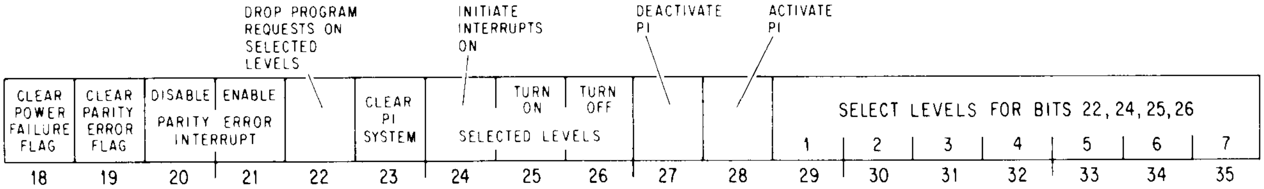
Interrupt Programming

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).



Bits 18–21 are actually for processor conditions (§5.3).

20 Prevent the setting of the Parity Error flag from requesting an interrupt on the level assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the level assigned to the processor.

22 On levels 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 levels, 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 levels selected by 1s in bits 29–35, and force the processor to accept them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a CONO that selects the same level and has a 1 in bit 22.

Remember that the processor allows the program to continue while it grants an interrupt. Thus when this bit forces acceptance of a request, another program instruction or two may be performed before the interrupt, even on the highest priority level. Moreover if the request is allowed to remain, additional instructions may be performed between successive interrupts. For other than the highest priority level, the greater the number of higher priority levels active, the greater the amount of program time available both initially and between successive interrupts. If the program forces an interrupt on the lowest priority level when all are active, there can be as much as 40 μs of program time between the CONO PI, and its interrupt.

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

26 Turn off the levels 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

70064	*I*	*X*	*Y*
0 … 12	13	14 … 17	18 … 35

Read the status of the priority interrupt (and nine 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 LEVELS						
											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

			INTERRUPT IN PROGRESS ON LEVELS							PI SYSTEM ON	LEVELS ON (ACTIVE)						
			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

Levels that are active are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 0–8 reflect the settings of various console operating switches; for information on these switches refer to Appendix F.1.

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 level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that level or any level of lower priority (requests, however, can be accepted on lower priority levels.).

A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be on when the JEN is given). This instruction restores the level on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority levels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, PUSHJ, or MUUO. If flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

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

Timing. The time a device must wait for an interrupt to start depends on the number of levels in use, and how long the service routines are for devices on higher priority levels. If only one device is using interrupts, it need never wait longer than 10 μs.

Special Considerations. On a return to an interrupted program, the processor always starts the interrupted instruction over from the beginning. This causes special problems in a BLT and in byte manipulation.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two-part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a PC word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an ILDB or IDPB would skip a byte. And if the routine restores the flag, the interrupted ILDB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In-out flag is set, a user program generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- No requests can be accepted, not even on higher priority levels, while an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- Most in-out devices are designed to drop an interrupt request when the program responds, usually with a DATAI or DATAO. If an interrupt is handled neither by a BLKI or BLKO interrupt instruction nor by a service routine, the programmer must make sure the device is configured to drop the request on receipt of whatever response the program does give.
- The interrupt instruction that calls the routine must save PC if there is to be a return to the interrupted program. Generally a JSR is used as it saves both PC and the flags, and it uses no accumulator.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. For example, computations that can be performed outside the routine should not be included within it.
- If the routine uses a UUO it must first save the contents of the pair of locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type. For an MUUO the routine must save locations 424 and 425 of the user process table. For an LUUO the routine must save location 40 in the executive process table and the location used by the UUO handler instruction to store the PC word.
- The routine must dismiss the interrupt (with a JEN) when returning to the interrupted program. The flags and UUO locations should be restored.

5.3 KI10 Processor Conditions

Page failures and overflow are handled by trapping, but there are a number of internal conditions that can signal the program by requesting an interrupt on a level assigned to the processor. The program can actually assign two levels — 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 (§5.2). Control over other conditions and inspection of all are handled by condition IO instructions that address the processor; the CONI also reads some console switches and maintenance functions. The processor also has a data-out instruction through which the program can perform margin checking of the system in both speed and voltage.

The error conditions are generally regarded as important enough to be assigned to the highest priority level. However for conditions that may be associated with user instructions (a parity error or unanswered memory reference), the common practice is for the error interrupt to switch over to the lowest priority level by means of a program-set request. Then the time to handle the situation, which may well be considerable, cannot interfere with high priority events.

One of the features controlled by the CONO for the processor is the automatic restart after power failure. This restart applies only when the levels on the power mains go below specification while the processor is running, and the power switch is on when power is restored — the machine never begins operation by itself when the operator turns the power switch on or off. Inadequate power, over temperature, etc. are indicated by the Power Failure flag. In order for the processor to restart itself, the program must respond in a particular way to the setting of Power Failure. If the program fails to respond properly, there is no restart.

The processor device code is 000, mnemonic APR.

CONO APR, Conditions Out, Arithmetic Processor

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

Assign the interrupt levels 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).

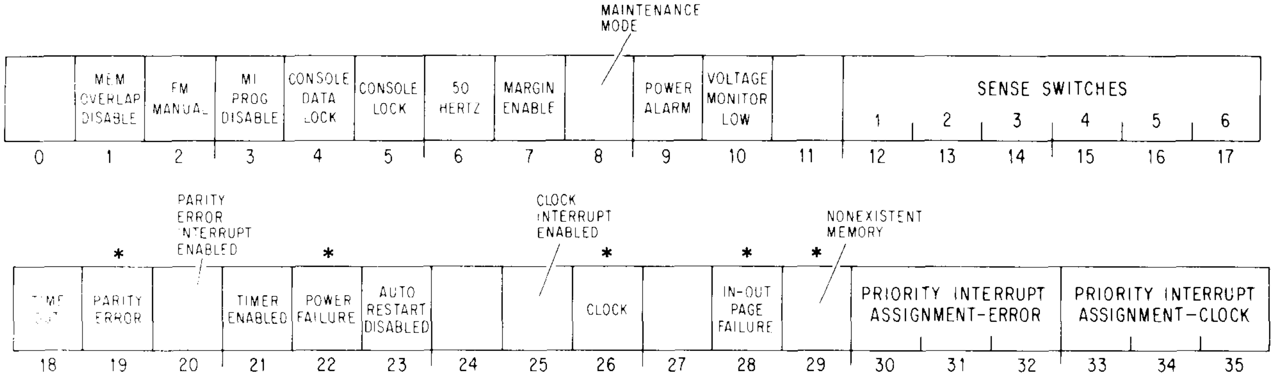
RESET TIMER	CLEAR ALL IN-OUT DEVICES	DISABLE TIMER	ENABLE TIMER	DISABLE AUTO RESTART	ENABLE AUTO RESTART	DISABLE CLOCK INTERRUPT	ENABLE CLOCK INTERRUPT	CLEAR CLOCK		CLEAR IN-OUT PAGE FAILURE	CLEAR NONEXISTENT MEMORY	PRIORITY INTERRUPT ASSIGNMENT-ERROR			PRIORITY INTERRUPT ASSIGNMENT-CLOCK		
18	19	20	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 (asterisks indicate bits that can cause interrupts).



Interrupts are requested on the error level (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 level (assigned by bits 33–35 of the CONO).

Bits 12–17 reflect the states of the console sense switches, which are specifically for operator communication with the program. Bits 1–5 reflect the settings of various console operating switches; for information on these switches refer to Appendix F.1. Bits 7–10 are maintenance functions[1] for which the reader should refer to Chapter 10 of the maintenance manual.

6 The system is operating on 50 Hz line power. This is important to the program, not only because some IO devices run slower on 50 Hz, but because the program must compensate for the time difference when using the line frequency clock (bit 26).

18 Bit 21 is 1 and the program has not reset the timer (CONO APR, bit 18) during the last 1.2 seconds (the period of the timer may vary from 1.2 to 1.5 seconds). The setting of this flag clears the processor and the peripheral equipment, and restarts the processor in kernel mode at location 70.[2]

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 level (see cautions below).

22 Ac power has failed. The program should save PC, the flags, mode information and fast memory in core, and halt the processor. Note that PC may point to an interrupt service routine rather than the main program.

[1] The processor does not actually have a maintenance mode — the bit is simply the OR function of a number of console switches, any cne of which being on implies that the processor is being operated for maintenance purposes.

[2] The timer provides a restart similiar to that following power failure. Running the machine under margins may result in significant logical errors. If the timer is enabled, failure of the program to reset it about every second allows it to time out. The restart instruction should set up PC, which would otherwise be clear.

The setting of this flag requests an interrupt on the error level. After 4 ms the processor is cleared. But at that time, if the power switch is on and the program has cleared Power Failure (CONO PI,400000) and enabled the auto restart (CONO APR,010000), then when adequate power levels are restored, the processor will resume normal operation by executing the instruction in location 70 in kernel mode. The restart instruction should set up PC, which would otherwise be clear.

26 This flag is set at the ac power line frequency and can thus be used for low resolution timing (the clock has high long term accuracy). If bit 25 is 1, the setting of the Clock flag requests an interrupt on the clock level.

28 A page failure has occurred in an interrupt instruction. The setting of this flag requests an interrupt on the error level. An interrupt page failure caused by the console address break switch also sets this flag instead of producing an address failure (§5.4).

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 program is expected to set up the interrupt instructions so that a failure simply cannot occur.

29 The processor attempted to access a memory that did not respond within 100 μs. The setting of this flag requests an interrupt on the error level (see cautions below).

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

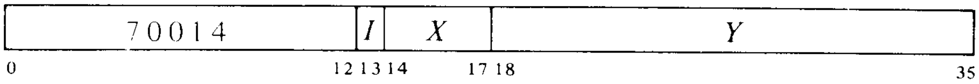
Programming Cautions. When handling parity error or nonexistent memory interrupts, the programmer should beware of the following.

- Should an error flag be set during an interrupt grant, the processor would handle a lower priority interrupt before getting to the processor interrupt. This means PC may be pointing to a lower level interrupt service routine rather than the program level at which the error occurred. (Remember that during the grant procedure, the interrupt system is otherwise static and the program continues. Moreover the processor is effectively at the far end of the bus.)
- Even without inadvertent interference from another level, it is quite likely the processor will perform one or perhaps two more instructions between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may well be pointing to the first or second instruction following the one in which the error occurred.
- A processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second processor interrupt before the error-handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another

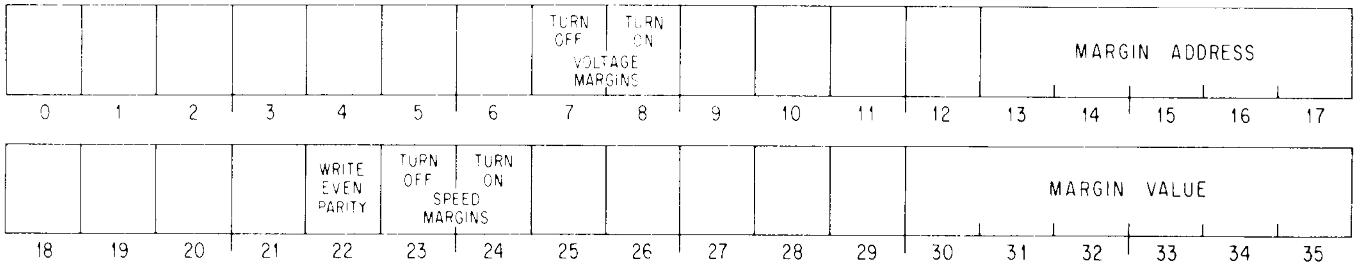
— consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what went wrong.

- The error may have originated from a console key function, and thus be hidden from any investigation by the program.

DATAO APR, Maintenance Data Out, Arithmetic Processor



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



The margin value specified by bits 30–35 of the output word is translated to a voltage in the range 0–10 volts by a D-A converter, whose output is available at pin 2S02V2. Running margins requires a slowdown capacitor in the converter. But turning off the margin enable switch cuts out the capacitor, making the converter output suitable for external use, such as for operating audio equipment to play Bach or rock or Bacharach.

Notes. This instruction is primarily for maintenance, for which further information is given in Chapter 10 of the *KI10 Maintainance Manual.*

5.4 KI10 Program and Memory Management

General information about the machine modes and paging procedures is given in Chapter 1, in particular in §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.

Paging

All of memory both virtual and physical is divided into pages of 512 words[3] 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 pager 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. For example 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

[3] 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 a PXCT (see below).

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. Then 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.

Figures 5.1 and 5.2 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 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 space for a given program by defining only certain pages as accessible.[4] The Monitor also specifies whether each page is public or not and writable or not. Each word in the page map has this format to supply the necessary information for two virtual pages.

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	Writable (not write-protected)
S	Software (not interpreted by the hardware)
X	Reserved for future use by DEC (do not use)

[4] 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.

Figure 5.1: Virtual Address Space and Page Map Layout

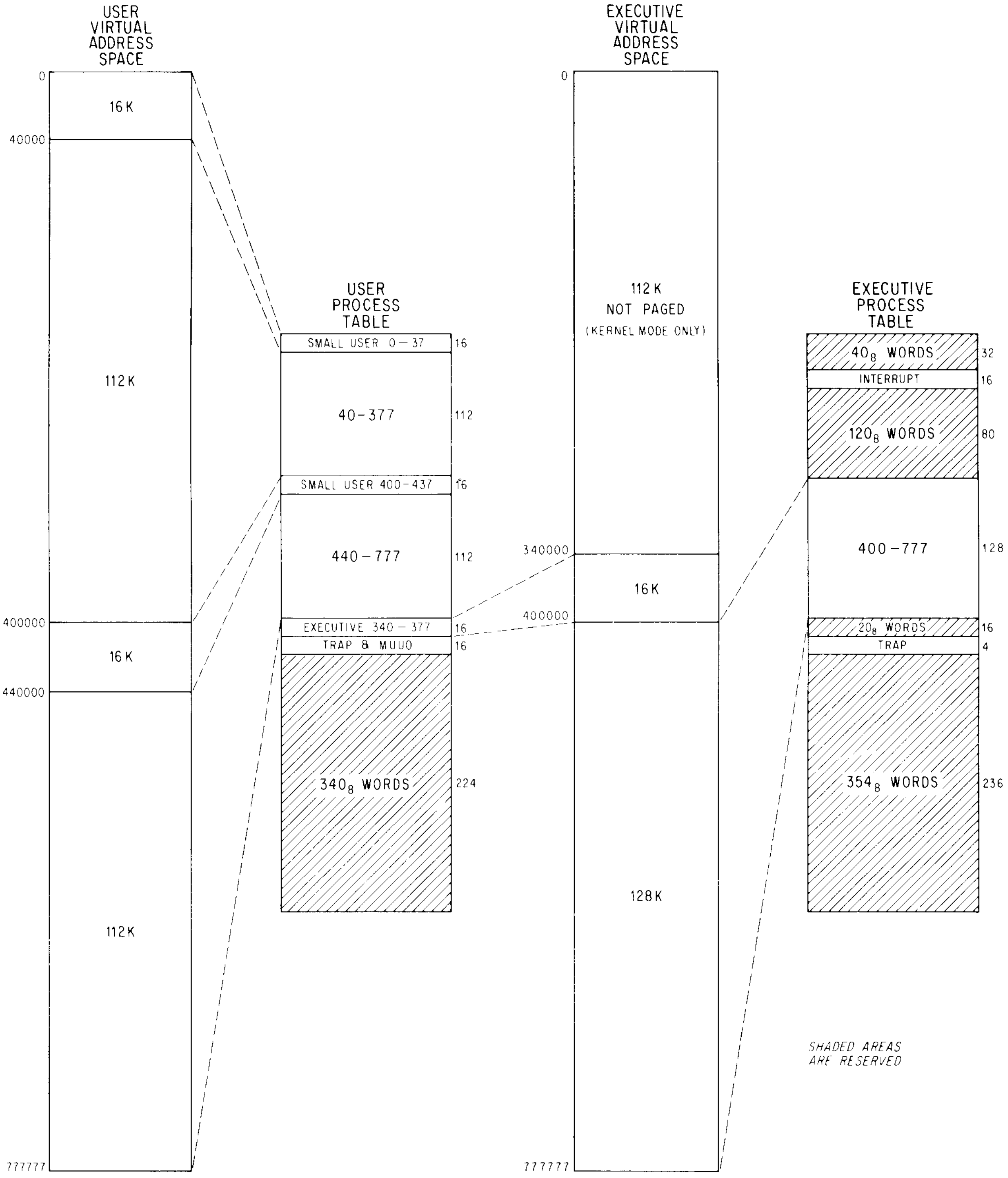


Figure 5.2: Process Table Configuration

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		
	RESERVED	
777		

EXECUTIVE PROCESS TABLE

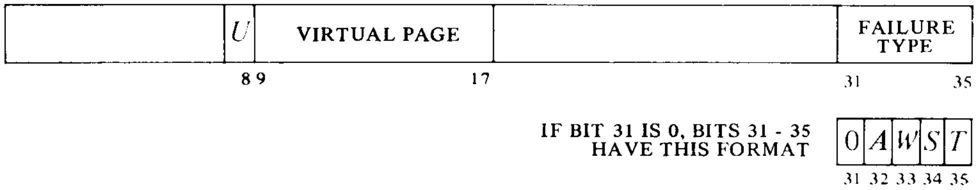
0		
	AVAILABLE TO SOFTWARE	
37		
40	EXECUTIVE LUUO STORED HERE	
41	LUUO HANDLER INSTRUCTION	
42		
	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
57		
60		
	RESERVED	
177		
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400		
	RESERVED	
417		
420	EXECUTIVE PAGE FAILURE TRAP INSTRUCTION	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE PUSHDOWN OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424		
	RESERVED	
777		

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 pager 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 (referred to as a "page refill cycle") 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 level 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.[5] 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.



Whether the violation occurred in user or executive virtual address space is indicated by a 1 or a 0 in bit 8. If bit 31 is 1, the number in bits 31–35 ($\geq$ 20) indicates the type of "hard" failure as follows.

23 Address failure — this is a simulated page failure caused by the satisfaction of an address condition selected from the console. It indicates that while the console address break switch was on and the Address Failure Inhibit flag was clear (bit 8 of the PC word), the processor initiated a page check for access to the memory location that was specified by the paging and address switches and for which a comparison was enabled (whether or not a comparison can be made is a function of the setting of the paging switches (Appendix F.1) and the state of the User Address Compare Enable flag (see below)), and the intended memory reference was for the purpose selected by the address condition switches as follows:

The instruction fetch switch was on and the requested access was for retrieval of an ordinary instruction, including an instruction executed by an XCT or an LUUO (address 41).

The data fetch switch was on and the requested access was for retrieval of an address word in an effective address calculation or read-only retrieval of an operand (other than in an XCT). This

[5] 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.

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.

switch can also cause a failure inadvertently[6] on the retrieval of a trap instruction or a PC word in an MUUO.

The write switch was on and the requested access was for writing,[6A] either write-only or read-modify-write, including writing by an LUUO (address 40). This switch also causes a failure on the first write in an MUUO if the address switches contain the effective address of the MUUO (even though that address is not used for the access), and can cause a failure inadvertently[6] on the second write.

The Address Failure Inhibit flag, which can be set only by a JRSTF or MUUO, prevents an address failure during the next instruction — the completion of the next instruction automatically clears it. If an interrupt or trap intervenes, the flag has no effect and it is saved and cleared if the PC word is saved. If it is not saved, it affects the instruction following the interrupt or trap. Otherwise it affects the instruction following a return in which it is restored with the PC word. Using this flag, the Monitor can return to a user instruction that caused an address failure and "get by it."

22 Page refill failure — this is a hardware malfunction. The paging hardware did not find the virtual page listed in the page table, so it loaded paging information from the page map into the table but still could not find it.

20 Small user violation — a small user has attempted to reference a location outside of the limited small user address space.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page or transfer control into a concealed page at an invalid entry point (one not containing a JRST 1,).

If the violation is not one of these, then bits 31–35 have the format shown above where *A*, *W* and *S* are simply the corresponding bits taken from the map half word for the page, and *T* indicates the type of reference in which the failure occurred — 0 for a read reference, 1 for a write or read-modify-write reference. The type of reference implies nothing about the cause of failure — it indicates only the reason the failed reference was being made.

The page fail trap instruction is set by the Monitor to transfer control to kernel mode. After rectifying the situation, the Monitor returns to the

[6] Virtual addresses are supplied to the paging hardware via the address bus. An inadvertent failure occurs when the bus is not used for an access, but it accidentally contains the number set into the address switches. The data fetch switch also catches the attempt to retrieve a dispatch interrupt instruction or inadvertently a standard interrupt instruction, but the page failure sets the In-out Page Failure flag instead of resulting in a trap for an address failure.

[6A] The write switch causes a failure on an instruction fetch if a read-modify-write precedes it immediately (e.g. if there is no intervening interrupt, the program is not being single stepped, etc).

interrupted instruction, which starts over again from the beginning.[7] 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.

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 writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered. Then in response to a write failure, the Monitor makes the page writable 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 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. If the Monitor shared block 0 with any users, it would have to store

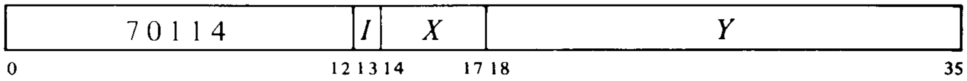
[7] In a soft page failure, the mapping entry for the page is removed from the page table on the assumption that the Monitor will change it. When the instruction is restarted, the hardware must go to the page map to get a new entry for the page table.

the user accumulators even when taking control only 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.

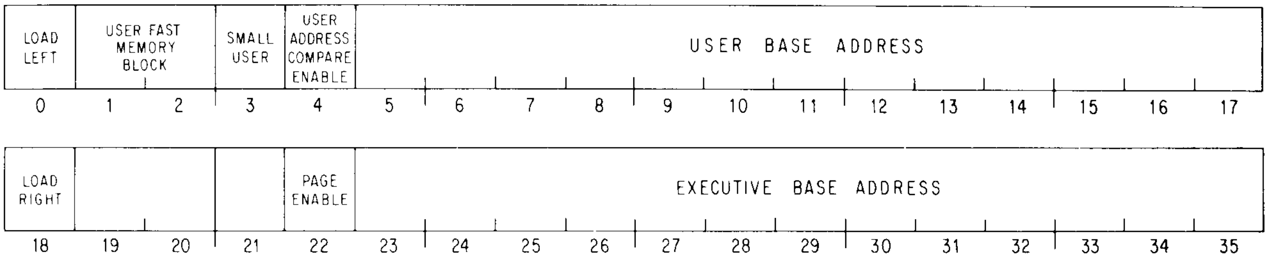
Even while User is set, the interrupt instructions are not part of the user program and are thus subject only to executive restrictions. (The page failure and overflow trap instructions are executed in the user address space if caused by the user.) As interrupt instructions, JSR, JSP and PUSHJ automatically take the processor out of user mode to jump to an executive service routine. An MUUO can also be used.

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

DATAO PAG, Data Out, Paging



Invalidate all data in the associative memory, and set up the paging hardware according to the contents of location *E* as shown. Invalidating all data in the associative memory means setting the Word Empty bit in each location to indicate that the rest of the word is meaningless and should not be used.



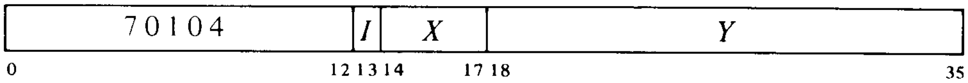
Bits 0 and 18 are change bits. If bit 0 is 0, ignore the rest of the left half word. But if bit 0 is 1, load bits 5–17 into the user base register to select the user process table, select the fast memory block specified by bits 1 and 2 for the user, limit the address space to that of a small user if bit 3 is 1, and enable address comparison if bit 4 is 1. The Address Compare Enable bit functions in conjunction with the console paging switches, as explained in Appendix F.1.

Similarly if bit 18 is 0, ignore the rest of the right half word. Otherwise load bits 23–35 into the executive base register to select the executive process table, and enable executive paging if bit 22 is 1. For normal operation of the system, bit 22 must be 1. A 0 in this bit disables overflow traps, and disables executive paging so there is no supervisor mode and no executive virtual addressing — in other words an executive program automatically runs in kernel mode with all access in the first 256K of physical memory unpaged.[8]

NOTE

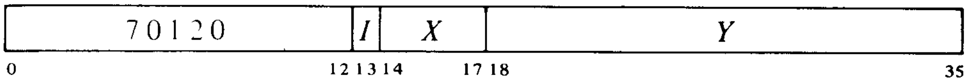
> Neither turning on power nor pressing the reset switch invalidates the data in the associative memory. Therefore, after power has been off, the starting kernel program must do a DATAO PAG, to clear the associative memory of random data before entering executive or user paged address space.

DATAI PAG, Data In, Paging

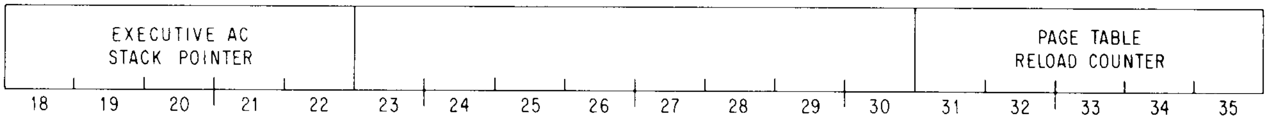


Read the status of the paging hardware into location *E*. The information read is the same as that supplied by a DATAO (bits 0 and 18 are 0).

CONO PAG, Conditions Out, Paging



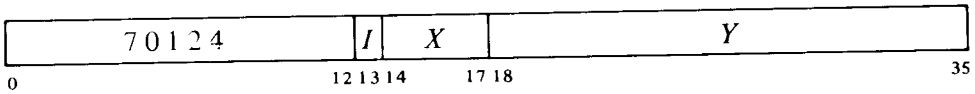
Load the executive stack pointer from bits 18–22 and the page table reload counter from bits 31–35 of the effective conditions *E* as shown.



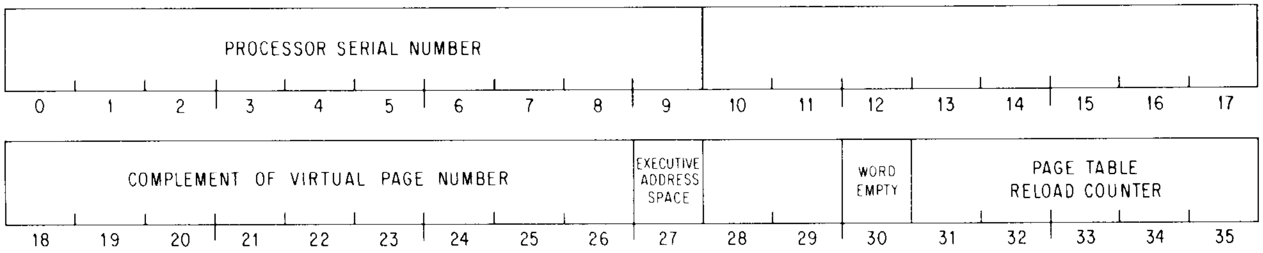
The executive stack pointer specifies a block of sixteen locations in the user process table by supplying the left five bits for a 9-bit address that references a location in the table; this function is used only for accessing stacked fast memory blocks in an instruction executed by a PXCT (see below). Loading the reload counter causes it to point to the specified location in the page table.

[8] An executive mode program that does not set bit 22 and avoids other special KI10 features will run on a KA10 as well. This is useful for hardware diagnostics and bootstrap loaders (see readin mode, §5.1).

CONI PAG, Conditions In, Paging

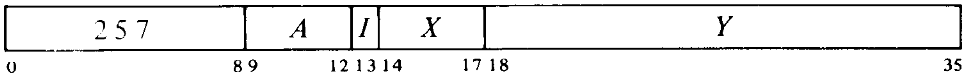


Read the processor serial number, the page table reload counter, and the contents of the location in the virtual page table specified by the counter into location *E* as shown.

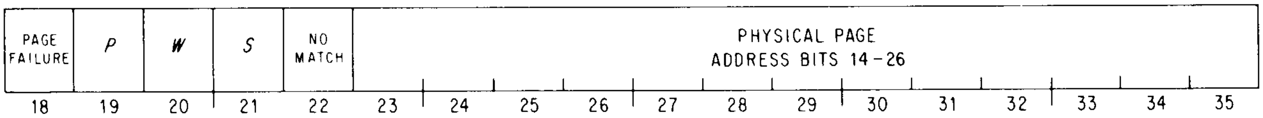


Note that bits 18–26 contain the complement of the virtual page number in the selected location. A 1 in bit 27 indicates the page is in the executive address space; a 1 in bit 30 means the information in bits 18–27 is invalid. It is possible for the reload counter to change between the CONI and the CONO, so the CONI might read a different location than was selected by the CONO.

MAP Map on Address



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, i.e. bits *P*, *W* and *S* in bits 19–21 and the physical page number in bits 23–35. Clear AC left.



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.

The last three instructions above 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.

Previous Context Execute

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. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero *A* bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous context. At any point in time, the previous context is essentially the circumstances in which the previous process was running. A PXCT can be given only in executive mode, but the previous context may be the user, as following a call to the Monitor by the user. The previous context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. But note that it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory operand references are made. To work as a PXCT, an XCT must be given in executive mode, and bits 11 and 12 in its *A* field (9–12) must not both be 0 (in user mode *A* is ignored). But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction all effective address calculation and accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a fast memory location.) 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.*

The previous context is specified by two flags. Just as the current mode is indicated by the User and Public flags, the mode in which the calling program was running is indicated by Previous Context User and Previous Context Public.[9] At a call these flags are set up by an MUUO PC word. Note that the restrictions on references made in the previous context are those of the previous context — not those in which the PXCT is given. Suppose the executive executes an instruction that references the concealed user area. Such a reference would fail if Previous Context Public were set; in other words the concealed area can be accessed by a PXCT only when such access is requested by the concealed program.

Which references in the executed instruction are made in the previous context is determined by 1s in bit 11 and 12 of the PXCT instruction word as follows: a 1 in bit 12 selects read and read-modify-write memory operand references; a 1 in bit 11 selects memory operand write references; and 1s in both bits selects all memory operand references. The meaning of previous context address space is obvious for core memory references, namely user or executive virtual address space. But this is not so for fast memory. When Previous Context User 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 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 Previous Context User is clear, fast memory references of the types specified 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,.

Previous Context Fast Memory References

Previous	*Fast Memory Block Selected*	
Context User	*Zero*	*Nonzero*
1	User shadow area	Selected user block
0	AC stack	AC stack

Individual Instruction Effects. The effects of execution by a PXCT on different types of instructions are as follows.

[9] Previous Context User and Previous Context Public are in the same flag bits that are used for User In-out and Overflow in user mode. The former has no meaning in executive mode, and the latter is not really necessary as the executive program is not ordinarily interested in performing extensive mathematical procedures.

- Instructions without memory operand references are not affected. This includes shifts, jumps, immmediate mode instructions, CONSO, CONO, and even an XCT. In fact not only is a PXCT not affected when executed by a PXCT, but the first destroys any effect the second would otherwise have on a third instruction (in other words, a pair of PXCTs 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, i.e. the load or deposit.

Philosophy. The purpose of the PXCT is to facilitate the handling of user requirements by the Monitor, but the selection made by Previous Context User of the references affected by the read and write bits is to allow the Monitor to make recursive calls to itself, i.e. to perform MUUOs in the process of carrying out an MUUO given by the user. Specifically the state of Previous Context User 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 Previous Context User 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 Previous Context User. 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 Previous Context User is left clear. The particular AC block used at any level is specified by the stack pointer, which makes a different set of sixteen words available at each level using the same adddresses. Hence the AC stack in the user process table is effectively a pushdown stack 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.

Example. Suppose that the Monitor has been called by an MUUO from the user (hence Previous Context User is set) and wishes to save the user's ACs in the shadow area. Assume that every user runs with AC block 1, 2 or 3, and that the Monitor always sets up executive virtual page 342 to point to the same physical page as user page 0. Using accumulator T in block 0, the Monitor saves the user ACs by giving these two instructions,

```
MOVEI    T,342000   ;Initialize pointer: from 0 to 342000
XCT      1,[BLT T,342017]
```

and restores them with these two.

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

5.5 KA10 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, i.e. 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 levels arranged in a priority chain, with assignment of devices to levels entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35). Levels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be connected to a single level, and some can be connected to two levels (e.g. a device may signal that data is ready on one level, that an error has occurred on another).

When a device requires service it sends an interrupt request signal over the in-out bus to its assigned level in the processor. The processor accepts the request depending upon certain conditions, such as that the level must be active (on). The request signal remains on the bus until turned off by the program (CONO, DATAO, or DATAI, depending on the device). Thus if a request is not accepted when made, it will be accepted when the conditions are satisfied. A single level 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 level 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 level 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. For example, 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.

Having accepted a request, the processor will do nothing further with it unless the priority interrupt system is on. But even with the system off, the processor will continue to accept requests on other levels; and when the system is finally turned on, it will respond as though all requests had just been accepted, handling the highest priority one first.

Starting an Interrupt. A request made to an active level is accepted at the next memory access unless the processor is starting an interrupt for any level or holding an interrupt for the same level. Once a request is accepted with the system on, the level must wait for the interrupt to start. The processor however cannot start an interrupt if it is already holding an interrupt for a level with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). When there is a higher priority level waiting, the processor stops the current program at the first allowable point to start an interrupt for the waiting level that has highest priority. Allowable stopping points are following the retrieval of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), and between transfers in a BLT. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

Two memory locations are associated with each level: unrelocated locations 40 + 2N and 41 + 2N, where N is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 57. The processor starts an interrupt for level N by executing the instruction in location 40 + 2N. Interrupt locations for a second processor on the same memory are 140 + 2N and 141 + 2N. Even though the processor may be in user mode when an interrupt occurs, interrupt instructions are performed in executive mode.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being performed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in location 40 + 2*N* or 41 + 2*N*, in direct response by the hardware (rather than by the program) to a request on level *N*. §2.18 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. There are two categories of interrupt instructions.

Non-IO Instructions. After executing a non-IO interrupt instruction, the processor holds an interrupt on the level 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).

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. For example, 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 + 2*N* is a BLKI or BLKO and the block is not finished (i.e. 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.

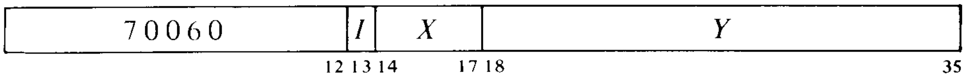
If the instruction in 40 + 2*N* is a BLKI or BLKO and the count does reach zero, the processor executes the instruction in location 41 + 2*N*. This cannot be an IO instruction and the actions that result from its execution as an interrupt instruction are those given above for non-IO instructions.

CAUTION

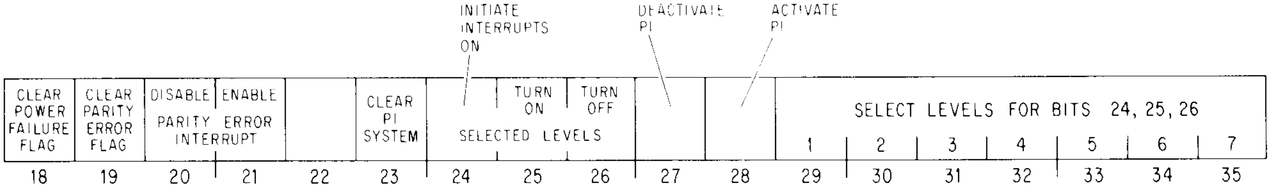
The execution, as an interrupt instruction, of a CONO, CONI, CONSO or CONSZ in location 40 + 2*N* or *any* IO instruction in location 41 + 2*N* hangs up the processor.

Interrupt Programming. The program can control the interrupt system by means of condition IO instructions. The device code is 004, mnemonic PI.

CONO PI, Conditions Out, Priority Interrupt



Perform the functions specified by the effective conditions E as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



Bits 18–21 are actually for processor conditions (§5.6).

20 Prevent the setting of the Parity Error flag from requesting an interrupt on the level assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the level assigned to the processor.

23 Deactivate the priority interrupt system, turn off all levels, 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 levels selected by 1s in bits 29–35, and force the processor to accept them even on levels that are off. There is at most one interrupt on a given level, and a request is lost if it is made by this means to a level on which an interrupt is already being held.

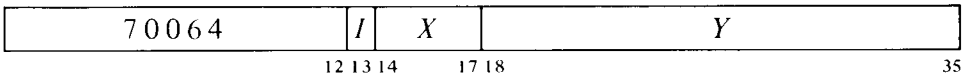
25 Turn on the levels selected by 1s in bits 29–35 so interrupt requests can be accepted on them.

26 Turn off the levels selected by 1s in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.

28 Activate the priority interrupt system so the processor can accept requests and can start, hold and dismiss interrupts.

CONI PI, Conditions In, Priority Interrupt



Read the status of the priority interrupt (and several bits of processor conditions) into location E as shown.

POWER FAILURE	PARITY ERROR	PARITY ERROR INTERRUPT ENABLED	INTERRUPT IN PROGRESS ON LEVELS							PI SYSTEM ON	LEVELS ON (ACTIVE)						
			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

Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held. A 1 in bit 28 means the interrupt system is on.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18–20 actually read processor status conditions (§5.6) as follows.

18 Ac power has failed. The program should save PC, the flags and fast memory in core, and halt the processor. Note that PC may point to an interrupt service routine rather than the main program.

The setting of this flag requests an interrupt on the level 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 level assigned to the processor, at which time PC points to the instruction being performed or to the one following it.

Dismissing an Interrupt. Automatic dismissal of an interrupt occurs only in a DATAI or DATAO, or in a BLKI or BLKO with an incomplete block. Following any non-IO interrupt instruction, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that level or any level of lower priority (requests, however, can be accepted on lower priority levels).

A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be on when the JEN is given). This instruction restores the level on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority levels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, or PUSHJ. If flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

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

Timing. The time a device must wait for an interrupt to start depends on the number of levels in use, and how long the service routines are for devices on higher priority levels. If only one device is using interrupts, it need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is made. The maximum time can be considered to be about 15 μs for FDVL, but a ridiculously long shift could take over 35 μs.

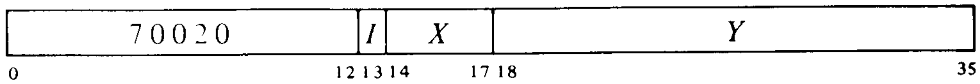
Special Considerations and Programming Suggestions. If the interrupt routine uses a UUO it must first save the contents of the pair of locations that will be changed by it in case the interrupted program was in the process of handling a UUO. Hence the routine must save unrelocated location 40 and the location used by the UUO handler instruction to store the PC word. In all other respects, the special considerations and programming suggestions given at the end of the section on the KI10 interrupt hold for the KA10 (§5.2).

5.6 KA10 Processor Conditions

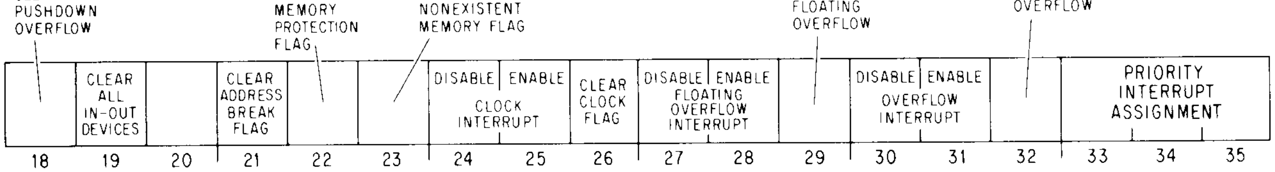
There are a number of internal conditions that can signal the program by requesting an interrupt on a level assigned to the processor. Most of these conditions are generally regarded as important enough to be assigned to the highest priority level. Except in the case of a power failure however, the common practice is for the processor interrupt to switch over to the lowest priority level by means of a program-set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

Flags for power failure and parity error are handled by the condition IO instructions that address the priority interrupt system (§5.5). 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 level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 18–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



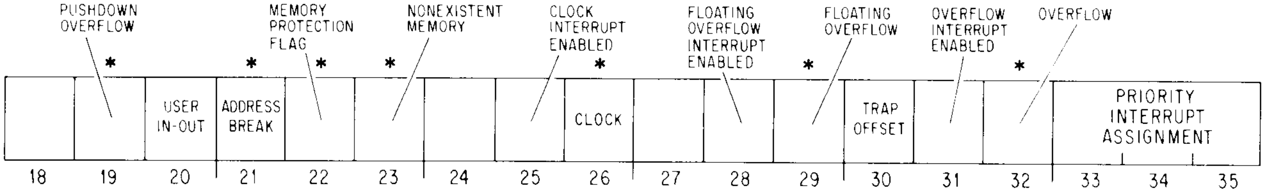
Enabling a particular flag to interrupt means that henceforth the setting of the flag will request an interrupt on the level assigned (by bits 33–35) to the processor. Disabling prevents the flag from triggering a request.

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

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

Read the status of the processor into the right half of location *E* as shown (all interrupt requests are made on the level assigned to the processor).



Bits that can cause interrupts on the level assigned to the processor are those indicated by asterisks, and also Power Failure and Parity Error, bits 18 and 19 read by a CONI PI,.

With the possible exception of an illegal memory reference on an instruction fetch, if the highest priority active level is assigned to the processor, then the occurrence of any processor interrupt condition is guaranteed to produce a processor interrupt with no lower priority interrupt intervening between it and the program level at which the processor condition occurred. The actual relationship between PC and the instruction associated with a given condition is as stated in its description.

19 Pushdown Overflow — in a PUSH or PUSHJ the count in AC left reached zero; or in a POP or POPJ the count reached -1. The setting of this flag requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred. The location of the offending instruction is implied by PC for PUSH or POP, is indicated by the last item in the stack for PUSHJ, but is indeterminate for POPJ.

20 User In-out — even if the processor is in user mode, there are no instruction restrictions (but memory restrictions still apply) (§5.7).

21 Address Break — while the console address break switch was on, the processor requested access to the memory location specified by the address switches and the memory reference was for the purpose selected by the address condition switches as follows:

The instruction switch was on and access was for retrieval of an instruction (including an instruction executed by an XCT or contained in an interrupt location or a trap for an unimplemented operation) or an address word in an effective address calculation.

The data fetch switch was on and access was for retrieval of an operand (other than in an XCT).

The write switch was on and access was for writing a word in memory, other than in a read-modify-write.

The setting of this flag requests an interrupt, at which time PC points to the instruction that was being executed or to the one following it. However PC bears no relation to the break if the access was requested for a console key function.

22 Memory Protection — a user program attempted to access a memory location outside of its area or to write in a write-protected part of its area, and the user instruction was terminated at that time. The setting of this flag requests an interrupt, at which time PC points either to the instruction that caused the violation or to the one following it, unless the illegal reference was for fetching an instruction. In this exceptional case it is possible for a lower level interrupt to occur between the violation and its interrupt, even with the processor assigned to the highest priority active level.

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 flag requests an interrupt, at which time PC points either to the instruction containing the unanswered reference or to the one following it. However 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 in §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 in §2.9. If bit 31 is set, the setting of Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.

CAUTION

For an address break, a memory protection violation, a parity error, or a nonexistent memory, a processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the processor will fetch the next user instruction before it accepts the program-set interrupt request. This makes it very likely that the same error will recur, producing a loop between the processor interrupt and the interrupted program.

5.7 KA10 Program and Memory Management

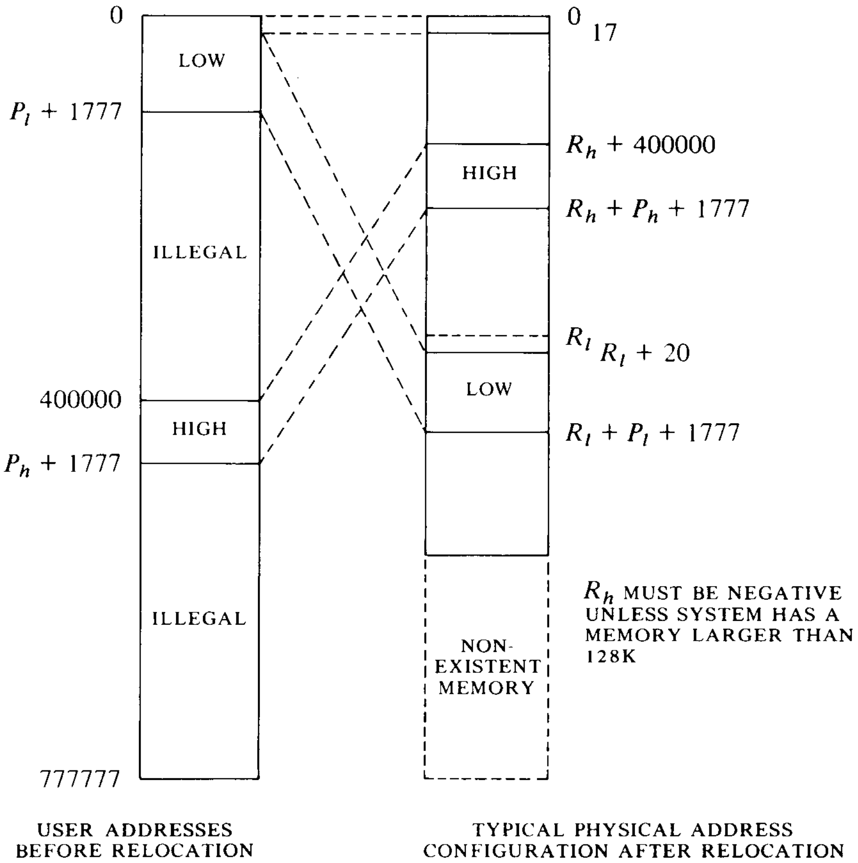
Every user is assigned a core area and the rest of core is protected from him — he cannot gain access to the protected area for either storage or retrieval of information. The assigned area is divided into two parts. The low part is unique to a given user and can be used for any purpose. The high part 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, i.e. he cannot write anything in it. The Monitor would do this when the high part is to be a pure procedure to be used reentrantly by several users. One high pure segment may be used with any number of low impure segments. The user can request that the Monitor write-protect the high part of a single program, e.g. 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 (i.e. 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, i.e. more than half the maximum memory capacity ($P_l \geqslant 400000$), the high part starts at the first location after the low part (at location P_l + 2000). The high part is limited to

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.


0
LOW
P_l + 1777
ILLEGAL
400000
HIGH
P_h + 1777
ILLEGAL
777777
0
17
R_h + 400000
HIGH
$R_h + P_h$ + 1777
R_l
R_l + 20
LOW
$R_l + P_l$ + 1777
NON-EXISTENT MEMORY
R_h MUST BE NEGATIVE UNLESS SYSTEM HAS A MEMORY LARGER THAN 128K
USER ADDRESSES BEFORE RELOCATION
TYPICAL PHYSICAL ADDRESS CONFIGURATION AFTER RELOCATION

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 level assigned to the processor (§5.6).

Addressing Summary. Let A_u be the address supplied by the user, and let A_p be the physical core address generated from it by the relocation hardware.

If $A_u \leqslant 17$, then $A_p = A_u$ (fast memory, no relocation).

If $20 \leqslant A_u \leqslant P_l + 1777$, then $A_p = (A_u + R_l) \bmod 2^{18}$.

If the greater of $\left\{ \begin{matrix} 400000 \\ P_l + 2000 \end{matrix} \right\} \leqslant A_u \leqslant P_h + 1777$,

then $A_p = (A_u + R_h) \bmod 2^{18}$.

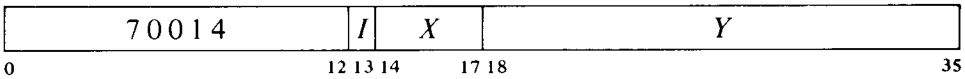
Any other value of A_u is illegal. These are $A_u < P_l + 1777$ if either $A_u <$ 400000 or $A_u > P_h + 1777$.

Note: If a relocated address is in the range 0–17, the reference is to core rather than fast memory.

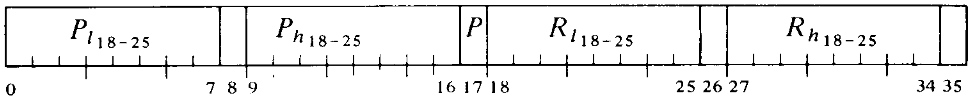
Monitor Programming

The Monitor must assign the core area for each user program, set up trap and interrupt locations, specify whether the user can give IO instructions, transfer control to the user program, and respond appropriately when an interrupt occurs or an instruction is executed in unrelocated 41 or 61. Core assignment is made by this instruction.

DATAO APR, Data Out, Arithmetic Processor



Load the protection and relocation registers from the contents of location E as shown, where P_l, P_h, R_l and R_h are the protection and relocation addresses defined above. If write-protect bit P (bit 17) is 1, do not allow the user to write in the high part of his area.



Notes. 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.

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. (The trap locations are 140-141 and 160-161 in a second KA10 processor.) 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.

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.

5.8 Real Time Clock DK10

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[10] 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 loaded by the program. Each time the count reaches the number held in the register, the clock requests an interrupt while the counter clears and begins a new count. With the internal clock source, whose period is 10 μs, the total count is about 2.6 seconds.

The program turns the clock on and off by enabling and disabling the counter. The clock has two modes of operation: with the User Time flag clear, the counter operates continuously; with User Time set, the counter stops while the processor is handling interrupts. Hence in the latter mode the clock discounts interrupt time and can be used to time user programs. In a system that contains two clocks, one can be used by the Monitor to time user programs while the other is used to keep the time of day.

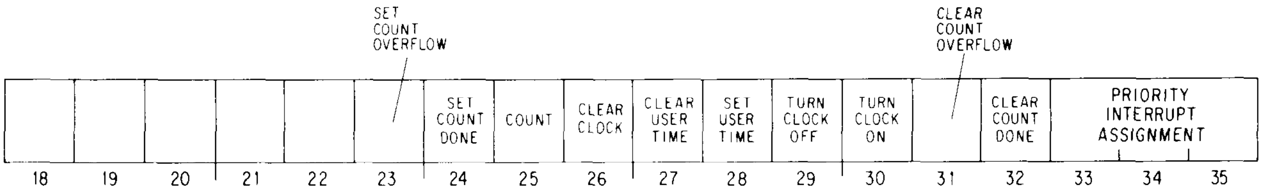
Instructions. The clock device code is 070, mnemonic CLK. A second clock would have device code 074.

CONO CLK, Conditions Out, Clock

70720	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt level 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).

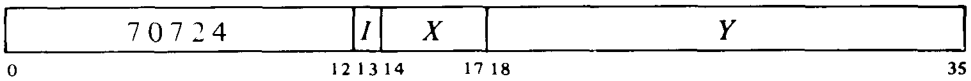
[10] The clock referred to throughout this section is the DK10 real time clock and should not be confused with the line frequency clock whose flag is one of the processor conditions (§5.3 or §5.6).



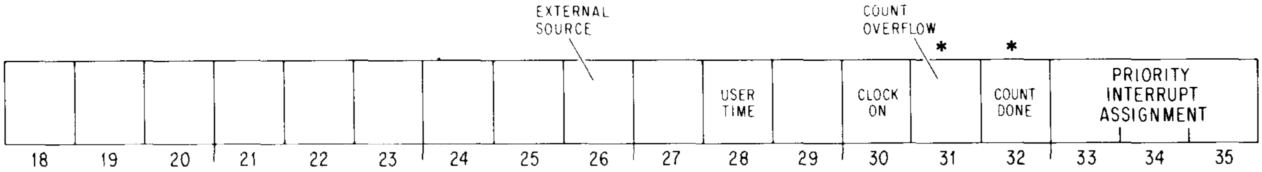
A 1 in bit 26 clears the clock counter and the Count Done, Count Overflow and User Time flags, turns off the clock, and drops the PI assignment (assigns zero). The effect of giving conflicting conditions is indeterminate.

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 (asterisks indicate bits that can cause interrupts).

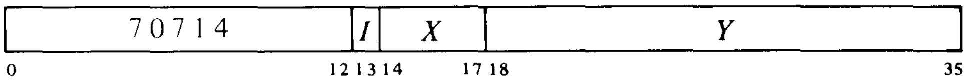


Interrupts are requested on the assigned level by the setting of Count Overflow and Count Done.

26 The counter is connected to an external source (0 indicates the internal source is connected).

28 The counter cannot be incremented while an interrupt is being held or a request has been accepted and the level is waiting for an interrupt to start. Note that to time a user properly, the Monitor must also compensate for any noninterrupt time taken from the user.

DATAO CLK, Data Out, Clock



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

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

DATAI CLK, Data In, Clock

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

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

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

Initially the program should give a CONO CLK,1000 to clear the clock, and then give a DATAO to select the interval and a CONO to turn on the clock, select the mode, and assign the interrupt level. Following turnon the first count may occur at any time up to the full period of the source. When the count reaches the specified interval, Count Done sets, requesting an interrupt on the assigned level. 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. Remember that although a CONO need not affect the mode or the clock state, every CONO must renew the PI assignment.

The interval can be changed at any time simply by giving a DATAO. However, if the program does not clear the counter at the same time, then it should make sure that the count has not yet reached the value of the new interval. If the count is already beyond that point, the counter will continue until it overflows. When the counter overflows, either because the count started too high, the program specified the maximum count (2^{18} is selected by loading zero), or there is a malfunction of some sort, Count Overflow sets, requesting an interrupt, and a new count begins.

To use the clock to time some operation, turn it on with the counter at zero. For a counter reading of *C*, the elapsed time is

$$T(C + nI)$$

where T is the period of the source, n is the number of clock interrupts since the clock was started, and I is the interval selected by the program. To cause the clock to request an interrupt after $T \times n$ μs, where $n \leq 2^{18}$ and T is the period of the source in microseconds, load the interval register with n expressed in binary. There is an average indeterminacy of half a count every time the counter starts and stops. Therefore, when the clock is keeping user time, there is an average indeterminacy of one count for every *group* of overlapping interrupts and requests (not for every interrupt, as the counter is inhibited while there is any request or interrupt being held).

For keeping the time of day, the program can use a memory location to maintain a count of the clock interrupts. The location should be cleared at midnight — note that an error of .01% amounts to 8.64 seconds in 24 hours — 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.

Appendix A
Instructions and Mnemonics

The drawing on the next two pages shows the formats of the various types of instructions, pointers, arithmetic operands, and other special words employed by the user in the KL10 and KS10 processors. On the two pages following this drawing is a similar illustration for the KI10 and KA10. The chart on pages A–6 and A–7 shows the derivation of the instruction mnemonics. Next are two tables that list all instruction mnemonics and their octal codes both numerically and alphabetically. For completeness, the tables include the MUUOs (indicated by an asterisk) that are recognized by MACRO for communication with the TOPS–10 or TOPS–20 Monitor (only JSYS is applicable to the latter). The great majority of the instruction codes are the standard ones in the PDP–10 instruction set, and they are available in all processors. A dagger (†) indicates a now-standard instruction code that is available in the KL10 and KS10 (and can be expected to be available in all future processors) but is unassigned in the earlier processors. Similarly a double dagger (‡) indicates an instruction that became available in KL10 microcode version 271 and will be in future machines, but is unassigned in all other circumstances. Footnotes explain other special situations and variations from one processor to another. KL10 codes listed as limited to a TOPS–10 system are actually a function of the individual microcode. Blanks in the numeric table are for codes not assigned in any processor, except that JRST blanks actually correspond to function combinations on the KI10 and KA10. (Note that 247 and 257 are executed as no-ops on the KA10.)

Device codes (which are of course meaningful only in the context of pre-KS10 IO instructions) are included in the listings only for internal devices, and they are indented to match their position in an instruction word. Combinations of IO mnemonics with internal device codes are included only in the numeric listing, as the numeric values for all combinations appropriate to each device are readily available in Appendix C.

Beginning on page A–16 is a list of all instructions showing their actions in symbolic form. On page A–27 is a table of the positive and negative powers of 2.

BASIC INSTRUCTIONS

INSTRUCTION CODE (INCLUDING MODE)	*A,F*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

KL10 IN–OUT INSTRUCTIONS

1 1 1	DEVICE CODE	INSTRUCTION CODE	*I*	*X*	*Y*
0 2	3 9	10 12	13	14 17	18 35

INSTRUCTIONS EXECUTED UNDER EXTEND

INSTRUCTION CODE	0 0 0 0	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

LOCAL INDIRECT WORD

1	0	RESERVED	*I*	*X*	*Y*
0	1		13	14 17	18 35

GLOBAL INDIRECT WORD

0	*I*	*X*	*Y*
0	1	2 5	6 35

LOCAL INDEX REGISTER

IN NONZERO SECTION MUST BE ≤0 OR BITS 6–17 = 0	LOCAL INDEX
0	18 35

GLOBAL INDEX REGISTER

0 0 0 0 0 0	GLOBAL INDEX WITH NONZERO SECTION NUMBER
0 5	6 35

SAVED FLAGS

OVERFLOW PREVIOUS CONTEXT✱ PUBLIC	CARRY 0	CARRY 1	FLOATING OVERFLOW	FIRST PART DONE	USER	USER IN–OUT PREVIOUS CONTEXT USER	✱ PUBLIC	✱ ADDRESS FAILURE INHIBIT	TRAP 2	TRAP 1	FLOATING OVERFLOW	NO DIVIDE
0	1	2	3	4	5	6	7	8	9	10	11	12

✱ KL10 ONLY

PC WORD

FLAGS	0 0 0 0 0	IN–SECTION PC
0 12	13 17	18 35

FLAG–PC DOUBLEWORD

FLAGS	0 0 0 0 0	PROCESSOR–DEPENDENT INFORMATION
0 0 0 0 0 0		
0 5 12	13 17	18 35

LOCAL STACK POINTER

CONTROL COUNT (IN NONZERO SECTION = 0 OR BITS 6–17 = 0)	IN–SECTION ADDRESS OF LAST ITEM
0 17	18 35

GLOBAL STACK POINTER

0 0 0 0 0 0	ADDRESS OF LAST ITEM (NONZERO SECTION)
0 5	6 35

ONE–WORD LOCAL BYTE POINTER

POSITION *P*	SIZE *S*	0	*I*	*X*	*Y*
0 5	6 11	12	13	14 17	18 35

ONE–WORD GLOBAL BYTE POINTER

P&S	30–BIT ADDRESS
0 5	6 35

TWO–WORD BYTE POINTER

POSITION *P*	SIZE *S*	1	RESERVED	AVAILABLE TO USER
INDIRECT WORD (GLOBAL OR LOCAL)				
0 1 2 5	6 11	12	13 17	18 35

BYTE STORAGE

	S BITS	*P* BITS	
	BYTE	NEXT BYTE	
0	35–*P*–*S*+1 35–*P*	35–*P*+1	

FIXED POINT OPERANDS (SINGLE PRECISION OR HIGH ORDER WORD)

SIGN 0 + 1 –	BINARY NUMBER (TWOS COMPLEMENT)
0	1 35

LOWER ORDER WORDS IN DOUBLE LENGTH FIXED POINT OPERANDS

SIGN 0 + 1 –	LOWER ORDER PART OF BINARY NUMBER (TWOS COMPLEMENT)
0	1 35

STANDARD RANGE FLOATING POINT OPERANDS (SINGLE PRECISION OR HIGH ORDER WORD)

SIGN 0 + 1 –	EXCESS 128 EXPONENT (ONES COMPLEMENT)	FRACTION (TWOS COMPLEMENT)
0	1 8	9 35

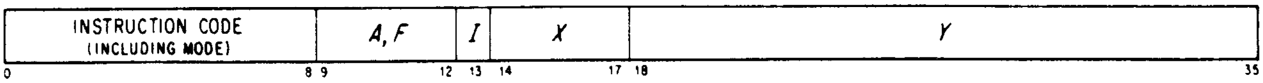
EXPANDED RANGE FLOATING POINT OPERANDS (HIGH ORDER WORD)

SIGN 0 + 1 –	EXCESS 2048 EXPONENT (ONES COMPLEMENT)	FRACTION (TWOS COMPLEMENT)
0	1 11	12 35

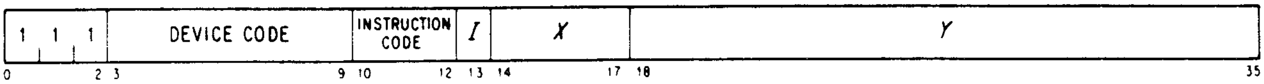
LOWER ORDER WORDS IN MULTIPLE LENGTH FLOATING POINT OPERANDS

0	LOWER ORDER EXTENSION OF FRACTION (TWOS COMPLEMENT)
0	1 35

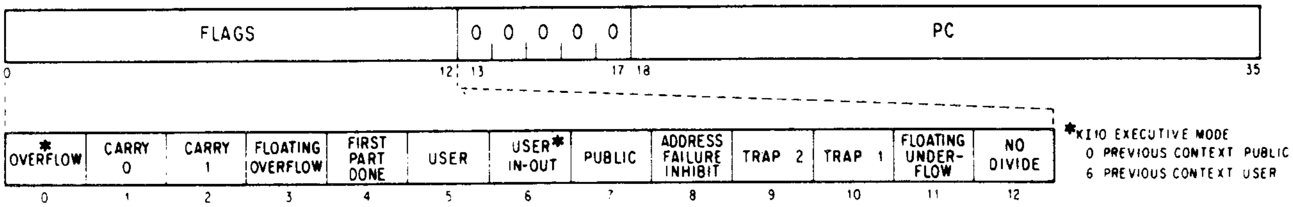
BASIC INSTRUCTIONS



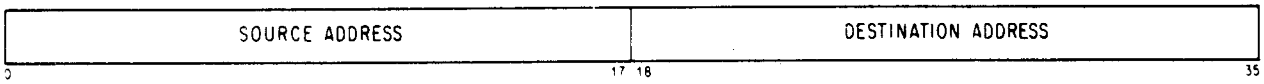
IN-OUT INSTRUCTIONS



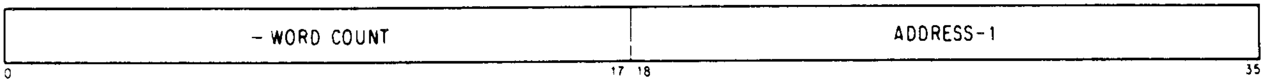
PC WORD



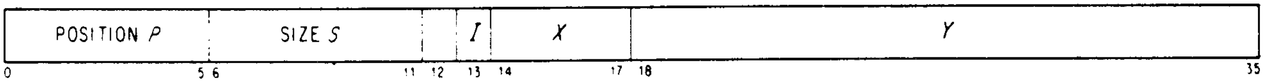
BLT POINTER {XWD}



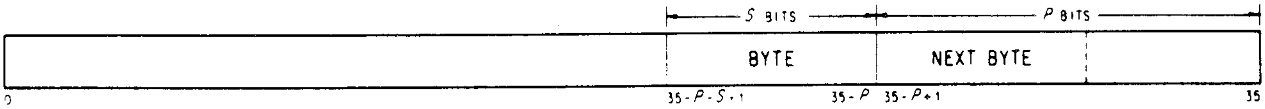
BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



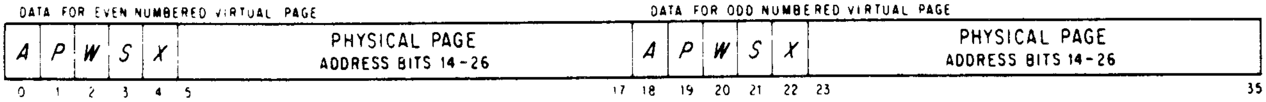
BYTE POINTER



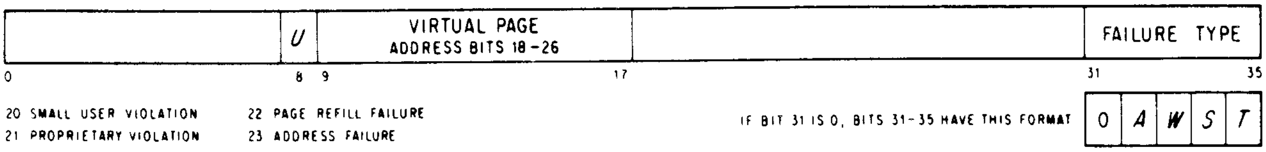
BYTE STORAGE



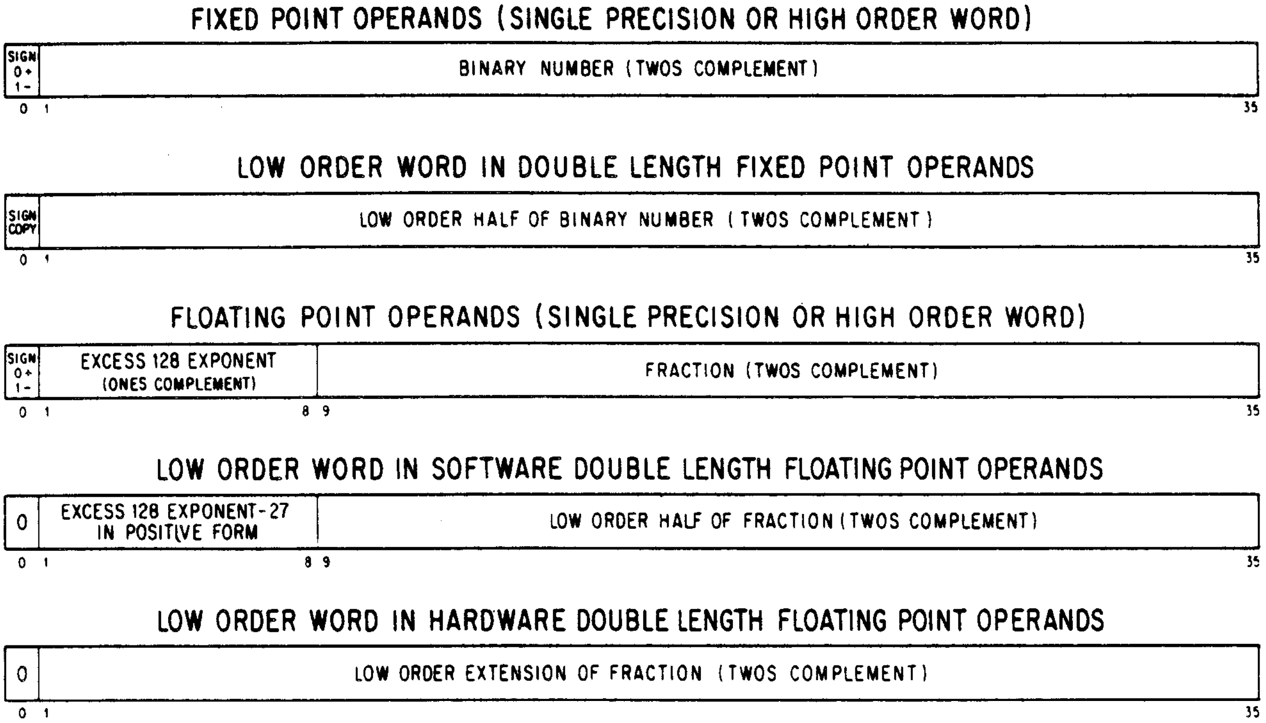
PAGE MAP WORD



PAGE FAIL WORD



KI10 and KA10 Formats — Instruction and Control Words



FIXED POINT OPERANDS (SINGLE PRECISION OR HIGH ORDER WORD)

SIGN 0+ 1-
BINARY NUMBER (TWOS COMPLEMENT)
0 1
35

LOW ORDER WORD IN DOUBLE LENGTH FIXED POINT OPERANDS

SIGN COPY
LOW ORDER HALF OF BINARY NUMBER (TWOS COMPLEMENT)
0 1
35

FLOATING POINT OPERANDS (SINGLE PRECISION OR HIGH ORDER WORD)

SIGN 0+ 1-
EXCESS 128 EXPONENT (ONES COMPLEMENT)
FRACTION (TWOS COMPLEMENT)
0 1
8 9
35

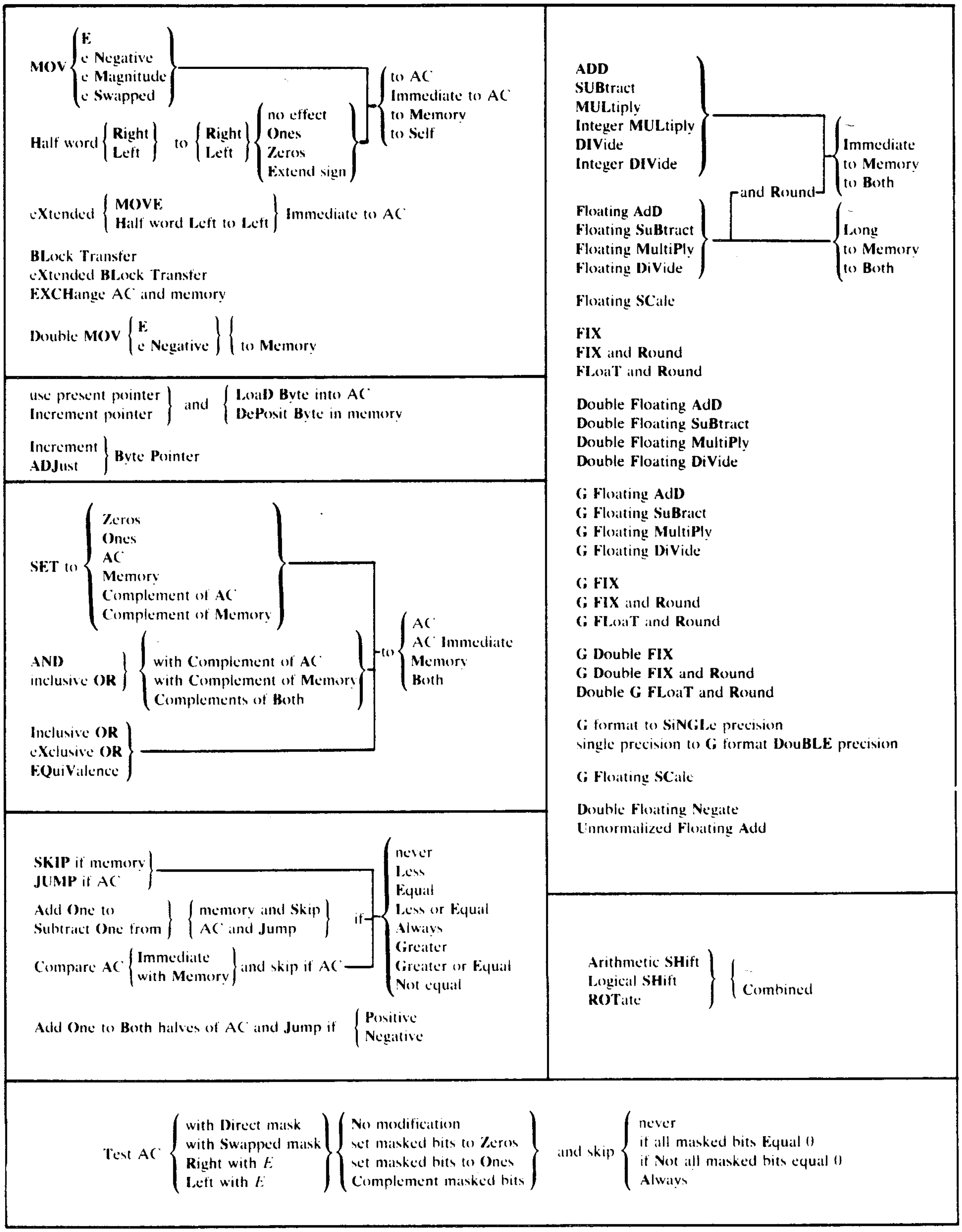
LOW ORDER WORD IN SOFTWARE DOUBLE LENGTH FLOATING POINT OPERANDS

0
EXCESS 128 EXPONENT-27 IN POSITIVE FORM
LOW ORDER HALF OF FRACTION (TWOS COMPLEMENT)
0 1
8 9
35

LOW ORDER WORD IN HARDWARE DOUBLE LENGTH FLOATING POINT OPERANDS

0
LOW ORDER EXTENSION OF FRACTION (TWOS COMPLEMENT)
0 1
35

KI10 and KA10 Formats — Arithmetic Operands



MOV
E
e Negative
e Magnitude
e Swapped

Half word
Right
Left
to
Right
Left
no effect
Ones
Zeros
Extend sign

to AC
Immediate to AC
to Memory
to Self

eXtended
MOVE
Half word Left to Left
Immediate to AC

BLock Transfer
eXtended BLock Transfer
EXCHange AC and memory

Double MOV
E
e Negative
to Memory

use present pointer
Increment pointer
and
LoaD Byte into AC
DePosit Byte in memory

Increment
ADJust
Byte Pointer

SET to
Zeros
Ones
AC
Memory
Complement of AC
Complement of Memory

AND
inclusive OR
with Complement of AC
with Complement of Memory
Complements of Both

Inclusive OR
eXclusive OR
EQuiValence

to
AC
AC Immediate
Memory
Both

SKIP if memory
JUMP if AC

Add One to
Subtract One from
memory and Skip
AC and Jump

Compare AC
Immediate
with Memory
and skip if AC

if
never
Less
Equal
Less or Equal
Always
Greater
Greater or Equal
Not equal

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

ADD
SUBtract
MULtiply
Integer MULtiply
DIVide
Integer DIVide

and Round

–
Immediate
to Memory
to Both

Floating AdD
Floating SuBtract
Floating MultiPly
Floating DiVide

–
Long
to Memory
to Both

Floating SCale

FIX
FIX and Round
FLoaT and Round

Double Floating AdD
Double Floating SuBtract
Double Floating MultiPly
Double Floating DiVide

G Floating AdD
G Floating SuBract
G Floating MultiPly
G Floating DiVide

G FIX
G FIX and Round
G FLoaT and Round

G Double FIX
G Double FIX and Round
Double G FLoaT and Round

G format to SiNGLe precision
single precision to G format DouBLE precision

G Floating SCale

Double Floating Negate
Unnormalized Floating Add

Arithmetic SHift
Logical SHift
ROTate
–
Combined

Test AC
with Direct mask
with Swapped mask
Right with E
Left with E

No modification
set masked bits to Zeros
set masked bits to Ones
Complement masked bits

and skip
never
if all masked bits Equal 0
if Not all masked bits equal 0
Always

PUSH
POP
~
and Jump

ADJust Stack Pointer

Jump
to SubRoutine
and Save PC
and Save AC
and Restore AC
if Find First One
on Flag and CLear it
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

MOVe String
Left Justified
Right Justified
Offset
Translated

CoMPare Strings and skip if
Less
Equal
Less or Equal
Greater
Greater or Equal
Not equal

ConVert
Decimal to Binary
Binary to Decimal
Offset
Translated

EDIT string

DATA
BLocK
CONditions
In
Out
in and Skip if
all masked bits Zero
some masked bit One

Bit Set
Bit Clear
ReaD
WRite
In-Out
Test In-Out
Equal
Not equal
~
Byte

INSTRUCTION MNEMONICS

NUMERIC LISTING

000	ILLEGAL
001 ⋮ 037	LUUO'S
CODES UNDER EXTEND	
001	†CMPSL
002	†CMPSE
003	†CMPSLE
004	†EDIT
005	†CMPSGE
006	†CMPSN
007	†CMPSG
010	†CVTDBO
011	†CVTDBT
012	†CVTBDO
013	†CVTBDT
014	†MOVSO
015	†MOVST
016	†MOVSLJ
017	†MOVSRJ
020	†XBLT
021	‡GSNGL
022	‡GDBLE
023	‡GDFIX
024	‡GFIX
025	‡GDFIXR
026	‡GFIXR
027	‡DGFLTR
030	‡GFLTR
031	‡GFSC
ALL OTHERS UNASSIGNED	
040	*CALL
041	*INIT
042 043 044 045 046	RESERVED MUUO'S
047	*CALLI
050	*OPEN
051	*TTCALL
052	
053	
054	
055	*RENAME
056	*IN
057	*OUT
060	*SETSTS
061	*STATO
062	*STATUS
062	*GETSTS
063	*STATZ
064	*INBUF
065	*OUTBUF
066	*INPUT
067	*OUTPUT
070	*CLOSE
071	*RELEAS
072	*MTAPE
073	*UGETF
074	*USETI
075	*USETO
076	*LOOKUP
077	*ENTER
100	*UJEN
101	
102	‡GFAD
103	‡GFSB
104	*JSYS
105	†ADJSP
106	‡GFMP
107	‡GFDV
110	[1]DFAD
111	[1]DFSB
112	[1]DFMP
113	[1]DFDV
114	†DADD
115	†DSUB
116	†DMUL
117	†DDIV
120	[1]DMOVE
121	[1]DMOVN
122	[1]FIX
123	†EXTEND
124	[1]DMOVEM
125	[1]DMOVNM
126	[1]FIXR
127	[1]FLTR
130	[2]UFA
131	[2]DFN
132	FSC
133	IBP ADJBP
134	ILDB
135	LDB
136	IDPB
137	DPB
140	FAD
141	[2]FADL
142	FADM
143	FADB
144	FADR
145	FADRI
146	FADRM
147	FADRB
150	FSB
151	[2]FSBL
152	FSBM
153	FSBB
154	FSBR
155	FSBRI
156	FSBRM
157	FSBRB
160	FMP
161	[2]FMPL
162	FMPM
163	FMPB
164	FMPR
165	FMPRI
166	FMPRM
167	FMPRB
170	FDV
171	[2]FDVL
172	FDVM
173	FDVB
174	FDVR
175	FDVRI
176	FDVRM
177	FDVRB
200	MOVE
201	MOVEI
202	MOVEM
203	MOVES
204	MOVS
205	MOVSI
206	MOVSM
207	MOVSS
210	MOVN
211	MOVNI
212	MOVNM
213	MOVNS
214	MOVM
215	MOVMI
216	MOVMM
217	MOVMS

[1] Not available in KA10.

[2] Used only in KA10, KI10 and TOPS–10 KL10.

220	IMUL	266	JSA	360	SOJ
221	IMULI	267	JRA	361	SOJL
222	IMULM	270	ADD	362	SOJE
223	IMULB	271	ADDI	363	SOJLE
224	MUL	272	ADDM	364	SOJA
225	MULI	273	ADDB	365	SOJGE
226	MULM	274	SUB	366	SOJN
227	MULB	275	SUBI	367	SOJG
230	IDIV	276	SUBM	370	SOS
231	IDIVI	277	SUBB	371	SOSL
232	IDIVM	300	CAI	372	SOSE
233	IDIVB	301	CAIL	373	SOSLE
234	DIV	302	CAIE	374	SOSA
235	DIVI	303	CAILE	375	SOSGE
236	DIVM	304	CAIA	376	SOSN
237	DIVB	305	CAIGE	377	SOSG
240	ASH	306	CAIN	400	SETZ
241	ROT	307	CAIG	401	SETZI
242	LSH	310	CAM	402	SETZM
243	JFFO	311	CAML	403	SETZB
244	ASHC	312	CAME	404	AND
245	ROTC	313	CAMLE	405	ANDI
246	LSHC	314	CAMA	406	ANDM
247		315	CAMGE	407	ANDB
250	EXCH	316	CAMN	410	ANDCA
251	BLT	317	CAMG	411	ANDCAI
252	AOBJP	320	JUMP	412	ANDCAM
253	AOBJN	321	JUMPL	413	ANDCAB
254	JRST	322	JUMPE	414	SETM
25404	PORTAL	323	JUMPLE	415	†XMOVEI
25410	JRSTF	324	JUMPA		SETMI
25414		325	JUMPGE	416	SETMM
25420	HALT	326	JUMPN	417	SETMB
25424	†XJRSTF	327	JUMPG	420	ANDCM
25430	†XJEN	330	SKIP	421	ANDCMI
25434	†XPCW	331	SKIPL	422	ANDCMM
25440		332	SKIPE	423	ANDCMB
25444		333	SKIPLE	424	SETA
25450	JEN	334	SKIPA	425	SETAI
25454		335	SKIPGE	426	SETAM
25460	†SFM	336	SKIPN	427	SETAB
25464		337	SKIPG	430	XOR
25470		340	AOJ	431	XORI
25474		341	AOJL	432	XCRM
255	JFCL	342	AOJE	433	XORB
25504	JFOV	343	AOJLE	434	IOR
25510	JCRY1	344	AOJA		OR
25520	JCRY0	345	AOJGE	435	IORI
25530	JCRY	346	AOJN		ORI
25540	JOV	347	AOJG	436	IORM
256	XCT	350	AOS		ORM
257	[2]MAP	351	AOSL	437	IORB
260	PUSHJ	352	AOSE		ORB
261	PUSH	353	AOSLE	440	ANDCB
262	POP	354	AOSA	441	ANDCBI
263	POPJ	355	AOSGE	442	ANDCBM
264	JSR	356	AOSN	443	ANDCBB
265	JSP	357	AOSG	444	EQV

445	EQVI
446	EQVM
447	EQVB
450	SETCA
451	SETCAI
452	SETCAM
453	SETCAB
454	ORCA
455	ORCAI
456	ORCAM
457	ORCAB
460	SETCM
461	SETCMI
462	SETCMM
463	SETCMB
464	ORCM
465	ORCMI
466	ORCMM
467	ORCMB
470	ORCB
471	ORCBI
472	ORCBM
473	ORCBB
474	SETO
475	SETOI
476	SETOM
477	SETOB
500	HLL
501	†XHLLI
	HLLI
502	HLLM
503	HLLS
504	HRL
505	HRLI
506	HRLM
507	HRLS
510	HLLZ
511	HLLZI
512	HLLZM
513	HLLZS
514	HRLZ
515	HRLZI
516	HRLZM
517	HRLZS
520	HLLO
521	HLLOI
522	HLLOM
523	HLLOS
524	HRLO
525	HRLOI
526	HRLOM
527	HRLOS
530	HLLE
531	HLLEI
532	HLLEM
533	HLLES
534	HRLE
535	HRLEI
536	HRLEM
537	HRLES
540	HRR
541	HRRI
542	HRRM
543	HRRS
544	HLR
545	HLRI
546	HLRM
547	HLRS
550	HRRZ
551	HRRZI
552	HRRZM
553	HRRZS
554	HLRZ
555	HLRZI
556	HLRZM
557	HLRZS
560	HRRO
561	HRROI
562	HRROM
563	HRROS
564	HLRO
565	HLROI
566	HLROM
567	HLROS
570	HRRE
571	HRREI
572	HRREM
573	HRRES
574	HLRE
575	HLREI
576	HLREM
577	HLRES
600	TRN
601	TLN
602	TRNE
603	TLNE
604	TRNA
605	TLNA
606	TRNN
607	TLNN
610	TDN
611	TSN
612	TDNE
613	TSNE
614	TDNA
615	TSNA
616	TDNN
617	TSNN
620	TRZ
621	TLZ
622	TRZE
623	TLZE
624	TRZA
625	TLZA
626	TRZN
627	TLZN
630	TDZ
631	TSZ
632	TDZE
633	TSZE
634	TDZA
635	TSZA
636	TDZN
637	TSZN
640	TRC
641	TLC
642	TRCE
643	TLCE
644	TRCA
645	TLCA
646	TRCN
647	TLCN
650	TDC
651	TSC
652	TDCE
653	TSCE
654	TDCA
655	TSCA
656	TDCN
657	TSCN
660	TRO
661	TLO
662	TROE
663	TLOE
664	TROA
665	TLOA
666	TRON
667	TLON
670	TDO
671	TSO
672	TDOE
673	TSOE
674	TDOA
675	TSOA
676	TDON
677	TSON
70000	[3]BLKI
	†APRID
70004	[3]DATAI
	[4]DATAI APR,
	[4]RSW
70010	[3]BLKO
	[5]WRFIL
70014	[3]DATAO
	[4]DATAO APR,
70020	[3]CONO
	[6]WRAPR
	[3]CONO APR,
70024	[3]CONI
	[6]RDAPR
	[3]CONI APR,
70030	[3]CONSZ
70034	[3]CONSO
70040	[5]RDERA

70050	[5]SBDIAG	70204	[6]RDCSB	70270	[6]WRHSB
70054	[4]DATAO PI,		[5]RDTIME	704	[6]UMOVE
70060	[6]WRPI	70210	[6]RDPUR	705	[6]UMOVEM
	[3]CONO PI,		[5]WRPAE	710	[6]TIOE
70064	[6]RDPI	70214	[6]RDCSTM	71054	[7]DATAO PTR,
	[3]CONI PI,	70220	[6]RDTIM	711	[6]TION
70104	[6]RDUBR		[5]CONO TIM,	712	[6]RDIO
	[1,3]DATAI PAG,	70224	[6]RDINT	713	[6]WRIO
70110	†CLRPT		[5]CONI TIM,	714	[6]BSIO
70114	[6]WRUBR	70230	[6]RDHSB	715	[6]BCIO
	[1,3]DATAO PAG,	70240	[6]WRSPB	720	[6]TIOEB
70120	[6]WREBR		[5]RDMACT	721	[6]TIONB
	[1,3]CONO PAG,	70244	[6]WRCSB	722	[6]RDIOB
70124	[6]RDEBR		[5]RDEACT	723	[6]WRIOB
	[1,3]CONI PAG,	70250	[6]WRPUR	724	[6]BSIOB
70144	[5]†SWPIA	70254	[6]WRCSTM	725	[6]BCIOB
70150	[5]†SWPVA	70260	[6]WRTIM	000	[3]APR
70154	[5]†SWPUA		[5]WRTIME	004	[3]PI
70164	[5]†SWPIO		[5]CONO MTR,	010	[1,3]PAG
70170	[5]†SWPVO	70264	[6]WRINT	014	[5]CCA
70174	[5]†SWPUO		[5]CONI MTR,	020	[5]TIM
70200	[6]RDSPB			024	[5]MTR
	[5]RDPERF				

[1] Not available in KA10.

[3] No longer used in KS10 and future machines.

[4] Used only in KA10 and KI10.

[5] Used only in KL10.

[6] Used only in KS10.

[7] Used only in KI10.

INSTRUCTION MNEMONICS

ALPHABETIC LISTING

ADD	270	[3]BLKO	70010	[1]DFSB	111
ADDB	273	BLT	251	‡DGFLTR	027
ADDI	271	[6]BSIO	714	DIV	234
ADDM	272	[6]BSIOB	724	DIVB	237
ADJBP	133	CAI	300	DIVI	235
ADJSP	105	CAIA	304	DIVM	236
AND	404	CAIE	302	[1]DMOVE	120
ANDB	407	CAIG	307	[1]DMOVEM	124
ANDCA	410	CAIGE	305	[1]DMOVN	121
ANDCAB	413	CAIL	301	[1]DMOVNM	125
ANDCAI	411	CAILE	303	†DMUL	116
ANDCAM	412	CAIN	306	DPB	137
ANDCB	440	*CALL	040	†DSUB	115
ANDCBB	443	*CALLI	047	†EDIT	004
ANDCBI	441	CAM	310	*ENTER	077
ANDCBM	442	CAMA	314	EQV	444
ANDCM	420	CAME	312	EQVB	447
ANDCMB	423	CAMG	317	EQVI	445
ANDCMI	421	CAMGE	315	EQVM	446
ANDCMM	422	CAML	311	EXCH	250
ANDI	405	CAMLE	313	FAD	140
ANDM	406	CAMN	316	FADB	143
AOBJN	253	[5]CCA	014	[2]FADL	141
AOBJP	252	*CLOSE	070	FADM	142
AOJ	340	†CLRPT	70140	FADR	144
AOJA	344	†CMPSE	002	FADRB	147
AOJE	342	†CMPSG	007	FADRI	145
AOJG	347	†CMPSGE	005	FADRM	146
AOJGE	345	†CMPSL	001	FDV	170
AOJL	341	†CMPSLE	003	FDVB	173
AOJLE	343	†CMPSN	006	[2]FDVL	171
AOJN	346	[3]CONI	70024	FDVM	172
AOS	350	[3]CONO	70020	FDVR	174
AOSA	354	[3]CONSO	70034	FDVRB	177
AOSE	352	[3]CONSZ	70030	FDVRI	175
AOSG	357	†CVTBDO	012	FDVRM	176
AOSGE	355	†CVTBDT	013	[1]FIX	122
AOSL	351	†CVTDBO	010	[1]FIXR	126
AOSLE	353	†CVTDBT	011	[1]FLTR	127
AOSN	356	†DADD	114	FMP	160
[3]APR	000	[3]DATAI	70004	FMPB	163
†APRID	70000	[3]DATAO	70014	[2]FMPL	161
ASH	240	†DDIV	117	FMPM	162
ASHC	244	[1]DFAD	110	FMPR	164
[6]BCIO	715	[1]DFDV	113	FMPRB	167
[6]BCIOB	725	[1]DFMP	112	FMPRI	165
[3]BLKI	70000	[2]DFN	131	FMPRM	166

[1] Not available in KA10.

[2] Used only in KA10, KI10 and TOPS–10 KL10.

[3] No longer used in KS10 and future machines.

[5] Used only in KL10.

[6] Used only in KS10.

FSB	150
FSBB	153
[2]FSBL	151
FSBM	152
FSBR	154
FSBRB	157
FSBRI	155
FSBRM	156
FSC	132
‡GDBLE	022
‡GDFIX	023
‡GDFIXR	025
*GETSTS	062
‡GFAD	102
‡GFDV	107
‡GFIX	024
‡GFIXR	026
‡GFLTR	030
‡GFMP	106
‡GFSB	103
‡GFSC	031
‡GSNGL	021
HALT	25420
HLL	500
HLLE	530
HLLEI	531
HLLEM	532
HLLES	533
HLLI	501
HLLM	502
HLLO	520
HLLOI	521
HLLOM	522
HLLOS	523
HLLS	503
HLLZ	510
HLLZI	511
HLLZM	512
HLLZS	513
HLR	544
HLRE	574
HLREI	575
HLREM	576
HLRES	577
HLRI	545
HLRM	546
HLRO	564
HLROI	565
HLROM	566
HLROS	567
HLRS	547
HLRZ	554
HLRZI	555
HLRZM	556
HLRZS	557
HRL	504
HRLE	534
HRLEI	535
HRLEM	536
HRLES	537
HRLI	505
HRLM	506
HRLO	524
HRLOI	525
HRLOM	526
HRLOS	527
HRLS	507
HRLZ	514
HRLZI	515
HRLZM	516
HRLZS	517
HRR	540
HRRE	570
HRREI	571
HRREM	572
HRRES	573
HRRI	541
HRRM	542
HRRO	560
HRROI	561
HRROM	562
HRROS	563
HRRS	543
HRRZ	550
HRRZI	551
HRRZM	552
HRRZS	553
IBP	133
IDIV	230
IDIVB	233
IDIVI	231
IDIVM	232
IDPB	136
ILDB	134
IMUL	220
IMULB	223
IMULI	221
IMULM	222
*IN	056
*INBUF	064
*INIT	041
*INPUT	066
IOR	434
IORB	437
IORI	435
IORM	436
JCRY	25530
JCRY0	25520
JCRY1	25510
JEN	25460
JFCL	255
JFFO	243
JFOV	25504
JOV	25540
JRA	267
JRST	254
JRSTF	25410
JSA	266
JSP	265
JSR	264
*JSYS	104
JUMP	320
JUMPA	324
JUMPE	322
JUMPG	327
JUMPGE	325
JUMPL	321
JUMPLE	323
JUMPN	326
LDB	135
*LOOKUP	076
LSH	242
LSHC	246
[1]MAP	257
MOVE	200
MOVEI	201
MOVEM	202
MOVES	203
MOVM	214
MOVMI	215
MOVMM	216
MOVMS	217
MOVN	210
MOVNI	211
MOVNM	212
MOVNS	213
MOVS	204
MOVSI	205
†MOVSLJ	016
MOVSM	206
†MOVSO	014
†MOVSRJ	017
MOVSS	207
†MOVST	015
*MTAPE	072
[5]MTR	024
MUL	224
MULB	227
MULI	225
MULM	226
*OPEN	050
OR	434
ORB	437
ORCA	454
ORCAB	457
ORCAI	455
ORCAM	456
ORCB	470
ORCBB	473
ORCBI	471
ORCBM	472
ORCM	464
ORCMB	467
ORCMI	465

ORCMM	466
ORI	435
ORM	436
*OUT	057
*OUTBUF	065
*OUTPUT	067
[1,3]PAG	010
[3]PI	004
POP	262
POPJ	263
PORTAL	25404
PUSH	261
PUSHJ	260
[6]RDAPR	70024
[6]RDCSB	70204
[6]RDCSTM	70214
[5]RDEACT	70244
[6]RDEBR	70124
[5]RDERA	70040
[6]RDHSB	70230
[6]RDINT	70224
[6]RDIO	712
[6]RDIOB	722
[5]RDMACT	70240
[5]RDPERF	70200
[6]RDPI	70064
[6]RDPUR	70210
[6]RDSPB	70200
[6]RDTIM	70220
[5]RDTIME	70204
[6]RDUBR	70104
*RELEAS	071
*RENAME	055
ROT	241
ROTC	245
[4]RSW	70004
[5]SBDIAG	70050
SETA	424
SETAB	427
SETAI	425
SETAM	426
SETCA	450
SETCAB	453
SETCAI	451
SETCAM	452
SETCM	460
SETCMB	463
SETCMI	461
SETCMM	462
SETM	414
SETMB	417
SETMI	415
SETMM	416
SETO	474
SETOB	477
SETOI	475
SETOM	476
*SETSTS	060
SETZ	400
SETZB	403
SETZI	401
SETZM	402
†SFM	25460
SKIP	330
SKIPA	334
SKIPE	332
SKIPG	337
SKIPGE	335
SKIPL	331
SKIPLE	333
SKIPN	336
SOJ	360
SOJA	364
SOJE	362
SOJG	367
SOJGE	365
SOJL	361
SOJLE	363
SOJN	366
SOS	370
SOSA	374
SOSE	372
SOSG	377
SOSGE	375
SOSL	371
SOSLE	373
SOSN	376
*STATO	061
*STATUS	062
*STATZ	063
SUB	274
SUBB	277
SUBI	275
SUBM	276
[5]SWPIA	70144
[5]SWPIO	70164
[5]SWPUA	70154
[5]SWPUO	70174
[5]SWPVA	70150
[5]SWPVO	70170
TDC	650
TDCA	654
TDCE	652
TDCN	656
TDN	610
TDNA	614
TDNE	612
TDNN	616
TDO	670
TDOA	674
TDOE	672
TDON	676
TDZ	630
TDZA	634
TDZE	632
TDZN	636
[5]TIM	020
[6]TIOE	710
[6]TIOEB	720
[6]TION	711
[6]TIONB	721
TLC	641
TLCA	645
TLCE	643
TLCN	647
TLN	601
TLNA	605
TLNE	603
TLNN	607
TLO	661
TLOA	665
TLOE	663
TLON	667
TLZ	621
TLZA	625
TLZE	623
TLZN	627
TRC	640
TRCA	644
TRCE	642
TRCN	646
TRN	600
TRNA	604
TRNE	602
TRNN	606
TRO	660
TROA	664
TROE	662
TRON	666
TRZ	620
TRZA	624
TRZE	622
TRZN	626
TSC	651
TSCA	655
TSCE	653

[1] Not available in KA10.

[2] Used only in KA10, KI10 and TOPS-10 KL10.

[3] No longer used in KS10 and future machines.

[4] Used only in KA10 and KI10.

[5] Used only in KL10.

[6] Used only in KS10.

TSCN	657
TSN	611
TSNA	615
TSNE	613
TSNN	617
TSO	671
TSOA	675
TSOE	673
TSON	677
TSZ	631
TSZA	635
TSZE	633
TSZN	637
*TTCALL	051
[2]UFA	130
*UGETF	073
*UJEN	100
*USETI	074
*USETO	075
[6]WRAPR	70020
[6]WRCSB	70244
[6]WRCSTM	70254
[6]WREBR	70120
[5]WRFIL	70010
[6]WRHSB	70270
[6]WRINT	70264
[6]WRIO	713
[6]WRIOB	723
[5]WRPAE	70210
[6]WRPI	70060
[6]WRPUR	70250
[6]WRSPB	70240
[6]WRTIM	70260
[5]WRTIME	70260
[6]WRUBR	70114
XCT	256
†XBLT	020
†XHLLI	501
†XJEN	25430
†XJRSTF	25424
†XMOVEI	415
XOR	430
XORB	433
XORI	431
XORM	432
†XPCW	25434

Algebraic Representation

The remaining pages of this appendix list, in symbolic form, the actual operations performed by the instructions. The grouping is the same as that used in Chapter 2, and the groups are in the same order.

Boolean	A–20	In-out	A–26
Byte manipulation	A–26	Program control	A–25
Fixed point arithmetic	A–18	Shift and rotate	A–21
Floating point arithmetic	A–19	Stack	A–25
Full word data transmission	A–18	Test, arithmetic	A–21
Half word data transmission	A–24	Test, logical	A–22

The string instructions are too complex to lend themselves in any reasonable manner to this type of presentation. For them the reader must use the complete descriptions given in §§2.12–2.14 (the last section includes a flowchart of EDIT).

The terminology and notation used vary somewhat from that in the body of the manual, as follows.

AC The accumulator address in bits 9–12 of the instruction word (presented by *A* in the instruction descriptions).

AC+*N* The address *N* greater than AC, except that accumulator addresses wrap around from 17, e.g. AC+3 is 1 if AC is 16.

E The result of the effective address calculation. When E is an address it has the number of bits appropriate to such use — depending on the type of processor, whether local or global, etc. E is eighteen bits unsigned when used as a half word operand, mask or output conditions; nine bits signed when used as a scale factor or shift number; and eighteen bits signed when used as an offset. For any signed quantity, the sign is always bit 18.

E_R The in-section part of E (the right eighteen bits).

E_L The section-number part of E (those bits, if any, at the left of bit 18).

E+*N* The address *N* greater than *E*, with a wraparound, where appropriate, from an in-section value of 777777 without changing the section number.

PC The 30-bit or 18-bit program counter; the symbol also represents the contents of PC when used as the source of an address.

PC+1 The address produced by adding 1 to the in-section part of PC with a wraparound from 777777 (the section number does not change).

(*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 doubleword operand.

$(X–Y)$ The contents of registers X to Y concatenated into a multiword operand.

$((X))$ The word contained in the register addressed by (X), i.e. 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 doublewords). For example,

$$(\mathrm{AC}) + (\mathrm{E}) \rightarrow (\mathrm{AC})$$

means the word in accumulator AC plus the word in memory location E replaces the word in AC.

(AC)(E) The word in AC and the word in E.

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

$+ - \times \div ||$ The arithmetic operators for addition, negation or subtraction, multiplication, division, and absolute value (magnitude).

Square brackets are used occasionally for grouping, but when they enclose an arithmetic computation they represent the "largest integer contained in" the enclosed quantity. With respect to the values of their terms, the equations for a given instruction are in chronological order; e.g. in the pair of equations

$$\begin{gathered}(\mathrm{AC}) + 1 \rightarrow (\mathrm{AC}) \\ \mathit{If}\,(\mathrm{AC}) = 0\text{: } \mathrm{E} \rightarrow (\mathrm{PC})\end{gathered}$$

the quantity tested in the second equation is the word in AC after it has been incremented by one.

Full Word Data Transmission

EXCH	250	(AC) ↔ (E)			
MOVE	200	(E) → (AC)	MOVS	204	$(E)_S$ → (AC)
MOVEI	201	0,E → (AC)	MOVSI	205	E,0 → (AC)
MOVEM	202	(AC) → (E)	MOVSM	206	$(AC)_S$ → (E)
MOVES	203	*If* AC ≠ 0: (E) → (AC)	MOVSS	207	$(E)_S$ → (E) *If* AC ≠ 0: (E) → (AC)
MOVN	210	− (E) → (AC)	MOVM	214	|(E)| → (AC)
MOVNI	211	− [0,E] → (AC)	MOVMI	215	0,E → (AC)
MOVNM	212	− (AC) → (E)	MOVMM	216	|(AC)| → (E)
MOVNS	213	− (E) → (E) *If* AC ≠ 0: (E) → (AC)	MOVMS	217	|(E)| → (E) *If* AC ≠ 0: (E) → (AC)
XMOVEI	415	*If not local AC reference:* E → (AC) *If local AC reference:* 1,E → (AC)			
DMOVE	120	(E,E+1) → (AC,AC+1)	DMOVEM	124	(AC,AC+1) → (E,E+1)
DMOVN	121	− (E,E+1) → (AC,AC+1)	DMOVNM	125	− (AC,AC+1) → (E,E+1)
BLT	251	*Move* $E_R - (AC)_R + 1$ *words starting with* $((AC)_L) \rightarrow ((AC)_R)$ *(see page **2-8**)*			
XBLT	020	*Move* |(AC)| *words (see page **2-10**)* *If* (AC) > 0: *start with* ((AC+1)) → ((AC+2)) *and go up* *If* (AC) < 0: *start with* ((AC+1) − 1) → ((AC+2) − 1) *and go down*			

Fixed Point Arithmetic

ADD	270	(AC) + (E) → (AC)	SUB	274	(AC) − (E) → (AC)
ADDI	271	(AC) + 0,E → (AC)	SUBI	275	(AC) − 0,E → (AC)
ADDM	272	(AC) + (E) → (E)	SUBM	276	(AC) − (E) → (E)
ADDB	273	(AC) + (E) → (AC) (E)	SUBB	277	(AC) − (E) → (AC) (E)
IMUL	220	(AC) × (E) → (AC)*	MUL	224	(AC) × (E) → (AC,AC+1)
IMULI	221	(AC) × 0,E → (AC)*	MULI	225	(AC) × 0,E → (AC,AC+1)
IMULM	222	(AC) × (E) → (E)*	MULM	226	(AC) × (E) → (E)†
IMULB	223	(AC) × (E) → (AC) (E)*	MULB	227	(AC) × (E) → (AC,AC+1) (E)
IDIV	230	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	DIV	234	(AC,AC+1) ÷ (E) → (AC) REMAINDER → (AC+1)
IDIVI	231	(AC) ÷ 0,E → (AC) REMAINDER → (AC+1)	DIVI	235	(AC,AC+1) ÷ 0,E → (AC) REMAINDER → (AC+1)
IDIVM	232	(AC) ÷ (E) → (E)	DIVM	236	(AC,AC+1) ÷ (E) → (E)
IDIVB	233	(AC) ÷ (E) → (AC) (E) REMAINDER → (AC+1)	DIVB	237	(AC,AC+1) ÷ (E) → (AC) (E) REMAINDER → (AC+1)

*The high order word of the product is discarded.

†The low order word of the product is discarded.

DADD	114	(AC,AC+1) + (E,E+1) → (AC,AC+1)
DSUB	115	(AC,AC+1) - (E,E+1) → (AC,AC+1)
DMUL	116	(AC,AC+1) × (E,E+1) → (AC–AC+3)
DDIV	117	(AC–AC+1) ÷ (E,E+1) → (AC,AC+1)
		REMAINDER → (AC+2,AC+3)

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) × (E) → (AC)	FMPR	164	(AC) × (E) → (AC)
FMPL	161	(AC) × (E) → (AC,AC+1)	FMPRI	165	(AC) × E,0 → (AC)
FMPM	162	(AC) × (E) → (E)	FMPRM	166	(AC) × (E) → (E)
FMPB	163	(AC) × (E) → (AC)(E)	FMPRB	167	(AC) × (E) → (AC)(E)
FDV	170	(AC) ÷ (E) → (AC)	FDVR	174	(AC) ÷ (E) → (AC)
FDVL	171	(AC) ÷ (E) → (AC)	FDVRI	175	(AC) ÷ E,0 → (AC)
		REMAINDER → (AC+1)			
FDVM	172	(AC) ÷ (E) → (E)	FDVRM	176	(AC) ÷ (E) → (E)
FDVB	173	(AC) ÷ (E) → (AC)(E)	FDVRB	177	(AC) ÷ (E) → (AC)(E)

DFAD	110	GFAD	102	(AC,AC+1) + (E,E+1) → (AC,AC+1)
DFSB	111	GFSB	103	(AC,AC+1) - (E,E+1) → (AC,AC+1)
DFMP	112	GFMP	106	(AC,AC+1) × (E,E+1) → (AC,AC+1)
DFDV	113	GFDV	107	(AC,AC+1) ÷ (E,E+1) → (AC,AC+1)

FSC	132	(AC) × 2^E → (AC)	GFSC	031	(AC,AC+1) × 2^E → (AC,AC+1)
FLTR	127	(E) *floated, rounded* → (AC)			
GFLTR	030	(E) *floated, rounded* → (AC,AC+1)			
DGFLTR	027	(E,E+1) *floated, rounded* → (AC,AC+1)			
FIX	122	(E) *fixed* → (AC)	FIXR	126	(E) *fixed, rounded* → (AC)
GFIX	024	(E,E+1) *fixed* → (AC)	GFIXR	026	(E,E+1) *fixed, rounded* → (AC)
GDFIX	023	(E,E+1) *fixed* → (AC,AC+1)			
GDFIXR	025	(E,E+1) *fixed, rounded* → (AC,AC+1)			
GSNGL	021	(E,E+1) *converted* → (AC)	GDBLE	022	(E) *converted* → (AC,AC+1)

UFA	130	(AC)+(E) → (AC+1) *without normalization*
DFN	131	- (AC,E) → (AC,E)

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)

Shift and Rotate

ASH	240	(AC) × 2^E → (AC)	ASHC	244	(AC,AC+1) × 2^E → (AC,AC+1)
ROT	241	*Rotate* (AC) E *places*	ROTC	245	*Rotate* (AC,AC+1) E *places*
LSH	242	*Shift* (AC) E *places*	LSHC	246	*Shift* (AC,AC+1) E *places*

Arithmetic Testing

AOBJP	252	(AC) + 1,1 → (AC) *If* (AC) ≥ 0: E → (PC)			
AOBJN	253	(AC) + 1,1 → (AC) *If* (AC) < 0: E → (PC)			
CAI	300	*No-op*	CAM	310	*No-op*
CAIL	301	*If* (AC) < E: *skip*	CAML	311	*If* (AC) < (E): *skip*
CAIE	302	*If* (AC) = E: *skip*	CAME	312	*If* (AC) = (E): *skip*
CAILE	303	*If* (AC) ≤ E: *skip*	CAMLE	313	*If* (AC) ≤ (E): *skip*
CAIA	304	*Skip*	CAMA	314	*Skip*
CAIGE	305	*If* (AC) ≥ E: *skip*	CAMGE	315	*If* (AC) ≥ (E): *skip*
CAIN	306	*If* (AC) ≠ E: *skip*	CAMN	316	*If* (AC) ≠ (E): *skip*
CAIG	307	*If* (AC) > E: *skip*	CAMG	317	*If* (AC) > (E): *skip*
JUMP	320	*No-op*	SKIP	330	*If* AC ≠ 0: (E) → (AC)
JUMPL	321	*If* (AC) < 0: E → (PC)	SKIPL	331	*If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*
JUMPE	322	*If* (AC) = 0: E → (PC)	SKIPE	332	*If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*
JUMPLE	323	*If* (AC) ≤ 0: E → (PC)	SKIPLE	333	*If* AC ≠ 0: (E) → (AC) *If* (E) ≤ 0: *skip*
JUMPA	324	E → (PC)	SKIPA	334	*If* AC ≠ 0: (E) → (AC) *Skip*
JUMPGE	325	*If* (AC) ≥ 0: E → (PC)	SKIPGE	335	*If* AC ≠ 0: (E) → (AC) *If* (E) ≥ 0: *skip*
JUMPN	326	*If* (AC) ≠ 0: E → (PC)	SKIPN	336	*If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
JUMPG	327	*If* (AC) > 0: E → (PC)	SKIPG	337	*If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*
AOJ	340	(AC) + 1 → (AC)	SOJ	360	(AC) − 1 → (AC)
AOJL	341	(AC) + 1 → (AC) *If* (AC) < 0: E → (PC)	SOJL	361	(AC) − 1 → (AC) *If* (AC) < 0: E → (PC)
AOJE	342	(AC) + 1 → (AC) *If* (AC) = 0: E → (PC)	SOJE	362	(AC) − 1 → (AC) *If* (AC) = 0: E → (PC)
AOJLE	343	(AC) + 1 → (AC) *If* (AC) ≤ 0: E → (PC)	SOJLE	363	(AC) − 1 → (AC) *If* (AC) ≤ 0: E → (PC)
AOJA	344	(AC) + 1 → (AC) E → (PC)	SOJA	364	(AC) − 1 → (AC) E → (PC)

AOJGE	345	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) \geqslant 0$: $E \rightarrow (PC)$	SOJGE	365	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) \geqslant 0$: $E \rightarrow (PC)$
AOJN	346	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$	SOJN	366	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$
AOJG	347	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) > 0$: $E \rightarrow (PC)$	SOJG	367	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) > 0$: $E \rightarrow (PC)$
AOS	350	$(E) + 1 \rightarrow (E)$ *If* $(AC) \neq 0$: $(E) \rightarrow (AC)$	SOS	370	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
AOSL	351	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) < 0$: *skip*	SOSL	371	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) < 0$: *skip*
AOSE	352	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) = 0$: *skip*	SOSE	372	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) = 0$: *skip*
AOSLE	353	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \leqslant 0$: *skip*	SOSLE	373	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \leqslant 0$: *skip*
AOSA	354	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *Skip*	SOSA	374	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *Skip*
AOSGE	355	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \geqslant 0$: *skip*	SOSGE	375	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \geqslant 0$: *skip*
AOSN	356	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \neq 0$: *skip*	SOSN	376	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \neq 0$: *skip*
AOSG	357	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) > 0$: *skip*	SOSG	377	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) > 0$: *skip*

Logical Testing and Modification

TLN	601	*No-op*	TRN	600	*No-op*
TLNE	603	*If* $(AC)_L \wedge E = 0$: *skip*	TRNE	602	*If* $(AC)_R \wedge E = 0$: *skip*
TLNA	605	*Skip*	TRNA	604	*Skip*
TLNN	607	*If* $(AC)_L \wedge E \neq 0$: *skip*	TRNN	606	*If* $(AC)_R \wedge E \neq 0$: *skip*
TLZ	621	$(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZ	620	$(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZE	623	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZE	622	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZA	625	$(AC)_L \wedge \sim E \rightarrow (AC)_L$ *skip*	TRZA	624	$(AC)_R \wedge \sim E \rightarrow (AC)_R$ *skip*
TLZN	627	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZN	626	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$

TLC	641	$(AC)_L \forall E \rightarrow (AC)_L$	TRC	640	$(AC)_R \forall E \rightarrow (AC)_R$
TLCE	643	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \forall E \rightarrow (AC)_L$	TRCE	642	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \forall E \rightarrow (AC)_R$
TLCA	645	$(AC)_L \forall E \rightarrow (AC)_L$ *skip*	TRCA	644	$(AC)_R \forall E \rightarrow (AC)_R$ *skip*
TLCN	647	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \forall E \rightarrow (AC)_L$	TRCN	646	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \forall E \rightarrow (AC)_R$
TLO	661	$(AC)_L \vee E \rightarrow (AC)_L$	TRO	660	$(AC)_R \vee E \rightarrow (AC)_R$
TLOE	663	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TROE	662	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TLOA	665	$(AC)_L \vee E \rightarrow (AC)_L$ *skip*	TROA	664	$(AC)_R \vee E \rightarrow (AC)_R$ *skip*
TLON	667	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TRON	666	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TDN	610	*No-op*	TSN	611	*No-op*
TDNE	612	*If* $(AC) \wedge (E) = 0$: *skip*	TSNE	613	*If* $(AC) \wedge (E)_S = 0$: *skip*
TDNA	614	*Skip*	TSNA	615	*Skip*
TDNN	616	*If* $(AC) \wedge (E) \neq 0$: *skip*	TSNN	617	*If* $(AC) \wedge (E)_S \neq 0$: *skip*
TDZ	630	$(AC) \wedge \sim (E) \rightarrow (AC)$	TSZ	631	$(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZE	632	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZE	633	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZA	634	$(AC) \wedge \sim (E) \rightarrow (AC)$ *skip*	TSZA	635	$(AC) \wedge \sim (E)_S \rightarrow (AC)$ *skip*
TDZN	636	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZN	637	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDC	650	$(AC) \forall (E) \rightarrow (AC)$	TSC	651	$(AC) \forall (E)_S \rightarrow (AC)$
TDCE	652	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \forall (E) \rightarrow (AC)$	TSCE	653	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \forall (E)_S \rightarrow (AC)$
TDCA	654	$(AC) \forall (E) \rightarrow (AC)$ *skip*	TSCA	655	$(AC) \forall (E)_S \rightarrow (AC)$ *skip*
TDCN	656	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \forall (E) \rightarrow (AC)$	TSCN	657	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \forall (E)_S \rightarrow (AC)$
TDO	670	$(AC) \vee (E) \rightarrow (AC)$	TSO	671	$(AC) \vee (E)_S \rightarrow (AC)$
TDOE	672	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSOE	673	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$
TDOA	674	$(AC) \vee (E) \rightarrow (AC)$ *skip*	TSOA	675	$(AC) \vee (E)_S \rightarrow (AC)$ *skip*
TDON	676	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSON	677	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$

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

HRLO	524	$(E)_R$,777777 → (AC)	HRLE	534	$(E)_R,[(E)_{18} \times 777777] \rightarrow (AC)$
HRLOI	525	E,777777 → (AC)	HRLEI	535	$E,[E_{18} \times 777777] \rightarrow (AC)$
HRLOM	526	$(AC)_R$,777777 → (E)	HRLEM	536	$(AC)_R,[(AC)_{18} \times 777777] \rightarrow (E)$
HRLOS	527	$(E)_R$,777777 → (E) *If* AC ≠ 0: (E) → (AC)	HRLES	537	$(E)_R,[(E)_{18} \times 777777] \rightarrow (E)$ *If* AC ≠ 0: (E) → (AC)
XHLLI	501	$E_L \rightarrow (AC)_L$			

Program Control

XCT	256	*Execute* (E)
JFFO	243	*If* (AC) = 0: 0 → (AC + 1) *If* (AC) ≠ 0: E → (PC) (*see page 2-64*)
JFCL	255	*If* AC ∧ FLAGS ≠ 0: E → (PC) ~AC ∧ FLAGS → FLAGS
JRST	25400	E → (PC)
PORTAL	25404	0 → PUBLIC E → (PC)
JRSTF	25410	$(X)_L$ *or* $(Y)_L$ → FLAGS E → (PC)
HALT	25420	E → (PC) *stop*
XJRSTF	25424	$(E)_L$ → FLAGS (E+1) → (PC)
XJEN	25430	*Dismiss* PI $(E)_L$ → FLAGS (E+1) → (PC)
XPCW	25434	FLAGS,0 → (E) PC+1 → (E+1) $(E+2)_L$ → FLAGS (E+3) → (PC)
JEN	25450	*Dismiss* PI $(X)_L$ *or* $(Y)_L$ → FLAGS E → (PC)
SFM	25460	FLAGS,0 → (E)
JSR	264	*If* $PC_L = 0$: FLAGS,PC_R+1 → (E) E+1 → (PC) *If* $PC_L \neq 0$: PC+1 → (E) E+1 → (PC)
JSP	265	*If* $PC_L = 0$: FLAGS,PC_R+1 → (AC) E → (PC) *If* $PC_L \neq 0$: PC+1 → (AC) E → (PC)
JSA	266	(AC) → (E) E_R,PC_R+1 → (AC) E+1 → (PC)
JRA	267	$((AC)_L)$ → (AC) E → (PC)
MAP	257	PHYSICAL MAP DATA → (AC)

Stack

PUSH	261	*If* $PC_L = 0$ *or* $(AC)_{0,6\text{-}17} \leqslant 0$: (AC) + 1,1 → (AC) (E) → $((AC)_R)$ I*f* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: (AC) + 1 → (AC) (E) → ((AC))
POP	262	*If* $PC_L = 0$ *or* $(AC)_{0,6\text{-}17} \leqslant 0$: $((AC)_R)$ → (E) (AC) - 1,1 → (AC) *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: ((AC)) → (E) (AC) - 1 → (AC)
PUSHJ	260	*If* $PC_L = 0$: (AC) + 1,1 → (AC) FLAGS,PC+1 → $((AC)_R)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} \leqslant 0$: (AC) + 1,1 → (AC) PC+1 → $((AC)_R)$ *If* $PC_L \neq 0$ *and* $(AC)_{0,6\text{-}17} > 0$: (AC) + 1 → (AC) PC+1 → ((AC)) E → (PC)

POPJ 263 *If* $\mathrm{PC_L} = 0$: $((\mathrm{AC})_\mathrm{R})_\mathrm{R} \rightarrow (\mathrm{PC})$ $(\mathrm{AC}) - 1{,}1 \rightarrow (\mathrm{AC})$
If $\mathrm{PC_L} \neq 0$ *and* $(\mathrm{AC})_{0,6\text{-}17} \leqslant 0$: $((\mathrm{AC})_\mathrm{R}) \rightarrow (\mathrm{PC})$ $(\mathrm{AC}) - 1{,}1 \rightarrow (\mathrm{AC})$
If $\mathrm{PC_L} \neq 0$ *and* $(\mathrm{AC})_{0,6\text{-}17} > 0$: $((\mathrm{AC})) \rightarrow (\mathrm{PC})$ $(\mathrm{AC}) - 1 \rightarrow (\mathrm{AC})$

ADJSP 105 *If* $\mathrm{PC_L} = 0$ *or* $(\mathrm{AC})_{0,6\text{-}17} \leqslant 0$: $(\mathrm{AC}) + [\pm]\mathrm{E_R},\mathrm{E_R} \rightarrow (\mathrm{AC})$
If $\mathrm{PC_L} \neq 0$ *and* $(\mathrm{AC})_{0,6\text{-}17} > 0$: $(\mathrm{AC}) + [\pm]\mathrm{E_R} \rightarrow (\mathrm{AC})$

Byte Manipulation

IBP 133 AC = 0
Linear operations on pointer in E *or* E,E+1
If $\mathrm{P} - \mathrm{S} \geqslant 0$: $\mathrm{P} - \mathrm{S} \rightarrow \mathrm{P}$
If $\mathrm{P} - \mathrm{S} < 0$: $\mathrm{Y} + 1 \rightarrow \mathrm{Y}$ $36 - \mathrm{S} \rightarrow \mathrm{P}$

ADJBP 133 AC ≠ 0
Array operations on pointer in E *or* E,E+1
Let A = REMAINDER $\dfrac{36 - \mathrm{P}}{\mathrm{S}}$
If $\mathrm{S} > 36 - A$: 1 → NO DIVIDE
If $\mathrm{S} = 0$: $(\mathrm{E}) \rightarrow (\mathrm{AC})$ *or* $(\mathrm{E,E{+}1}) \rightarrow (\mathrm{AC,AC{+}1})$
If $0 < \mathrm{S} < 36 - A$: *make copy* C *of* (E) *or* (E,E+1)

$$\textit{Compute}\ (\mathrm{AC}) + \left[\frac{36 - \mathrm{P}}{\mathrm{S}}\right] = \mathrm{Q} \times \text{BYTES/WORD} + \mathrm{R}$$

$$1 \leqslant \mathrm{R} \leqslant \text{BYTES/WORD} = \left[\frac{36 - \mathrm{P}}{\mathrm{S}}\right] + \left[\frac{\mathrm{P}}{\mathrm{S}}\right]$$

$$\mathrm{Y}\{\mathrm{C}\} + \mathrm{Q} \rightarrow \mathrm{Y}\{\mathrm{C}\}$$

$$36 - \mathrm{R} \times \mathrm{S} - A \rightarrow \mathrm{P}\{\mathrm{C}\}$$

$$\mathrm{C} \rightarrow (\mathrm{AC})\ \textit{or}\ (\mathrm{AC,AC{+}1})$$

LDB	135	BYTE IN ((E)) → (AC)
DPB	137	BYTE IN (AC) → BYTE IN ((E))
ILDB	134	IBP *and* LDB
IDPB	136	IBP *and* DPB

In-out

CONO	70020	E → COMMAND	CONSZ	70030	*If* $\mathrm{STATUS_R} \wedge \mathrm{E} = 0$: *skip*
CONI	70024	STATUS → (E)	CONSO	70034	*If* $\mathrm{STATUS_R} \wedge \mathrm{E} \neq 0$: *skip*
DATAO	70014	(E) → DATA	DATAI	70004	DATA → (E)

BLKO	70010	$(\mathrm{E}) + 1{,}1 \rightarrow (\mathrm{E})$	$((\mathrm{E})_\mathrm{R}) \rightarrow$ DATA	*If* $(\mathrm{E})_\mathrm{L} \neq 0$: *skip*
BLKI	70000	$(\mathrm{E}) + 1{,}1 \rightarrow (\mathrm{E})$	DATA $\rightarrow ((\mathrm{E})_\mathrm{R})$	*If* $(\mathrm{E})_\mathrm{L} \neq 0$: *skip*

POWERS OF TWO

2^N	N	2^{-N}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

Appendix B
Character Codes

The table on pages B–3 to B–5 lists the complete 1977 ASCII code (ANSI X3.4–1977). The software handles the full character set, and for a program that does not handle lower case, it translates input codes 140–174 into the corresponding upper case codes (100–134) and translates both 175 and 176 into 033, escape. The actual character sets available on different terminals vary greatly, but usually a terminal without lower case will accept lower case codes, printing the corresponding upper case character. The definitions of the control codes are those given by ASCII; most control codes, however, have no effect on most typical terminals, and the definitions bear no necessary relation to the use of the codes in conjunction with any software.

CAUTION

> Output codes are ordinarily passed as they are, with the expectation that the terminal will ignore irrelevant control codes, and that a terminal that lacks lower case will print the corresponding upper case. A terminal that fails to live up to these assumptions will generally not operate satisfactorily with TOPS–10 or TOPS–20. Brackets enclose earlier definitions of control codes (mostly 1963 ASCII). The table includes bit 8 as an even parity bit, the form regularly used for paper tape and asynchronous operations; odd parity is regularly used for magnetic tape and synchronous operations.

There are three line printer controllers: the LP20 and LP100, which can handle any printer, and the BA10, which handles only the LP10 models. With the LP100 and BA10, ten fixed control characters are used for format control, and the BA10 also recognizes null for fill and delete for selecting hidden characters. Control characters recognized by the LP20 are program selectable. Remote stations (not considered in the tables beginning on page B–6) generally recognize only line feed and form feed and do not convert lower case codes to upper case. The 64-character print set includes the figures and upper case; lower case is added for the 95-character set (with the smaller print set, giving a lower case code prints the upper case character). The LP05, LP07 and LP14 are available with either set; those LP10 printers with the larger set (models D and H) can have an optional 96th character hidden under delete. The printable characters are generally those defined by ASCII, with little if any variation. The 128-character printer (LP10E) uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control codes that affect the printer and also under null and delete.

The first two pages of the table of card codes (pages B–10 to B–14) list the column punches required to represent characters in the ASCII card code. When reading cards, the software translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. There are also a few control hole patterns that the software responds to but does not translate. The next page lists two earlier DEC card codes that have only the figure and upper case character subset, plus a few control punches. The remaining pages of the table show the relationship among the early DEC card codes, the corresponding characters in the ASCII set, and several IBM card keypunches. Each column punch is produced by a single key on any keypunch for which a character is listed, the character being that printed at the top of the card.

ASCII CODE

Control Characters

Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Class[1]	Remarks
0	000	000	NUL		Null, tape feed. Control @ (control shift P[2]).
1	001	001	SOH	CC	Start of heading [SOM, start of message]. Control A.
1	002	002	STX	CC	Start of text [EOA, end of address]. Control B.
0	003	003	ETX	CC	End of text [EOM, end of message]. Control C.
1	004	004	EOT	CC	End of transmission; shuts off TWX machines and disconnects some data sets. Control D.
0	005	005	ENQ	CC	Enquiry [WRU, "Who are you?"]. Triggers identification ("Here is . . .") at remote station if so equipped. Control E.
0	006	006	ACK	CC	Acknowledge [RU, "Are you . . .?"]. Control F.
1	007	007	BEL		Bell (audible or attention signal). Control G.
1	008	010	BS	FE	Backspace. Control H.
0	009	011	HT	FE	Horizontal tabulation. Control I.
0	010	012	LF[3]	FE	Line feed. Control J.
1	011	013	VT[3]	FE	Vertical tabulation. Control K.
0	012	014	FF[3]	FE	Form feed (to top of next page). Control L.
1	013	015	CR	FE	Carriage return (to beginning of line). Control M.
1	014	016	SO		Shift out; change character set or change ribbon color to red. Control N.
0	015	017	SI		Shift in; return to standard character set or color. Control O.
1	016	020	DLE	CC	Data link escape [DC0]. Control P.
0	017	021	DC1		Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	018	022	DC2		Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	019	023	DC3		Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	020	024	DC4		Device control 4 (stop), turns punch or auxiliary off. Control T (~~TAPE~~, AUX OFF).
1	021	025	NAK	CC	Negative acknowledge [ERR, error]. Control U.
1	022	026	SYN	CC	Synchronous idle [SYNC]. Control V.
0	023	027	ETB	CC	End of transmission block [LEM, logical end of medium]. Control W.
0	024	030	CAN		Cancel [S_0]. Control X.
1	025	031	EM		End of medium [S_1]. Control Y.
1	026	032	SUB		Substitute [S_2]. Control Z.
0	027	033	ESC		Escape, prefix [S_3]. Control [(control shift K[2]).
1	028	034	FS	IS	File separator [S_4]. Control \ (control shift L[2]).
0	029	035	GS	IS	Group separator [S_5]. Control] (control shift M[2]).
0	030	036	RS	IS	Record separator [S_6]. Control ^ (control shift N[2]).
1	031	037	US	IS	Unit separator [S_7]. Control - (control shift O[2]).

[1] CC communication control, FE format effector, IS information separator.

[2] On LT33, LT35 and similar units.

[3] Includes a carriage return on some equipment, but not on standard DEC units.

Graphic Characters

Figures				Upper Case				Lower Case			
Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character[3]
1	032	040	SP	1	064	100	@	0	096	140	`[2]
0	033	041	!	0	065	101	A	1	097	141	a
0	034	042	"	0	066	102	B	1	098	142	b
1	035	043	#	1	067	103	C	0	099	143	c
0	036	044	$	0	068	104	D	1	100	144	d
1	037	045	%	1	069	105	E	0	101	145	e
1	038	046	&	1	070	106	F	0	102	146	f
0	039	047	'	0	071	107	G	1	103	147	g
0	040	050	(	0	072	110	H	1	104	150	h
1	041	051	)	1	073	111	I	0	105	151	i
1	042	052	*	1	074	112	J	0	106	152	j
0	043	053	+	0	075	113	K	1	107	153	k
1	044	054	,	1	076	114	L	0	108	154	l
0	045	055	–	0	077	115	M	1	109	155	m
0	046	056	.	0	078	116	N	1	110	156	n
1	047	057	/	1	079	117	O	0	111	157	o
0	048	060	Ø[1]	0	080	120	P	1	112	160	p
1	049	061	1	1	081	121	Q	0	113	161	q
1	050	062	2	1	082	122	R	0	114	162	r
0	051	063	3	0	083	123	S	1	115	163	s
1	052	064	4	1	084	124	T	0	116	164	t
0	053	065	5	0	085	125	U	1	117	165	u
0	054	066	6	0	086	126	V	1	118	166	v
1	055	067	7	1	087	127	W	0	119	167	w
1	056	070	8	1	088	130	X	0	120	170	x
0	057	071	9	0	089	131	Y	1	121	171	y
0	058	072	:	0	090	132	Z	1	122	172	z
1	059	073	;	1	091	133	[	0	123	173	{
0	060	074	<	0	092	134	\[2]	1	124	174	|
1	061	075	=	1	093	135	]	0	125	175	}[4]
1	062	076	>	1	094	136	^[2]	0	126	176	~[2,5]
0	063	077	?	0	095	137	_	1	127	177	DEL[6]

[1] Zero–slash absent on many units.

[2] Under study by responsible American National Standards Committee for possible change at next revision of ASCII (ca. 1982).

[3] Codes 140-173 first defined in 1965. For a full ASCII character set the operating system accepts codes 140-176 as lower case. For a program requiring a character set that lacks lower case, the operating system translates input codes 140-174 into the corresponding upper case codes (100-134) and translates both 175 and 176 into 033, escape. Early versions of the DECsystem-10 Monitor used 175 as the escape code and translated both 176 and 033 to it.

[4] Unassigned control character (usually ALT MODE) before 1965. Code generated by ALT MODE key on some DEC units, especially earlier ones; on some more recent units, the ALT key generates the stanc'ard escape code, 033.

[5] Control character ESC before 1965; code generated by ESC key on some DEC units designed at that time.

[6] Delete, rub out (not part of lower case set).

Remarks on Special Graphic Characters

SP Space – normally nonprinting.

! Exclamation point.

" Quotation mark, diaeresis.

Number sign. £ on some (non-DEC) units.

$ Dollar sign.

% Percent.

& Ampersand.

' Apostrophe, closing single quotation mark, acute accent. ´ in appearance on some DEC units.

(Opening parenthesis.

) Closing parenthesis.

* Asterisk.

+ Plus.

, Comma, cedilla.

- Hyphen, minus.

. Period, decimal point.

/ Slant, slash, solidus.

: Colon.

; Semicolon.

< Less than.

= Equals.

> Greater than.

? Question mark.

@ Commercial at. ` 1965-67, but never on DEC units.

[Opening bracket. Shift K on LT33, LT35 and similar units.

\ Reverse slant. ~ 1965-67, but never on DEC units. Shift L on LT33, LT35 and similar units.

] Closing bracket. Shift M on LT33, LT35 and similar units.

^ Circumflex, upward arrow head. ↑ before 1965, but used until 1972 on DEC units.

_ Underline, underscore. ← before 1965, but used until 1972 on DEC units.

` Grave accent, opening single quotation mark. @ 1965-67, but never on DEC units.

{ Opening brace.

| Vertical line. Control character ACK before 1965; ¬ 1965-67, but never on DEC units; ¦ in appearance 1968-1977, but generally not on DEC units.

} Closing brace. Unassigned control character (usually ALT MODE) before 1965.

~ Overline, tilde, general accent. Control character after ESC before 1965; | 1965-67, but never on DEC units.

LINE PRINTER CODE: LP10A, B, C, D, E
Basic Character Set

Control				Figures				Upper Case			
Hex	Decimal	Octal	Character	Hex	Decimal	Octal	Character	Hex	Decimal	Octal	Character
09	009	011	HT	20	032	040	SP	40	064	100	@
0A	010	012	LF	21	033	041	!	41	065	101	A
0B	011	013	VT	22	034	042	"	42	066	102	B
0C	012	014	FF	23	035	043	#	43	067	103	C
0D	013	015	CR	24	036	044	$	44	068	104	D
				25	037	045	%	45	069	105	E
10	016	020	DLE	26	038	046	&	46	070	106	F
11	017	021	DC1	27	039	047	'	47	071	107	G
12	018	022	DC2	28	040	050	(	48	072	110	H
13	019	023	DC3	29	041	051	)	49	073	111	I
14	020	024	DC4	2A	042	052	*	4A	074	112	J
				2B	043	053	+	4B	075	113	K
00	000	000	NUL	2C	044	054	,	4C	076	114	L
7F	127	177	DEL	2D	045	055	–	4D	077	115	M
				2E	046	056	.	4E	078	116	N
				2F	047	057	/	4F	079	117	O
				30	048	060	0	50	080	120	P
				31	049	061	1	51	081	121	Q
				32	050	062	2	52	082	122	R
				33	051	063	3	53	083	123	S
				34	052	064	4	54	084	124	T
				35	053	065	5	55	085	125	U
				36	054	066	6	56	086	126	V
				37	055	067	7	57	087	127	W
				38	056	070	8	58	088	130	X
				39	057	071	9	59	089	131	Y
				3A	058	072	:	5A	090	132	Z
				3B	059	073	;	5B	091	133	[
				3C	060	074	<	5C	092	134	\
				3D	061	075	=	5D	093	135	]
				3E	062	076	>	5E	094	136	↑
				3F	063	077	?	5F	095	137	←

BA10 recognizes all twelve control codes, LP100 recognizes the first ten. LP20 control codes are program selectable.

Additional Characters – 95, 96 and 128 Character Sets

LP10D, E				LP10E			
Hex	Decimal	Octal	Character	Hex	Decimal	Octal	Character
60	096	140	`	7F/00	127/000	177/000	· *NUL*
61	097	141	a	01	001	001	↓ *SOH*
62	098	143	b	02	002	002	α *STX*
63	099	143	c	03	003	003	β *ETX*
64	100	144	d	04	004	004	Λ *EOT*
65	101	145	e	05	005	005	¬ *ENQ*
66	102	146	f	06	006	006	ϵ *ACK*
67	103	147	g	07	007	007	π *BEL*
68	104	150	h	08	008	010	λ *BS*
69	105	151	i	7F/09	127/009	177/011	γ *HT*
6A	106	152	j	7F/0A	127/010	177/012	δ *LF*
6B	107	153	k	7F/0B	127/011	177/013	∫ *VT*
6C	108	154	l	7F/0C	127/012	177/014	± *FF*
6D	109	155	m	7F/0D	127/013	177/015	⊙ *CR*
6E	110	156	n	0E	014	016	∞ *SO*
6F	111	157	o	0F	015	017	∂ *SI*
70	112	160	p	7F/10	127/016	177/020	⊂ *DLE*
71	113	161	q	7F/11	127/017	177/021	⊃ *DC1*
72	114	162	r	7F/12	127/018	177/022	∩ *DC2*
73	115	163	s	7F/13	127/019	177/023	∪ *DC3*
74	116	164	t	7F/14	127/020	177/024	∀ *DC4*
75	117	165	u	15	021	025	∃ *NAK*
76	118	166	v	16	022	026	⊗ *SYN*
77	119	167	w	17	023	027	↔ *ETB*
78	120	170	x	18	024	030	^ *CAN*
79	121	171	y	19	025	031	→ *EM*
7A	122	172	z	1A	026	032	_ *SUB*
7B	123	173	{	1B	027	033	≠ *ESC*
7C	124	174	|	1C	028	034	⩽ *FS*
7D	125	175	}	1D	029	035	⩾ *GS*
7E	126	176	~	1E	030	036	≡ *RS*
7F/7F	127/127	177/177	¬ *DEL*	1F	031	037	∨ *US*

Code pairs indicate hidden characters. For characters after the 95th, corresponding ASCII control characters are given in italics to facilitate generating codes at a keyboard. Use of 177 for a hidden character is optional on the LP10D.

LINE PRINTER CODE: LP10F and H, LP05, LP07, LP14
Basic Character Set

Control

Hex	Decimal	Octal	Character
09	009	011	HT
0A	010	012	LF
0B	011	013	VT
0C	012	014	FF
0D	013	015	CR
10	016	020	DLE
11	017	021	DC1
12	018	022	DC2
13	019	023	DC3
14	020	024	DC4
00	000	000	NUL
7F	127	177	DEL

Table gives character set of LP05, LP07 and LP14, and EDP character set of LP10FE and HE. Brackets enclose substitutions for scientific set, LP10FF and HF.

Figures

Hex	Decimal	Octal	Character
20	032	040	SP
21	033	041	!
22	034	042	″
23	035	043	#
24	036	044	$
25	037	045	%
26	038	046	&
27	039	047	′
28	040	050	(
29	041	051	)
2A	042	052	*
2B	043	053	+
2C	044	054	,
2D	045	055	–
2E	046	056	.
2F	047	057	/
30	048	060	0 [Ø]
31	049	061	1
32	050	062	2
33	051	063	3
34	052	064	4
35	053	065	5
36	054	066	6
37	055	067	7
38	056	070	8
39	057	071	9
3A	058	072	:
3B	059	073	;
3C	060	074	<
3D	061	075	=
3E	062	076	>
3F	063	077	?

Upper Case

Hex	Decimal	Octal	Character
40	064	100	@
41	065	101	A
42	066	102	B
43	067	103	C
44	068	104	D
45	069	105	E
46	070	106	F
47	071	107	G
48	072	110	H
49	073	111	I
4A	074	112	J
4B	075	113	K
4C	076	114	L
4D	077	115	M
4E	078	116	N
4F	079	117	O
50	080	120	P
51	081	121	Q
52	082	122	R
53	083	123	S
54	084	124	T
55	085	125	U
56	086	126	V
57	087	127	W
58	088	130	X
59	089	131	Y
5A	090	132	Z [Ƶ]
5B	091	133	[
5C	092	134	\
5D	093	135	]
5E	094	136	^
5F	095	137	_

Additional Characters – LP10H, LP05, LP07, LP14

Hex	Decimal	Octal	Character	Hex	Decimal	Octal	Character
60	096	140	`	70	112	160	p
61	097	141	a	71	113	161	q
62	098	142	b	72	114	162	r
63	099	143	c	73	115	163	s
64	100	144	d	74	116	164	t
65	101	145	e	75	117	165	u
66	102	146	f	76	118	166	v
67	103	147	g	77	119	167	w
68	104	150	h	78	120	170	x
69	105	151	i	79	121	171	y
6A	106	152	j	7A	122	172	z
6B	107	153	k	7B	123	173	{
6C	108	154	l	7C	124	174	|
6D	109	155	m	7D	125	175	}
6E	110	156	n	7E	126	176	~
6F	111	157	o	7F/7F	127/127	177/177	←*DEL*

Character with code 177 is available as an option on LP10H only and is hidden under delete.

ASCII CARD CODE

Octal	Character	Column Punch	Octal	Character	Column Punch	Octal	Character	Column Punch	Octal	Character	Column Punch
000	NUL	12 0 9 8 1	040	SP	*None*	100	@	8 4	140		8 1
001	SOH	12 9 1	041	!	12 8 7	101	A	12 1	141	a	12 0 1
002	STX	12 9 2	042	"	8 7	102	B	12 2	142	b	12 0 2
003	ETX	12 9 3	043	#	8 3	103	C	12 3	143	c	12 0 3
004	EOT	9 7	044	$	11 8 3	104	D	12 4	144	d	12 0 4
005	ENQ	0 9 8 5	045	%	0 8 4	105	E	12 5	145	e	12 0 5
006	ACK	0 9 8 6	046	&	12	106	F	12 6	146	f	12 0 6
007	BEL	0 9 8 7	047	'	8 5	107	G	12 7	147	g	12 0 7
010	BS	11 9 6	050	(	12 8 5	110	H	12 8	150	h	12 0 8
011	HT	12 9 6	051	)	11 8 5	111	I	12 9	151	i	12 0 9
012	LF	0 9 5	052	*	11 8 4	112	J	11 1	152	j	12 11 1
013	VT	12 9 8 3	053	+	12 8 6	113	K	11 2	153	k	12 11 2
014	FF	12 9 8 4	054	,	0 8 3	114	L	11 3	154	l	12 11 3
015	CR	12 9 8 5	055	–	11	115	M	11 4	155	m	12 11 4
016	SO	12 9 8 6	056	.	12 8 3	116	N	11 5	156	n	12 11 5
017	SI	12 9 8 7	057	/	0 1	117	O	11 6	157	o	12 11 6
020	DLE	12 11 9 8 1	060	0	0	120	P	11 7	160	p	12 11 7
021	DC1	11 9 1	061	1	1	121	Q	11 8	161	q	12 11 8
022	DC2	11 9 2	062	2	2	122	R	11 9	162	r	12 11 9
023	DC3	11 9 3	063	3	3	123	S	0 2	163	s	11 0 2
024	DC4	9 8 4	064	4	4	124	T	0 3	164	t	11 0 3
025	NAK	9 8 5	065	5	5	125	U	0 4	165	u	11 0 4
026	SYN	9 2	066	6	6	126	V	0 5	166	v	11 0 5
027	ETB	0 9 6	067	7	7	127	W	0 6	167	w	11 0 6
030	CAN	11 9 8	070	8	8	130	X	0 7	170	x	11 0 7
031	EM	11 9 8 1	071	9	9	131	Y	0 8	171	y	11 0 8
032	SUB	9 8 7	072	:	8 2	132	Z	0 9	172	z	11 0 9
033	ESC	0 9 7	073	;	11 8 6	133	[	12 8 2	173	{	12 0
034	FS	11 9 8 4	074	<	12 8 4	134	\	0 8 2	174	|	12 11
035	GS	11 9 8 5	075	=	8 6	135	]	11 8 2	175	}	11 0
036	RS	11 9 8 6	076	>	0 8 6	136	^	11 8 7	176	~	11 0 1
037	US	11 9 8 7	077	?	0 8 7	137	_	0 8 5	177	DEL	12 9 7

When reading or punching cards, the software translates between the octal codes and column punches listed here. The software also recognizes the following control punches.

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

Column Punch	Character	Column Punch	Character	Column Punch	Character	Column Punch	Character
None	SP	11 8	Q	12 0 4	d	11 8 3	$
12	&	11 9	R	12 0 5	e	11 8 4	*
11	–	0 1	/	12 0 6	f	11 8 5	)
0	0	0 2	S	12 0 7	g	11 8 6	;
1	1	0 3	T	12 0 8	h	11 8 7	^
2	2	0 4	U	12 0 9	i	0 9 5	LF
3	3	0 5	V	12 9 1	SOH	0 9 6	ETB
4	4	0 6	W	12 9 2	STX	0 9 7	ESC
5	5	0 7	X	12 9 3	ETX	0 8 2	\
6	6	0 8	Y	12 9 5	HT	0 8 3	,
7	7	0 9	Z	12 9 7	DEL	0 8 4	%
8	8	9 2	SYN	12 8 2	[	0 8 5	_
9	9	9 7	EOT	12 8 3	.	0 8 6	>
12 11	|	8 1	`	12 8 4	<	0 8 7	?
12 0	{	8 2	:	12 8 5	(	9 8 4	DC4
12 1	A	8 3	#	12 8 6	+	9 8 5	NAK
12 2	B	8 4	@	12 8 7	!	9 8 7	SUB
12 3	C	8 5	′	11 0 1	~	12 9 8 3	VT
12 4	D	8 6	=	11 0 2	s	12 9 8 4	FF
12 5	E	8 7	″	11 0 3	t	12 9 8 5	CR
12 6	F	12 11 1	j	11 0 4	u	12 9 8 6	SO
12 7	G	12 11 2	k	11 0 5	v	12 9 8 7	SI
12 8	H	12 11 3	l	11 0 6	w	11 9 8 1	EM
12 9	I	12 11 4	m	11 0 7	x	11 9 8 4	FS
11 0	}	12 11 5	n	11 0 8	y	11 9 8 5	GS
11 1	J	12 11 6	o	11 0 9	z	11 9 8 6	RS
11 2	K	12 11 7	p	11 9 1	DC1	11 9 8 7	US
11 3	L	12 11 8	q	11 9 2	DC2	0 9 8 5	ENQ
11 4	M	12 11 9	r	11 9 3	DC3	0 9 8 6	ACK
11 5	N	12 0 1	a	11 9 6	BS	0 9 8 7	BEL
11 6	O	12 0 2	b	11 9 8	CAN	12 11 9 8 1	DLE
11 7	P	12 0 3	c	11 8 2	]	12 0 9 8 1	NUL

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

EARLY DEC CARD CODES

Character	7-Bit Octal	DEC 029	DEC 026	Character	7-Bit Octal	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

†The Monitor accepts either punch for input but outputs only the triple punch.

These two DEC card codes provide a representation for the figure and upper case character subset. DEC 029 is not available in all programs, but it is almost identical to the ASCII subset, differing only in the four column punches indicated by asterisks as follows:

DEC 029	11 8 2	11 8 7	0 8 2	12 8 7
ASCII	12 8 7	0 8 2	11 8 2	11 8 7

The next two pages show the relationship among the various character sets for the column punches listed above, and where they exist, the corresponding single-key punch configurations and printed characters for several IBM key punches.

Column Punch	Character	Column Punch	Character
None	*Space*	12 9	I
0	0	11 1	J
1	1	11 2	K
2	2	11 3	L
3	3	11 4	M
4	4	11 5	N
5	5	11 6	O
6	6	11 7	P
7	7	11 8	Q
8	8	11 9	R
9	9	0 1	/
12 1	A	0 2	S
12 2	B	0 3	T
12 3	C	0 4	U
12 4	D	0 5	V
12 5	E	0 6	W
12 6	F	0 7	X
12 7	G	0 8	Y
12 8	H	0 9	Z

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	ASCII
12	&	+	&	+	&	&
11	–	–	–	–	–	–
12 0				?		
11 0				:		
8 2			:	_	:	:
8 3	#	=	#	=	#	#
8 4	@	-	@	@	@	@
8 5			'	^	'	'
8 6			=	'	=	=
8 7			"	\	"	"
12 8 2			¢	?	[	[
12 8 3			.	.	.	.

above, and where they exist, the corresponding single-key punch configurations and printed characters for several IBM key punches.

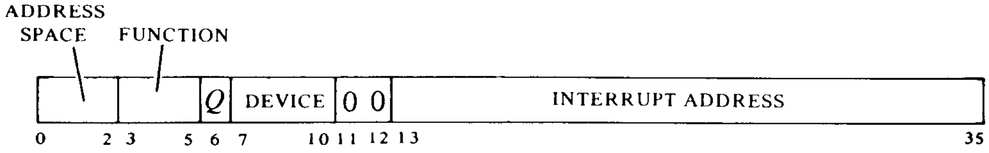
Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	ASCII
12 8 4	¤	)	<	)	<	<
12 8 5			(	]	(	(
12 8 6			+	<	+	+
12 8 7			|	!	^	!
11 8 2			!	:	!	]
11 8 3	$	$	$	$	$	$
11 8 4	*	*	*	*	*	*
11 8 5		'	)	[	)	)
11 8 6			;	>	;	;
11 8 7			¬	&	\	^
0 8 2			*See note*	;	]	\
0 8 3	,	,	,	,	,	,
0 8 4	%	(	%	(	%	%
0 8 5			←	"	_	_
0 8 6			>	#	>	>
0 8 7			?	%	?	?
7 9				*Binary*	*Binary*	EOT
12 0 2 4 6 8				*Mode Switch*	*Mode Switch*	
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.

Appendix C
Internal Device Bit Assignments

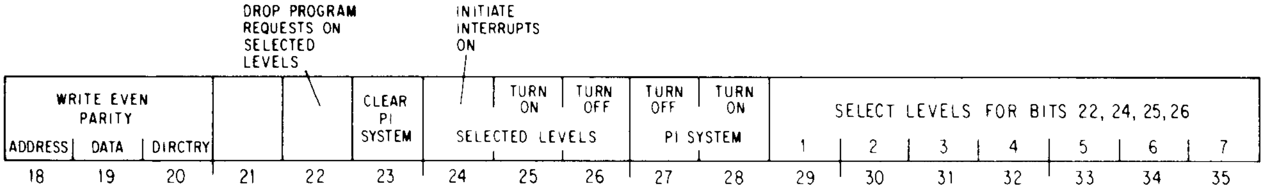
The drawings on the following pages define the meanings of the bits in the half words and full words of control and status information handled by the system instructions for the internal devices in the KL10, KS10, KI10 and KA10 processors (bits that cause interrupts are indicated by asterisks). The section on each device also lists all other instructions for that device, showing the operations performed in symbolic form using the conventions defined for the representation of the user instructions in Appendix A (see page A–16).

Priority Interrupt PI 004

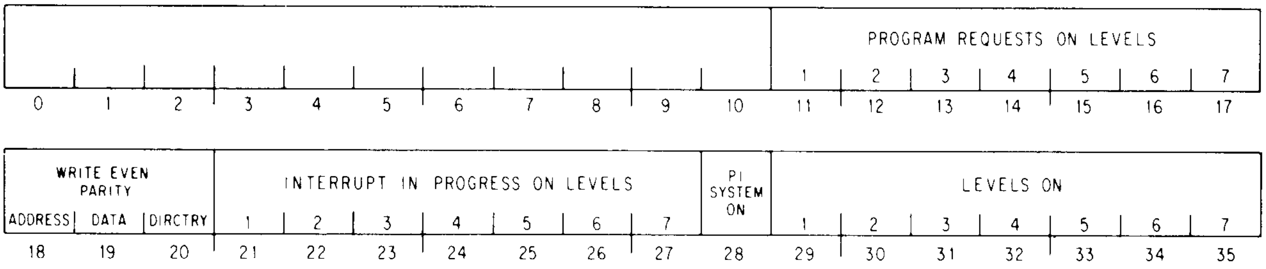


FUNCTION WORD

CONO PI, 70060



CONI PI, 70064

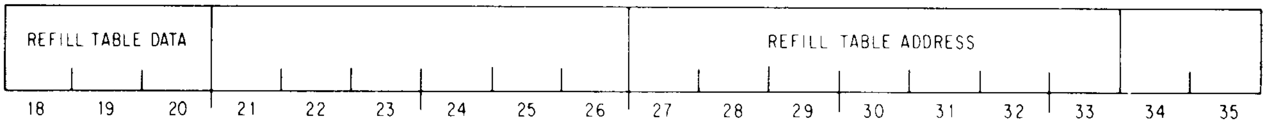


Cache CCA 014 APR 000

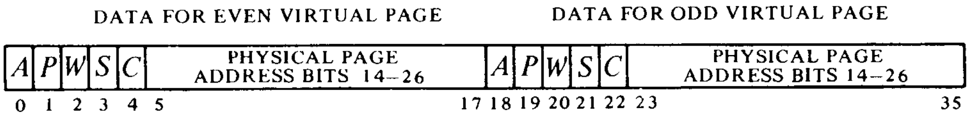
All sweeps initially set Sweep Busy, and at termination they clear Sweep Busy and set Sweep Done.

SWPIA	70144	*Invalidate all pages*	SWPIO	70164	*Invalidate page* E
SWPVA	70150	*Validate all pages*	SWPVO	70170	*Validate page* E
SWPUA	70154	*Unload all pages*	SWPUO	70174	*Unload page* E

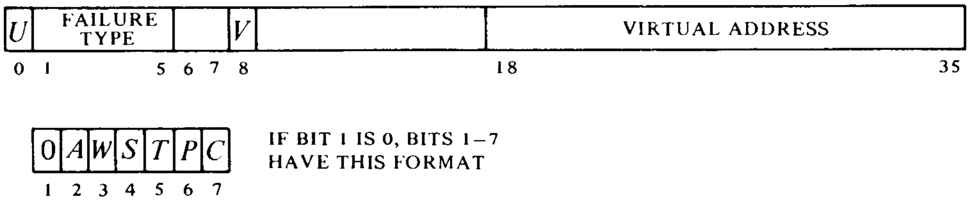
WRFIL 70010



TOPS-10 Paging

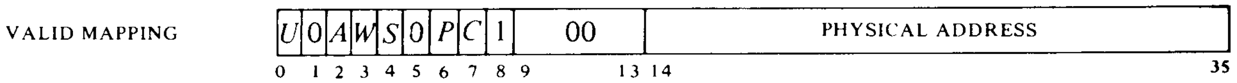

DATA FOR EVEN VIRTUAL PAGE
DATA FOR ODD VIRTUAL PAGE
A P W S C
PHYSICAL PAGE ADDRESS BITS 14–26
A P W S C
PHYSICAL PAGE ADDRESS BITS 14–26
0 1 2 3 4 5
17 18 19 20 21 22 23
35

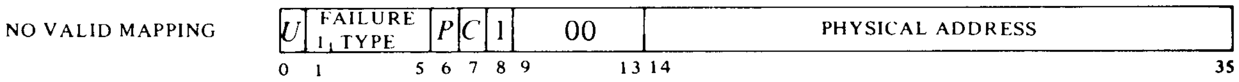
PAGE MAP WORD



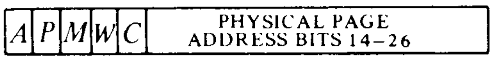
PAGE FAIL WORD

MAP 257



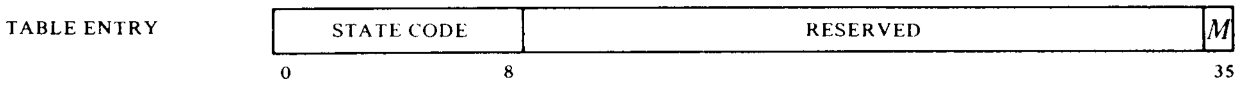


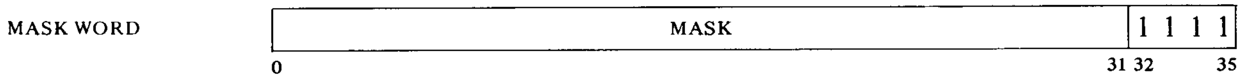
TOPS-20 Paging



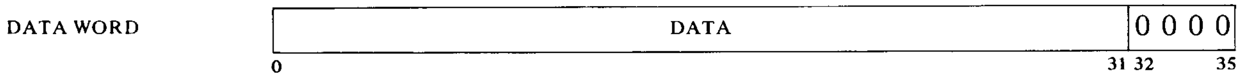
PAGE MAPPING

CST Words



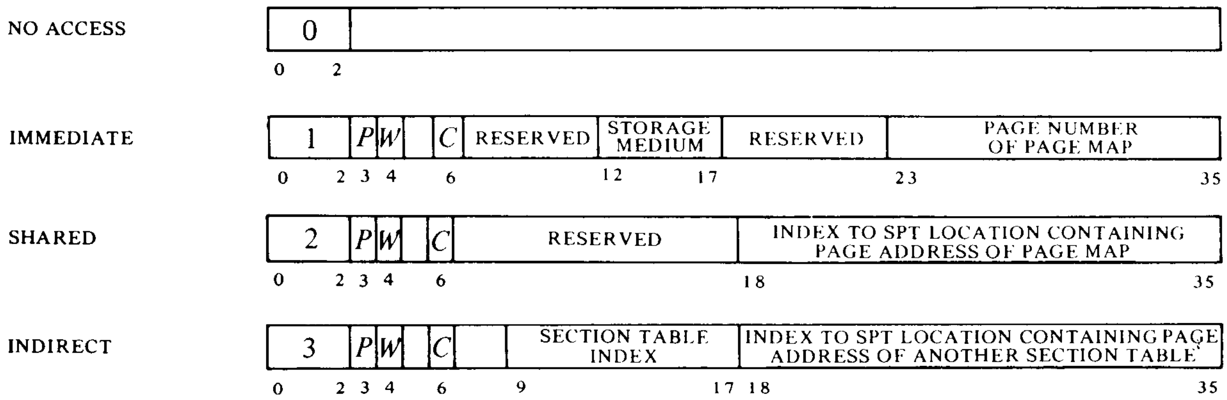


CST MASK WORD

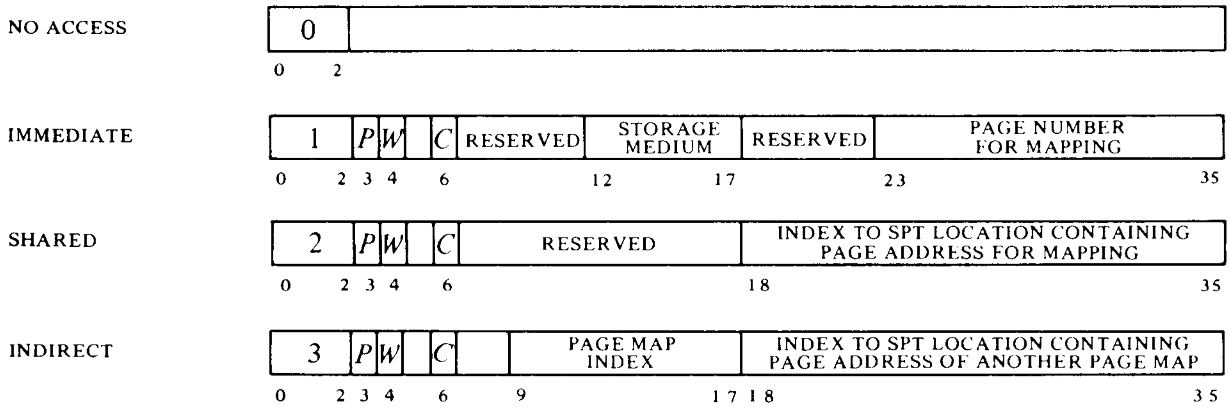


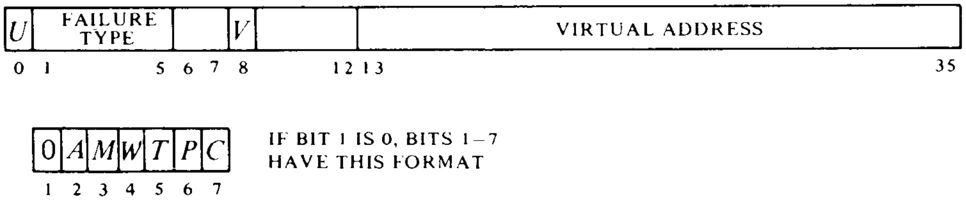
CST DATA WORD

Section Pointers


NO ACCESS
0
0 2
IMMEDIATE
1 P W C RESERVED STORAGE MEDIUM RESERVED PAGE NUMBER OF PAGE MAP
0 2 3 4 6 12 17 23 35
SHARED
2 P W C RESERVED INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF PAGE MAP
0 2 3 4 6 18 35
INDIRECT
3 P W C SECTION TABLE INDEX INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER SECTION TABLE
0 2 3 4 6 9 17 18 35

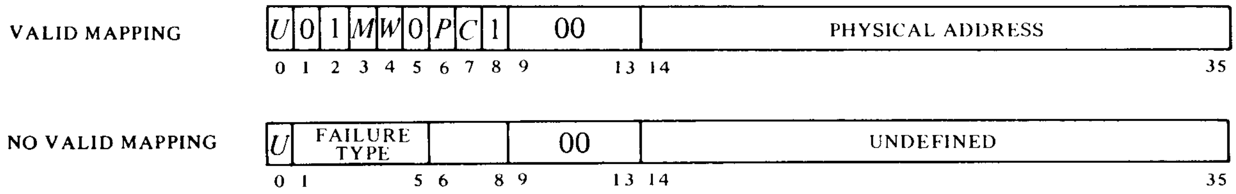
Map Pointers


NO ACCESS
0
0 2
IMMEDIATE
1 P W C RESERVED STORAGE MEDIUM RESERVED PAGE NUMBER FOR MAPPING
0 2 3 4 6 12 17 23 35
SHARED
2 P W C RESERVED INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS FOR MAPPING
0 2 3 4 6 18 35
INDIRECT
3 P W C PAGE MAP INDEX INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER PAGE MAP
0 2 3 4 6 9 17 18 35

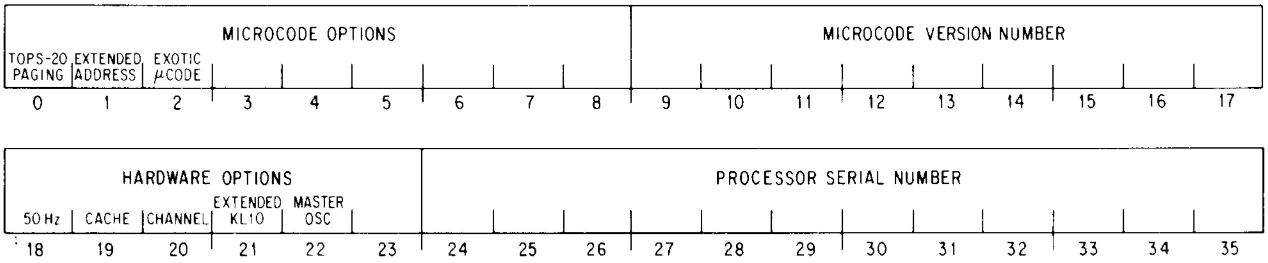

U FAILURE TYPE V VIRTUAL ADDRESS
0 1 5 6 7 8 12 13 35
0 A M W T P C
1 2 3 4 5 6 7
IF BIT 1 IS 0, BITS 1–7 HAVE THIS FORMAT

PAGE FAIL WORD

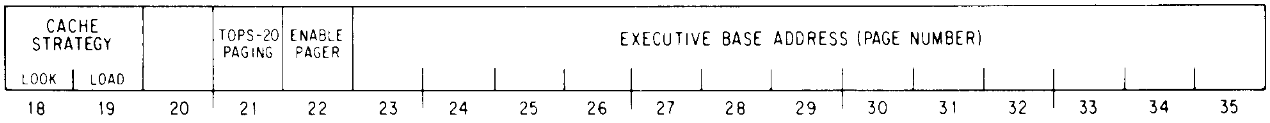
MAP 257



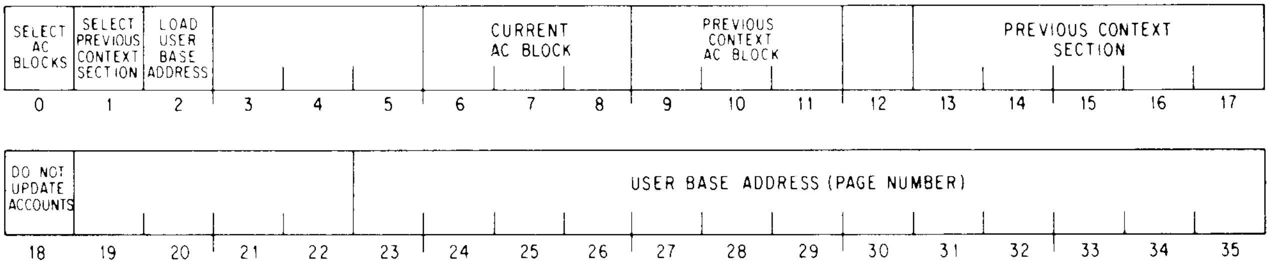
APRID   70000



CONO PAG,   70120
CONI PAG,   70124

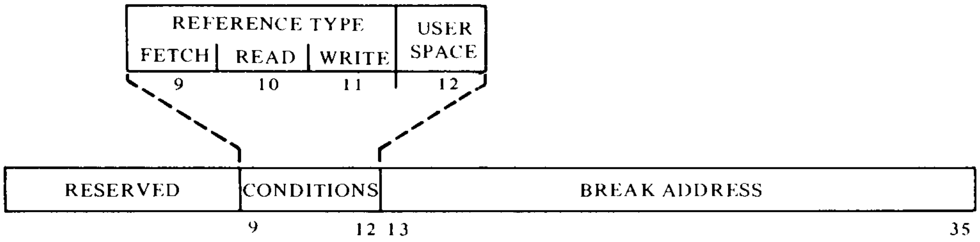


DATAO PAG,   70114
DATAI PAG,   70104



CLRPT   70110   *Invalidate page table line(-pair)* E_{18-23}

DATAO APR,   70014
DATAI APR,   70004



EVEN NUMBERED WORD	ODD NUMBERED WORD		
HIGH ORDER PART OF COUNT	0	LOW ORDER PART OF COUNT	RESERVED
0 – 35	0	1 – 23	24 – 35
		36 – 58	
		COUNTER	
		43 – 58	

DOUBLEWORD METER COUNT

CONO MTR, 70260

			ACCOUNTING			TIME BASE									PRIORITY INTERRUPT ASSIGNMENT		
SET UP ACCOUNTS			EXECUTIVE PI ACCOUNT	EXECUTIVE NON-PI ACCOUNT	TURN ON	TURN OFF	TURN ON	CLEAR									
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

CONI MTR, 70264

			ACCOUNTING												PRIORITY INTERRUPT ASSIGNMENT		
			EXECUTIVE PI ACCOUNT	EXECUTIVE NON-PI ACCOUNT	ON		TIME BASE ON										
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

RDTIME 70204 (EPT 510,511) + (COUNTER) → (E,E+1)

CONO TIM, 70220

CLEAR INTERVAL COUNTER			TURN INTERVAL COUNTER ON	CLEAR INTERVAL FLAGS		INTERVAL PERIOD											
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

CONI TIM, 70224

						INTERVAL COUNT											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
				*													
			INTERVAL COUNTER			INTERVAL PERIOD											
			ON	DONE	OVERFLOW												
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

RDEACT 70244 (UPT 504,505) + (COUNTER) → (E,E+1)

RDMACT 70240 (UPT 506,507) + (COUNTER) → (E,E+1)

WRPAE 70210

												†	†	†	†		
SELECT CHANNELS									IGNORE µCODE	SELECT PROBE		SELECT MEMORY CONDITIONS					
0	1	2	3	4	5	6	7	IGNORE		LOW	IGNORE	E BOX WAIT	CACHE MISS	CACHE WRITE BACK	SWEEP WRITE	IGNORE	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

SELECT INTERRUPT LEVELS									SELECT MODE		SELECT EVENT METHOD	CLEAR COUNTER					
0	1	2	3	4	5	6	7	NONE	USER	IGNORE							
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

†*0s select conditions in these bits*

RDPERF 70200 (EPT 512,513) + (COUNTER) → (E,E+1)

Processor Error Logic APR 000 PI 004

CONO APR, 70020

	CLEAR ALL IN-OUT DEVICES	ENABLE	DISABLE	CLEAR	SET	SELECT FLAGS FOR BITS 20 - 23									PRIORITY INTERRUPT ASSIGNMENT		
		SELECTED FLAGS				S BUS ERROR	NO MEMORY	IN-OUT PAGE FAILURE	MB PARITY	CACHE DIRCTRY	ADDRESS PARITY	POWER FAILURE	SWEEP DONE				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

CONI APR, 70024

						FLAGS ENABLED TO INTERRUPT											
						S BUS ERROR	NO MEMORY	IN-OUT PAGE FAILURE	MB PARITY	CACHE DIRCTRY	ADDRESS PARITY	POWER FAILURE	SWEEP DONE				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

						*	*	*	*	*	*	*	*				
	SWEEP BUSY					S BUS ERROR	NO MEMORY	IN-OUT PAGE FAILURE	MB PARITY ERROR	CACHE DIRCTRY PARITY ERROR	ADDRESS PARITY ERROR	POWER FAILURE	SWEEP DONE	INTRUPT REQUEST	PRIORITY INTERRUPT ASSIGNMENT		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

RDERA 70040

WORD NUMBER		REFERENCE IDENTIFICATION					INDETERMINATE		0	0	0	0	0	HIGH ORDER ADDRESS BITS			
		SWEEP	CHANNEL	DATA	SOURCE	WRITE											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

PHYSICAL ADDRESS OF FIRST WORD OF TRANSFER																	
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

IO Address

00000	C	REGISTER ADDRESS
0 13	14 17	18 35

Priority Interrupt

WRPI 70060

				DROP PROGRAM REQUESTS ON SELECTED LEVELS		INITIATE INTERRUPTS ON											
					CLEAR PI SYSTEM		TURN ON	TURN OFF	TURN OFF	TURN ON	SELECT LEVELS FOR BITS 22, 24, 25,26						
						SELECTED LEVELS			PI SYSTEM		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

RDPI 70064

											PROGRAM REQUESTS ON LEVELS						
											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

			INTERRUPT IN PROGRESS ON LEVELS							PI SYSTEM ON	LEVELS 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

TOPS-10 Paging

DATA FOR EVEN VIRTUAL PAGE							DATA FOR ODD VIRTUAL PAGE						
A		*W*	*S*	*C*		PHYSICAL PAGE ADDRESS BITS 17–26	*A*		*W*	*S*	*C*		PHYSICAL PAGE ADDRESS BITS 17–26
0	1	2	3	4		8 17	18	19	20	21	22		26 35

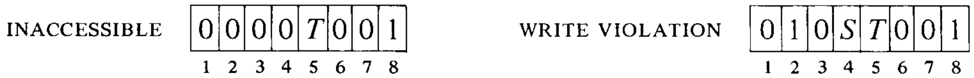
PAGE MAP WORD

U	36 OR 37	0 0	*P*	000		ADDRESS
0	1 5		8		17	18 35

HARD PAGE FAIL WORD 36 OR 37

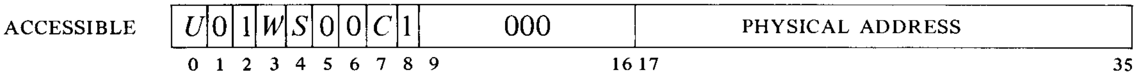
U	20	0	0	1	0	1	0	0	*B*	IO ADDRESS
0	1 5			8		10			13	14 35

HARD PAGE FAIL WORD 20

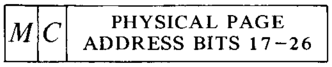


SOFT PAGE FAIL WORD

MAP 257

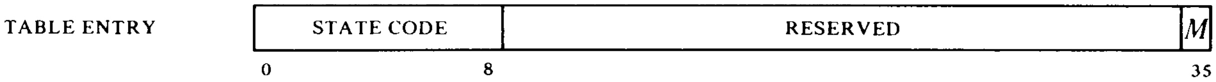


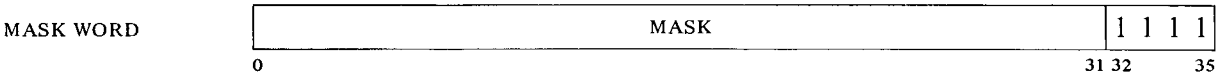
TOPS-20 Paging

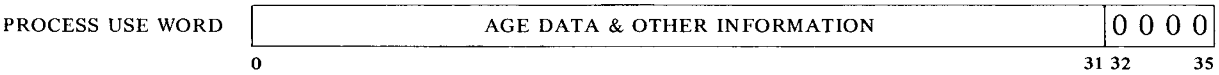


PAGE MAPPING

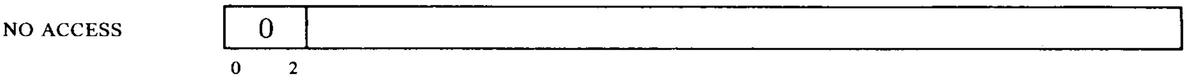
CST Words

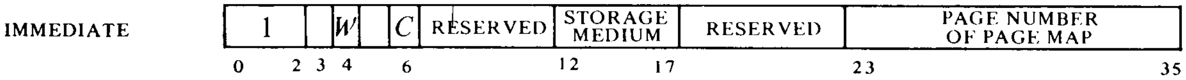


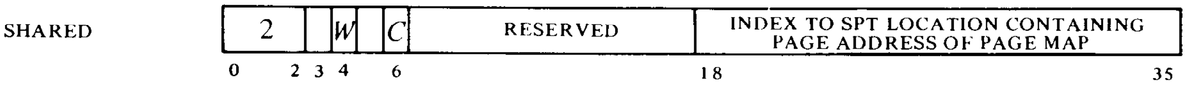


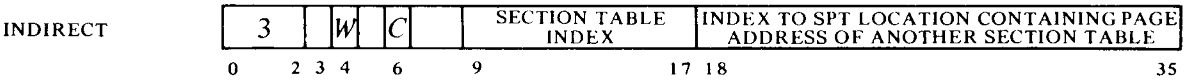


Section Pointers

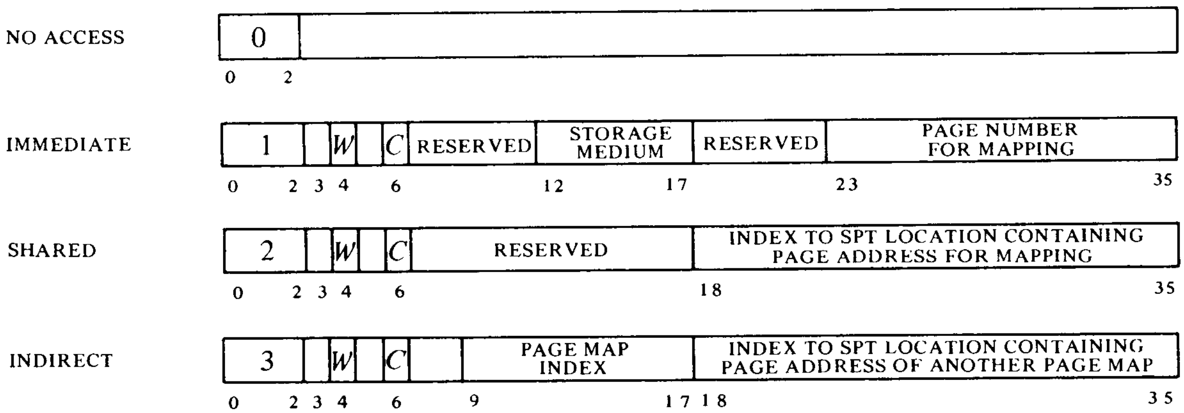


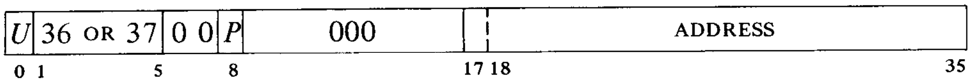




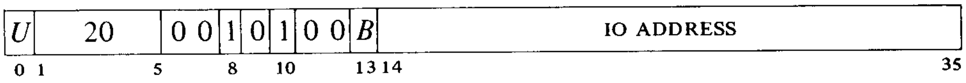

INDIRECT
3
W
C
SECTION TABLE INDEX
INDEX TO SPT LOCATION CONTAINING PAGE ADDRESS OF ANOTHER SECTION TABLE
0
2 3 4
6
9
17 18
35

Map Pointers

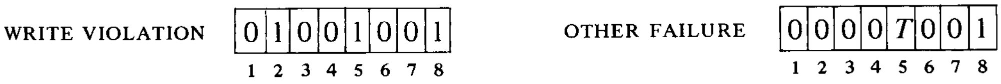




HARD PAGE FAIL WORD 36 OR37

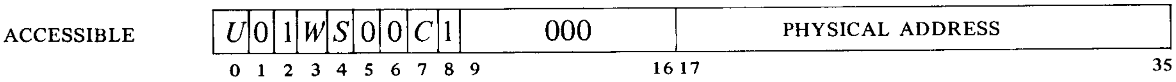


HARD PAGE FAIL WORD 20



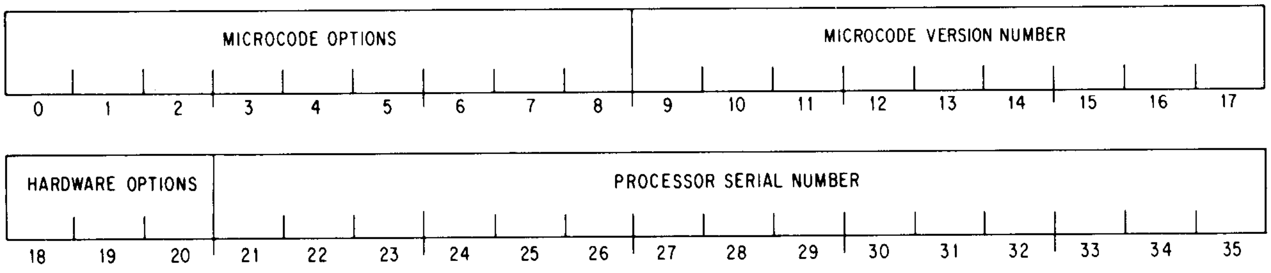
SOFT PAGE FAIL WORD

MAP 257

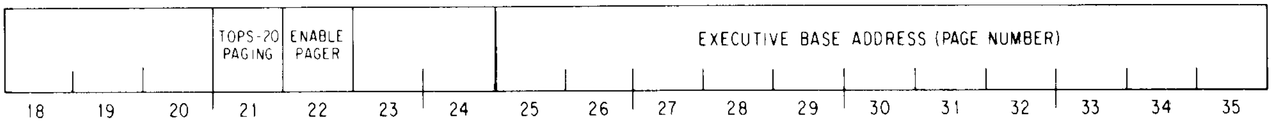


Memory Management

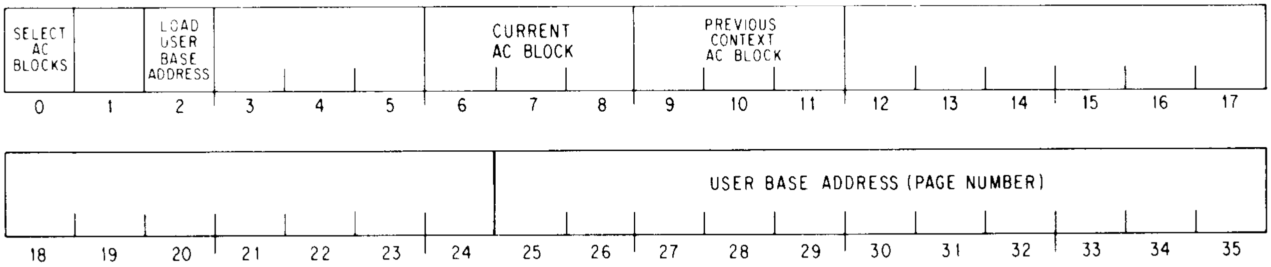
APRID 70000



WREBR 70120
RDEBR 70124



WRUBR 70114
RDUBR 70104



CLRPT	70110	*Invalidate page table line* $E_{18\text{-}26}$ *Invalidate cache*
WRSPB	70240	(E) → (SPT BASE)
RDSPB	70200	(SPT BASE) → (E)
WRCSB	70244	(E) → (CST BASE)
RDCSB	70204	(CST BASE) → (E)
WRCSTM	70254	(E) → (CST MASK)
RDCSTM	70214	(CST MASK) → (E)
WRPUR	70250	(E) → (PROCESS USE)
RDPUR	70210	(PROCESS USE) → (E)
UMOVE	704	PXCT 4,[MOVE *A*,*E*]
UMOVEM	705	PXCT 4,[MOVEM *A*,*E*]

System Timing

WRTIM	70260	$(E,E+1_{0\text{-}23}) \rightarrow$ (TIME BASE) $0 \rightarrow$ (TIME BASE$_{24\text{-}35}$)
RDTIM	70220	(TIME BASE) + (COUNTER) → (E,E+1)
WRINT	70264	(E) → (INTERVAL)
RDINT	70224	(INTERVAL) → (E)

Halt Status

HALT CODE → (0)
(PC) → (1)
(REGISTER FILE & VMA) → (HALT STATUS BLOCK)

WRHSB	70270	(E) → (HALT STATUS BASE) *If* $(E)_0 = 0$: *disable status storage*
RDHSB	70230	(HALT STATUS BASE) → (E)

System Flags

WRAPR 70020

		ENABLE	DISABLE	CLEAR	SET	SELECT FLAGS FOR BITS 20 – 23									PRIORITY INTERRUPT ASSIGNMENT		
		SELECTED FLAGS				FLAG 24	INTRUPT CONSOLE	POWER FAILURE	NO MEMORY	BAD MEMORY DATA	CORECTD MEMORY DATA	INTERVAL DONE	CONSOLE INTRUPT				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

RDAPR 70024

						FLAGS ENABLED TO INTERRUPT											
						FLAG 24	INTRUPT CONSOLE	POWER FAILURE	NO MEMORY	BAD MEMORY DATA	CORECTD MEMORY DATA	INTERVAL DONE	CONSOLE INTRUPT				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

						*	*	*	*	*	*	*	*				
						FLAG 24	0	POWER FAILURE	NO MEMORY	BAD MEMORY DATA	CORECTD MEMORY DATA	INTERVAL DONE	CONSOLE INTRUPT	INTRUPT REQUEST	PRIORITY INTERRUPT ASSIGNMENT		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Memory Status

Read

ERROR HOLD	UNCORRECTABLE ERROR HOLD	REFRESH ERROR	PARITY ERROR	ECC ON	ERROR CORRECTION CODE							POWER FAILED		HIGH ORDER ADDRESS BITS			
					CP	C40	C20	C10	C4	C2	C1						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

LAST ADDRESS OR FIRST ERROR ADDRESS																	
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Write

CLEAR ERROR HOLD (1)		CLEAR REFRESH ERROR (1)	PARITY ERR									CLEAR POWER FAILED (0)					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

										FORCE CHECK BITS							ECC OFF
										CP	C40	C20	C10	C4	C2	C1	
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

KI10 PROCESSOR

Console APR 000 PI 004 PTR 104

DATAI APR, 70004 (DS) → (E) (RSW)

DATAO PI, 70054 *If* MI PROG DIS = 0: (E) → (MI)
1 → PROGRAM DATA

DATAO PTR, 71054

INST FETCH	DATA FETCH	WRITE		ADDRESS BREAK	EXEC PAGING	USER PAGING		
0	1	2	3	4	5	6	7	8

		ADDRESS SWITCHES
0	6	14 35

Priority Interrupt PI 004

	FUNCTION	INCREMENT	INTERRUPT ADDRESS
3	5	6 17	18 35

FUNCTION WORD

CONO PI, 70060

CLEAR POWER FAILURE FLAG	CLEAR PARITY ERROR FLAG	DISABLE	ENABLE	DROP PROGRAM REQUESTS ON SELECTED LEVELS	CLEAR PI SYSTEM	INITIATE INTERRUPTS ON	TURN ON	TURN OFF	DEACTIVATE PI	ACTIVATE PI	SELECT LEVELS FOR BITS 22, 24, 25, 26						
		PARITY ERROR INTERRUPT				SELECTED LEVELS					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

CONI PI, 70064

INST FETCH	DATA FETCH	WRITE	ADDRESS STOP	ADDRESS BREAK	EXEC PAGING	USER PAGING	PAR STOP	NXM STOP			PROGRAM REQUESTS ON LEVELS						
											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

			INTERRUPT IN PROGRESS ON LEVELS							PI SYSTEM ON	LEVELS ON (ACTIVE)						
			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

MR-0745

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

PAGE MAP WORD

	U	VIRTUAL PAGE		FAILURE TYPE
	8	9 17		31 35

IF BIT 31 IS 0, BITS 31-35 HAVE THIS FORMAT

0	*A*	*W*	*S*	*T*
31	32	33	34	35

PAGE FAIL WORD

DATAO PAG, 70114 0 → ASSOCIATIVE MEMORY
DATAI PAG, 70104

LOAD LEFT	USER FAST MEMORY BLOCK	SMALL USER	USER ADDRESS COMPARE ENABLE	USER BASE ADDRESS
0	1 2	3	4	5 6 7 8 9 10 11 12 13 14 15 16 17

LOAD RIGHT			PAGE ENABLE	EXECUTIVE BASE ADDRESS
18	19 20	21	22	23 24 25 26 27 28 29 30 31 32 33 34 35

CONO PAG, 70120

EXECUTIVE AC STACK POINTER		PAGE TABLE RELOAD COUNTER
18 19 20 21 22	23 24 25 26 27 28 29 30	31 32 33 34 35

CONI PAG, 70124

PROCESSOR SERIAL NUMBER	
0 1 2 3 4 5 6 7 8 9	10 11 12 13 14 15 16 17

COMPLEMENT OF VIRTUAL PAGE NUMBER	EXECUTIVE ADDRESS SPACE		WORD EMPTY	PAGE TABLE RELOAD COUNTER
18 19 20 21 22 23 24 25 26	27	28 29	30	31 32 33 34 35

MR-0746

MAP 257

PAGE FAILURE	*P*	*W*	*S*	NO MATCH	PHYSICAL PAGE ADDRESS BITS 14-26												
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Processor Conditions APR 000

CONO APR, 70020

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

CONI APR, 70024

	MEM OVERLAP DISABLE	FM MANUAL	MI PROG DISABLE	CONSOLE DATA LOCK	CONSOLE LOCK	50 HERTZ	MARGIN ENABLE	MAINTENANCE MODE	POWER ALARM	VOLTAGE MONITOR 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

	*			*				*		*	*						
TIME OUT	PARITY ERROR	PARITY ERROR INTERRUPT ENABLED	TIMER ENABLED	POWER FAILURE	AUTO RESTART DISABLED		CLOCK INTERRUPT ENABLED	CLOCK		IN-OUT PAGE FAILURE	NONEXISTENT MEMORY	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

DATAO APR, 70014

							TURN OFF	TURN ON					MARGIN ADDRESS				
							VOLTAGE MARGINS										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

				WRITE EVEN PARITY	TURN OFF	TURN ON						MARGIN VALUE					
					SPEED MARGINS												
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

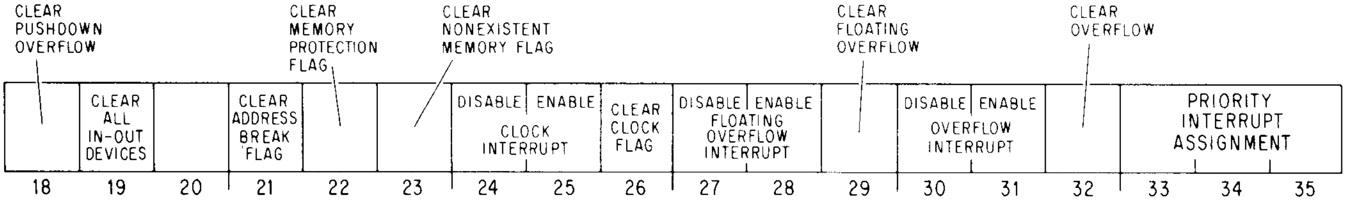
MR-0747

Console APR 000

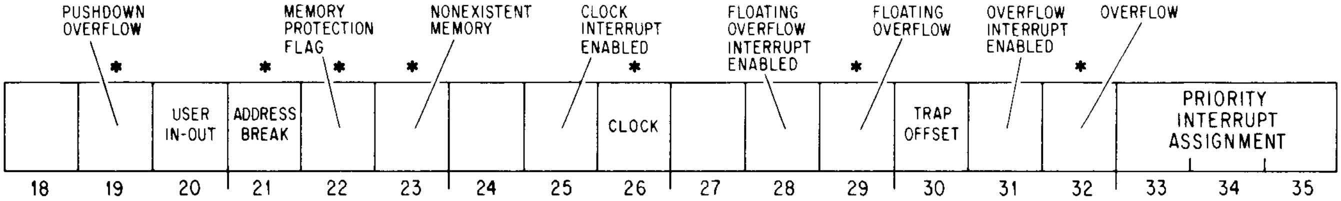
DATAI APR, 70004 (DS) → (E) (RSW)

Processor Conditions APR 000

CONO APR, 70020



CONI APR, 70024



Priority Interrupt PI 004

CONO PI, 70060

CLEAR POWER FAILURE FLAG	CLEAR PARITY ERROR FLAG	DISABLE	ENABLE		CLEAR PI SYSTEM	INITIATE INTERRUPTS ON	TURN ON	TURN OFF	DEACTIVATE PI	ACTIVATE PI	SELECT LEVELS FOR BITS 24, 25, 26						
		PARITY ERROR INTERRUPT				SELECTED LEVELS					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

CONI PI, 70064

POWER FAILURE	PARITY ERROR	PARITY ERROR INTERRUPT ENABLED	INTERRUPT IN PROGRESS ON LEVELS							PI SYSTEM ON	LEVELS ON (ACTIVE)						
			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

Memory Management APR 000

DATAO APR, 70014

$P_{l_{18-25}}$		$P_{h_{18-25}}$	P	$R_{l_{18-25}}$		$R_{h_{18-25}}$	
0 7	8	9 16	17	18 25	26	27 34	35

MR-0749

Appendix D
Timing

This appendix contains tables and charts for determining the time taken by instructions in PDP–10 processors. But some advice is in order before the reader turns to the timing information. It is quite likely that inspecting tables and charts for the fastest instructions takes more time than is saved using them. Moreover there are other considerations of far greater import to good programming than speed. The primary objective should be to generate code that is correct, clear and concise. Only after producing code having these qualities should the programmer concern himself with the timing; and even then there are general principles whose employment is considerably more valuable than searching through charts and tables.

There are two levels at which time enters into the running of programs: the execution of the code itself, and the service the program requires of the Monitor. The latter is principally a function of the organization of the program and is discussed in some detail in the final paragraphs of §2.19. In terms of writing fast code, the most important single factor is to pick a fast algorithm. Here are some guidelines.

Loops are slow.

Always try to use one big loop instead of many small ones.

Put loops in subroutines, rather than subroutine calls in loops.

Set up data structures to minimize searching; instead index directly into tables. If direct indexing is not feasible, divide tables into sublists wherever possible.

Avoid testing inside loops for conditions that are constant throughout the loop.

Avoid recomputing constants.

Use pointers rather than moving blocks of data.

UUOs are slow. If an extra ten instructions will save you from doing a UUO half the time, use them.

IO is very slow. Keep information in memory as much as possible. When it is not possible, move data to disk, but always keep in memory that part most likely to be needed next. In any event always keep track of what information is where; nothing wastes time more than bringing in the same data twice. It is also a good idea to keep track of what is in your buffers if there is any chance you will have to back up.

After finding the best procedure for doing a particular job, try to minimize the number of instructions, and following that the number of memory references. These two are obviously related, as every instruction requires at least the memory reference to fetch it. Generally speaking the fewer instructions, the faster the code will go; and there is the added benefit of fewer places for coding errors. But such rules should never be followed blindly, as there are various tradeoffs that must be taken into consideration. It is seldom useful to decrease the number of instructions or references by 10%, only to double the storage requirement in the process. For an exposition of such tradeoffs, refer to the discussion of parity procedures given in §2.15. And also keep in mind that an increase in storage space may mean an increase in Monitor service.

The reason that minimizing the number of instructions and/or memory references tends to minimize time is that for most of the simpler instructions — data handling, Boolean, test — memory access is the dominant factor in instruction time. This is not true for those instructions that contain extensive repetitive procedures, such as multiply and divide. A single MUL may entail a dozen additions, but it requires no more memory time than a single ADD. Substituting an MUL for two or three Boolean or test instructions rarely saves time. However use of the cache can reduce total memory access time by 90%. The programmer can help achieve such reduction by using the same locations for temporary storage in different programs, and by making multiple use of subroutines rather than repeating common code. A good rule of thumb for approximating program time is to figure about a microsecond per memory reference, minus a suitable saving for cache use, and then add a factor for shift operations. The KL10 has a shift matrix, so in it the shift, rotate and byte instructions require no iterative shifting. But iterative shifting does occur in multiply, divide, JFFO, unnormalizing one operand with respect to the other in floating add and subtract, fixing and floating a number, and normalizing a floating point result.

After coding using the above guidelines and suggestions, run SNOOPY and TATTLE to determine where the program is really spending its time and whether further local optimization is worthwhile. It usually is not, but where it is, use the information given in the following pages. A note of caution, however, concerning that information. No instruction times are measured. The timing charts are constructed from delay times and published times for various circuits used in the hardware. Specific instruction times listed in tables and elsewhere are calculated from the timing charts. There are therefore manifest sources of inaccuracy in the information without considering the obvious fact that no two machines ever run exactly alike. Be especially leery of attaching any significance to the last digit.

Do not assume that the fastest algorithm on one type of PDP–10 processor will be the fastest on another, or that instruction times given for one processor can be regarded as a relative indication of the times for another. Each new processor is faster than its predecessors, but different instructions are speeded up different amounts. Since the shorter instructions are dependent almost entirely on memory cycle time, they go faster only with a faster memory but are affected greatly by use of a cache. Among the slower instructions, it is the most frequently used that get speeded up the most in a new machine. One thing a programmer can safely assume is that with each new machine, equivalent instructions will tend toward taking the same amount of time. In all processors the fastest jump is JRST; the fastest no-op and absolute skip are as follows.

	No-op	*Absolute Skip*
KL10	TRN	TRNA
KS10	TRN	TRNA
KI10	JFCL	TRNA
KA10	JFCL	CAIA

KI10 Instruction Times

The table on the next two pages lists the processor execution time in microseconds for each instruction beginning with its address calculation. The times do not include the instruction fetch (.89 microsecond), as this is overlapped with the preceding instruction execution; in each case the processor time needed to complete the instruction fetch depends upon the extent of the overlap, a factor that varies from one instruction to another. The time listed is that required for direct addressing without indexing (i.e. with no effective address calculation), and assuming E addresses an ME10 or MF10 core location (1 μs cycle), except in DFN and UFA, which are most frequently used with E equal to A+1. It is further assumed that no conflicts develop in memory access — in other words there are a number of interleaved memories, and data blocks are kept in separate memories from instructions, so the processor need never wait for operand access while a given memory completes a cycle from an instruction fetch.

To arrive at more complete execution times for various circumstances, make the following adjustments to the figures given in the table. For indexing add 0.06; for indirect addressing add 1.02 for each address cycle without indexing, 1.08 for each with indexing. If the final address cycle includes indexing, add 0.12 to JRA, POP and POPJ, and add 0.11 to any instruction that does not fetch a memory operand. If memory operand storage is in fast memory, subtract 0.08 unless there is also storage in a second accumulator, in which case add 0.03. For more esoteric considerations: add 1.11 for each page refill cycle; add 0.09 to every page check in an instruction executed by a PXCT, from the console, or in an interrupt; and add 0.20 per read access if the machine is running with the parity stop switch on. The time given for MAP assumes no page failure.

Following the table is a chart (with intervals in nanoseconds) that can be used for calculating the instruction time in almost any circumstances, with any memory, etc. However neither table nor chart includes any information about interrupts, page failures, bus conflicts between interrupts and IO instructions, or other special situations.

Memory access by the processor is divided into three parts: page check, request setup, and the actual access cycle over the memory bus. In an instruction fetch, the first two of these can be overlapped with operand storage, but not the third. The effect of this on instruction fetch time is as follows. If an instruction does not store a memory operand (either because it has no operand or stores the result in an accumulator), probably the next instruction fetch will be overlapped entirely: hence the second instruction will be ready by the time the first is done. If an instruction stores a memory operand, there is no overlap on the bus, but most likely the page check and request setup will already have been performed (these show up as the 175 preceding each read access in the chart). After the write access is complete an instruction has 147 nanoseconds to go, during which period the read access for the next instruction can begin. Finally some instructions put off triggering the next instruction fetch until near the very end, but even in the worst case (BLT) there is a minimum overlap of 87 nanoseconds, which is enough for the page check (81).

KI10 INSTRUCTION TIMES

Full Words

EXCH	1.61	MOVE MOVS	1.26
		MOVEI MOVSI	.45
BLT	1.35	MOVEM MOVSM	1.06
+ *per word*		MOVES MOVSS	1.61
Memory → memory	1.59	MOVN MOVM	1.32
AC → memory	1.21	MOVNI MOVMI	.51
Memory → AC	1.42	MOVNM MOVMM	1.12
		MOVNS MOVMS	1.67

Half Words

Basic	1.26
Immediate	.45
Memory, no action	1.72
Memory, some action	1.06
Self	1.61

Byte

IBP	1.90
LDB	2.87–6.72
DPB	3.12–4.99
ILDB	3.54–7.39
IDPB	3.80–5.67

Pushdown

PUSH	1.94
POP	2.16
PUSHJ	1.12
POPJ	1.43

Arithmetic Test

AOBJP AOBJN	.57
CAI	.62
CAM	1.43
JUMP	.56
AOJ SOJ	.62
SKIP	1.37
AOS SOS	1.78

In-out

BLKO	1.51 + DATAO	
BLKI	1.51 + DATAI	
	Fast	*Slow*
DATAO	3.13	4.12
DATAI CONI	2.05	3.37
CONO	2.32	3.31
CONSO CONSZ	1.55	2.87

Program Control

JSR	.95
JSP	.45
JRST	.34
JEN	.34
PORTAL	.34
JRSTF	.45
JSA	1.06
JRA	1.59
JFCL	.34
JFFO	.79–2.66
XCT	.34
LUUO	1.06
MUUO	2.83
MAP	.60

Logical Test

TLN	.45	TRN	.34	TDN	1.15	TSN	1.26
TLNE	.62	TRNE	.51	TDNE	1.32	TSNE	1.43
TLNA	.56	TRNA	.45	TDNA	1.26	TSNA	1.37
TLNN	.62	TRNN	.51	TDNN	1.32	TSNN	1.43
TLZ	.56	TRZ	.45	TDZ	1.26	TSZ	1.37
TLZE	.73	TRZE	.62	TDZE	1.43	TSZE	1.54
TLZA	.67	TRZA	.56	TDZA	1.37	TSZA	1.48
TLZN	.73	TRZN	.62	TDZN	1.43	TSZN	1.54
TLC	.56	TRC	.45	TDC	1.26	TSC	1.37
TLCE	.73	TRCE	.62	TDCE	1.43	TSCE	1.54
TLCA	.67	TRCA	.56	TDCA	1.37	TSCA	1.48
TLCN	.73	TRCN	.62	TDCN	1.43	TSCN	1.54
TLO	.67	TRO	.56	TDO	1.32	TSO	1.48
TLOE	.73	TROE	.62	TDOE	1.43	TSOE	1.54
TLOA	.67	TROA	.56	TDOA	1.37	TSOA	1.48
TLON	.73	TRON	.62	TDON	1.43	TSON	1.54

Boolean

		AND ANDCA ANDCM	
	SETZ SETO	ANDCB SETM SETCM	IOR ORCA
	SETA SETCA	XOR EQV	ORCM ORCB
Basic	.45	1.26	1.37
Immediate	.45	.45	.56
Memory, Both	.95	1.61	1.72

Shift-rotate

Left	1.14–5.10
Right	1.19–3.17
Left long	1.14–9.06
Right long	1.19–5.15

Fixed Point Arithmetic

ADD SUB	1.32	MUL	3.63–7.14	DIV	8.10–8.51
ADDI SUBI	.51	MULI	2.82–6.33	DIVI	7.29–7.70
ADDM SUBM	1.67	MULM	3.76–7.27	DIVM	8.23–8.64
ADDB SUBB	1.67	MULB	3.87–7.38	DIVB	8.34–8.75
		IMUL	3.47–6.38	IDIV	8.16–8.45
		IMULI	2.66–5.57	IDIVI	7.35–7.64
		IMULM	3.82–6.73	IDIVM	8.29–8.58
		IMULB	3.82–6.73	IDIVB	8.40–8.69

No Divide	
Immediate	.90–1.08
Other	1.71–1.89

Single Precision Floating Point Arithmetic

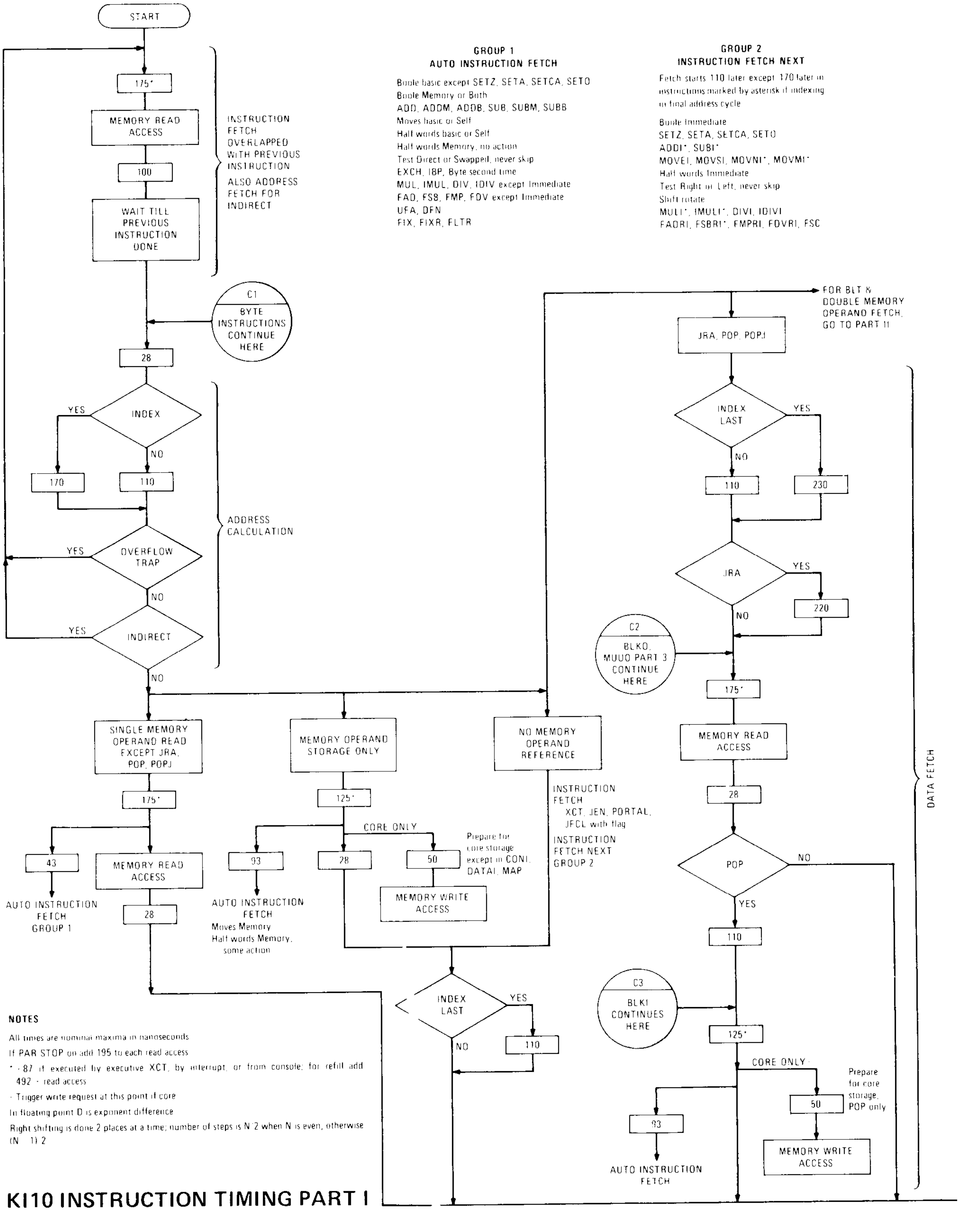
FAD	2.45–6.20	FSB	2.62–6.37	FMP	3.65–4.83
FADL	2.79–6.54	FSBL	2.96–6.71	FMPL	3.99–5.17
FADM FADB	2.80–6.55	FSBM FSBB	2.97–6.73	FMPM FMPB	4.00–5.18
FADR	2.45–6.26	FSBR	2.62–6.43	FMPR	3.65–4.89
FADRI	1.75–5.56	FSBRI	1.98–5.79	FMPRI	2.95–3.59
FADRM FADRB	2.80–6.61	FSBRM FSBRB	2.98–6.79	FMPRM FMPRB	4.00–5.24

DFN	1.50	FDV	7.12–7.75
UFA	1.91–3.86	FDVL	7.77–8.52
FSC	1.02, 1.19	FDVM FDVB	7.47–8.10
FIX FIXR	1.72–3.31	FDVR	7.49–7.95
FLTR	2.10–6.07	FDVRI	6.79–7.25
		FDVRM FDVRB	7.84–8.30

No Divide	
FDVL	2.11, 2.29
FDVRI	1.24, 1.30
Other	1.94, 2.00

Double Precision Floating Point Arithmetic

DFAD	2.59–7.00	DFMP	6.89–10.59	DMOVE	1.73
DFSB	2.59–7.19	DFDV	14.88–15.24	DMOVEM	1.85
		No Divide	2.62, 2.70	DMOVN	2.07, 2.13
				DMOVNM	2.31


START
175*
MEMORY READ ACCESS
100
WAIT TILL PREVIOUS INSTRUCTION DONE
INSTRUCTION FETCH OVERLAPPED WITH PREVIOUS INSTRUCTION
ALSO ADDRESS FETCH FOR INDIRECT
C1
BYTE INSTRUCTIONS CONTINUE HERE
28
INDEX
YES
NO
170
110
ADDRESS CALCULATION
OVERFLOW TRAP
YES
NO
INDIRECT
YES
NO
SINGLE MEMORY OPERAND READ EXCEPT JRA, POP, POPJ
175*
43
AUTO INSTRUCTION FETCH GROUP 1
MEMORY READ ACCESS
28
MEMORY OPERAND STORAGE ONLY
125*
CORE ONLY
93
AUTO INSTRUCTION FETCH
Moves Memory
Half words Memory, some action
28
50
Prepare for core storage except in CONI, DATAI, MAP
MEMORY WRITE ACCESS
NO MEMORY OPERAND REFERENCE
INSTRUCTION FETCH
XCT, JEN, PORTAL, JFCL with flag
INSTRUCTION FETCH NEXT GROUP 2
INDEX LAST
YES
NO
110
FOR BLT & DOUBLE MEMORY OPERAND FETCH, GO TO PART II
JRA, POP, POPJ
INDEX LAST
YES
NO
110
230
JRA
YES
NO
220
C2
BLKO, MUUO PART 3 CONTINUE HERE
175*
MEMORY READ ACCESS
28
POP
NO
YES
110
C3
BLKI CONTINUES HERE
125*
CORE ONLY
93
AUTO INSTRUCTION FETCH
50
Prepare for core storage, POP only
MEMORY WRITE ACCESS
DATA FETCH

GROUP 1
AUTO INSTRUCTION FETCH

Boole basic except SETZ, SETA, SETCA, SETO
Boole Memory or Both
ADD, ADDM, ADDB, SUB, SUBM, SUBB
Moves basic or Self
Half words basic or Self
Half words Memory, no action
Test Direct or Swapped, never skip
EXCH, IBP, Byte second time
MUL, IMUL, DIV, IDIV except Immediate
FAD, FSB, FMP, FDV except Immediate
UFA, DFN
FIX, FIXR, FLTR

GROUP 2
INSTRUCTION FETCH NEXT

Fetch starts 110 later except 170 later in instructions marked by asterisk if indexing in final address cycle

Boole Immediate
SETZ, SETA, SETCA, SETO
ADDI*, SUBI*
MOVEI, MOVSI, MOVNI*, MOVMI*
Half words Immediate
Test Right or Left, never skip
Shift rotate
MULI*, IMULI*, DIVI, IDIVI
FADRI, FSBRI*, FMPRI, FDVRI, FSC

NOTES

All times are nominal maxima in nanoseconds

If PAR STOP on add 195 to each read access

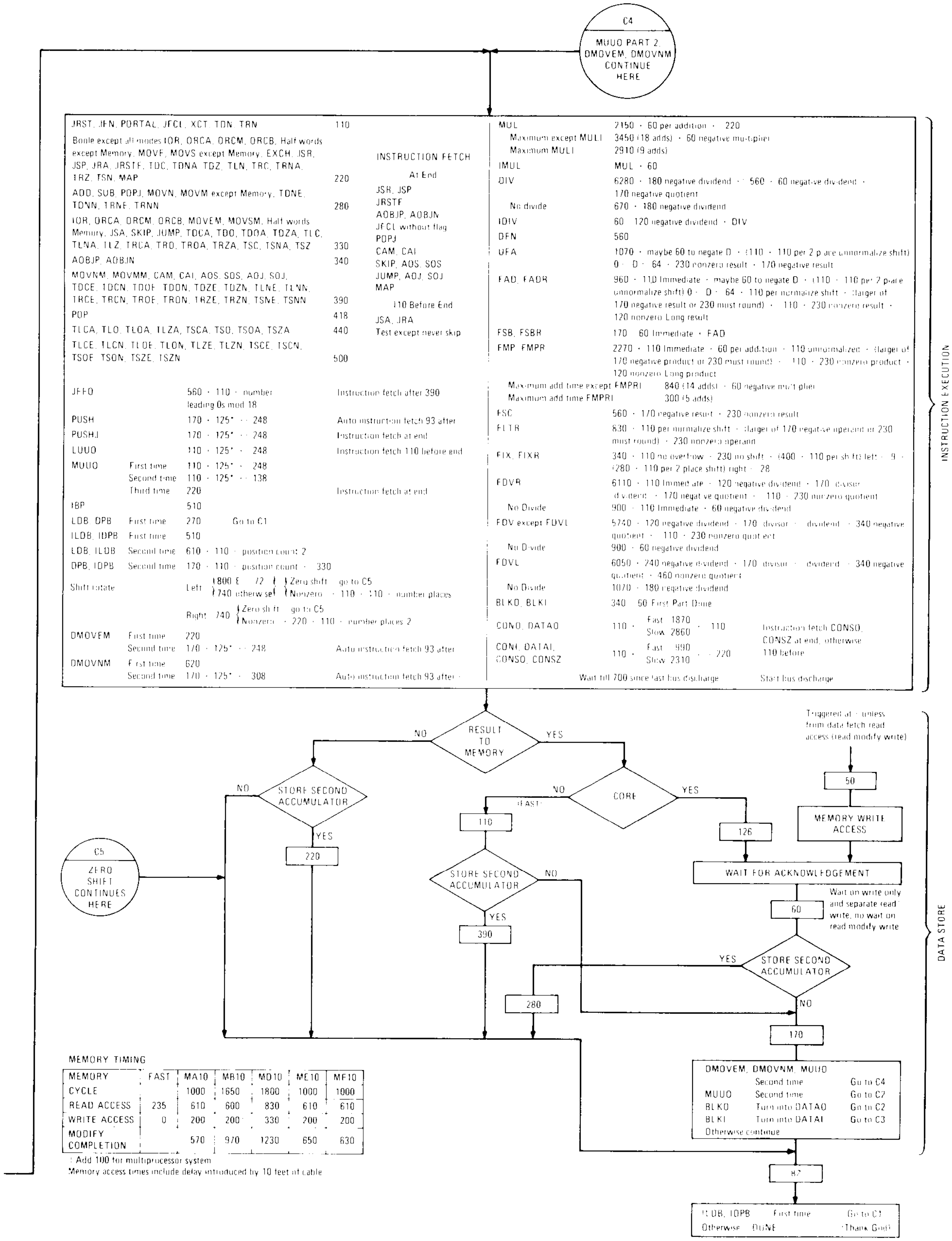
* - 87 if executed by executive XCT, by interrupt, or from console; for refill add 492 - read access

· Trigger write request at this point if core

In floating point D is exponent difference

Right shifting is done 2 places at a time; number of steps is N/2 when N is even, otherwise (N - 1) 2

KI10 INSTRUCTION TIMING PART I



JRST, JEN, PORTAL, JFCL, XCT, TDN, TRN	110	
Boole except all modes IOR, ORCA, ORCM, ORCB, Half words except Memory, MOVE, MOVS except Memory, EXCH, JSR, JSP, JRA, JRSTF, TDC, TDNA, TDZ, TLN, TRC, TRNA, TRZ, TSN, MAP	220	
ADD, SUB, POPJ, MOVN, MOVM except Memory, TDNE, TDNN, TRNE, TRNN	280	
IOR, ORCA, ORCM, ORCB, MOVEM, MOVSM, Half words Memory, JSA, SKIP, JUMP, TDCA, TDO, TDOA, TDZA, TLC, TLNA, TLZ, TRCA, TRO, TROA, TRZA, TSC, TSNA, TSZ	330	
AOBJP, AOBJN	340	
MOVNM, MOVMM, CAM, CAI, AOS, SOS, AOJ, SOJ, TDCE, TDCN, TDOE, TDON, TDZE, TDZN, TLNE, TLNN, TRCE, TRCN, TROE, TRON, TRZE, TRZN, TSNE, TSNN	390	
POP	418	
TLCA, TLO, TLOA, TLZA, TSCA, TSO, TSOA, TSZA	440	
TLCE, TLCN, TLOE, TLON, TLZE, TLZN, TSCE, TSCN, TSOE, TSON, TSZE, TSZN	500	

INSTRUCTION FETCH

At End
JSR, JSP
JRSTF
AOBJP, AOBJN
JFCL without flag
POPJ
CAM, CAI
SKIP, AOS, SOS
JUMP, AOJ, SOJ
MAP

110 Before End
JSA, JRA
Test except never skip

JFFO		560 + 110 + number leading 0s mod 18	Instruction fetch after 390
PUSH		170 + 125* + 248	Auto instruction fetch 93 after
PUSHJ		170 + 125* + 248	Instruction fetch at end
LUUO		110 + 125* + 248	Instruction fetch 110 before end
MUUO	First time	110 + 125* + 248	
	Second time	110 + 125* + 138	
	Third time	220	Instruction fetch at end
IBP		510	
LDB, DPB	First time	270 Go to C1	
ILDB, IDPB	First time	510	
LDB, ILDB	Second time	610 + 110 + position count/2	
DPB, IDPB	Second time	170 + 110 + position count + 330	
Shift rotate		Left {800 E > 72 / 740 otherwise} {Zero shift go to C5 / Nonzero + 110 + 110 + number places}	
		Right 740 {Zero shift go to C5 / Nonzero + 220 + 110 + number places/2}	
DMOVEM	First time	220	
	Second time	170 + 125* + 248	Auto instruction fetch 93 after
DMOVNM	First time	620	
	Second time	170 + 125* + 308	Auto instruction fetch 93 after

MUL	2150 + 60 per addition + 220
Maximum except MULI	3450 (18 adds) + 60 negative multiplier
Maximum MULI	2910 (9 adds)
IMUL	MUL − 60
DIV	6280 + 180 negative dividend + (560 + 60 negative dividend + 170 negative quotient
No divide	670 + 180 negative dividend
IDIV	60 + 120 negative dividend + DIV
DFN	560
UFA	1070 + maybe 60 to negate D + (110 + 110 per 2 place unnormalize shift) 0 < D < 64 + 230 nonzero result + 170 negative result
FAD, FADR	960 − 110 Immediate + maybe 60 to negate D + (110 + 110 per 2 place unnormalize shift) 0 < D < 64 + 110 per normalize shift + (larger of 170 negative result or 230 must round) + 110 + 230 nonzero result + 120 nonzero Long result
FSB, FSBR	170 − 60 Immediate + FAD
FMP, FMPR	2270 − 110 Immediate + 60 per addition + 110 unnormalized + (larger of 170 negative product or 230 must round) + 110 + 230 nonzero product + 120 nonzero Long product
Maximum add time except FMPRI	840 (14 adds) + 60 negative multiplier
Maximum add time FMPRI	300 (5 adds)
FSC	560 + 170 negative result + 230 nonzero result
FLTR	830 + 110 per normalize shift + (larger of 170 negative operand or 230 must round) + 230 nonzero operand
FIX, FIXR	340 + 110 no overflow + 230 no shift + (400 + 110 per shift) left < 9 + (280 + 110 per 2 place shift) right < 28
FDVR	6110 − 110 Immediate + 120 negative dividend + 170 (divisor < dividend) + 170 negative quotient + 110 + 230 nonzero quotient
No Divide	900 − 110 Immediate + 60 negative dividend
FDV except FDVL	5740 + 120 negative dividend + 170 (divisor < dividend) + 340 negative quotient + 110 + 230 nonzero quotient
No Divide	900 + 60 negative dividend
FDVL	6050 + 240 negative dividend + 170 (divisor < dividend) + 340 negative quotient + 460 nonzero quotient
No Divide	1070 + 180 negative dividend
BLKO, BLKI	340 − 60 First Part Done
CONO, DATAO	110 + Fast 1870 / Slow 2860 + 110
CONI, DATAI, CONSO, CONSZ	110 + Fast 990 / Slow 2310 + 220

Instruction fetch CONSO, CONSZ at end, otherwise 110 before

Wait till 700 since last bus discharge

Start bus discharge

DMOVEM, DMOVNM, MUUO

	Second time	Go to C4
MUUO	Second time	Go to C2
BLKO	Turn into DATAO	Go to C2
BLKI	Turn into DATAI	Go to C3

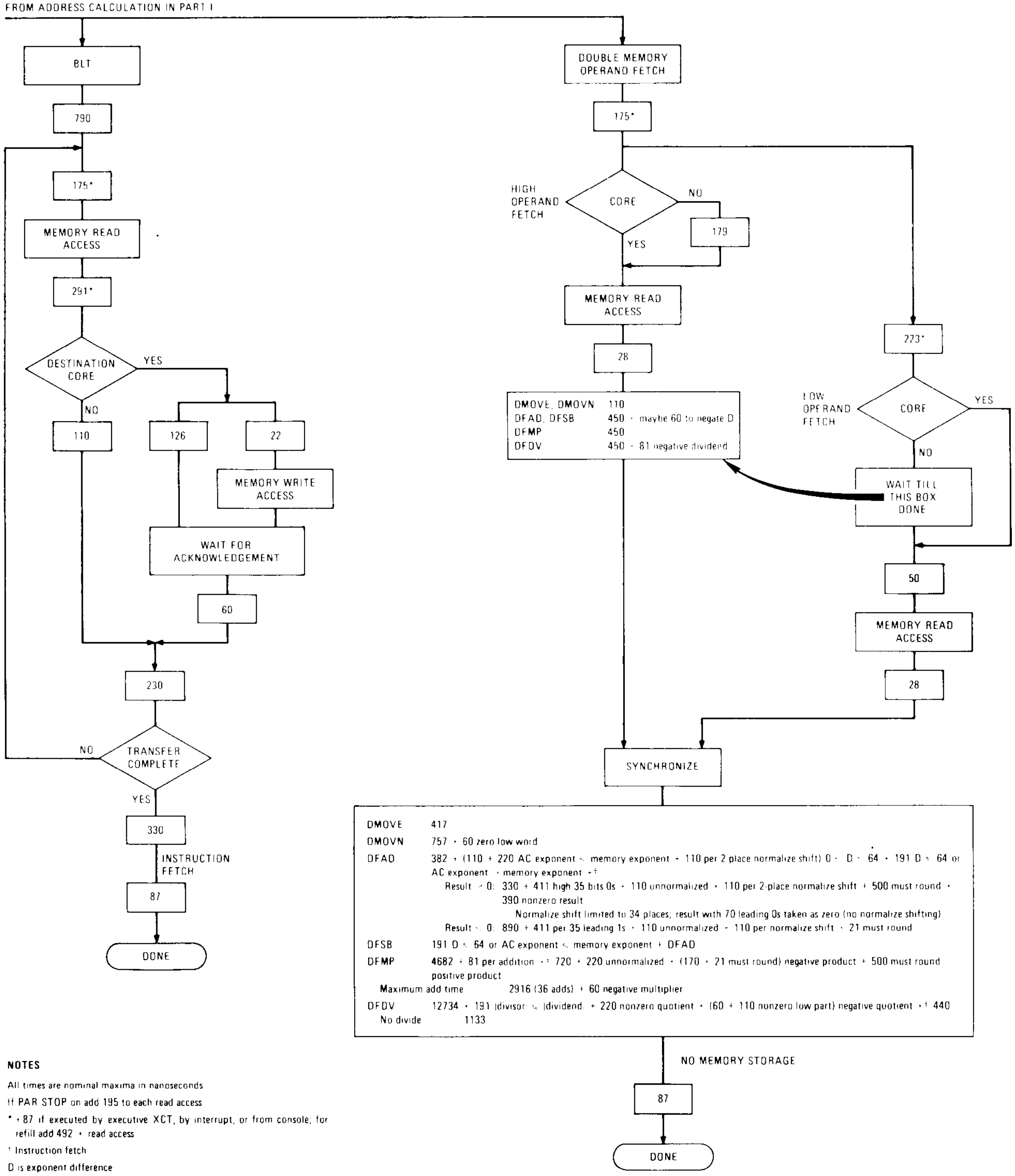
Otherwise continue

ILDB, IDPB	First time	Go to C1
Otherwise	DONE	(Thank God)

MEMORY TIMING

MEMORY	FAST	MA10	MB10	MD10	ME10	MF10
CYCLE		1000	1650	1800	1000	1000
READ ACCESS	235	610	600	830	610	610
WRITE ACCESS	0	200	200	330	200	200
MODIFY COMPLETION		570	970	1230	650	630

* Add 100 for multiprocessor system
Memory access times include delay introduced by 10 feet of cable



DMOVE, DMOVN 110
DFAD, DFSB 450 + maybe 60 to negate D
DFMP 450
DFDV 450 + 81 negative dividend

DMOVE 417
DMOVN 757 + 60 zero low word
DFAD 382 + (110 + 220 AC exponent < memory exponent + 110 per 2 place normalize shift) 0 < D < 64 + 191 D ≥ 64 or AC exponent < memory exponent +†
Result ≥ 0: 330 + 411 high 35 bits 0s + 110 unnormalized + 110 per 2-place normalize shift + 500 must round + 390 nonzero result
Normalize shift limited to 34 places; result with 70 leading 0s taken as zero (no normalize shifting)
Result < 0: 890 + 411 per 35 leading 1s + 110 unnormalized + 110 per normalize shift + 21 must round
DFSB 191 D ≥ 64 or AC exponent < memory exponent + DFAD
DFMP 4682 + 81 per addition +† 720 + 220 unnormalized + (170 + 21 must round) negative product + 500 must round positive product
Maximum add time 2916 (36 adds) + 60 negative multiplier
DFDV 12734 + 191 |divisor| ≤ |dividend| + 220 nonzero quotient + (60 + 110 nonzero low part) negative quotient +† 440
No divide 1133

NOTES

All times are nominal maxima in nanoseconds

‡ If PAR STOP on add 195 to each read access

* + 87 if executed by executive XCT, by interrupt, or from console; for refill add 492 + read access

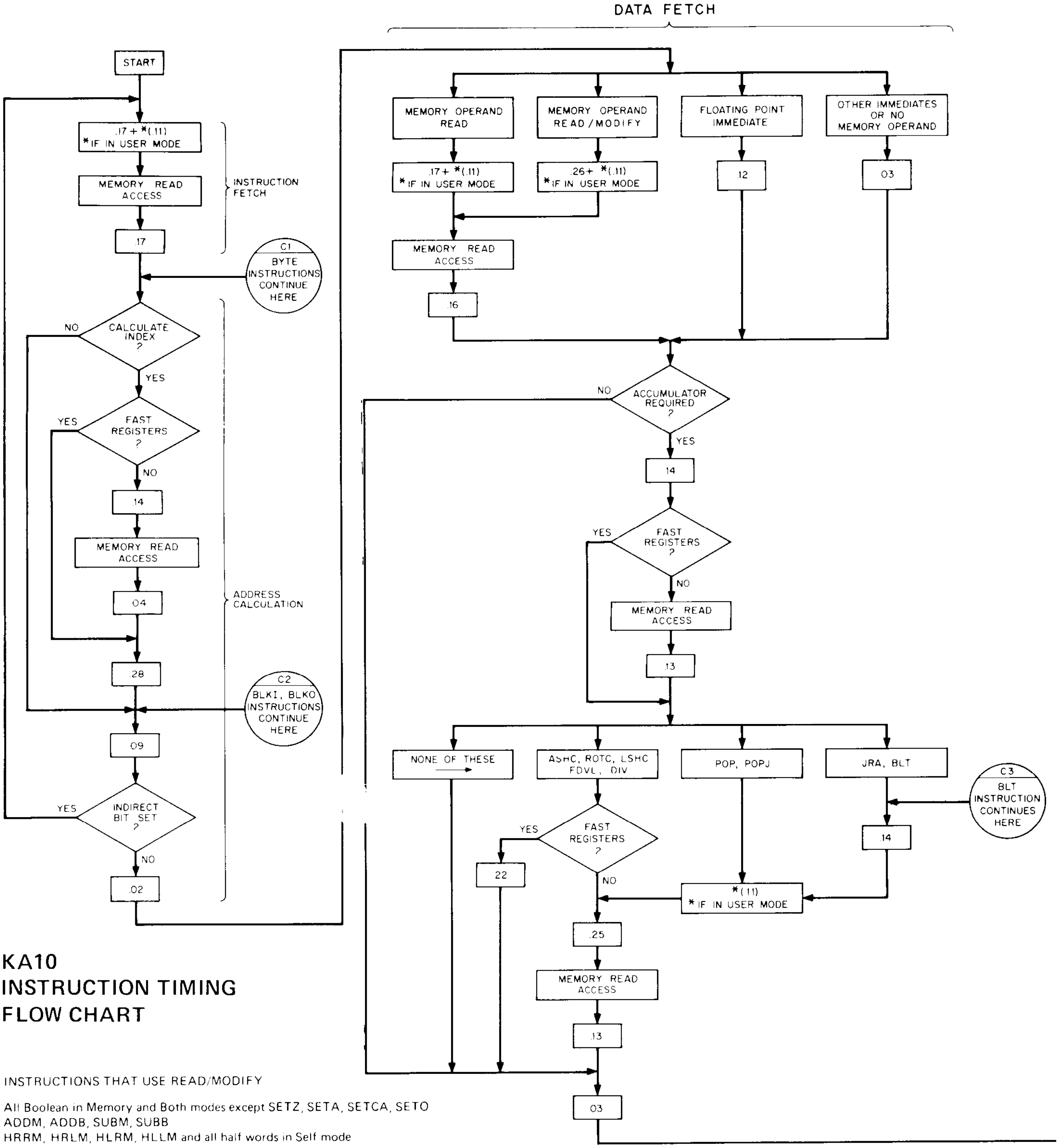
† Instruction fetch

D is exponent difference

KI10 INSTRUCTION TIMING PART II

KA10 Instruction Times

Instruction times for the KA10 can be calculated from the chart on the next two pages (intervals are in microseconds). Times derived from this chart are given with the instruction descriptions in the original *PDP–10 System Reference Manual*, which should be available to you if your system is based on a KA10. For more exact times than those given in that manual, add 0.06 to the listed time, plus an additional 0.03 for each memory operand read access, and another 0.03 if the instruction does not write a result in memory.



KA10 INSTRUCTION TIMING FLOW CHART

INSTRUCTIONS THAT USE READ/MODIFY

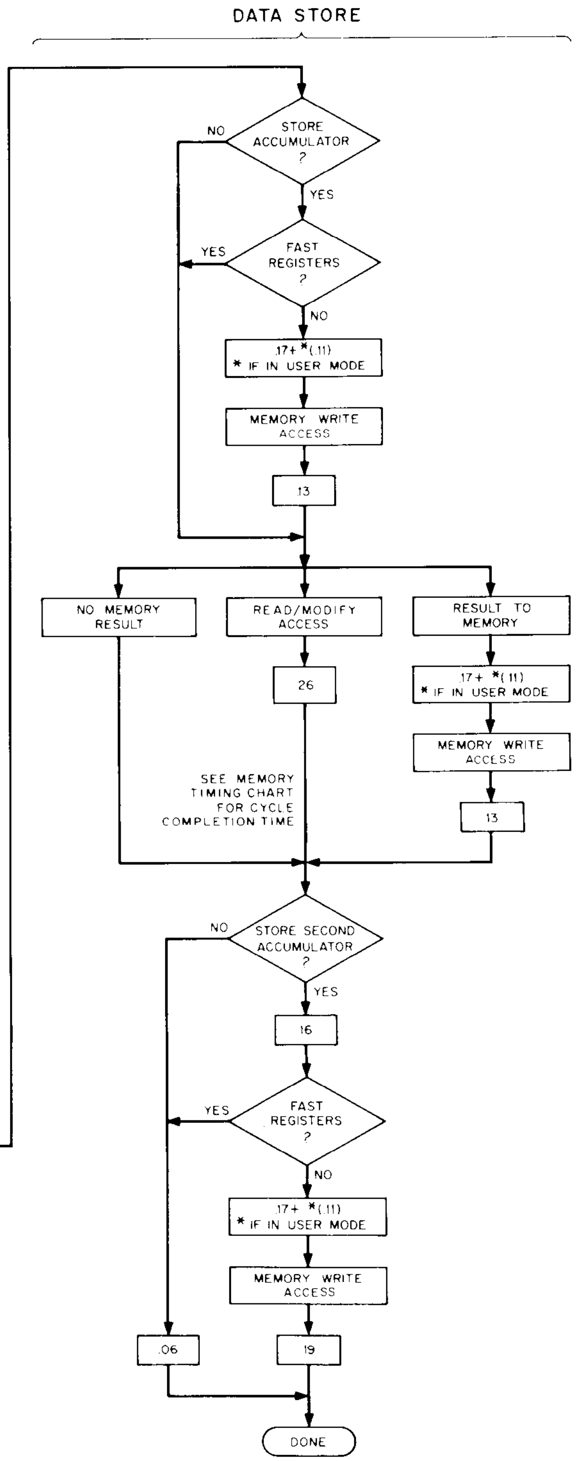
All Boolean in Memory and Both modes except SETZ, SETA, SETCA, SETO
ADDM, ADDB, SUBM, SUBB
HRRM, HRLM, HLRM, HLLM and all half words in Self mode
MOVES, MOVNS, MOVMS, MOVSS
ILDB, IDPB (first time only)
IBP, BLKI, BLKO, DFN, EXCH
AOS, SOS in all modes

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 Average except MULI		6.02 8.36	+ .13 per transition (18 transitions for 2.34)
IMUL Average except IMULI		6.34 7.51	+ .13 per transition (9 transitions for 1.17)
FMP Average except FMPRI		6.39 8.21	+ 13 per transition (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 Average		2.38 4.33	{ + .15 per shift to unnormalize + .25 per shift to normalize
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 CONO, CONI, DATAO, DATAI CONSO, CONSZ		.12 +2.69 +2.90	Then wait until 4.50 has passed since last here
BLKO, BLKI		.60	Then turn into DATAO, DATAI and go to C2

03

DATA STORE



MEMORY TIMING

MEMORY	MA10	MB10	MB10	FAST	MD10	ME10	MF10
PROCESSORS	SINGLE OR MULTI	SINGLE	MULTI	SINGLE (BUILT IN)	SINGLE OR MULTI	SINGLE OR MULTI	SINGLE OR MULTI
CYCLE	1 00	1 65	1.75	—	1.8	1 00	1.00
READ ACCESS	.61	.60	70	21	83	.61	.61
WRITE ACCESS	20	20	30	.21	.33	.20	.20
MODIFY COMPLETION	57	97	97	—	1.23	.65	.63

NOTES

MEMORY ACCESS TIMES INCLUDE DELAY INTRODUCED BY 10 FEET OF CABLE

ALL TIMES ARE NOMINAL MAXIMA IN MICROSECONDS

Appendix E
Processor Compatibility

The table beginning on the next page identifies the user programming differences among the various central processors. The reader is forewarned not to assume that he can program a new processor simply by glancing through this table. Simpler differences, principally some of those associated with individual user instructions, are explained adequately in the table entries. But in more complex cases, the table entries serve only to identify the areas of difference and refer the reader to the real substance in Chapter 2. In particular, all programmers, regardless of previous experience with other processors, should read Chapter 1.

The table is limited to *user* programming differences. Depending on the area, system differences vary from minor to extreme, and every system programmer must read the system operations chapter for his processor. Operating differences are so extensive, that upon approaching a new processor an operator must read the complete operating information given for it in Appendix F or the appropriate operator's guide.

Complete conditions that affect the program flags are given in §2.9 and are not listed here, although the table does indicate any differences from one processor to another in which flags are present, how they are read, and how a particular flag is affected by a given instruction. The entry "Same as *X*" means the situation for that processor is the same as indicated in the column headed *X*. Column B, Extended KL10 Section 0, applies also to the KS10 except for minor differences indicated in the individual entries or in notes at the end of the table.

	A	B	C	D	E
	Extended KL10 Nonzero Sections	Extended KL10 Section 0	Single-section KL10	KI10	KA10
Address word (§ *1.5*)	Global or local	Local only	Same as B	Same as B	Same as B
ADJBP	Yes	Yes	Yes	No	No
ADJSP	Yes	Yes	Yes	No	No
AOBJN, AOBJP	The two halves of AC are incremented independently	Same as A	Same as A	Same as A	AC is incremented by adding 1000001
BLKI, BLKO (*also see IO instructions*)	The two halves of the pointer are incremented independently	Same as A KS10: not available	Same as A	Same as A	The pointer is incremented by adding 1000001
BLT	At the end AC left and right contain addresses one greater than final source and destination locations, except funny addresses in attempted reverse BLT	Same as A KS10: same as A except no funny stuff in attempted reverse BLT	Same as A	At the end AC is indeterminate unless interrupt and pager both off, in which case it is unaffected	Same as D except pager condition not applicable
Byte pointer (§ *2.11*)	One word or two; as of microcode 271, one-word pointer local or global	One word local only	Same as B	Same as B	One word only; address overflow carries into index field
Carry flags	Set up by DMOVN and DMOVNM	Same as A	Same as A	Not affected by DMOVN or DMOVNM	Not applicable
CMPS*x*	Yes	Yes	Yes	No	No
Concealed mode	Yes	Yes	Yes	Yes	No
CVTBDO, CVTBDT	*M* and *N* can be affected by an aborted instruction	Same as A KS10: an aborted instruction cannot affect *M* and *N*	Same as A	Not available	Not available
CVTDBO, CVTDBT	Yes	Yes	Yes	No	No
DADD, DSUB, DMUL, DDIV	Yes	Yes	Yes	No	No
DFAD, DFSB	Test for zero inspects entire fraction	Same as A	Same as A	Test for zero inspects only high order 70 bits of fraction	Not available
DFDV	Normalizes and rounds	Same as A	Same as A	Neither normalizes nor rounds	Not available
DFMP	DFAD zero test; does ordinary normalization	Same as A	Same as A	DFAD zero test; at most one normalization shift (except skips entire high order word if zero)	Not available
DFN	*See software double precision floating point*				
DGFLTR	Yes as of microcode 271	Same as A KS10: no	No	No	No
DMOVE, DMOVEM	Yes	Yes	Yes	Yes	No

	A Extended KL10 Nonzero Sections	B Extended KL10 Section 0	C Single-section KL10	D KI10	E KA10
DMOVN, DMOVNM (§ *2.1*)	Overflow if negate -2^{70}	Same as A	Same as A	No overflow test	Not available
Double precision arithmetic	Fixed and floating	Same as A	Same as A	Floating only	No
EDIT	Yes	Yes	Yes	No	No
EXTEND	Yes	Yes	Yes	No	No
FAD, FSB	Zero substitution problem with exponent difference in range 54–72	Same as A	Same as A	Range 54–64	Same as D
FADL, FSBL	*See FAD and software double precision floating point*				
FDV	Negative quotient is ones complement, except twos complement when remainder is zero	Same as A	Same as A	Negative quotient is always twos complement	Same as D
FDVL	*See FDV and software double precision floating point*				
FIX, FIXR	Yes	Yes	Yes	Yes	No
Flag-PC doubleword	Yes	Yes	No	No	No
Flags (§ *2.9*)	Saved alone — never combined with PC in single word	Saved alone or in PC word KS10: bits 7 and 8 not used	Saved only in PC word	Same as C	Same as D but bits 7–10 not used
Floating Overflow (§ *2.9*)	With flags and JFCL	Same as A	Same as A	Same as A	Also in processor conditions
FLTR	Yes	Yes	Yes	Yes	No
FMPL	*See software double precision floating point*				
Front end	Yes	Yes	Yes	No	No
FSC	No problem	Same as A	Same as A	Extreme overflows not detected properly	Same as D
G format floating point	Yes as of microcode 271	Same as A KS10: no	No	No	No
HALT	Illegal; microcode enters halt loop	Same as A KS10: stores halt code and status block before entering halt loop	Same as A	Illegal; AR lights display address one greater than that of instruction that caused halt	Illegal unless User In-out set; MA lights display address
HLLI	= XHLLI	Yes	Yes	Yes	Yes
IBP, IDBP, ILDB	*See byte pointer*				
IDIV	No Divide on $-2^{35} \div -1$ as of microcode 271, earlier on $-2^{35} \div \pm 1$	Same as A KS10: $-2^{35} \div -1 = -2^{35}$ with no error indication	No Divide on $-2^{35} \div \pm 1$	Same as C	Same as C

	A Extended KL10 Nonzero Sections	B Extended KL10 Section 0	C Single-section KL10	D KI10	E KA10
Illegal basic instructions (*also see IO instructions*)	JRSTF, HALT, JEN; XJEN, XPCW, JRST 10, MAP unless User In-out set	[2]XBLT, HALT; XJEN, XPCW, JEN, JRST10, SFM, MAP unless User In-out set	HALT, XJRSTF, XJEN, XPCW, SFM; JRST10, JEN, MAP unless User In-out set	HALT, JRST 10, JEN	HALT, JRST 10, JEN unless User In-out set
Indirect word (§*1.6*)	Global or local	Local only	Same as B	Same as B	Same as B
Index	Global if bit 0 is 0 and bits 6–17 nonzero; otherwise local	Local only (bots 0–17 ignored)	Same as B	Same as B	Same as B
IO instructions	Legal if device code ⩾ 740 or User In-out set	Same as A KS10: same as E	Same as A	Same as A	Legal only if User In-out set
JEN	Illegal	Legal if User In-out set	Same as B	Same as A	Same as B
JRST (*see illegal instructions*)	Bits 9–12 decoded for 16 functions (*see* §*2.9*)	Same as A	Same as A	Four functions selected by 1s in bits 9–12 (*see* §*2.9*)	Same as D
JRSTF	Illegal	Legal	Same as B	Same as B	Same as B
JSP, JSR	Saves extended PC in bits 6–35	Saves flags and in-section PC in PC word	Same as B	Same as B	Same as B
JSYS	MUUO used by TOPS–20	Same as A	Same as A	Unassigned code	Same as D
Long mode	*See software double precision floating point*				
LUUO (§*2.16*)	Stores data in LUUO block and jumps to location given by block	Stores LUUO in virtual location 40 and executes location 41	Same as B	Same as B	Same as B
MAP	Legal only if User In-out set	Same as A	Same as A	Legal everywhere	Not available
Monitor	TOPS–20, but can be TOPS–10 as of microcode 271	Same as A KS10: same as C	TOPS–10 or TOPS–20	TOPS–10	Same as D
MOVSLJ, MOVSO, MOVSRJ, MOVST	Yes	Yes	Yes	No	No
MUL	AC supplies multiplier	Same as A	Same as A	Same as A	AC supplies multiplicand, which if -2^{35} is treated as though it were $+2^{35}$
MUUO (§ *2.16*)	104. Stores data in 424–427 of user process table; sets up previous context flags, clears others, jumps as specified by PC list	040–051, 055–077, 104. Same action as A KS10 action: TOPS–20 same as A; TOPS–10 same as C	040–051, 055–077; also 104 with TOPS–20. Stores data in 425–427 of user process table except 424–426 if TOPS–10; sets up flags and jumps as specified by PC-word list	040–051, 055–077. Stores data in 424–425 of user process table; sets up flags and jumps as specified by PC-word list	040–051, 055–100. Action same as LUUO except uses unrelocated 40–41 (140–141 if trap offset)

	A	B	C	D	E
	Extended KL10 Nonzero Sections	**Extended KL10 Section 0**	**Single-section KL10**	**KI10**	**KA10**
Overflow	Handled by trapping: arithmetic sets Overflow and Trap 1; stack sets Trap 2	Same as A	Same as A	Same as A	Handled by interrupt: arithmetic sets Overflow; stack sets Pushdown Overflow
Overflow flag	With flags and JFCL; sets Trap 1	Same as A	Same as A	Same as A	Also in processor conditions; causes interrupt
Overflow trapping	Yes	Yes	Yes	Yes	No
PC word	Replaced by extended PC without flags (JSP, JSR, PUSHJ) or flag-PC doubleword (MUUO)	With JSP, JSR, PUSHJ, JRSTF (MUUO uses doubleword)	With JSP, JSR, PUSHJ, MUUO, JRSTF	Same as C	Same as C – but flag bits 7–10 not used
POP, POPJ (*see stack pointer and §2.10*)	Action depends on whether pointer global or local	Local action only	Same as B	Same as B	Same as B
PORTAL (JRST 1,)	Clears Public when fetched from non-public area, so is valid entry	Same as A KS10: =JRST 0,	Same as A	Same as A	Enter user mode
Public	Yes	Yes KS10: no	Yes	Yes	No flag — user always public
PUSH, PUSHJ (*see stack pointer and §2.10*)	Action depends on whether pointer global or local	Local action only	Same as B	Same as B	Same as B
Pushdown Overflow	No (*see stack pointer*)	No	No	No	Yes
SETMI	= XMOVEI	Yes	Yes	Yes	Yes
SFM	Yes	Illegal unless User In-out set KS10: illegal	Illegal	No	No
Small User	No	No	No	Yes	No
Software double precision floating point	Only if specially implemented in microcode	Same as A	Same as A	Yes	Yes
Stack pointer (§ *2.10*)	Global if bit 0 is 0 and bits 6–17 nonzero; otherwise local, in which case the two halves are incremented or decremented independently, and overflow sets Trap 2	Local only	Same as B	Same as B	Local only, but incremented or decremented by adding or subtracting 1000001, and overflow sets Pushdown Overflow
String instructions	Yes	Yes	Yes	No	No
Trap flags	Yes	Yes	Yes	Yes	No
Trapping	Overflow, page failures, UUO	Same as A	Same as A	Same as A	Only UUO
UFA	*See FAD and software double precision floating point*				
UJEN	No	No	No	No	Yes

	A	B	C	D	E
	Extended KL10 Nonzero Sections	**Extended KL10 Section 0**	**Single-section KL10**	**KI10**	**KA10**
Unassigned codes	As of microcode 271: 052–054, 100, 101, 247; extended codes 032–777; JRST functions 3, 11, 13, 15–17; 001–037 in executive mode; 130, 131, 141, 151, 161, 171 unless software double precision floating point implemented in microcode. With earlier microcode also 102, 103, 106, 107 and extended codes 021–031 (G format floating point)	Same as A except 001–037 are LUUOs in any mode KS10: G format and software double precision floating point not implemented; in TOPS–10 104 unassigned	Same as B except: G floating and extended code 020 unassigned; in TOPS–10 104 unassigned	052–054, 100–107, 114–117, 123, 247	052–054, 101–127. Execute like MUUOs but use executive locations 60–61 (160–161 if trap offset). 247 and 257 are not regarded as unassigned and execute as no-ops unless implemented by special hardware
Unimplemented operations	All assigned codes implemented in hardware except G format fix instructions (when present) simulated by Monitor	Same as A KS10: all assigned codes implemented in hardware	All assigned codes implemented in hardware	Same as C	Turning on FP TRP switch causes floating point and byte codes 130–177 to act like unassigned codes
User In-out	Allows IO instructions with device codes under 740 and certain otherwise *illegal basic instructions* to be performed in user mode. With flags	Same as A KS10: applies to all IO instructions (no device codes)	Same as A	Allows IO instructions with device codes under 740 to be performed in user mode. With flags	Allows all IO instructions and otherwise *illegal basic instructions* to be performed in user mode. With flags and processor conditions
XBLT	Yes	Illegal	No	No	No
XCT	No problem	Same as A	Same as A	A private XCT cannot execute an instruction in a public area	Not applicable
XHLLI	Yes	= HLLI	Same as B	Same as B	Same as B
XJEN	Illegal unless User In-out set	Same as A KS10: illegal	Illegal	No	No
XJRSTF	Yes	Yes	Illegal	No	No
XMOVEI	Yes	= SETMI	Same as B	Same as B	Same as B
XPCW	Illegal unless User In-out set	Same as A KS10: illegal	Illegal	No	No

KS10 Notes

1. For console only (microprocessor).
2. HALT, XJEN, XPCW, SFM; JEN, JRST10, MAP and all system instructions unless User In-out set.
3. TOPS–10 MUUO; flag bits 7 and 8 not used.

	A	B	C	D	E
	Extended KL10 Nonzero Sections	Extended KL10 Section 0	Single-section KL10	KI10	KA10
PC word	Replaced by extended PC without flags (JSP, JSR, PUSHJ) or flag-PC doubleword (MUUO)	[3]With JSP, JSR, PUSHJ, JRSTF (MUUO uses doubleword)	With JSP, JSR, PUSHJ, MUUO, JRSTF	Same as C	Same as C – but flag bits 7–10 not used
POP, POPJ *(see stack pointer and section 2.10)*	Action depends on whether pointer global or local	Local action only	Same as B	Same as B	Same as B
PORTAL (JRST 1,)	Clears Public when fetched from non-public area, so is valid entry	Same as A KS10: = JRST 0,	Same as A	Same as A	Enter user mode
Public	Yes	Yes KS10: no	Yes	Yes	No flag – user always public
PUSH, PUSHJ *(see stack pointer and section 2.10)*	Action depends on whether pointer global or local	Local action only	Same as B	Same as B	Same as B
Pushdown Overflow	No *(see stack pointer)*	No	No	No	Yes
SETMI	= XMOVEI	Yes	Yes	Yes	Yes
SFM	Yes	Illegal unless User In-out set KS10: illegal	Illegal	No	No
Small User	No	No	No	Yes	No
Software double precision floating point	Only if specially implemented in microcode	Same as A	Same as A	Yes	Yes

	A	B	C	D	E
	Extended KL10 Nonzero Sections	**Extended KL10 Section 0**	**Single-section KL10**	**KI10**	**KA10**
Stack pointer *(section 2.10)*	Global if bit 0 is 0 and bits 6–17 nonzero; otherwise local, in which case the two halves are incremented or decremented independently, and overflow sets Trap 2	Local only	Same as B	Same as B	Local only, but incremented or decremented by adding or subtracting 1000001, and overflow sets Pushdown Overflow
String instructions	Yes	Yes	Yes	No	No
Trap flags	Yes	Yes	Yes	Yes	No
Trapping	Overflow, page failures, UUO	Same as A	Same as A	Same as A	Only UUO
UFA	*See FAD and software double precision floating point*				
UJEN	No	No	No	No	Yes
Unassigned codes	052–054, 100–103, 106, 107, 247; extended codes 021–777; JRST functions 3, 11, 13, 15–17; 001–037 in executive mode; 130, 131, 141, 151, 161, 171 unless software double precision floating point implemented in microcode	Same as A except 001–037 are LUUOs in any mode KS10: software double precision floating point not implemented; in TOPS-10 104 unassigned	Same as B except: extended code 020 unassigned; in TOPS-10 104 unassigned	052–054, 100–107, 114–117, 123, 247	052–054, 101–127. Execute like MUUOs but use executive locations 60-61 (160-161 if trap offset). 247 and 257 are not regarded as unassigned and execute as no-ops unless implemented by special hardware
Unimplemented operations	All assigned codes implemented in hardware	Same as A	Same as A	Same as A	Turning on FP TRP switch causes floating point and byte codes 130–177 to act like unassigned codes

	A	B	C	D	E
	Extended KL10 Nonzero Sections	**Extended KL10 Section 0**	**Single-section KL10**	**KI10**	**KA10**
User In-out	Allows IO instructions with device codes under 740 and certain otherwise *illegal basic instructions* to be performed in user mode. With flags	Same as A KS10: applies to all IO instructions (no device codes)	Same as A	Allows IO instructions with device codes under 740 to be performed in user mode. With flags	Allows all IO instructions and otherwise *illegal basic instructions* to be performed in user mode. With flags and processor conditions
XBLT	Yes	Illegal	No	No	No
XCT	No problem	Same as A	Same as A	A private XCT cannot execute an instruction in a public area	Not applicable
XHLLI	Yes	= HLLI	Same as B	Same as B	Same as B
XJEN	Illegal unless User In-out set	Same as A KS10: illegal	Illegal	No	No
XJRSTF	Yes	Yes	Illegal	No	No
XMOVEI	Yes	= SETMI	Same as B	Same as B	Same as B
XPCW	Illegal unless User In-out set	Same as A KS10: illegal	Illegal	No	No

KS10 Notes

1. For console only (microprocessor).
2. HALT, XJEN, XPCW, SFM; JEN, JRST10, MAP and all system instructions unless User In-out set.
3. TOPS-10 MUUO; flag bits 7 and 8 not used.

Appendix F
Processor Operation

The two sections of this appendix explain the switches and indicators used in normal operation and program debugging on the KI10 and KA10 processors. Mounted on the front of the right bay of each processor are console operator and maintenance panels. Indicator panels are at the tops of the processor bays. The real time clock is included in both processor discussions.

More recent processors have no operator panels and no controls beyond perhaps a simple power switch. The KL10 is operated from the master front end PDP–11; the KS10 is operated via a terminal that communicates with a microprocessor in the console module. For information on how to bootstrap (including loading the microcode) and operate these systems, refer to the *TOPS–10 Operator's Guide* or *TOPS–20 Operator's Guide*, whichever is appropriate.

Operating information for memories is given in Appendix G. Note that the KS10 memory is built in and is not operable in the usual sense; program information about it is included in Chapter 4. For information on the running of hardware diagnostics and the use of diagnostic functions or switches and indicators for hardware troubleshooting, refer to the appropriate maintenance manual.

Cleaning the Equipment

The exterior of all equipment in the system should be cleaned at least weekly. Vacuum all outside surfaces including cabinet tops and, where possible, underneath the cabinets. Pay special attention to air intake gratings.

CAUTION

When cleaning, be careful not to change the position of any switches as this could easily cause a software crash. Also be very careful not to jar any disk or drum equipment as serious head problems may result.

It is alright to use spray cleaner on exposed vertical surfaces, but do not use it around switches, near intake gratings, or near any other openings, because the "guck" can cause severe problems if it gets inside the equipment. The "alright" in this caution applies to the sheet metal. Whether the carcinogens that come out of aerosol cans are alright for your lungs is up to you to decide. It has never been shown that the presence or absence of fingermarks or other stains has any effect whatever on the operation of the system. And anyway, it is probably much healthier to get a little exercise using an ordinary cleaner.

Interior cleaning is necessary only for certain items of peripheral equipment. Specific instructions for such cleaning are given in the various operator's guides and maintenance manuals.

F.1 KI10 Operation

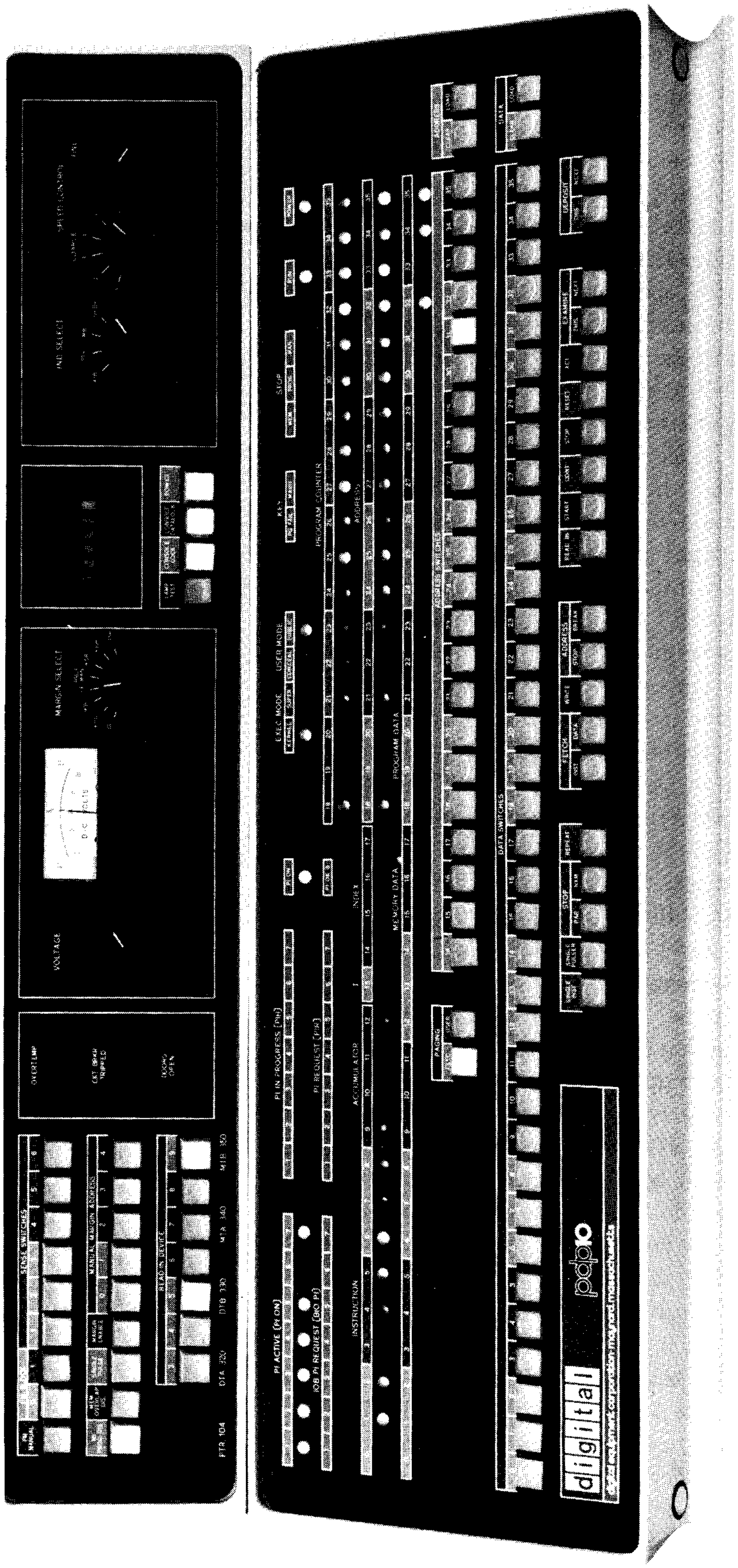
Most of the controls and indicators used for normal operation of the processor and for program debugging are located on the console operator panel, shown on the next page with the maintenance panel above it. In the upper half of the operator panel are four rows of indicators, and below them are three rows of two-position keys and switches. Physically both are pushbuttons, but the keys are momentary contact whereas the switches are alternate action. Relative to the internal logic, the switches are actually flip-flops that are controlled by the buttons but which in many cases can also be "operated" by the program. A switch is on or represents a 1 when it is illuminated. Buttons that actually trigger operating sequences in the processor are the operating keys, which are located in the right half of the bottom row. Operating switches are those that supply control levels for governing various processor operations; these include the buttons in the left half of the bottom row (except SINGLE PULSER), the paging switches at the left end of the third row, and the buttons at the left in the top two rows at the left end of the maintenance panel. The remaining buttons are sense switches, groups that constitute switch registers, and various other special keys and switches that supply information to the program or to specific hardware functions, or perform special functions of various sorts separate from the normal processor operating sequence.

A panel indicator is worthless if the bulb is burned out. Before attempting to use the information presented by the panels, press the LAMP TEST button below the counter on the maintenance panel; this turns on all of the lamps so any that are burned out can easily be detected.

The thirty-six numbered switches in the second row from the bottom on the operator panel and the twenty-two numbered switches in the row above them are 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. At the right end of each of these switch registers is a pair of keys that clear or load all the switches in the register together. The load button sets up the switches according to the contents of the corresponding bits of the memory indicators (MI) in the fourth row. At the left end of the maintenance panel are switches to select the device for readin mode and a set of sense switches, which can be interrogated by the program.

The center section of the maintenance panel contains a voltmeter and controls for margin checking, and the right section contains speed controls for slowing down the program. Between these is a counter that registers the total time processor power has been on (the counter reads hours if the line frequency is 50 Hz, but at 60 Hz it counts six for every five hours). Below the counter are four special buttons, two of which are locks that are used to prevent inadvertent manipulation of the keys and switches while the processor is running: the console data lock disables the data and sense switches; the console lock disables all other buttons except those that are mechanical, which group comprises the four under the counter and the readin device switches.

Power is supplied to the system by means of the switch at the right end in the group under the counter. This switch is lit while power is on, but the power light in the upper right corner of the operator panel is lit only when the system is actually in operation or is ready for operation; after power turn-on the light does not come on until power is stabilized in the correct range.



At the left of the margin check controls are three red lights that indicate an overtemperature condition somewhere in the processor logic, a tripped circuit breaker, or a cooling assembly door open. Whenever any of these lights goes on the Power Failure flag sets and power automatically shuts down.

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, in certain memory references, or following every clock pulse (the last allows extremely slow speed operation with the clock running slowly or each clock pulse triggered individually by the operator).

Of the large groups of lights on the operator panel, the right half of the second row displays the contents of PC, the third row displays the instruction being executed or just completed, and the fourth row is the memory indicators. The left third of the third row displays IR; in an IO instruction the left three instruction lights are on, the remaining instruction lights and the left accumulator light are the device code, and the remaining accumulator lights complete the instruction code. The right half of the row displays the virtual address on the address bus, and the I and index lights reflect the states of the corresponding bits of the memory buffer. Hence the right two thirds of the row changes with every memory reference, and the I and

Opposite: Console Operator and Maintenance Panels

index lights actually display the indirect bit and the index register address only following an instruction fetch or an indirect reference in an effective address calculation.

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 addresses used for memory reference are compared with the contents of the address switches in a manner determined by the paging switches and the User Address Compare Enable flag. Whenever the two addresses are equal and the comparison is enabled, 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. 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, which is shown in the PI GEN lights at the left end in the bottom row on the indicator panel at the top of the console bay, 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 an IN PROGRESS light goes on, the corresponding REQUEST goes off and cannot go on again until IN PROGRESS goes off when the interrupt is dismissed. PI ON indicates the priority interrupt system is active, so interrupts can be started (this corresponds to CONI PI, bit 28). PI OK 8 indicates that there is no interrupt being held and no channel waiting for an interrupt; this signal is used by the real time clock to discount interrupt time while timing user programs.

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

The four lights at the center of the top row indicate the processor mode. One and only one of these lights can be on and they represent the combined states of the User and Public flags. The rest of the top row contains the power light and the following control indicators.

RUN
The processor is in normal operation with one instruction following another (although the light remains on at a stop in a memory reference). When the light goes off, the processor stops.

STOP MAN
The operator has stopped the processor by pressing STOP or RESET.

STOP PROG
The processor has been stopped by a HALT instruction. At the completion of the instruction the address lights display the jump address (the location from which the next instruction will be taken if the operator presses the continue key), and the AR lights at the top of bay 2 display an address one greater than that of the location containing the instruction that caused the halt.

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

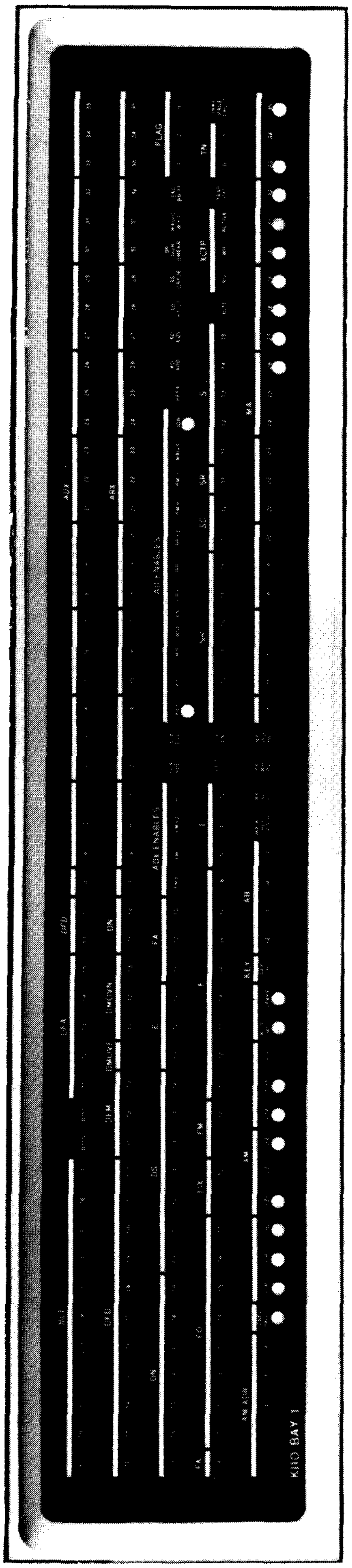
KEY MAINT
One of the following switches is on (this light is equivalent to CONI APR, bit 8): FM MANUAL, MEM OVERLAP DIS, SINGLE PULSE, MARGIN ENABLE, SINGLE INST, STOP PAR. Any one of these switches being on implies that the processor is being operated for maintenance purposes, and is not running at maximum speed.

KEY PG FAIL
A key function has caused a page failure. No page fail trap is executed in response to a key-induced failure; if the processor is running, it continues the program.

The remaining processor lights are on the indicator panels at the tops of the bays [*illustrated on next page*]. The large groups of lights on the panel at the top of bay 2 display the contents of the adder, the AR, BR and MQ registers, and the selected location in fast memory. The bottom row displays the AR flags — FXU is Floating (exponent) Underflow, DCK is No Divide (divide check). FXU HOLD is a nonprogram flag that plays a role in determining underflow conditions. At the end is the flipflop that inhibits the clock.

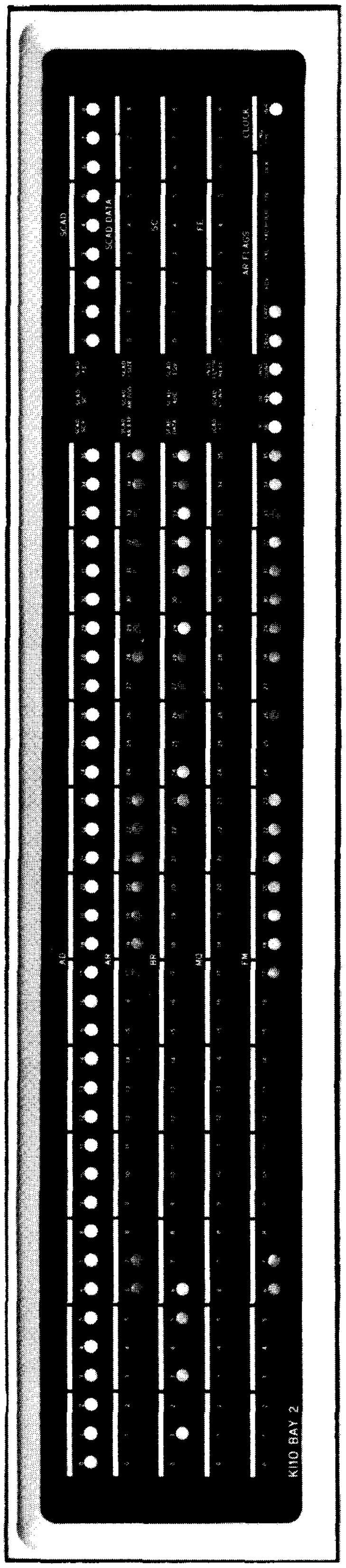
The right halves of the top two rows of the bay 1 panel display the contents of the AD and AR extensions. BYF6 in the top row is the First Part Done flag; the TN lights at the right end of the fourth row are the trap flags (TN 0 is Trap 2). The right half of the bottom row displays the physical address for each memory reference and the type of memory request. At the left are the lights for the associative memory. The AB 14–17 lights at the center are always either off or reflect the states of address switches 14–17.

The lights in the top row of the panel on the console bay (bay 3) display either the contents of the in-out bus, the paper tape reader buffer, MB, or the information supplied by the last DATAO PAG, as selected by the 4-position switch in the right section of the maintenance panel. The large groups of lights in the second row display the user and executive base registers; at the left end are the Small User and User Address Compare Enable flags, and a



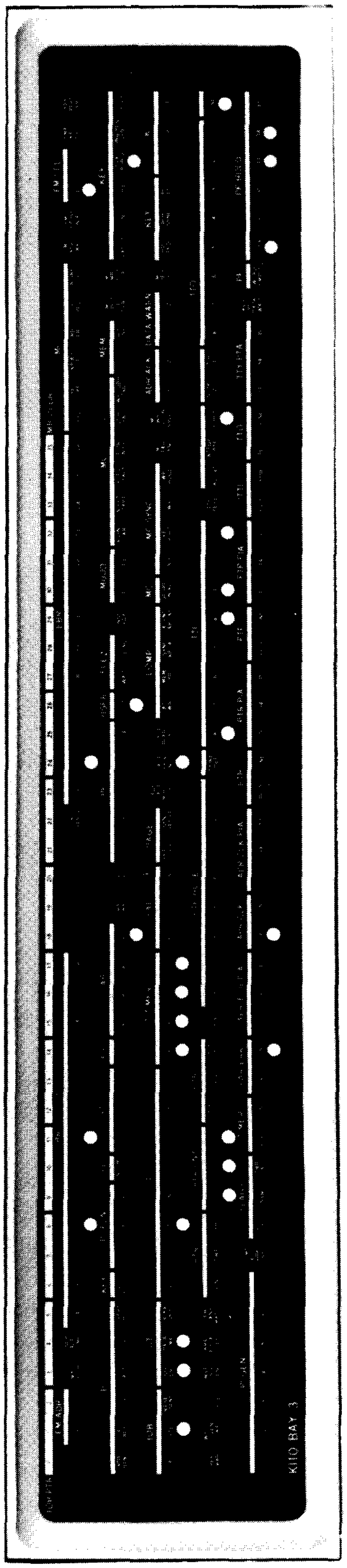
6369-1

Indicator Panel, KI10 Arithmetic Processor, Bay 1



6369-2

Indicator Panel, KI10 Arithmetic Processor, Bay 2



6369-3

Indicator Panel, KI10 Arithmetic Processor, Console Bay pair of lights that indicate which fast memory block is currently selected for the user program. The bottom two rows include the indicators for reader, punch and console terminal (see DECsystem-10 manual, Appendix H) and the processor flags. Note that the TRAP ENABLE light at the center of the second row is the Page Enable flag, which also enables overflow traps (DATAI PAG, bit 22). PAGE LAST MUUO PUB at the very center of the panel is the Disable Bypass flag. The User IOT flag is in the middle of the third row, and COMP ADR BRK INH near the left end of the bottom row is Address Failure Inhibit.

The remaining lights on the panels are for maintenance. If the operator must use them, he should consult the maintenance manual and the flow charts.

Operating Keys

The operating keys can be used whether RUN is on or off. If the processor is running when a key is pressed, it simply pauses at an appropriate point in the program to perform a key cycle to execute the function. These key functions are effectively of three types. The first three keys on the left are for the initiating functions, read in, start, and continue: these functions place the processor in operation under conditions determined primarily by the function itself. The next two keys are for the terminating functions, stop and reset: if the processor is running, these functions stop it. The last five keys are for the independent functions, execute, examine, examine next, deposit, and deposit next. These functions have no inherent effect on processor operation: if the processor is not running it simply performs a key cycle and stops; if it is running, it pauses to perform a key cycle and continues the program. (However the data deposited or the instruction executed may have an effect.) Moreover the independent functions are affected by the setting of the paging switches, which determine the address space in which the function is performed.

The logic responds to the keys in two stages. When a key is pressed or several are pressed simultaneously, the logic latches them. From among the buttons latched, the processor then accepts the request for the function that has priority; the priority order is the same as the order of the keys from left to right on the panel except that reset has first priority. As soon as a function request is accepted, the corresponding button lights up and remains lit until the function is completed. If the processor is not already in operation, it performs the accepted function immediately; otherwise it saves the function until it can be performed. While any button is lit, however, no function request can be accepted; in other words, although the processor will interrupt the program to perform a key function, it will not interrupt one key function for another. It will however do one key latch while a key is lit and accept the highest priority latched function once the current function is done. Provision is also made in the logic so that the RESET key can be used to stop the processor no matter what.

CAUTION

READ IN does not clear the associative memory, whose contents are unpredictable at power turnon.

READ IN
Clear all IO devices and all processor flags. Turn on RUN and EXEC MODE KERNEL (trapping and paging will both be disabled as TRAP ENABLE at the top of the console bay will be off). Execute DATAI *D*,0 where *D* is the

device code specified by the readin device switches at the left end of the maintenance panel. Then execute a series of BLKI *D*,0 instructions until the left half of location 0 reaches zero. After storing the last word in the block, fetch that word as an instruction from the location in which it was stored as specified by PC. Since RUN has been set the processor begins normal operation at the location containing the last word. [*For information on the data format refer to* §5.1].

Codes of readin devices are: PTR 104, DTC 320, DTC2 330, TMC 340, TMC2 350.

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

Note that the key function lasts throughout the processing of the entire block. This means that read in cannot be interrupted for another key function. Hence if it must be stopped (*eg* because of a crumpled tape), press RESET.

START

Turn on RUN and EXEC MODE KERNEL, and begin normal operation by fetching the instruction at the location specified by address switches 18–35. The memory subroutine for the instruction fetch loads the address into PC for the program to continue. This function does not disturb the flags or the IO equipment.

CONT (Continue)

If STOP MEM is on begin normal operation at the point at which the processor is stopped in a memory subroutine. Otherwise turn on RUN and begin normal operation by fetching an instruction from the location specified by PC.

The memory restart is not a key function in the sense defined above. In other words, use of CONT to continue at a memory stop is not subject to the restrictions given above for use of the operating keys.

STOP

Turn off RUN so the processor stops with STOP MAN on. At the stop PC points to the location of the instruction that will be fetched if CONT is pressed (this is the instruction that would have been done next had the processor not stopped).

The processor may stop in the middle of a two-part instruction, but pressing CONT restarts the instruction without repeating any first-part actions that would adversely affect the result.

RESET

Clear all IO devices, disable auto restart, high speed operation and margin programming, clear the processor flags (lighting EXEC MODE KERNEL), turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA), turn off RUN and stop the processor. Do *not* clear the associative memory.

If this function is not performed within 10 ms (*eg* because READ IN is lit), the key triggers a panic reset that produces all of the standard reset actions and also clears all but the mechanical console keys and switches.

If STOP ever fails to stop the processor, pressing this key will, but not without destroying information. To save the processor state, stop by pressing SINGLE INST and SINGLE PULSE simultaneously.

XCT

Execute the contents of the data switches as an instruction without incrementing PC, even if a skip condition is satisfied in the instruction. If PAGING USER is on and PAGING EXEC is off, execute the instruction in user virtual address space; otherwise use executive address space. If the instruction is an XCT or LUUO, the instruction called by it is also executed.

Note that an instruction executed from the console can alter the processor state like an instruction in the program: it can halt the processor, can change PC by jumping, alter the flags, or even cause a non-existent memory stop (but not a page fail trap, even if it turns on the KEY PG FAIL light).

NOTE

The remaining key functions all reference memory. They can therefore light KEY PG FAIL and set such flags as Nonexistent Memory and Parity Error, and they all turn on the triangular light beside MEMORY DATA, turning off the light beside PROGRAM DATA. *Performing one of these functions with the ADDRESS STOP switch on stops the processor in the memory subroutine (with STOP MEM on).*

These functions use an address supplied by the address switches, and the way that address is interpreted is determined by the paging switches. If both paging switches are off, the function uses a 22-bit absolute physical address supplied by address switches 14–35, and fast memory references are made to the block selected by the FM block switches at the left end of the maintenance panel. If either paging switch is set, the function uses a virtual address supplied by address switches 18–35 and the FM block switches have no effect (in other words the function has access to one of the virtual address spaces defined for a normal program). If PAGING EXEC is on, the function has access to executive address space; if PAGING EXEC is off and PAGING USER is on, the function has access to user address space.

EXAMINE THIS
Display the contents of the location specified by the paging and address switches in the memory indicators.

EXAMINE NEXT
Add 1 to the address displayed in the address switches, and display the contents of the location then specified by the paging and address switches in the memory indicators.

DEPOSIT
Deposit the contents of the data switches in the location specified by the paging and address switches, and display the word deposited in the memory indicators.

DEPOSIT NEXT
Add 1 to the address displayed in the address switches, deposit the contents of the data switches in the location then specified by the paging and address switches, and display the word deposited in the memory indicators.

Operating Switches

Besides defining the address space for the independent key functions, the paging switches also perform this service for address comparison and for the group of five switches just at the left of the operating keys. Whenever the processor references memory or an accumulator, it may compare the virtual address used with that specified by address switches 18–35 and may take some action if the two are identical. There are a number of conditions that affect the comparison. First, comparison can be made only for memory references and accumulator write references — there is never a comparison for an index register reference or an accumulator read reference. Given the proper type of reference, the comparison must be enabled by the paging switches and the User Address Compare Enable flag, as described below. In a reference of the correct type with the comparison enabled, if the virtual address on the address bus or the fast memory address is identical to the address in switches 18–35, the processor displays the contents of the addressed location or accumulator in the memory indicators (unless the light beside PROGRAM DATA is on).

Except in an AC reference, the same situation that causes the word display can also be made to stop the processor or produce an address failure, depending upon the purpose of the reference as selected by the three address condition switches. The logic that implements the address stop conditions differs from that for the address break conditions in the data fetch case (the break conditions are a subset of the stop conditions). However the differences in the statement of the conditions appear quite large. This is because the conditions are stated in terms of their consequences. And the consequences differ considerably because an address failure occurs in the page check that is done when a memory reference is requested, whereas an address stop occurs after a memory reference is actually made.

The address conditions for a failure are explained in detail in §2.15. Whenever there is a page check for a memory reference that satisfies both the comparison conditions and any selected address condition, ADDRESS BREAK being on causes an address failure except in an instruction performed while COMP ADR BRK INH is on.

Whenever the processor actually makes a memory reference that satisfies both the comparison conditions and any selected address condition, ADDRESS STOP being on halts the processor with STOP MEM on and PC pointing to the instruction that is being performed (running with ADDRESS STOP on slows down the processor). The stop conditions selected by the address condition switches are as follows:

When the ADDRESS BREAK and ADDRESS STOP switches are both on, the former has precedence because the page failure cancels the requested access.

FETCH INST selects access for retrieval of an ordinary instruction, including an instruction executed by an XCT or an LUUO (address 41), and a page refill for same.

FETCH DATA selects access for retrieval of an address word in an effective address calculation, any retrieval of an operand other than in an XCT (read-only and in the read part of a read-modify-write), retrieval of a dispatch interrupt instruction, and a page refill for any of these and for any of the conditions selected by the WRITE switch (*ie* any reference except an instruction fetch). This switch can also cause a stop inadvertently on the retrieval of a trap instruction, a PC word in an MUUO, or a standard interrupt instruction.

WRITE selects access for writing, both write-only and read-modify-write, including writing by an LUUO (address 40), a page refill for any of these, and also for retrieval of the operand in a read-modify-write — in other words the processor stops separately on the read and write parts of a read-modify-write. This switch also causes a stop on the first write in an

MUUO if the address switches contain the effective address of the MUUO (even though that address is not used for the access), and can cause a failure inadvertently on the second write.

ADDRESS STOP also stops any examine or deposit function in the memory subroutine.

The way the paging switches enable the comparison is as follows. If PAGING EXEC is on and PAGING USER is off, the comparison is enabled for executive address space. If PAGING EXEC is off and PAGING USER is on, the comparison is enabled for user address space provided the program has turned on USER ADR COMP (User Address Compare Enable flag) in the upper left corner of the bay 3 indicator panel. If both paging switches are on, the comparison is enabled for executive address space, provided USER ADR COMP is on (in other words with both switches on, PAGING USER applies the flag condition to PAGING EXEC).

Besides controlling USER ADR COMP with a DATAO PAG, a debugging program can directly manipulate the paging, address, address condition, and address break switches by means of a DATAO PTR,. But for the program to control address stopping (other than by USER ADR COMP), the operator must turn the switch on, and the program can then inhibit its effect by turning off all three address condition switches. Should it be preferred that the address condition be controlled solely by the operator, the program can still disable the stop by setting the address switches to a number that is unlikely to appear on the address bus, such as zero, or better still an address greater than any used in the program. It might seem that an address all 1s is a good candidate for this purpose, but it is in fact a very poor choice and results in advertent stops at traps, MUUOs and the like. The reason for this is that various types of special access do not use the address bus; and when the bus is not used, it is generally left free to follow the adder, which in turn puts out all 1s when neither of its input mixers is enabled.

Displaying the contents of a selected location and catching a particular type of reference to a selected location, as described above, are traditional debugging techniques. The paging switches allow these techniques to be used more flexibly in a large system that handles many users. The configuration PAGING EXEC on and PAGING USER off would be used for debugging the Monitor itself or some other executive program, which quite likely would be the only program running. PAGING EXEC off and PAGING USER on limits the procedures to user address space; and control over the comparison by the executive through a flag allows debugging an individual user program without interfering with either the executive or other users. Similarly both switches on allows investigation of that part of the executive associated with a given user, interfering with neither the rest of the executive nor any user. One who uses these switches often works in conjunction with a debugging or diagnostic program, and in the flag-limited cases one would be more apt to use the address break than the address stop, as the latter terminates all operations.

Conditions associated with the comparison are displayed by the COMP lights in the middle of the bay 3 indicator panel. From left to right these indicate an accumulator write reference, a memory read reference, equal addresses in a synchronous reference (an operand reference, but limited to the first in a double operand), and equal addresses in an asynchronous reference (an instruction fetch or the second in a double operand).

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

SINGLE 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, 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.

Note that read in cannot be done in single instruction mode, as the function extends over many instructions and there is thus no way to continue.

APR CLK FLAG (Clock flag) on the bay 3 indicator panel is held off to prevent clock interrupts while SINGLE INST is on. Otherwise interrupts would occur at a faster rate than the instructions.

Caution

It is not generally worthwhile to attempt to use the interrupt system in single instruction mode except with the slowest start-stop devices, such as reader, punch and teletypewriter. In any event an interrupt hangs up the processor, and the operator must dispose of it manually before single instruction operation can continue.

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

SINGLE PULSE
Inhibit the clock so that a single clock pulse is generated each time SINGLE PULSER is pressed. If the processor is not already in operation, an operating key must be pressed before SINGLE PULSER can be used. If the processor is running, it converts to single pulse operation at the beginning of the instruction cycle; hence the clock will not stop if the processor does not reach the instruction cycle, say because it is hung up in a multiply or divide sequence. To force the processor into single pulse operation regardless of its position in the operating sequence, turn on both SINGLE INST and SINGLE PULSE — this stops the processor dead in its tracks.

This type of stop destroys no information, the way pressing RESET would.

STOP PAR
Stop with STOP MEM 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: PAR ERR FLAG (Parity Error flag) is on in the bottom row on the bay 3 indicator panel; and among the PAR lights in the third row from the bottom, ERR is on, IGN (ignore parity) is off, and BIT displays the parity bit for the word read. MA points to the location in which the error occurred.

If IGN is on (it displays a signal from the memory), parity errors are not detected and no stop can occur. Running with STOP PAR on slows down the processor.

STOP NXM
Stop with STOP MEM on if a memory reference is attempted but the memory does not respond within 100 μs. This type of stop is indicated by FLAGS NXM (Nonexistent Memory flag) being on in the bottom row on the bay 3 indicator panel.

REPEAT
If SINGLE PULSE is on and the processor is placed in operation, slow down the clock so that the processor runs at a clock rate determined by the speed controls at the right end of the maintenance panel. If the processor is not already running, it can be placed in single-pulse repeat operation by pressing an operating key and then pressing SINGLE PULSER. If the processor is running and the switches are turned on in the order REPEAT/SINGLE PULSE, then it goes into single pulse operation automatically at the beginning of the instruction cycle. If the processor is running with REPEAT off, it stops at the beginning of the instruction cycle when SINGLE PULSE is turned on; to restart it, turn on REPEAT and then press SINGLE PULSER twice. The lamp in the SINGLE PULSER button goes off at each clock pulse and turns back on each time the clock is retriggered; hence the button glows

with an intensity that is relative to the clock duty cycle (*eg* for a given speed, the light will be dimmer for a program with many memory references). When either REPEAT or SINGLE PULSE is turned off, operation terminates after one more clock.

If SINGLE PULSE is off and any operating key is pressed, then every time the repeat delay can be retriggered, wait a period of time determined by the setting of the speed control and repeat the given key function. The point at which the processor can restart the repeat delay depends upon the type of key function being repeated as follows.

So long as REPEAT remains on, the selected key remains lit and its function continues in effect. In other words the operating keys are disabled.

For an initiating function the delay starts when the processor stops with RUN off. This is either when the program gives a HALT instruction (STOP PROG) or following the first instruction if SINGLE INST is on.

For an independent function the delay starts every time the function is done whether RUN is on or off.

A terminating function stops the processor and the delay starts every time the function is repeated. Reset is generally used only to provide a chain of reset pulses on the IO bus, and stop is used to troubleshoot the clock.

In any case continue to repeat the function until REPEAT is turned off.

The function is often repeated once more after the switch is turned off, but this is noticeable only with very long repeat delays.

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	200 ns to 2 μs
2	2 μs to 20 μs
3	20 μs to 500 μs
4	500 μs to 6 ms
5	6 ms to 160 ms
6	160 ms to 4 seconds

The remaining switches are located at the left end of the maintenance panel.

FM MANUAL

All fast memory references for any purpose (index register, accumulator, memory) and under any conditions are made to the fast memory block selected by the FM BLOCK switches. When FM MANUAL is off, the block switches control fast memory references only in examine and deposit type key functions with both paging switches off (*ie* with the function using physical addressing). Turning on FM MANUAL overrides all other conditions so that all fast memory references are controlled by the block switches.

MI PROG DIS

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

MEM OVERLAP DIS
Prevent memory control from overlapping cycles on the memory bus.

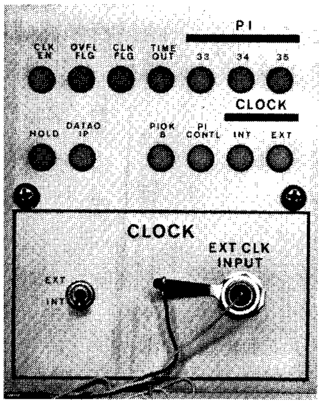
This has no effect on pipelining within memory control, such as overlapping the page checking of consecutive memory subroutines.

MARGIN ENABLE
Enable maintenance operation, including writing with even parity in memory and checking speed or voltage margins. Maintenance actions attempted by the program are indicated by the last four lights on the left end of the second row from the bottom on the bay 3 indicator panel. With maintenance operation enabled, writing with even parity and checking speed margins are otherwise entirely under program control. Voltage margins may be checked by the program or the operator.

For information on maintenance operation, including use of the MARGIN SELECT and MANUAL MARGIN ADDRESS switches, refer to Chapter 10 of the maintenance manual.

Real Time Clock DK10

The real time clock for the KI10 is usually installed under the console operator panel in bay 3 and has a small control panel mounted directly on the logic behind the cabinet door. 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

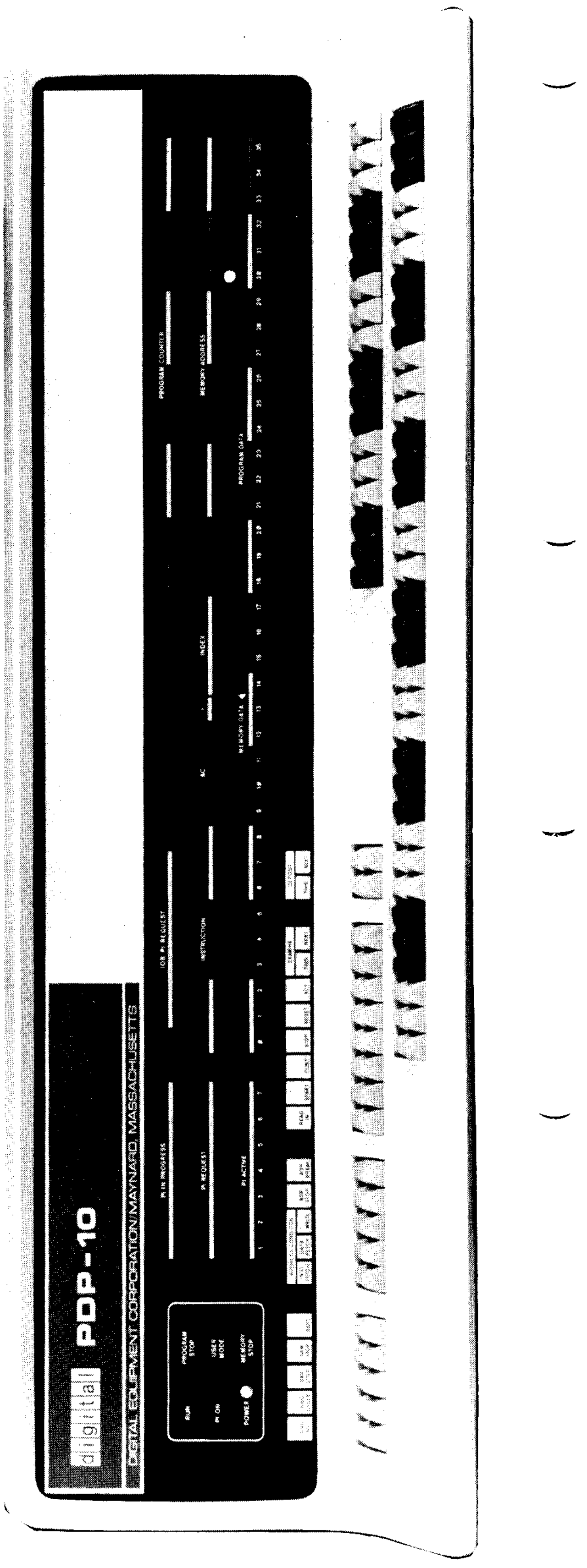
The leftmost light in the upper row at the top of the panel indicates when the clock is on (*ie* when the counter is enabled). The next two lights are the Count Overflow and Count Done flags. TIME OUT indicates when the numbers in the interval register and the clock counter are identical — this light 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. PI OK 8 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.

F.2 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.

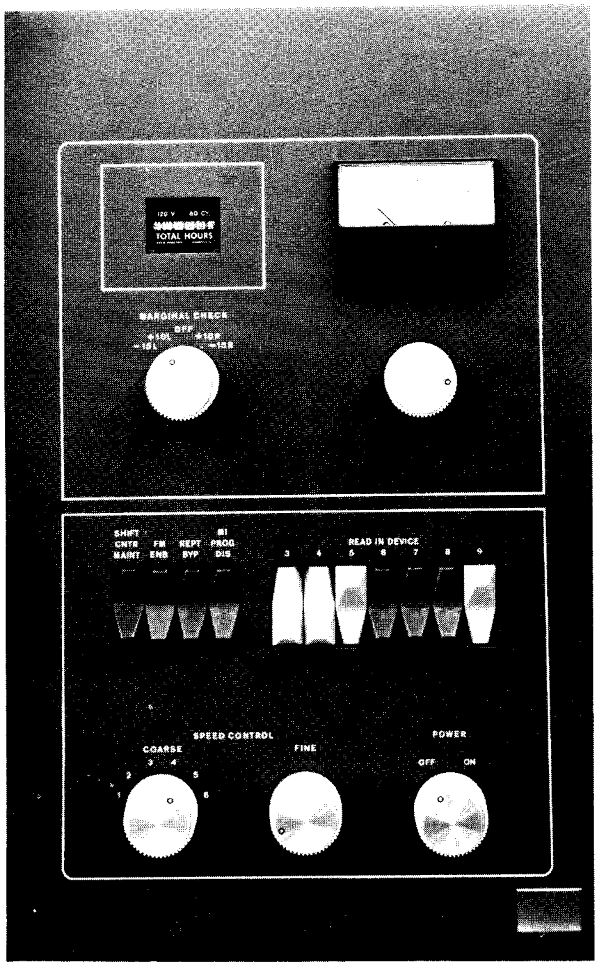
Also of interest to the operator is the small panel shown overleaf, which is located above the main panel at the left of the tape reader. The upper section of this panel contains a total hours meter and the margin-check controls. The lower section contains the power switch, speed controls for slowing 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.



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

Note: If a REQUEST light stays on indefinitely with the associated IN 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.

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 relocation 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 levels. The PI ACTIVE lights indicate which levels are on. The IOB PI REQUEST lights indicate which levels are receiving request signals over the in-out bus; the PI REQUEST lights indicate levels 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 levels on which interrupts are currently being held; the level that is actually being serviced is the lowest-numbered one whose light is on. When an IN PROGRESS light goes on, the corresponding REQUEST goes off and cannot go on again until IN PROGRESS goes off when the interrupt is dismissed.

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

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 next page*]. 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.

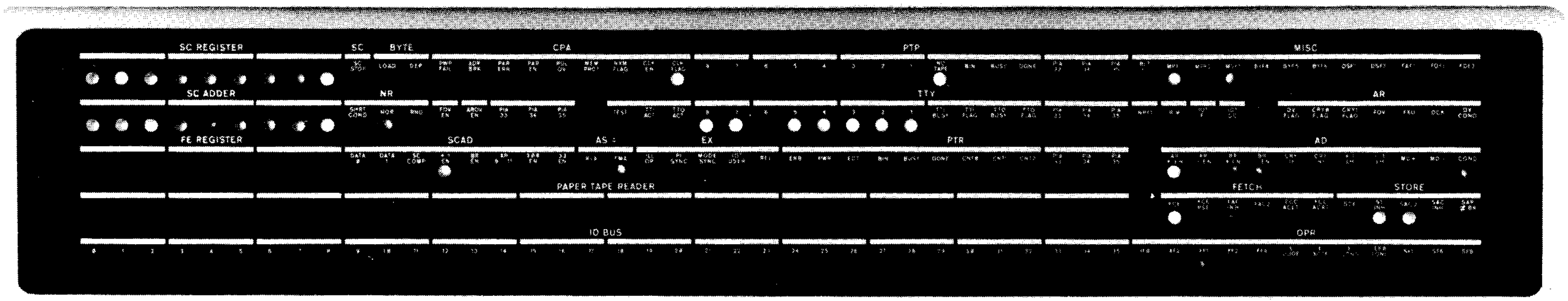
The upper four rows on the bay 1 panel include the indicators for reader, punch and terminal (see Appendix H1, DECsystem–10 manual). The bottom row displays the information on the data lines in the IO bus. The AR lights at the upper right are the flags — FXU is Floating (exponent) Underflow, DCK is No 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 on the panels are for maintenance. If the operator must use them, he should consult the maintenance manual and the flow charts.

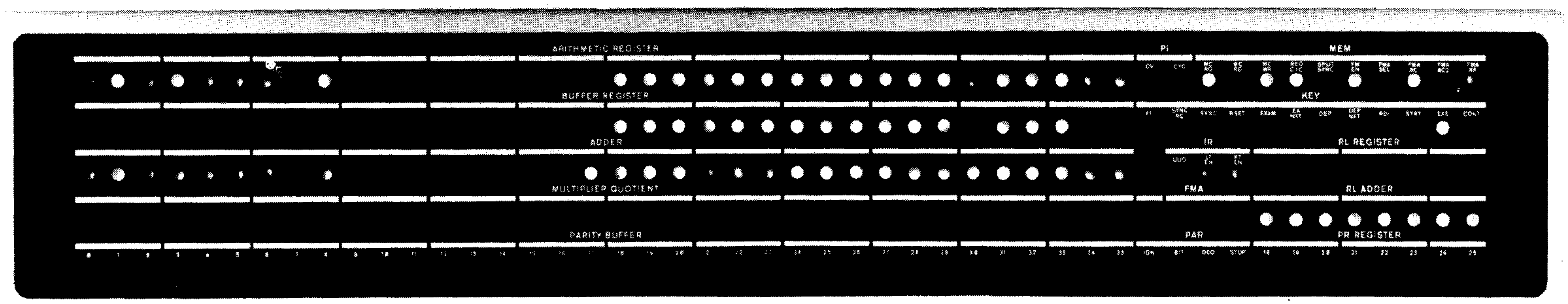
Operating Keys

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.

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



Indicator Panel, KA10 Arithmetic Processor, Bay 1



Indicator Panel, KA10 Arithmetic Processor, Bay 2

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* §5.1.]

Codes of readin devices are: PTR 104, DTC 320, DTC2 330, TMC 340, TMC2 350.

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

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 μs, 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.

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 non-existent-memory stop.

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

> 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 (read-only or read-modify-write). WRITE selects access for writing only. 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).

If the interrupt for an address break is started before the completion of the instruction that caused it, that instruction will be restarted upon the return from the interrupt routine unless provision is made by the program to do otherwise. In such a case, the address break will recur, producing a loop between the processor interrupt and the interrupted program. The operator can free the processor by momentarily releasing the break switch.

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

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

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.

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.

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.

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 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

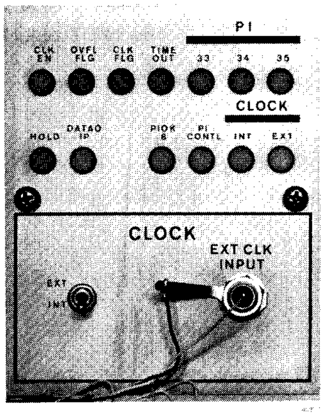
This switch is left on for normal operation with a fast memory. Turning it *off* (lower part in) substitutes the first sixteen core locations for the fast memory. The switch is left off if there is no fast memory, and it can be used to allow stopping or breaking at fast memory references.

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.

Real Time Clock DK10

The real time clock for the KA10 is usually installed in a DECtape cabinet and has a small control panel mounted directly on the logic behind the cabinet door. 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.

Appendix G
Handling Memory

A number of different types of memory units are available for use with PDP–10 processors. In a DECsystem–10, all are core memories that are external to the processor (on an external bus) and can be shared with other processors, central or peripheral. In a DECSYSTEM–20, all memory is internal to the individual system; in other words it is not directly accessible to any external processor, although provision is made for transfers of data between memory and IO devices via internal channels or some similar mechanism. The earlier internal memories are core, but all current ones are MOS. External memories are set up and controlled by switches; internal memory is handled entirely by software. Mixing of various kinds of units is more likely to occur in older systems as a result of upgrading or adding new models, whereas the greatly increased capacity and efficiency of current memories makes mixing less likely in newer systems.

Although the programmer usually regards an address simply as the number of a location somewhere in memory, the memory system interprets the address in two or more parts depending on the physical configuration of the memory units. With the traditional configuration of a single controller and associated storage module in each memory, the memory system interprets the address in two parts: the high order part is the number of a memory, and the low order part is the number of a location in the memory selected by the high order part. Every such memory has switches for selecting its number, i.e. the number to which that particular memory will respond when it appears in the appropriate bits on the address bus. For a given address length, the number of bits used for the memory number depends upon the size of the individual memory — the larger the size, the more bits are needed to specify the location within the memory and the fewer that are needed to select the memory itself. In more recent units, a single controller may have up to four storage modules, and some memories contain two controllers. In these cases the address word is divided into several parts, where the highest order part selects the controller, the next selects the module, and the remaining bits select the location. With KL10 internal memories, the two least significant bits are used only for the starting position in a 4-word group, and which words are accessed in the group is determined by the request signals. The newest KL10 MOS memory, the MF20, has a single controller with three storage modules, each of which acts like an independent memory unit, with 4-word access so interleaving is unnecessary.

Every unit has some means for taking it off line or "deselecting" it. To deselect a memory relative to a given processor means to remove that memory logically from the bus for that processor; in other words for the given processor that memory no longer exists. In some cases only the entire unit can be deselected, but with more recent memories, controllers and even storage modules can be deselected individually. If a storage module fails, it must be deselected if the system is to continue to run. Moreover, if the processor is a KA10 or the Monitor is a TOPS–10 version earlier than 5.06, the deselected memory must be replaced so there is no gap in the physical address space. This may be done by resetting the switches on the highest numbered memory so it fills the gap left by the deselected one. When any system is installed, the system administrator (in consultation with Field Service) should work out a separate procedure to be followed in the event that any given storage module fails. In other words there should be a set of procedures, and the set should contain as many procedures as there are modules in the system. A procedure might be as simple as filling a gap, but it may also involve the software or entail other complications. Whenever the organization of the system is changed in any way, that fact should be recorded, and the administrator should review, and if necessary change, the procedures to make sure they are appropriate to the new configuration. When a failed module is deselected, all other modules interleaved with it must also be deselected if speed is to be maintained. But if memory capacity is more important that speed, the other modules can be run without interleaving, or in a 4-way situation, two of the remaining modules can be 2-way interleaved.

G.1 DECsystem–10 Memories

There are seven types of external memories available for use in a DECsystem–10. §1.10 lists the types and gives their size and timing characteristics when used in a system based on a KA10 or KI10 processor. §1.7 lists the characteristics of the several among them that are suitable for use with the KL10. An MG10 or MH10 memory may be accessed by up to eight different processors; in other words one of these memories may be connected to eight memory buses and thus be part of the memory systems of eight different processors. The earlier memories are limited to four processor ports. Most memories are designed for use with either a 22- or 18-bit address and may therefore be used with any PDP–10 processor that can handle external memory. The earliest memories were designed specifically for use with a KA10; these are limited to an 18-bit address and can be used on a KI10 memory bus only by interfacing to it through a KI10–M adapter[1]. The more recent external memories can be used with a KL10 by having them on a KI10 memory bus that is interfaced to the KL10 S bus by a DMA20 adapter. The KI10–M is transparent to programmer and operator, but the DMA20 (discussed at the end of this section) must be set up by the software, and it supplies error and other information to the software.

Besides address switches, every memory has a power switch, interleave and deselect switches, a restart or reset switch, and a single step switch. With the MF10 and earlier memories, each unit actually has a separate set of address, interleave and deselect switches for each of the four processors to whose buses it may be connected. Hence a given memory may be number 2 for processor 0 but be number 7 for processor 1; by the same token it may be interleaved with some other memory relative to processor 1 but be deselected altogether from processors 2 and 3. A given memory should, however, have the same number with respect to all processors controlled by a single Monitor; e.g. if the Monitor running in central processor 0 sets up a direct-access processor to move data in or out of the memory connected to processor 0, those memory units used by both processors should have the same numbers.

Memory setup and operation differs among types, and these are discussed separately in the following pages. However, the MG10 and MH10 memories are described together as they are almost identical. Moreover, these two most recent memories employ a different interleave procedure than that used by all of the others. Among the earlier memories, some can be interleaved only in pairs, whereas others can be interleaved in either pairs or quadruplets. In any event the memories in a set that is to be interleaved must all be the same size and must occupy a contiguous area of the overall address space. For a 2-way interleave, the pair of memories must be numbered n and $n+1$, where n is even. The interleaving is accomplished by setting the interleave switches for the same processor at both memories to the INTL position. This action interchanges the least significant bits of the memory number and the location, so the least significant address switch at the memory is actually selecting a state for memory address bit 35. Hence all even addresses given by the processor in the interleaved set actually address the even-numbered memory, and all odd addresses address the odd-numbered memory. A 4-way interleave must be done on a group of memories numbered n, $n+1$, $n+2$ and $n+3$, where n is divisible by four. The interleave is accomplished by interchanging both the least significant bits of the memory number and location and the next more significant bits of those two quantities. The illustration on the next page shows the complete address structure for memories of all sizes, with and without interleaving.

Most or all of the switches on a memory are located on a switch panel mounted with the logic wiring inside the front door of the bay. The lights are always on an outside panel at the top of the bay. Every indicator panel

[1] The same considerations apply to use with direct access processors; e.g. using an 18-bit memory with a DF10C or a KI10-type bus on a DL10 requires an adapter.

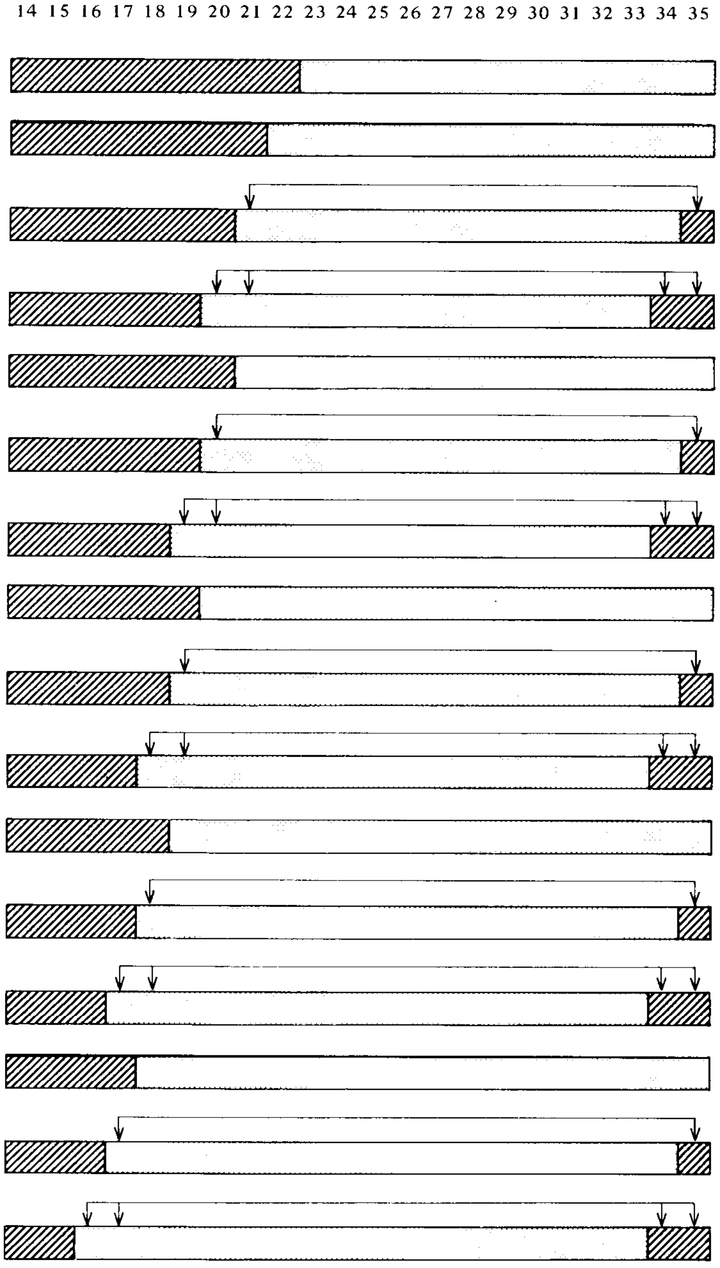
ADDRESS STRUCTURE

▨ *Memory number*

□ *Stack address (location)*

Memory Size	Address Bit Pairs Switched	Minimum Total Memory
8K	None	
16K	None	
16K	1	32K
16K	2	64K
32K	None	
32K	1	64K
32K	2	128K
64K	None	
64K	1	128K
64K	2	256K
128K	None	
128K	1	256K
128K	2	512K
256K	None	
256K	1	512K
256K	2	1024K

Address Bit Configuration

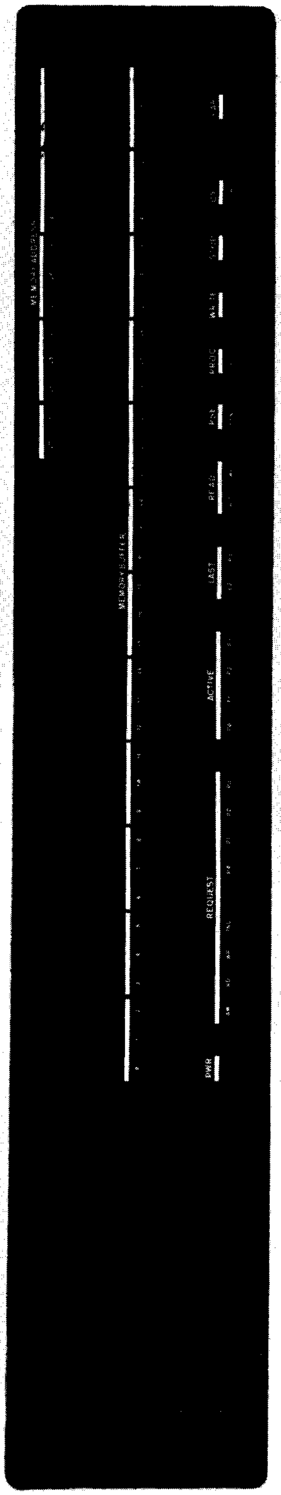


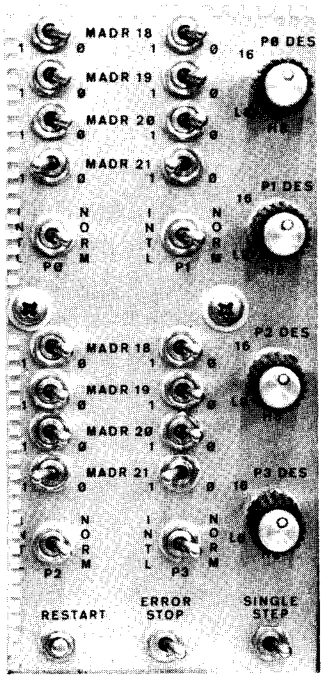
has sets of memory address and memory buffer lights. These indicate the last location accessed and display the information read from or written into that location (except the buffer lights are off after the read part of a read-modify-write). All transfers between bus and core are made through the buffer. On the MF10 and earlier memories, the four ACTIVE lights identify the processor that currently has access, and the two LAST lights indicate which of processors 2 and 3 more recently had access. Priority among processors is their numerical order, with 0 first. However adhering strictly to this priority in a memory used by four processors might easily lead to the total exclusion of the lowest priority processor. To make this occurrence less likely, in a conflict between processors 2 and 3, access is granted to the one that has had it less recently. The panel also has a power indicator, lights that identify the type of request, and a parity bit. A memory in operation but idle is indicated by the AW light, meaning the memory is awaiting a request. STOP goes on only following completion of a cycle in single step mode. In a read access a memory completes its cycle without need of further communication from a processor. Following the read part of a write or read-modify-write however, the memory waits with SYNC on for a write restart, receipt of which is indicated by a light often labeled RS. Failure of the processor to supply the restart turns on the INC light (if present). Such an incomplete request may or may not hang up a specific type of memory, but it always results in leaving the addressed location clear. Since the system uses odd parity, a subsequent read at that location will result in a bad parity zero. Completion of a cycle is sometimes indicated by a CYC DONE light. Some memories identify all processors currently making requests. There are also lights that reflect the internal state of the memory (for further information refer to the appropriate maintenance manual).

The system administrator should be aware that even if the hardware and software are capable of dealing with noncontiguous memory, some of the programs being run may require that the memory be contiguous. This could be necessitated only by real time programs, but in general it is best to avoid having memory gaps unless they are known to be of no consequence. Note that to avoid gaps in a system with different size memories requires arranging them so the smaller memories are at the top or are grouped so as to fill the spaces between the larger memories. Consider a system with two 32K memories and one 16K memory. The 32K memories must be numbered 0 and 1, and using the same numbering scheme, the 16K unit must be numbered 2.0 (really 4 in 16K terms). If there were two 16K units we could number them 2 and 3, with their space straddled by the 32K modules numbered 0 and 2 (i.e. the 16K modules are numbered 1 and 1½ in 32K terms).

MA10 CORE MEMORY

This unit has a capacity of 16K words, a cycle time of 1.00 μs, and operates only with an 18-bit address, of which four bits select the unit. At the left inside the front door are two tall panels of margin check switches, all of which should be set left. The switch panel is in the lower right corner. Note that there is no power switch: power is controlled by the circuit breaker on the power control panel on the rear plenum door. The three switches at the bottom of the panel are for all processors. Ordinarily an incomplete cycle does not affect memory operation; the unit simply drops the unfinished cycle and awaits the next request. But running with ERROR STOP on causes the memory to cease operation with INC on when a processor fails to supply a write restart. To free the memory so it may await further processor requests, push the RESTART button. Pressing RESTART while SINGLE STEP is on allows the memory to respond to just one processor request. Once this single cycle has been completed, STOP goes on and the memory will acknowledge no further requests, thus giving a nonexistent memory indication in any processor that makes one. Pressing RESTART again allows the memory to respond to one more request. It is possible for a power line transient to hang up the memory in a request; to free it, turn the power off and back on again by means of the circuit breaker at the back.





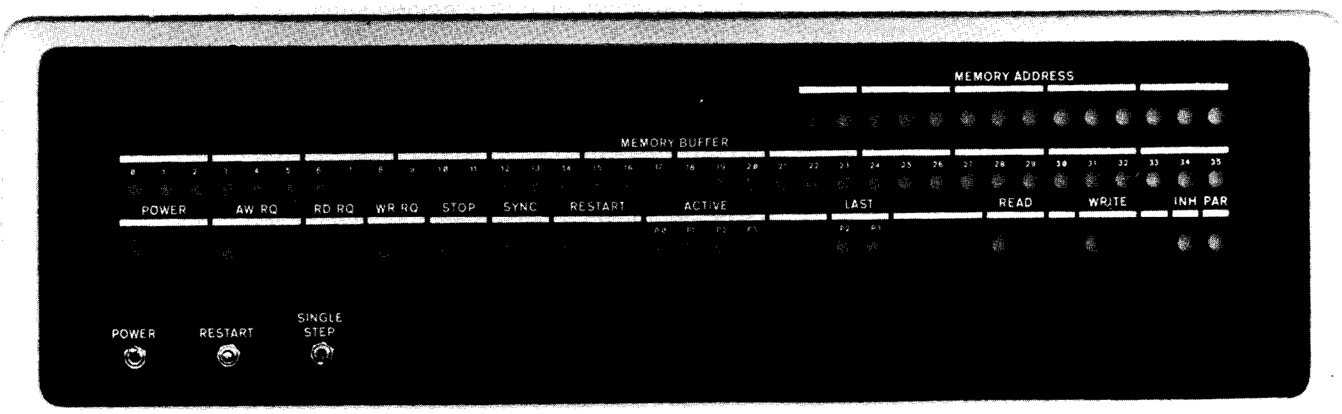
6908-A

Above: Switch Panel
Left: Indicator Panel

The remaining switches are in four sets for the individual processors. In each set the MADR switches allow selection of the memory number for address bits 18–21. Setting the fifth toggle to INTL interchanges address bits 21 and 35 for a two-way interleave. Setting the deselect switch to the 16 position deselects the whole unit from the corresponding processor. However the switch also has positions for deselecting the lower or higher half of the memory, resulting in an 8K memory with a 5-bit number. *Eg* setting the switch to L8 deselects the lower half and selects a 1 for address bit 22; the unit is then an 8K memory whose locations are addressable in the upper half of the original 16K address space. For 8K operation the interleave switch should always be set to NORM.

MB10 CORE MEMORY

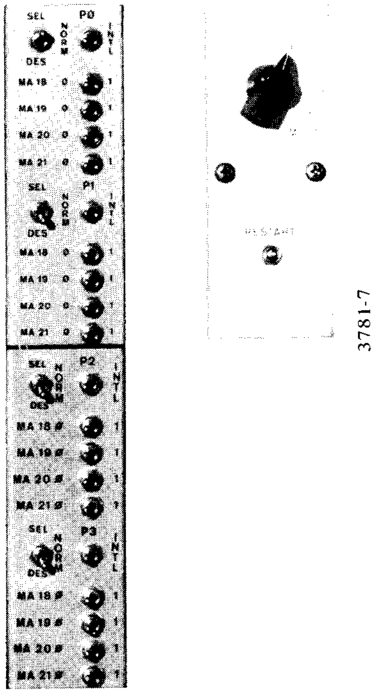
This unit has a capacity of 16K words, a cycle time of 1.65 μs, and operates only with an 18-bit address, of which four bits select the unit.



Indicator Panel

Several switches for all processors are located on the indicator panel. Inside the front door are three switch panels. The one in the upper right corner has a rotary switch for selecting operation with any single processor or selecting the more usual multiprocessor operation. Also on this small panel is a button labeled RESTART, even though there is another button with the same label on the indicator panel. Should a processor fail to supply a write restart, the memory ceases operation with SYNC remaining on; to free the memory so it may await further processor requests, push both restart buttons at the same time. Pressing the indicator panel RESTART while SINGLE STEP is on allows the memory to respond to just one processor request. Once this single cycle has been completed, STOP goes on and the memory will acknowledge no further requests, thus giving a nonexistent memory indication in any processor that makes one. Pressing the top RESTART again allows the memory to respond to one more request. It is possible for a power line transient to hang up the memory in a request; to free it, turn the power off and back on again.

On the two small panels in the lower right corner are four sets of switches for the individual processors. Each set contains a deselect switch, an interleave switch, and four MA switches that allow selection of the memory number for address bits 18–21. Setting the interleave toggle to INTL interchanges address bits 21 and 35 for a two-way interleave.

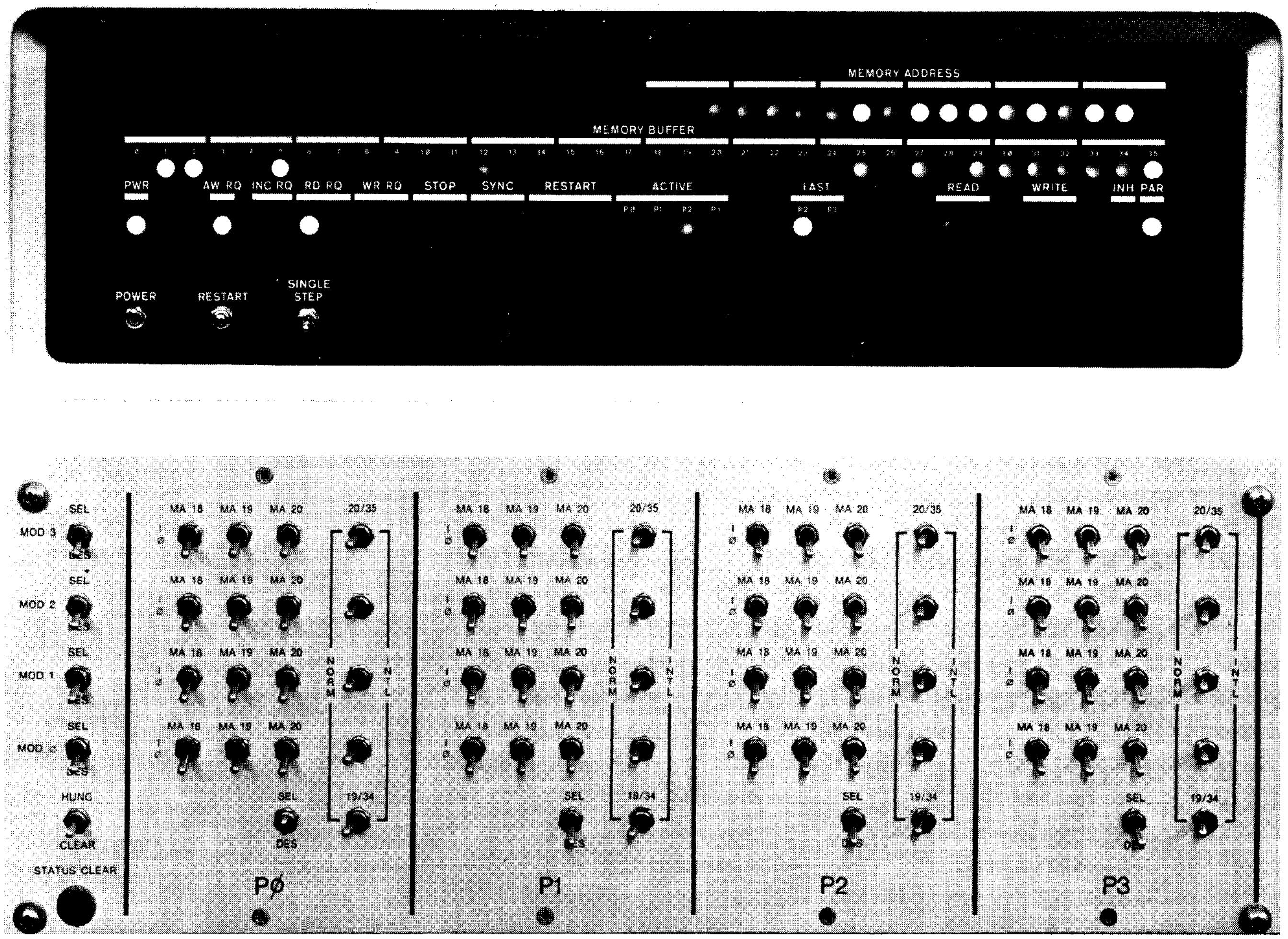


Switch Panels

The toggles at the left end of the logic rows are margin check switches, all of which should be set down. The toggle on the little panel between rows U and V should be set left (+10 FXD).

MD10 CORE MEMORY

This unit may contain up to four 32K memory modules, which are numbered and interleaved independently, but which share a common interface with the bus and therefore otherwise act as a single memory of 32, 64, 96 or 128K words. The cycle time is 1.8 μs or less, and the unit operates only with an 18-bit address, of which four bits select the individual module. Interleaving among modules within a single MD10 is useless, because the whole unit is tied up whenever any module is performing a cycle even if it has already disconnected from the bus, as while a word is being written. Hence interleaving must be done between a module and one or three other 32K memories, which may themselves be modules in other MD10s. Note that there is no requirement of continuity of the address space within a given MD10 — if there were, interleaving would be impossible. Suppose a memory system consisted of two full-size MD10s with complete two-way interleaving. The modules in one unit could just as well be numbered 0, 3, 5 and 6, with the remaining numbers applied to the modules in the other unit.



Upper: Indicator Panel
Lower: Switch Panel

Several switches for all processors are located on the indicator panel. Pressing RESTART while SINGLE STEP is on allows the memory to respond to just one processor request. Once this single cycle has been completed, STOP goes on and the memory will acknowledge no further request, thus giving a nonexistent memory indication in any processor that makes one. Pressing RESTART again allows the memory to respond to one more request.

The remaining switches are on the big panel inside the front door. This panel has four large sections for the individual processors and a column at the left end for all processors. The upper four toggles in the column allow deselecting a single module from all processors. The button at the bottom is not used, and the remaining switch actually has three positions, including an unmarked center one. With this switch in the unmarked position, an incomplete cycle does not affect memory operation; the unit simply drops the unfinished cycle and awaits the next request. But running with the switch up in the HUNG position causes the memory to cease operation with INC RQ on when a processor fails to supply a write restart. To free the memory so it may await further processor requests, press the switch down to the momentary-contact CLEAR position.

Above the switch panel is a narrow panel with three margin check switches, all of which should be in the center NOM position. Above that is a power supply panel whose switch is not operative. The margin toggles at the bottom left should all be set down.

Each of the large sections of the panel contains four sets of MA switches for independently selecting the numbers of the individual modules for address bits 18–21. (Note that each of the upper four rows of switches extending across the entire panel affects the single module specified by the label at the left end.) Each deselect switch allows the operator to disconnect the entire unit from the specified processor. The right column of each section contains five interleave switches, of which the upper four are for two-way interleaving of individual modules and the bottom one is for four-way interleaving of all modules together. Setting one of the upper four switches right interchanges address bits 20 and 35 only for the given module with respect to the specified processor. Setting the bottom toggle to the right interchanges address bits 19 and 34 only for the specified processor but for all modules in the unit.

Note that one can deselect all modules from one processor, or one module from all processors, but not a single module from a single processor.

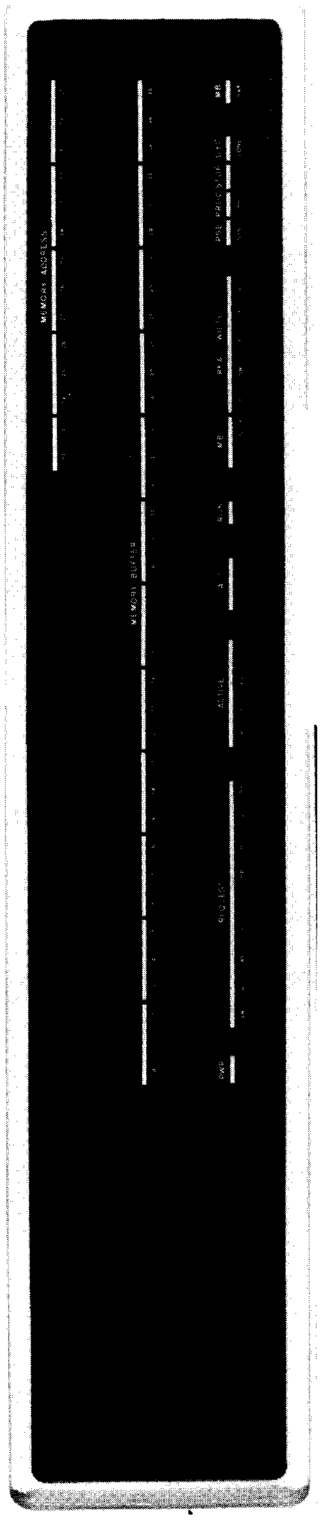
Hence for a given processor some modules can be used normally while others are interleaved on a two-way basis with other memories outside the unit. But if one module enters into a four-way interleave with respect to a given processor, all modules must. This means that when a unit is used in a four-way interleave for a given processor, among the switches for that processor the 19/34 switch and all four 19/35 switches must be set to INTL. If four-way interleaving is used among MD10s, there must be four of them and each must contain the same number of modules.

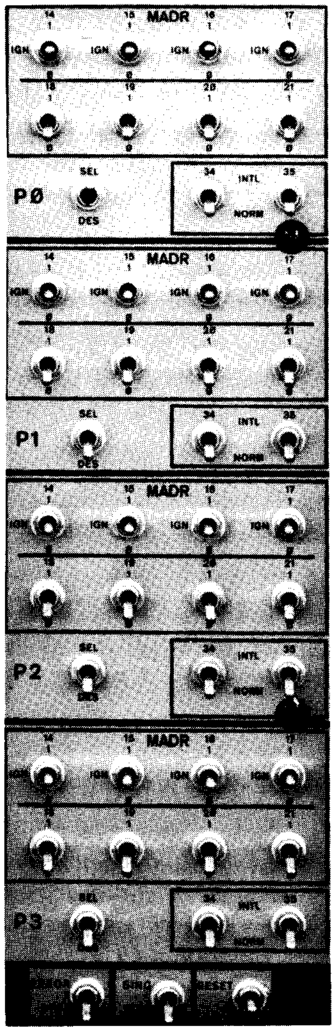
The restriction on number of units can be bypassed through worthless interleaving among modules within a single unit. This produces somewhat lopsided interleaving — *eg* one could have a three-way interleave of three MD10s with 32K, 32K and 64K.

ME10 CORE MEMORY

This unit has a capacity of 16K words, a cycle time of 1.00 μs, and operates with either a 22- or 18-bit address, of which eight or four bits select the unit. At the left inside the front door is a tall panel of margin check switches, all of which should be set left. The switch panel is at the lower right. Note that there is no power switch: power is controlled by the circuit breaker on the power control panel on the rear plenum door. The three switches at the bottom of the panel are for all processors. Ordinarily an incomplete cycle does not affect memory operation; the unit simply drops the unfinished cycle and awaits the next request. But running with ERROR STOP on causes the memory to cease operation with INC on when a processor fails to supply a write restart. To free the memory so it may await further processor requests, push up the RESET switch. Pressing RESET up while SING STEP is on allows the memory to respond to just one processor request. Once this single cycle has been completed, STOP goes on and the memory will acknowledge no further requests, thus giving a nonexistent memory indication in any processor that makes one. Pressing RESET again allows the memory to respond to one more request.

The rest of the panel is in four sections for the individual processors. Each section has two rows of MADR switches for selecting the memory number. If the bus supplies a 22-bit address, all eight MADR switches are used to select the memory number for address bits 14–21. For an 18-bit address the upper row must all be set to the center IGN position, and the lower row is used to select the memory number for address bits 18–21. Completing each panel section are a deselect switch and a pair of interleave switches. Setting the 35 toggle to INTL interchanges address bits 21 and 35 for a two-way interleave. Setting the 34 toggle up interchanges bits 20 and 34 for a four-way interleave.





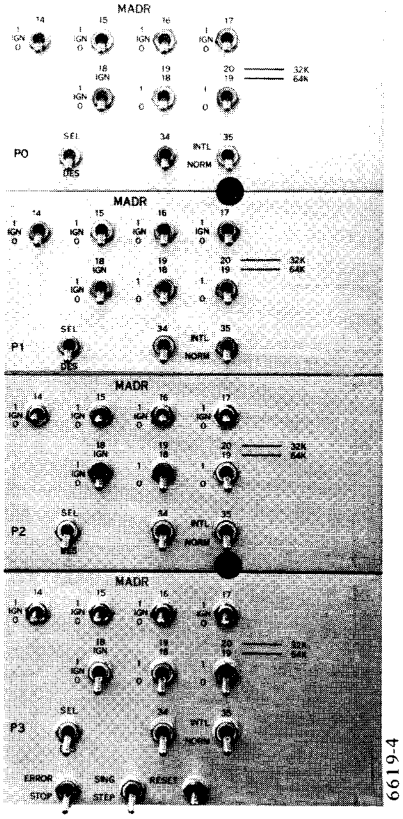
5041-9

Above: Switch Panel
Left: Indicator Panel

MF10 CORE MEMORY

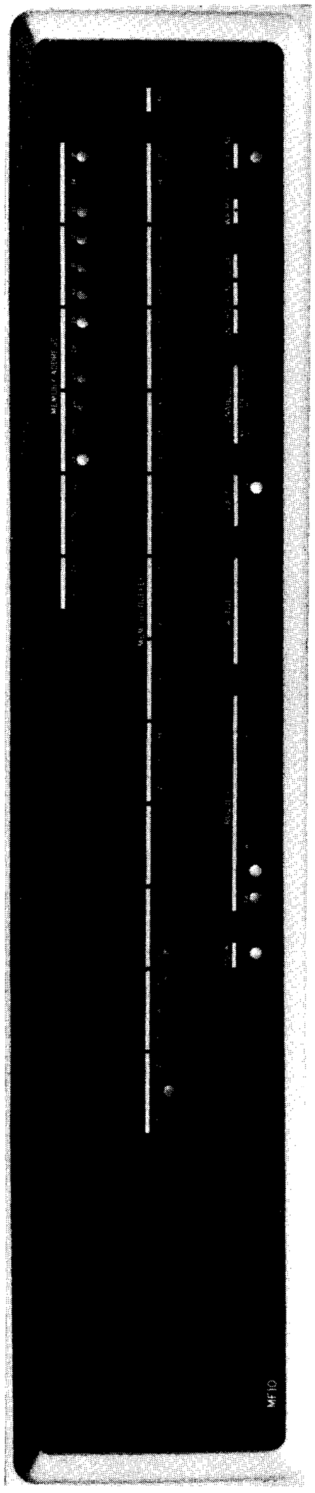
This unit has a capacity of either 32K or 64K words, a cycle time of 1.00 μs, and operates with either a 22- or 18-bit address. The number of bits that select the unit depends on both the address length and the unit capacity. All switches are on panels in the lower half of the unit inside the front door. The tall panel at the left has margin check switches, all of which should be set left. At the upper right is a small maintenance panel, of interest in that it contains a size switch, which is set by Field Service to make the unit operate in a manner appropriate to the installed capacity, and whose position therefore indicates that capacity. Note that there is no power switch: power is controlled by the circuit breaker on the power control panel on the rear plenum door.

The four-part switch panel is at the lower right. The three switches at the bottom are for all processors. Ordinarily an incomplete cycle does not affect memory operation; the unit simply drops the unfinished cycle and awaits the next request. But running with ERROR STOP on causes the memory to cease operation with INC RQ on when a processor fails to supply a write restart. To free the memory so it may await further processor requests, push up the RESET switch. Pressing RESET up while SING STEP is on allows the memory to complete only one cycle in response to a single processor request. In the absence of a second request in the interim, pressing RESET again allows another request-cycle combination. However, if following completion of a cycle, a processor makes a request before RESET is pressed, the memory will return an acknowledgement, thus avoiding a nonexistent memory indication. Of course the memory will hang in the second cycle, either waiting to write or with the processor awaiting a read restart. Pressing RESET in this circumstance causes the memory to complete its cycle leaving it free to acknowledge yet another request.



6619-4

Above: Switch Panel
Right: Indicator Panel



6908-B

The remaining switches are in four sets for the individual processors. Each set has two rows of MADR switches for selecting the memory number. If the bus supplies a 22-bit address, the upper row is used to select the left four bits (address bits 14–17) in the 6- or 7-bit memory number. For an 18-bit address the upper switches must all be set to the

IGN position. The configuration of the switches in the lower row depends only on memory size. For a 32K unit these switches correspond to address bits 18–20 and are used to select a 3-bit memory number or the right three bits in a 7-bit number. For a 64K unit the left switch must be set to the center IGN position; the other two correspond to address bits 18 and 19 and are used to select a 2-bit memory number or the right two bits in a 6-bit number. Completing the switch complement for each processor are a deselect switch and a pair of interleave switches. Setting the 35 toggle to INTL interchanges either address bit 19 or 20 (depending on unit size) with bit 35 for a two-way interleave. Setting the 34 toggle up interchanges either address bit 18 or 19 with bit 34 for a four-way interleave.

MG10 AND MH10 CORE MEMORIES

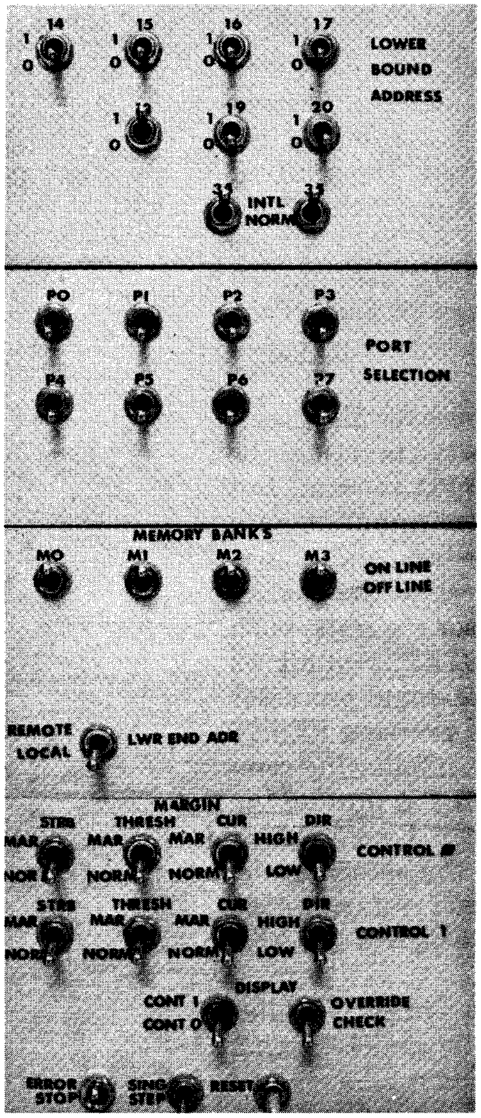
These two units are very similar. Both contain two controllers, each with one module, and both operate with either a 22- or 18-bit address. The two have identical indicator panels, and their switch panels, located at the lower right inside the front door, are almost identical as well. The MH10 is slightly slower but has twice the capacity: the basic cycle time of the MG10 is 1000 ns and its module capacity is 64K; corresponding statistics for the MH10 are 1180 ns and 128K. Besides the usual buffer and address lights on the indicator panel, there are two sets of control lights, one for each of the controllers in the unit. No power switch is visible: it is located on the power control panel on the rear plenum door.

The three switches at the bottom of the switch panel are for all processors. Ordinarily an incomplete cycle does not affect memory operation; the unit simply drops the unfinished cycle and awaits the next request. But running with ERROR STOP on causes the memory to cease operation with INC RQ on when a processor fails to supply a write restart, and it also causes a stop on detection of a parity error or other error in the control logic. To free the memory so it may await further processor requests, push up the RESET switch. Pressing RESET up while SING STEP is on allows the memory to complete only one cycle in response to a single processor request. In the absence of a second request in the interim, pressing RESET again allows another request-cycle combination. However, if following completion of a cycle, a processor makes a request before RESET is pressed the memory will return an acknowledgement, thus avoiding a nonexistent memory indication. Of course the memory will hang in the second cycle, either waiting to write or with the processor awaiting a read restart. Pressing RESET in this circumstance causes the memory to complete its cycle leaving it free to acknowledge yet another request.

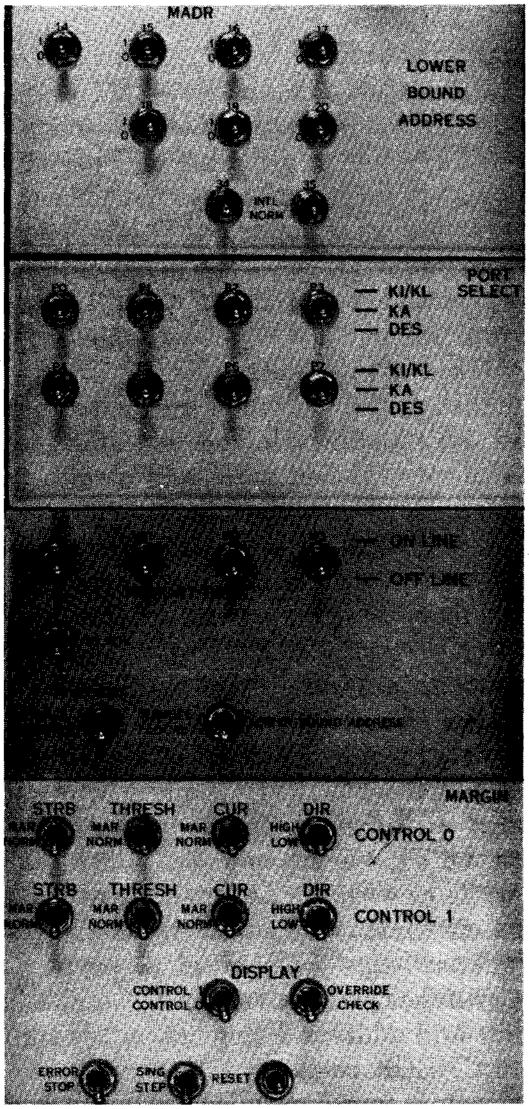
In the pair of switches just above, the left one selects which controller shall have its buffer and address displayed on the indicator panel. Setting the right switch to CHECK causes the latches for the buffer, address and various control indicators in a controller to freeze. Should the controller in which the error occurred not be the one currently displayed, that situation can be remedied simply by flicking the left switch. Setting the right switch to OVERRIDE frees the lights. In the upper half of the bottom panel section are separate rows of maintenance switches for the two controllers. For normal operation all of these switches should be set down.

On the second panel section from the bottom on the MH10 is a switch for deselecting the whole unit from all processors. The bank switches above allow deselection of single banks (each bank is a half module, 0 and 1 with controller 0, 2 and 3 with controller 1). Memory banks in use are identified by lights at the top of the indicator panel. The other two switches are not connected. The MG10 has bank switches, but the only other thing on the same panel section is a lower bound address switch. This switch is, however, connected the same as the memory switch on the MH10: setting it to LOCAL deselects the unit.

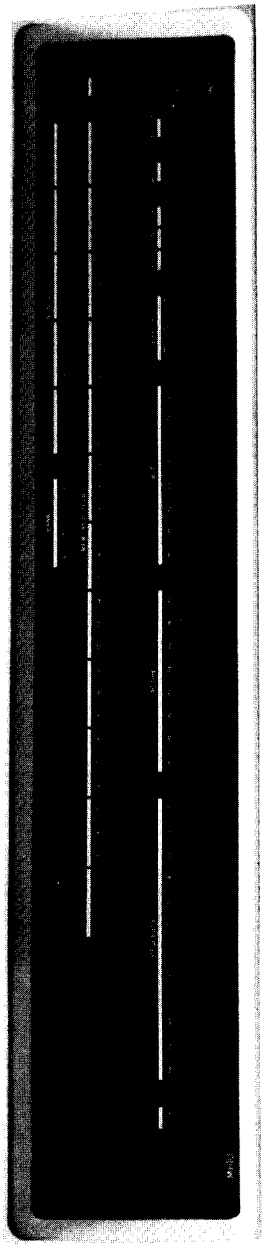
On the third part of the panel are eight switches for selecting the characteristics of the unit for each processor separately. The operator can deselect the unit, select KA10 operation (18-bit address), or select operation



MG10 Switch Panel 7370-13



MH10 Switch Panel 8202-2



Indicator Panel
8202-5

with a KI10 or KL10 (22-bit address). The indicator panel also has separate active and request lights for the eight processor ports. The processors have priority not in a fixed order, but rather in three sets: 0 and 1, 2 and 3, and 4–7. In the first two sets priority alternates between the two processors in the set. In the third set the priority order is a loop — 4, 5, 6, 7, 4, ... — in which whenever the top priority processor gains access, top priority moves to the next position. For example the priority order is initially 4 to 7, where 5 has priority over 6. Once 4 gains access the order becomes 5, 6, 7, 4, where 5 now has top priority and 6 has priority over 4.

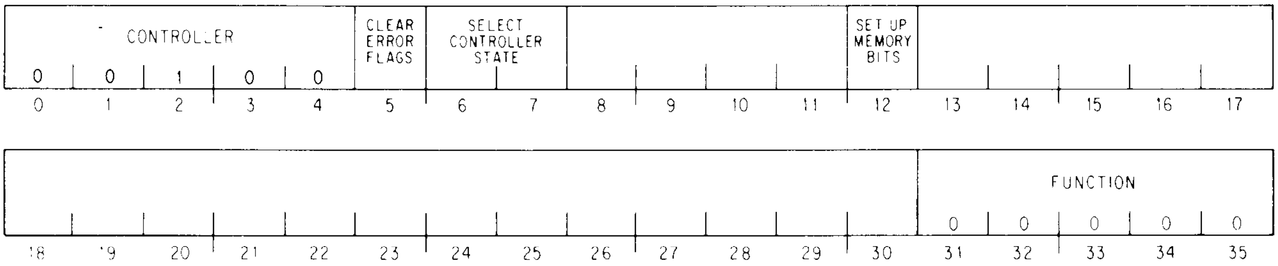
The MADR switches at the top of the panel are for selecting the memory number and controlling interleaving for all processors. Generally bits 19 and 20 are not used, and bit 18 is used only in the MG10, unless a unit has only one module or the capacity is decreased by deselecting banks. Of course bits 14–17 are irrelevant for operations through any processor port whose selection switch is set to KA. The number set into MADR switches 14–20 defines the lowest address (lower bound) in the unit, and addresses are assigned through the two modules in order, skipping over any deselected banks. However, if switch 35 is set to INTL, the even addresses are assigned to controller 0 and the odd ones to controller 1, producing 2-way interleaving within the unit. If both 34 and 35 are up, addresses are assigned like 2-way interleaving, but only the even addresses are used at all if the memory number is even and only the odd ones if the memory number is odd. It is assumed in this case that some other memory is assigned a number that differs only in the LSB, producing 4-way interleaving in the pair.

DMA20 MEMORY BUS ADAPTER

External memories can be used with a KL10 processor by employing a DMA20 adapter to interface the KI10 external memory bus to the KL10 S bus. Although every external memory has its own controller and must be set up via its own switches, the DMA20 serves as the S bus controller for the whole external memory system, and it must be set up by the software for the kind of interleaving used. The individual external memories that are to be interleaved in a set of two or four must be connected to different KI10-bus ports, of which the DMA20 has four.

Setup is accomplished by means of the S bus diagnostic instruction, to which the DMA20 is controller number 4. The diagnostic cycle is also used to get error information and check data transfers between the processor and buffers in the external memories. For use with the DMA20 are the following two functions, 0 and 1, distinguished by the LSB of the function code.

Function 0, To Memory



5 Clear the controller error flags (see bits 3–5 of the return word).

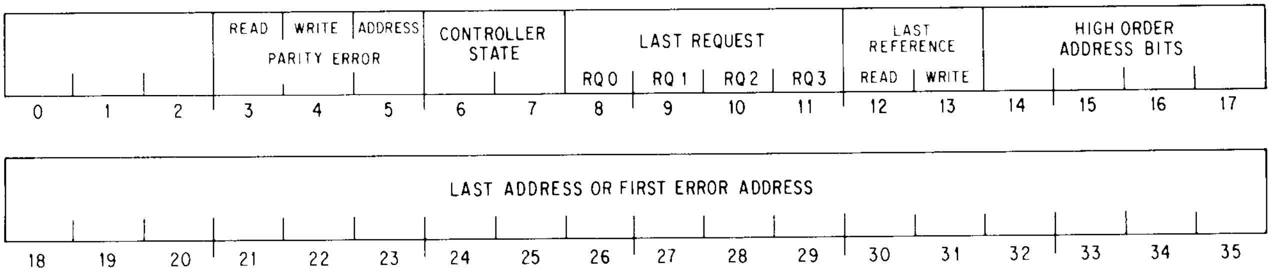
6–12 If bit 12 is 0, ignore bits 6 and 7. But if it is 1, select the interleave mode for the storage modules according to bits 6 and 7.

NOTE

State selection by bits 6 and 7 applies to the DMA20, even though the interleave characteristics of external storage modules must be selected manually.

Bits 6–7	*State Selected*
00	Off line
01	No interleave
10	2-way interleave
11	4-way interleave

Function 0, Return



3–5 These are error flags for the storage modules connected to this controller. Setting a flag locks bits 8–35 on the identification of the current reference.

3 A word with even parity was read from a storage module.

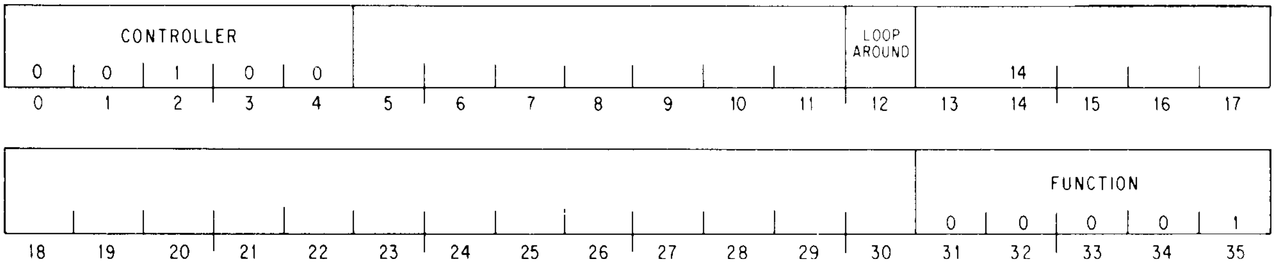
4 A word with even parity was written in a storage module.

5 The controller has received an address with even parity over the S bus. The address intended for the reference is available in ERA.

6–7 These bits identify the current controller state as set by the function word to memory.

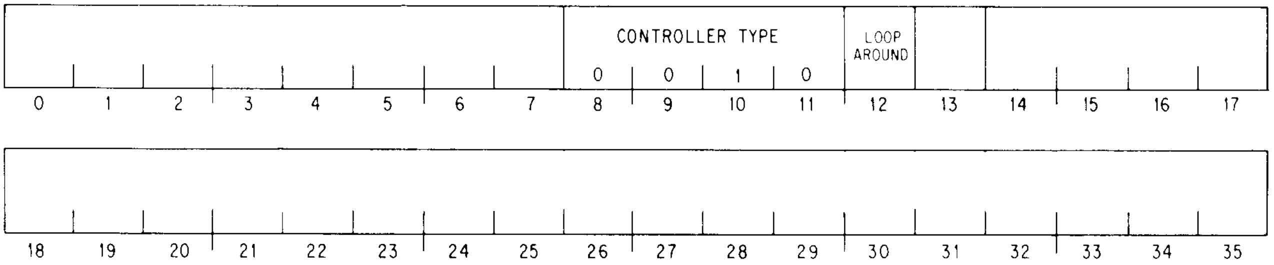
8–35 These bits regularly give the address of the most recent reference, whether it was read, write or both, and which words in the 4-word group were requested. However, when one of the error flags (bits 3–5) is set, these bits lock onto the identifying information for that reference. The bits remain static, even with additional errors, until the program unlocks the register by clearing the error flags (a function word to memory with a 1 in bit 5).

Function 1, To Memory



12 For checking the transfer logic between the processor and the buffer in a storage module, a 1 in this bit causes the next pair of references, write followed by read, just to load the buffer and then read it without affecting any memory location.

Function 1, Return



12 The controller is set up to loop around a write-read pair of refernces through a module buffer. Doing a single looparound clears the bit.

G.2 DECSYSTEM–20 Memories

All DECSYSTEM–20 memory is internal and is handled entirely by the program — there are no switches associated with it (power is controlled through the main power system).

The DECSYSTEM–2020 has only one type of MOS memory; it is integral with the system and has no setup. Although it is expandable by adding array boards, it is treated as a single unit with all access in single words via the KS10 bus. General information about the memory, including size and timing, is given in §1.8. §4.8 gives detailed information about memory status, which is the link between the software and the memory controller: the software can read error information, handle the memory flags, and force errors in order to check the storage array correction circuits. Should errors be discovered in an array board, the Monitor may be able to work around it simply by not using the bad board. However, there is no way to tell the Monitor initially that a board is bad, and hence there is no way to prevent the Monitor from storing information — perhaps critical information — in it. The best procedure therefore is to remove the bad board and replace it with the last board in the array, so that all of storage is physically contiguous.

Several types of memories are available for use in other DECSYSTEM–20s. Information about the characteristics of these memories is given in §1.7, and the remainder of this appendix discusses the way the software handles them through the S bus diagnostic cycle. The earlier of these memories are the MA20 and MB20, which have core storage modules that can be interleaved and which are so similar they are described together. The latest storage equipment is the MF20 MOS memory, available only with an extended processor. In an MF20, a single controller can handle three storage modules, each of which is treated as a separate memory unit. Interleaving is unnecessary as data is handled in 4-word groups (referred to in the MF20 discussion as "quadwords," as the word "group" has a specific meaning in the MF20 hardware). In the descriptions of the S bus diagnostic functions, bits for memory setup are indicated by an asterisk.

MA20 and MB20 Memories

The MB20 is slightly slower than the MA20 but has twice the capacity: the basic cycle time of the MA20 is 1000 ns and its module capacity is 16K; corresponding statistics for the MB20 are 1180 ns and 32K. A KL10 memory system can have two of these units. Each unit contains two controllers (numbered 0 and 1 or 2 and 3), and each controller can handle four storage modules (0–3).

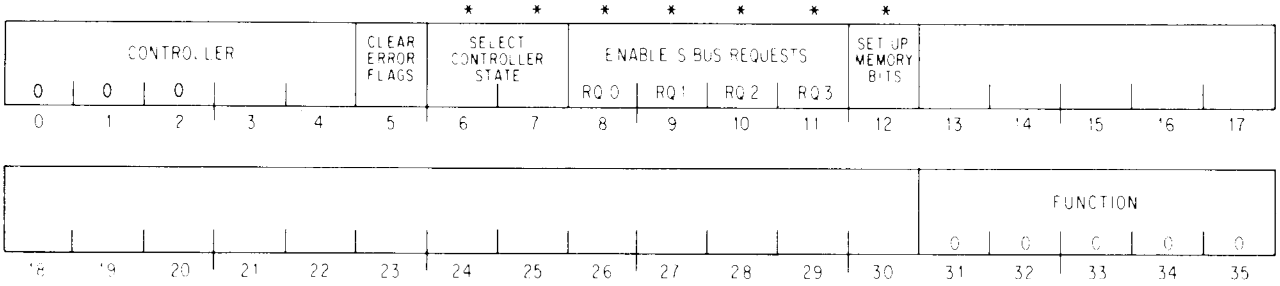
The memory system is set up by assigning addresses to the controllers, specifying the lower and upper address bounds for the storage modules on the controllers, and specifying which requests within a 4-word group each controller is to respond to. The controller address is simply the most significant four bits of a memory address, and the address bounds correspond to the next three or four bits of the memory address depending on whether the modules are 32K or 16K. Bits 34 and 35 of an address indicate the particular word for which access is being made, but the request lines indicate which words within the 4-word group are to be read or written. As an example consider a memory with 32K modules (MB20), so each controller has the full complement of 128K words associated with it. For no interleave, the program could simply assign the same address to arbitrary pairs of controllers, give lower and upper bounds for different halves of the address range (000–011 and 100–111) to the paired controllers, and specify that every controller should respond to all requests. With interleaving, two controllers are accessed simultaneously; and with 4-way interleaving each of the accessed controllers operates two storage modules simultaneously. Hence for interleaving, the program assigns the same address to an even-odd pair of controllers, specifying that the even controller shall respond to the even requests (0 and 2), whereas the odd controller shall respond to the odd requests (1 and 3). Since each controller then responds to only half the addresses in a 4-word group, the address bounds assigned must cover a range equal to twice the number of words of storage; in other words for the

128K the range must be given as 256K (000–111). The way the modules are set up inside the unit is handled automatically depending on whether the program specifies a controller state of 2-way or 4-way interleaving. For 2-way the address range is continuous and both requests are assigned to every module. For 4-way the modules are handled in pairs: in each pair one module is assigned half the range and one of the requests, the other is assigned the same half of the range and the other request. For more complete information along with detailed diagrams for all the cases, refer to the appropriate maintenance manual.

If the last module on a controller fails, it can be deselected by limiting the address range, i.e. lowering the upper bound. If speed must be maintained, the other three modules interleaved with it should also be deselected. On the other hand, if capacity is more important, the matching good modules in the two controllers can be 2-way interleaved, and the fourth module of the other controller can be used without interleaving. If any module other than the last fails, there is no way to deselect it without also deselecting everything beyond it. Should this happen, in order to keep as much memory functioning as possible, the bad module (three boards) should be removed and the last module on that controller put in its place.

Memory setup as well as errors and various maintenance functions are handled through the SBDIAG instruction, to which the four MA20 or MB20 controllers are numbered 0–3. For use with these controllers are the following two functions, 0 and 1, distinguished by the LSB of the function code.

Function 0, To Memory



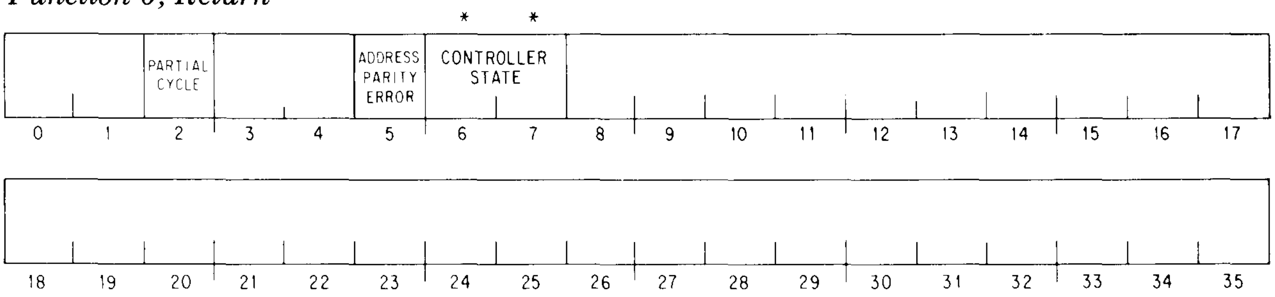
5 Clear the controller error flags (see bits 2 and 5 of the return word).

6–12* If bit 12 is 0, ignore bits 6–11. But if it is 1, select the interleave mode for the storage modules according to bits 6 and 7, and enable the word requests to which the controller will respond as specified by bits 8–11.

Bits 6–7	*State Selected*
01	No interleave
10	2-way interleave
11	4-way interleave

Requests enabled must be consistent with the interleave selection: no interleave, enable all requests; 2- or 4-way enable RQ0 and RQ2 in the even controller, RQ1 and RQ3 in the odd. Enabling no requests takes the controller off line.

Function 0, Return



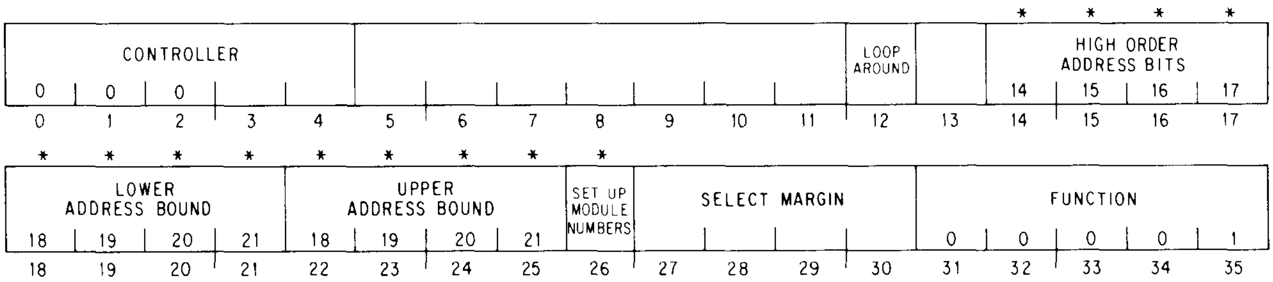
2, 5 These are error flags for the storage modules connected to this controller.

2 A storage module failed to complete a cycle, or the processor failed to send data for the write part of a read-pause-write within 8 μs.

5 The controller has received an address with even parity over the S bus. The address intended for the reference is available in ERA.

6–7* These bits identify the current controller state as set by the function word to memory.

Function 1, To Memory



12 For checking the transfer logic between the processor and the buffer in a storage module, a 1 in this bit causes the next pair of references, write followed by read, just to load the buffer and then read it without affecting any memory location.

14–26* If bit 26 is 0, ignore bits 14–25. Otherwise assign numbers to the storage modules as specified by bits 14–25. Essentially these bits define the range of addresses to which the modules will respond. Bits 14–17 give the four high order address bits, which are the same for all modules on a controller. Selection of individual modules is by the next three or four address bits depending on whether the modules are 32K or 16K. Configurations of these bits are assigned to the modules in physical order over the range from lower to upper, with duplication depending on the interleave characteristics.

27–30 These bits select a margin for diagnostic operation of the system.

0000	No-op — do not change present margin (if any)
0001	Return to normal operation (turn off margin)
0010	Select low current margin
0011	Select high current margin
0100	Select low strobe margin
0101	Select high strobe margin
1000	Select low threshold margin
1001	Select high threshold margin

Function 1, Return

*	*	*	*	*	*	*	*							*	*	*	*
STORAGE MODULES INSTALLED								CONTROLLER TYPE				LOOP AROUND		HIGH ORDER ADDRESS BITS			
7	6	5	4	3	2	1	0	0	0		1			14	15	16	17
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

*	*	*	*	*	*	*	*										
LOWER ADDRESS BOUND				UPPER ADDRESS BOUND								RUNNING ON MARGIN		REQUESTS ENABLED			
18	19	20	21	18	19	20	21							RQ 0	RQ 1	RQ 2	RQ 3
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

0–7* Storage modules installed on the controller are indicated by 1s in these bits. At present the maximum is four.

8–11 These bits specify the type of controller.

(0	Customer unit)
1	MA20
(2	DMA20)
3	MB20

12 The controller is set up to loop around a write-read pair of references through a module buffer. Doing a single looparound clears the bit.

Remaining bits indicate the way the controller and its modules have been set up by previous functions.

MF20 Memory

A single MF20 controller handles three storage modules of 256K each. A module is referred to as a "group", and storage is in groups, blocks and subblocks. A group contains four blocks, and a block contains four subblocks, which are the basic storage units manipulated by the program in setting up the memory. Each block of 64K words is four times the RAM size, i.e. a block is 4 × 44 RAMs, and a subblock is one set of 44 RAMs containing 16K words. The subblocks in a block are selected however by address bits 34 and 35. Hence a given subblock contains every fourth word in the address space corresponding to a block, and in access for a quadword, one of the words is in each of the subblocks. This is equivalent to builtin 4-way interleaving. The words are actually read or written in rapid succession, but from the point of view of the system the memory appears to be four words wide.

A subblock contains 16K sets of 44 bits, each set containing a 43-bit memory word. Bits 0–35 are the 36 data bits, bits 36–41 are a 6-bit error correction code (ECC), bit 42 is a parity bit for the 43-bit word, and bit 43 is a spare that can be substituted for any bad bit in the word. Although there is a given physical order in the numbering of the blocks and subblocks, the correspondence of these to parts of the physical address space is determined entirely by the software, which can set up a continuous address space out of bits and snatches all over the array. This assignment, the spare substitution, and many other operations are handled through the S bus diagnostic cycle. Besides the usual communication with the memory controllers, the SBDIAG instruction also allows access in a single step mode wherein all control signals, including even the clock, are simulated by the processor. Information about how to handle these various operations is included in the descriptions of the SBDIAG functions, to which most of the rest of this section is devoted. There are eleven functions, 0–12, to which the MF20 controllers are numbered 10–17. At the end of the section is a discussion on recovering from double bit errors.

Function 0, To Memory

CONTROLLER					CLEAR ERROR FLAGS												
0	1																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

													FUNCTION				
													0	0	0	0	0
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

5 Clear the controller error flags (see bits 0–5 of the return word). Clearing these flags unlocks the error identification information in the return words for this function and functions 2 and 6.

Function 0, Return

CONTROL RAM PARITY ERROR	ERROR CORRECTD	PARTIAL CYCLE	READ ERROR	WRITE	ADDRESS	CONTROLLER STATE		LAST REQUEST				LAST REFERENCE		HIGH ORDER ADDRESS BITS			
				PARITY	ERROR	1	1	RQ0	RQ1	RQ2	RQ3	READ	WRITE				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

LAST ADDRESS OR FIRST ERROR ADDRESS																	
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

0–5 These are error flags for this controller and the storage groups connected to it. Setting a flag latches bits 8–35 on the identification of the current reference.

0 A parity error was encountered in a control RAM (see function 2).

1 An incorrect word was read from storage (bit 3 is 1), but the memory circuits were able to correct it.

2 The controller is hung up, or the processor failed to send data for the write part of a read-pause-write within 8 μs.

3 An incorrect word was read from storage.

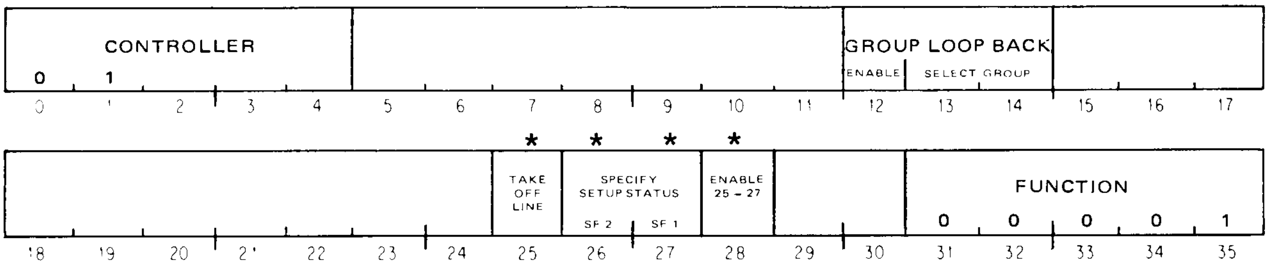
4 A word with even parity was received from the bus, and it was written in storage with a forced 2-bit error.

5 The controller has received an address with even parity over the S bus. The address intended for the reference is available in ERA.

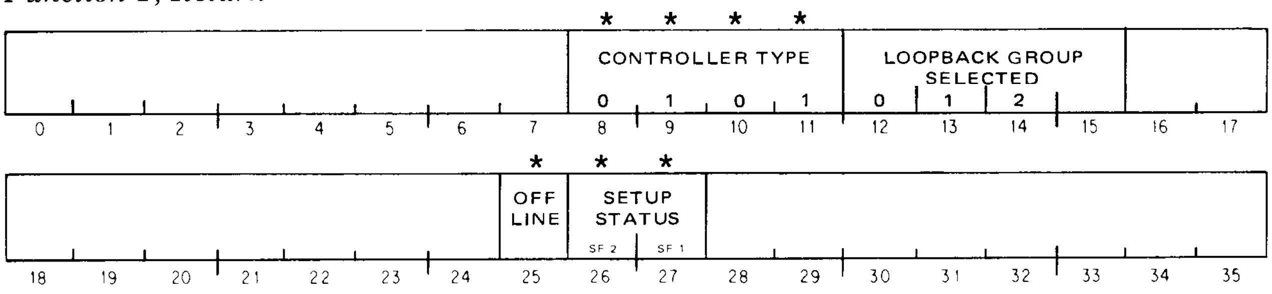
6–7 For compatibility these bits are both 1 to identify the controller state as 4-way interleave. The programmer can regard the memory as four words wide, but physically this is accomplished by interleaving the four subblocks in each block.

8–35 These bits regularly give the address of the most recent reference, whether it was read, write or both, and which words in the quadword were requested. However, when one of the error flags (bits 0–5) is set, the identifying information for that reference is latched into these bits. The bits remain static, even with additional errors, until the program unlatches the register by clearing the error flags (through a function 0 word with a 1 in bit 5). Note that the address is that given for the access even though the error may be in some other word in the quadword (identified in function 2).

Function 1, To Memory



Function 1, Return



8–11 In the return word these bits have the configuration 0101, indicating the controller is an MF20.

12–14 A 1 in bit 12 sent out substitutes a single register for the array boards in the group selected by bits 13 and 14. Hence the processor can exercise all the logic, including the error correction circuits, without affecting the MOS array. A 1 in bits 12–14 in the return indicates which group, if any, is currently selected for loopback operation.

25–28 A 1 in bit 28 sent out enables the contents of bits 25–27 to have an effect; otherwise they are disabled. A 0 in bit 25 places the controller on line; a 1 takes it off line. The same bit in the return word indicates the current state of the controller.

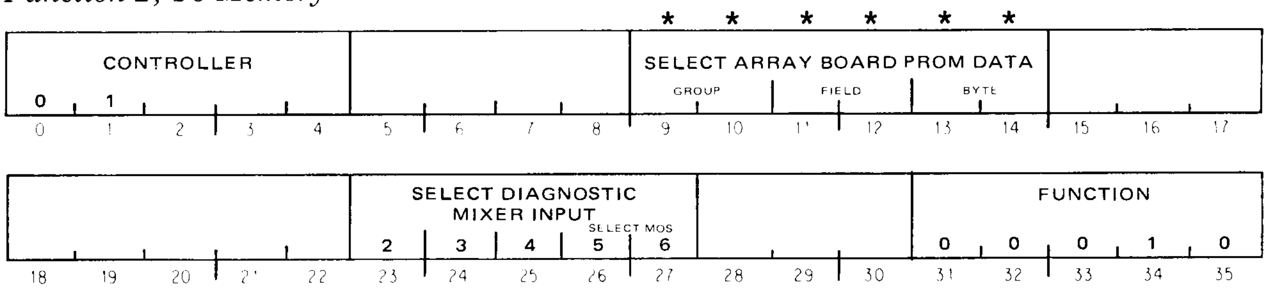
Bits 26 and 27 exercise no hardware control. They simply set up a pair of software flags that are read in the return. At powerup the hardware clears these flags. The meaning of the configurations given for them by the software is as follows.

01 All RAMs except address response are loaded, and any required patching (including bit substitution) is done.

10 Controller configuration is complete.

11 Controller configuration is complete, and TGHA has been run.

Function 2, To Memory

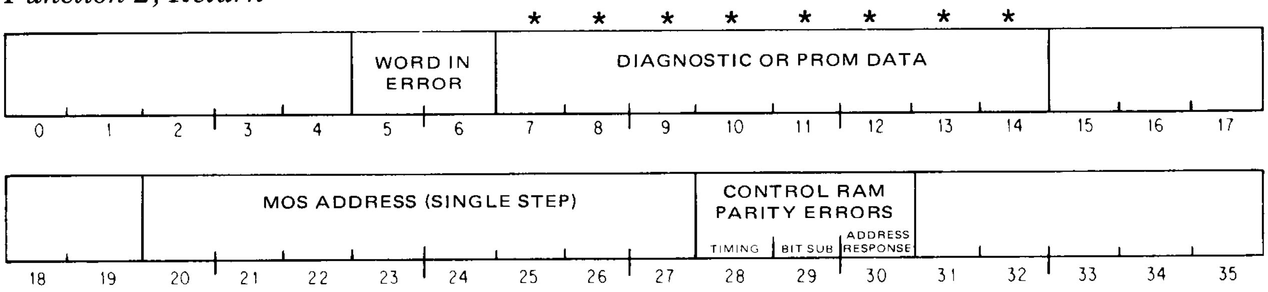


9–14 On each array board is a PROM containing four bytes that give the board serial number and week of manufacture, the size of the MOS RAMs used on the board (always 16K), and the name of the MOS RAM vendor. To read this information, the program can use these bits to select an array board (identifying it by group and field) and to select an individual byte in the PROM on that board.

23–27 The leftmost 1 in this set of bits selects the indicated input to the diagnostic mixer in the controller for supplying a byte of data in the return word. All 0s in these bits selects the return of a byte from an array board PROM as chosen by bits 9–14. Bits 26 and 27 also select various MOS signals for return, even though they may be in use for selecting a mixer input.[2]

[2] Mixer inputs 0, 1 and 7 are used by other functions.

Function 2, Return



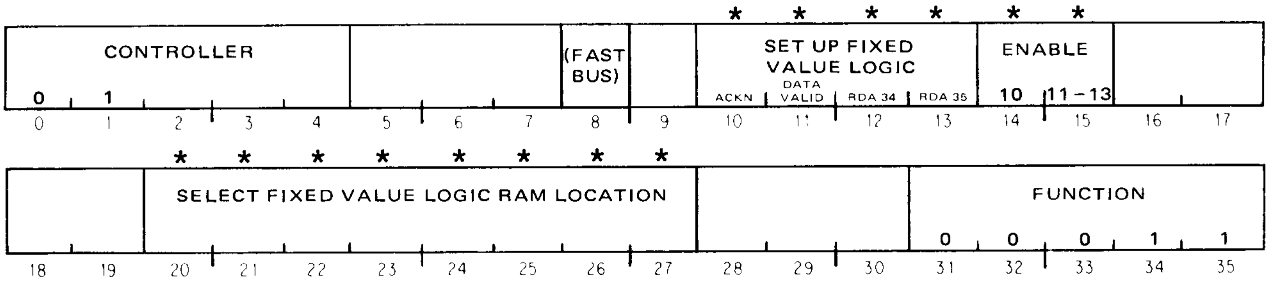
5–6 When an error occurs, the address available through function 0 is that given for the access. The actual word, within the quadword, on which the error occurred is indicated by these bits.

7–14 The data byte from the array board PROM or the diagnostic mixer as selected by the function.

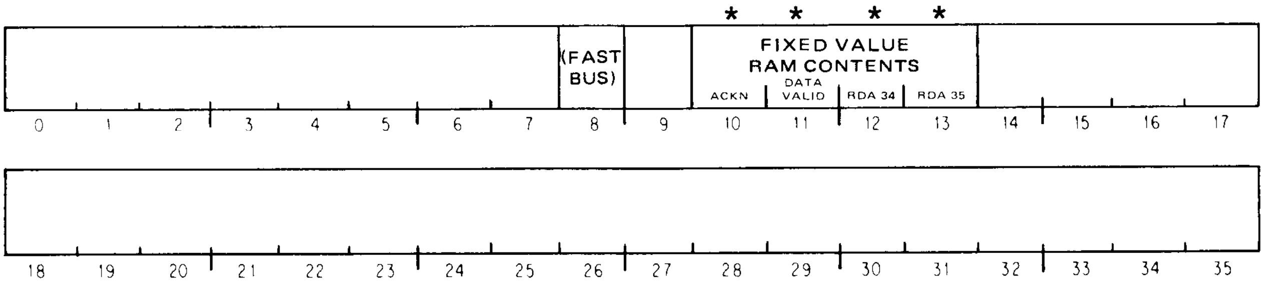
20–27 In single step operation these bits give information about the signals supplied to the array board. Configurations 0–3 in bits 26 and 27 of the function word select respectively the row, column and refresh address, and a group of miscellaneous signals.

28–30 A 1 in one of these bits indicates a parity error in the timing, bit substitution, or address response RAM. An error indication here is always accompanied by a 1 in bit 0 of the function 0 return.

Function 3, To Memory



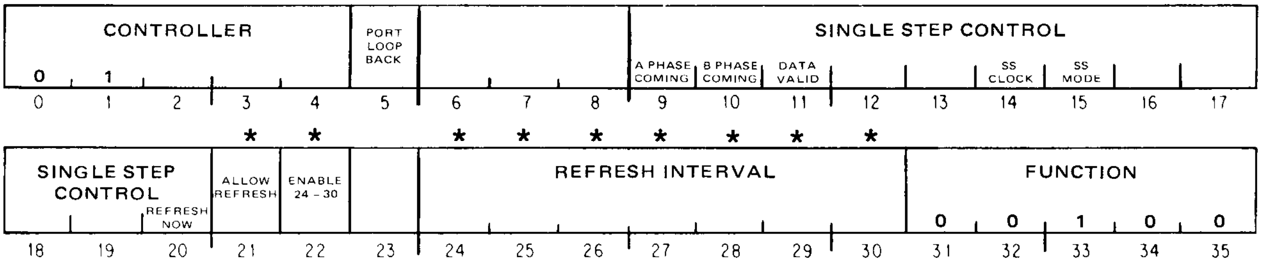
Function 3, Return



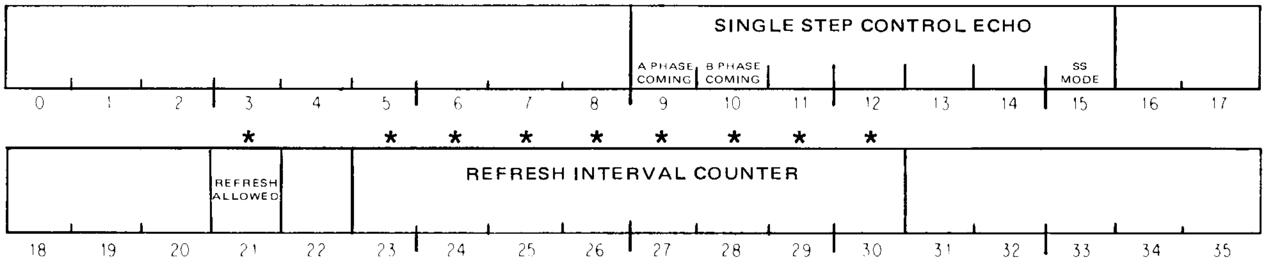
Each controller has a set of four RAMs for supplying logic signals whose values are fixed by the program for the characteristics of the particular system. These RAMs give the acknowledge and data valid signals for the bus, and various configurations of the two least significant read address bits for determining which word to read next during processing of a quadword. Bits 20–27 of the function word select among the 256 locations in the

RAMs loaded from bits 11–13, but the acknowledge RAM (loaded from bit 10) has only 128 locations selected by bits 21–27. There are separate load enables for bit 10 and the remaining bits so the acknowledge RAM can be loaded independently of the others. The only information in the return is the contents of the four RAMs at the location specified by the function.[3]

Function 4, To Memory



Function 4, Return



5 A 1 causes the controller to fail to respond to any information received over the bus except to echo it back. The loopback thus exercises bus reception and transmission in the controller without affecting any circuitry associated with the storage array. Note that even SBDIAG functions are simply echoed back and do not generate a normal return. The condition can be cleared only by an S bus reset given from the front end.

9–20 These bits are used to simulate various memory control signals during single step operation. Note that individual ticks of the clock are produced by giving a sequence of SBDIAGs each with this function with a 1 in bit 14. To keep the controller in single step mode, bit 15 must be 1 in every instance of this function. Several of the bits are echoed back in the return.

21–30 These bits control the refreshing of the MOS data, wherein a 1 in bit 21 enables refreshing. If bit 22 is 1, bits 24–30 select the refresh interval in units of about .5 μs. (The interval is given as a count on a clock that is the 30 MHz clock divided by 16.) The return includes the value given by the function in bit 21, but bits 23–30 supply the current contents of the refresh counter (including an overflow bit that sets when the specified interval is reached).

[3] The fast bus bit is not used and is read as a 0.

Function 5, To Memory

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
CONTROLLER						SINGLE STEP SIMULATED BUS SIGNALS									ENABLE ALL BITS		
0	1					START A PHASE	START B PHASE	RQ 0	RQ 1	RQ 2	RQ 3	READ	WRITE				

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
		SINGLE STEP SIMULATED ADDRESS											FUNCTION				
		14	15	16	17	18	19	20	21		34	35	0	0	1	0	1

Function 5, Return

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
						ROW ADDRESS STROBES (SINGLE STEP)											
						0	1	2	3								

Single step operation exercises all of the storage logic, but a buffer substitutes for the actual MOS array. Hence the single step address supplied by this function includes only the bits that select the array group and the individual word in the set of four (row and column addresses to the array are unneeded). Bits 6–13 simulate signals that would be received by the controller over the bus. Note that this function has an overall enable: none of the other bits have any effect unless bit 15 is 1. This is so the function can be given just to get the information in the return word in which bits 24–27 contain the present values of the row address strobes to the four array boards in the group (meaningful only in single step).

Function 6, To Memory

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
CONTROLLER							DATA FOR ECC COMMANDS 2, 4, 5								ENABLE 7 - 13 FOR 4 & 5		
0	1						32	16	8	4	2	1	P				

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
							ECC COMMAND						FUNCTION				
													0	0	1	1	0

Function 6, Return (except ECC command 2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
							DATA READ BY ECC COMANDS 0, 1, 4, 5, 6										
							32	16	8	4	2	1	P	SPARE			

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
						COLUMN ADDRESS STROBES (SINGLE STEP)											
						0	1	2	3								

Bits 24–27 of the return contain the present values of the column address strobes to the four array boards in the group selected by the single step address given by function 5. Other information in the return (in some or all of bits 7–14 or occupying the entire word) depends on the ECC command in bits 25–27 of the function. These commands are as follows.

0 Read the contents of the ECC register in return bits 7–13. This register regularly receives each ECC code read from storage, but the setting of any function 0 error flag locks it, and it remains static until the error flags are cleared or ECC command 2 is performed.

1 Read the syndrome register in return bits 7–12. This register regularly receives the syndrome produced by comparing each ECC code read from storage with that generated by the ECC circuits for the word read. But the setting of any function 0 error flag locks the register, and it remains static until the error flags are cleared or ECC command 2 is performed. The syndrome actually identifies which bit is in error, provided there is only one, and is zero when there is no error.

2 Using function bits 7–13 as an ECC code (as though read from storage), run through a correction pass on a zero data word, and send back the corrected data word as the return for the function.

3 No-op (may be used to read the column address strobes).

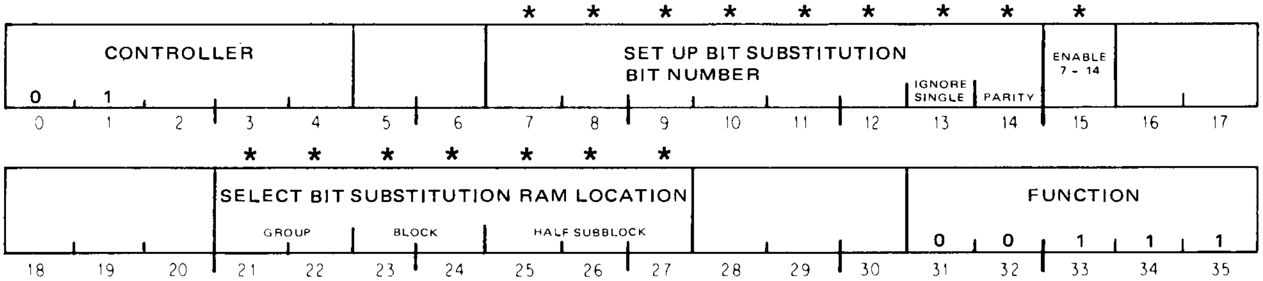
4 If bit 15 out is 1, load bits 7–13 into the complement register; in any event read the contents of this register in return bits 7–13. Before each word is written in storage, bits of its ECC code corresponding to 1s in the complement register are complemented. This of course makes it appear that the word is in error when read — for normal operation the complement register is kept clear.

5 Do ECC command 4, but then cause the very next write to store the contents of the complement register as the ECC code in place of that generated by the ECC circuits.

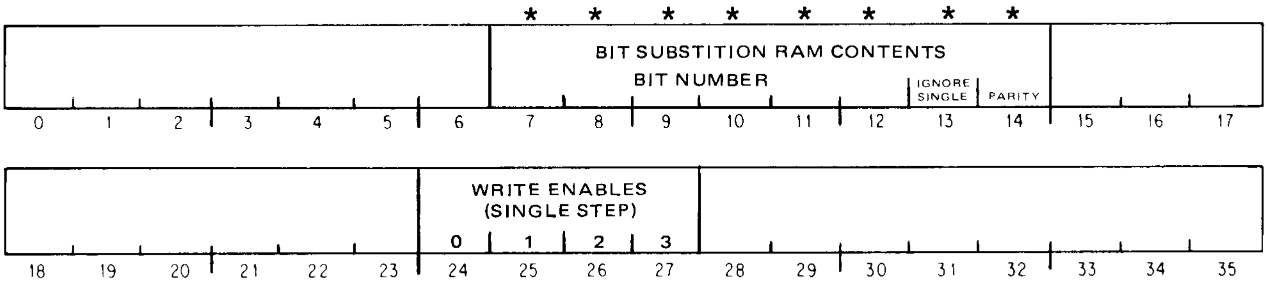
6 Read the saved ECC code and spare bit in return bits 7–14 (see command 7).

7 On the next write, save the generated ECC code and whatever is written in the spare bit for subsequent reading by command 6.

Function 7, To Memory



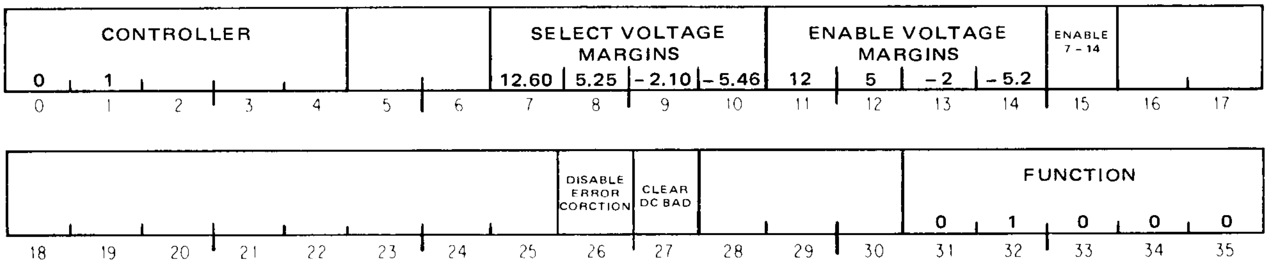
Function 7, Return



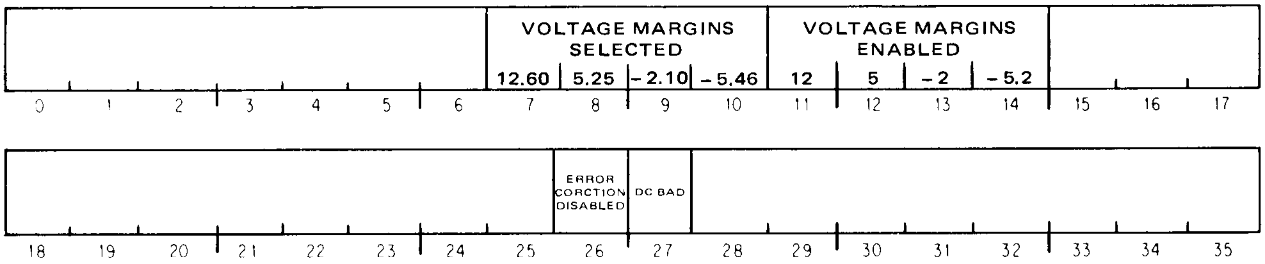
The software can use the spare bit as a substitute for any other bit in a MOS word, but the substitution must be the same for all words in a given 8K half subblock. If bit 15 is 1, the contents of bits 7–14 are loaded into the substitution RAM at the location for the half subblock selected by bits 21–27. In the data for the RAM, bits 7–12 give the number of the MOS-word bit for which the spare is substituted, and a 1 in bit 13 causes the controller to ignore single errors — the controller still corrects such errors, but does not set the related flags (bits 1 and 3 in the function 0 return). Bit 14 must be adjusted to produce odd parity in the contents of the RAM location. Note that a RAM location must be set up for every half subblock used in the MOS array, even if only to substitute the spare bit for itself (bit 43) and to ensure correct parity. Note also that the selections made in bits 21–27 correspond to physical parts of the storage array that respond in a given access, not to the addresses supplied over the S bus (see function 12).

The return word supplies the contents of the RAM location selected by the function, and the present values of the write enable signals to the four array boards in the group selected by the single step address given by function 5.

Function 10, To Memory



Function 10, Return

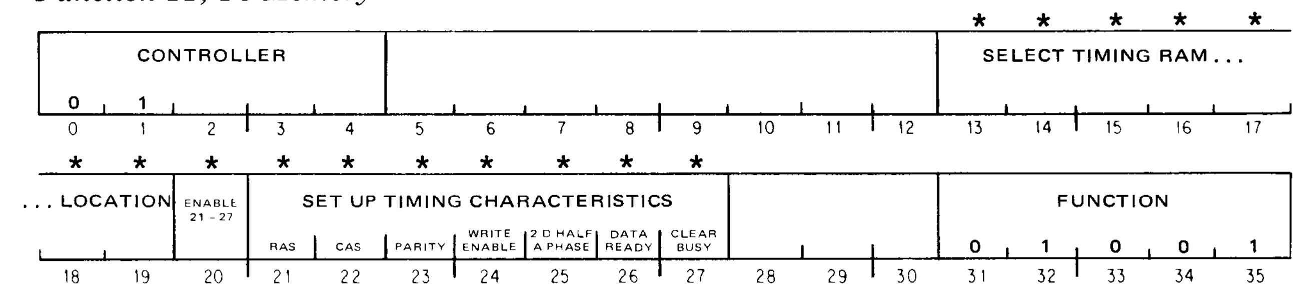


7–15 If bit 15 out is 1, 1s in bits 11–14 select which voltages are to be margined in subsequent operation; and for any voltage running on margin, a 1 or 0 in the corresponding bit among bits 7–10 selects the margin value. E.g. if bit 12 is 1 selecting a margin for 5 volts, a 1 in bit 8 selects 5.25 volts, whereas a 0 selects 4.75. In other words, 1s select high margins in absolute value, 0s select low margins (11.40, 4.75, –1.90, –4.94). Margins enabled and selected are identified in the return word.

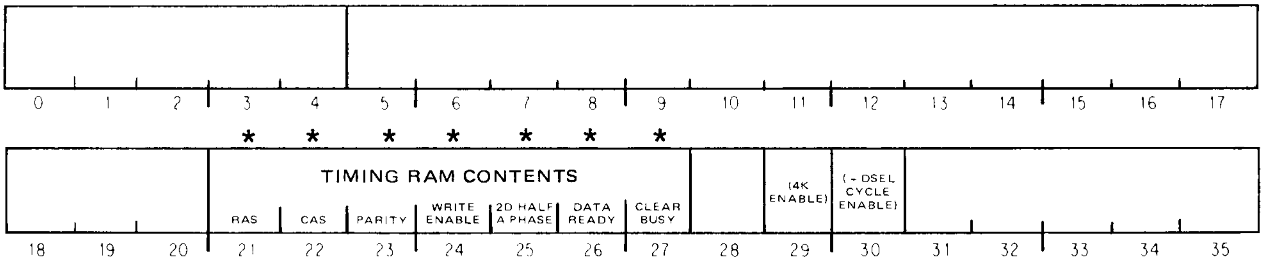
26 A 1 in this bit sent out disables the correction circuits, so that although errors are still detected, any incorrect word is sent back over the bus as is. A 0 reestablishes error correction, and the same bit in the return indicates the current state of the correction logic.

27 A 1 in the return bit indicates bad dc levels in the storage unit, meaning battery backup has failed. A 1 in the bit sent out clears the flag.

Function 11, To Memory



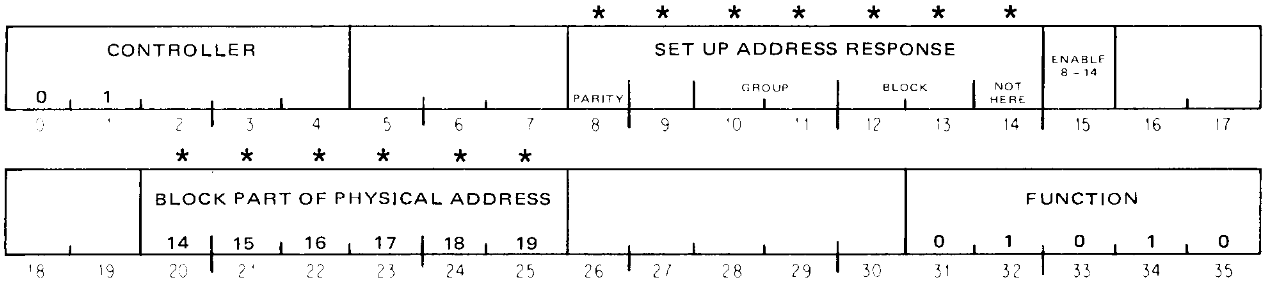
Function 11, Return



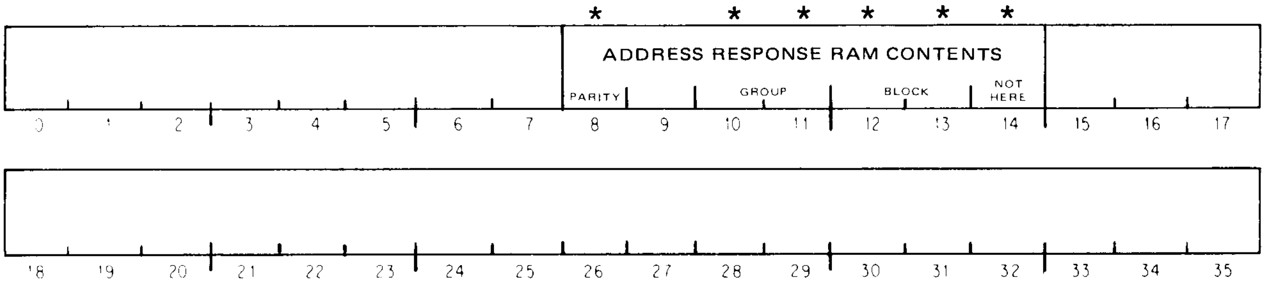
The software must set up the timing characteristics of each controller for the MOS RAMs used with it. To time various signals the clock steps through the locations of a timing RAM, whose bits control the on and off states of those signals. If bit 20 out is 1, the contents of bits 21–27 (adjusted for odd parity) are loaded into the timing RAM at the location specified by bits 13–19. Whether loaded or not, the contents of the selected location are returned in the same bits.[4]

[4] The signals represented by bits 29 and 30 in the return are not used and are read as 0 and 1 respectively.

Function 12, To Memory



Function 12, Return



The high order six bits in a physical address on the S bus specify a particular block of storage, but there is no predetermined relationship between the actual blocks and those bits. Instead when an address appears on the bus, the left six bits are used, in every controller, to access a location in an address response RAM whose contents indicate whether the corresponding block of storage is connected to that controller, and if so, which one it is. The software can create a contiguous address space with any desired correspondence to storage, even deleting blocks to avoid unacceptable error levels. If bit 15 out is 1, the contents of bits 8–14 (adjusted for odd parity) are loaded into the address response RAM at the location specified by bits 20–25, which represent the left six bits of a physical address. If a storage block with this controller is to respond to the given physical block address, the data given for that RAM location must identify it by group and block; otherwise a 1 in bit 14 inhibits any response to that address at this controller. The return word supplies the contents of the selected RAM location.

When a word read from memory has good parity but a nonzero syndrome (see function 6), there must be two bits that are in error. When this happens the memory sends the bad word to the processor along with an indication that the error is uncorrectable. All may not be lost however — it is possible to recover the data provided that at least one of the bad bits is a hard failure. To do this we must know the syndromes for the bits, and these are as follows, where the syndromes are given as three octal digits corresponding to their position in bits 7–12 of the return word for function 6.

Bit	*Syndrome*	*Bit*	*Syndrome*	*Bit*	*Syndrome*
0	014	14	120	28	214
1	024	15	124	29	220
2	030	16	130	30	224
3	034	17	134	31	230
4	044	18	140	32	234
5	050	19	144	33	240
6	054	20	150	34	244
7	060	21	154	35	250
8	064	22	160	36	200
9	070	23	164	37	100
10	074	24	170	38	040
11	104	25	174	39	020
12	110	26	204	40	010
13	114	27	210	41	004

Then follow this procedure.

1. Get the bad word and its syndrome. Since the bad bits can be anywhere, also get the ECC for the word. (Available through function 6).
2. Write the complement of the bad word back into the failed location.
3. Read the complement and take the equivalence of that word with the original bad word (including their ECCs).
4. If the result is zero or contains more than two 1s, nothing can be done.
5. If exactly two bits are set in the equivalence, then exclusive OR it into the original bad word and the result is the correct data.
6. If there is one bit set in the equivalence, then it identifies one of the bad bits in the original word. Get the syndrome for it from the table above and look up that syndrome in the left column of the large table below. Among the entries for that syndrome find the pair of numbers that includes the number of the known bad bit. The other number in the pair identifies the other bad bit. (If no such bit pair can be found, you're out of luck).

Syndromes vs Error Pairs

004	00,40	01,39	02,03	04,38	05,06	07,08	09,10	11,37	12,13	14,15	16,17
	18,19	20,21	22,23	24,25	26,36	27,28	29,30	31,32	33,34	41,42	
010	00,41	01,03	02,39	04,06	05,38	07,09	08,10	11,13	12,37	14,16	15,17
	18,20	19,21	22,24	23,25	26,28	27,36	29,31	30,32	33,35	40,42	
014	00,42	01,02	03,39	04,05	06,38	07,10	08,09	11,12	13,37	14,17	15,16
	18,21	19,20	22,25	23,24	26,27	28,36	29,32	30,31	34,35	40,41	
020	00,03	01,41	02,40	04,08	05,09	06,10	07,38	11,15	12,16	13,17	14,37
	18,22	19,23	20,24	21,25	26,30	27,31	28,32	29,36	39,42		
024	00,02	01,42	03,40	04,07	05,10	06,09	08,38	11,14	12,17	13,16	15,37
	18,23	19,22	20,25	21,24	26,29	27,32	28,31	30,36	39,41		

030	00,01	02,42	03,41	04,10	05,07	06,08	09,38	11,17	12,14	13,15	16,37
	18,24	19,25	20,22	21,23	26,32	27,29	28,30	31,36	39,40		
034	00,39	01,40	02,41	03,42	04,09	05,08	06,07	10,38	11,16	12,15	13,14
	17,37	18,25	19,24	20,23	21,22	26,31	27,30	28,29	32,36		
040	00,06	01,08	02,09	03,10	04,41	05,40	07,39	11,19	12,20	13,21	14,22
	15,23	16,24	17,25	18,37	26,34	27,35	33,36	38,42			
044	00,05	01,07	02,10	03,09	04,42	06,40	08,39	11,18	12,21	13,20	14,23
	15,22	16,25	17,24	19,37	26,33	28,35	34,36	38,41			
050	00,04	01,10	02,07	03,08	05,42	06,41	09,39	11,21	12,18	13,19	14,24
	15,25	16,22	17,23	20,37	27,33	28,34	35,36	38,40			
054	00,38	01,09	02,08	03,07	04,40	05,41	06,42	10,39	11,20	12,19	13,18
	14,25	15,24	16,23	17,22	21,37	26,35	27,34	28,33			
060	00,10	01,04	02,05	03,06	07,42	08,41	09,40	11,23	12,24	13,25	14,18
	15,19	16,20	17,21	22,37	29,33	30,34	31,35	38,39			
064	00,09	01,38	02,06	03,05	04,39	07,41	08,42	10,40	11,22	12,25	13,24
	14,19	15,18	16,21	17,20	23,37	29,34	30,33	32,35			
070	00,08	01,06	02,38	03,04	05,39	07,40	09,42	10,41	11,25	12,22	13,23
	14,20	15,21	16,18	17,19	24,37	29,35	31,33	32,34			
074	00,07	01,05	02,04	03,38	06,39	08,40	09,41	10,42	11,24	12,23	13,22
	14,21	15,20	16,19	17,18	25,37	30,35	31,34	32,33			
100	00,13	01,15	02,16	03,17	04,19	05,20	06,21	07,22	08,23	09,24	10,25
	11,41	12,40	14,39	18,38	37,42						
104	00,12	01,14	02,17	03,16	04,18	05,21	06,20	07,23	08,22	09,25	10,24
	11,42	13,40	15,39	19,38	37,41						
110	00,11	01,17	02,14	03,15	04,21	05,18	06,19	07,24	08,25	09,22	10,23
	12,42	13,41	16,39	20,38	37,40						
114	00,37	01,16	02,15	03,14	04,20	05,19	06,18	07,25	08,24	09,23	10,22
	11,40	12,41	13,42	17,39	21,38						
120	00,17	01,11	02,12	03,13	04,23	05,24	06,25	07,18	08,19	09,20	10,21
	14,42	15,41	16,40	22,38	37,39						
124	00,16	01,37	02,13	03,12	04,22	05,25	06,24	07,19	08,18	09,21	10,20
	11,39	14,41	15,42	17,40	23,38						
130	00,15	01,13	02,37	03,11	04,25	05,22	06,23	07,20	08,21	09,18	10,19
	12,39	14,40	16,42	17,41	24,38						
134	00,14	01,12	02,11	03,37	04,24	05,23	06,22	07,21	08,20	09,19	10,18
	13,39	15,40	16,41	17,42	25,38						
140	00,21	01,23	02,24	03,25	04,11	05,12	06,13	07,14	08,15	09,16	10,17
	18,42	19,41	20,40	22,39	37,38						
144	00,20	01,22	02,25	03,24	04,37	05,13	06,12	07,15	08,14	09,17	10,16
	11,38	18,41	19,42	21,40	23,39						
150	00,19	01,25	02,22	03,23	04,13	05,37	06,11	07,16	08,17	09,14	10,15
	12,38	18,40	20,42	21,41	24,39						
154	00,18	01,24	02,23	03,22	04,12	05,11	06,37	07,17	08,16	09,15	10,14
	13,38	19,40	20,41	21,42	25,39						
160	00,25	01,19	02,20	03,21	04,15	05,16	06,17	07,37	08,11	09,12	10,13
	14,38	18,39	22,42	23,41	24,40						
164	00,24	01,18	02,21	03,20	04,14	05,17	06,16	07,11	08,37	09,13	10,12
	15,38	19,39	22,41	23,42	25,40						
170	00,23	01,21	02,18	03,19	04,17	05,14	06,15	07,12	08,13	09,37	10,11
	16,38	20,39	22,40	24,42	25,41						
174	00,22	01,20	02,19	03,18	04,16	05,15	06,14	07,13	08,12	09,11	10,37
	17,38	21,39	23,40	24,41	25,42						
200	00,28	01,30	02,31	03,32	04,34	05,35	26,41	27,40	29,39	33,38	36,42
204	00,27	01,29	02,32	03,31	04,33	06,35	26,42	28,40	30,39	34,38	36,41
210	00,26	01,32	02,29	03,30	05,33	06,34	27,42	28,41	31,39	35,38	36,40
214	00,36	01,31	02,30	03,29	04,35	05,34	06,33	26,40	27,41	28,42	32,39
220	00,32	01,26	02,27	03,28	07,33	08,34	09,35	29,42	30,41	31,40	36,39
224	00,31	01,36	02,28	03,27	07,34	08,33	10,35	26,39	29,41	30,42	32,40
230	00,30	01,28	02,36	03,26	07,35	09,33	10,34	27,39	29,40	31,42	32,41

234	00,29	01,27	02,26	03,36	08,35	09,34	10,33	28,39	30,40	31,41	32,42
240	04,26	05,27	06,28	07,29	08,30	09,31	10,32	33,42	34,41	35,40	36,38
244	00,35	04,36	05,28	06,27	07,30	08,29	09,32	10,31	26,38	33,41	34,42
250	00,34	04,28	05,36	06,26	07,31	08,32	09,29	10,30	27,38	33,40	35,42
254	00,33	04,27	05,26	06,36	07,32	08,31	09,30	10,29	28,38	34,40	35,41
260	01,34	02,35	04,30	05,31	06,32	07,36	08,26	09,27	10,28	29,38	33,39
264	01,33	03,35	04,29	05,32	06,31	07,26	08,36	09,28	10,27	30,38	34,39
270	02,33	03,34	04,32	05,29	06,30	07,27	08,28	09,36	10,26	31,38	35,39
274	01,35	02,34	03,33	04,31	05,30	06,29	07,28	08,27	09,26	10,36	32,38
300	11,26	12,27	13,28	14,29	15,30	16,31	17,32	18,33	19,34	20,35	36,37
304	11,36	12,28	13,27	14,30	15,29	16,32	17,31	18,34	19,33	21,35	26,37
310	11,28	12,36	13,26	14,31	15,32	16,29	17,30	18,35	20,33	21,34	27,37
314	11,27	12,26	13,36	14,32	15,31	16,30	17,29	19,35	20,34	21,33	28,37
320	11,30	12,31	13,32	14,36	15,26	16,27	17,28	22,33	23,34	24,35	29,37
324	11,29	12,32	13,31	14,26	15,36	16,28	17,27	22,34	23,33	25,35	30,37
330	11,32	12,29	13,30	14,27	15,28	16,36	17,26	22,35	24,33	25,34	31,37
334	11,31	12,30	13,29	14,28	15,27	16,26	17,36	23,35	24,34	25,33	32,37
340	11,34	12,35	18,36	19,26	20,27	21,28	22,29	23,30	24,31	25,32	33,37
344	11,33	13,35	18,26	19,36	20,28	21,27	22,30	23,29	24,32	25,31	34,37
350	12,33	13,34	18,27	19,28	20,36	21,26	22,31	23,32	24,29	25,30	35,37
354	11,35	12,34	13,33	18,28	19,27	20,26	21,36	22,32	23,31	24,30	25,29
360	14,33	15,34	16,35	18,29	19,30	20,31	21,32	22,36	23,26	24,27	25,28
364	14,34	15,33	17,35	18,30	19,29	20,32	21,31	22,26	23,36	24,28	25,27
370	14,35	16,33	17,34	18,31	19,32	20,29	21,30	22,27	23,28	24,36	25,26
374	15,35	16,34	17,33	18,32	19,31	20,30	21,29	22,28	23,27	24,26	25,36

Index

DECsystem-10
DECSYSTEM-20
Processor Reference Manual
AD-H391A-T1

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify) ____________

Name ____________ Date ____________

Organization ____________ Telephone ____________

Street ____________

City ____________ State ________ Zip Code or Country ________

Do Not Tear – Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1–2/L12
MARLBOROUGH, MASSACHUSETTS 01752

Do Not Tear – Fold Here and Tape

Cut Along Dotted Line