+---------------------------+
! ! ! ! ! ! ! ! ! ! !
! d ! i ! g ! i ! t ! a ! l !
! ! ! ! ! ! ! ! ! ! !
+---------------------------+

interoffice

memorandum

To: KD10 review list

Date: 18 Jul 83
From: Mike Uhler
Dept: L.S.E.G.
DTN: (8-)231-6448
Loc/Mail stop: MR01-1/L26
Net mail: UHLER at IO

Subject: KD10 Architectural Design Specification

This spec describes the architectural design of the KD10 system. It also covers the hardware, microcode, software, and diagnostic changes required to build the system.

The KD10 design, as described in this memo, assumes certain priorities about the goals of the project. These goals, in priority order, are as follows:

- Runs TOPS-10 and TOPS-20 with minimum software changes.
- Time-to-market of 18 months or less.
- Is compatible with existing PDP-10 processors, but includes most architectural changes approved by the Architecture Committee during the course of the Jupiter project.
- Uses existing hardware, software, and microcode designs whenever possible in an attempt to use minimum engineering resources.
- Performance in the range 0.3 to 0.5 times a KL10, with most timesharing loads in the upper part of the range.
- Low manufacturing and sale cost.
- Low risk, high reliability technology.

Certain parts of the design of the machine are, in all probability, sub-optimal. In most cases, this is the result of decisions made with time-to-market in mind. We've tried to indicate such decisions in the text, with possible alternatives.

It is assumed that the reader has also read "A Low-Cost, Space-Efficient PDP-10 Technical Proposal", dated 16 Jun 83, by Pat Sullivan and Mike Uhler.

Table of Contents

CHAPTER 4 EXTENDED ADDRESSING

CHAPTER 1

SYSTEM DESCRIPTION

This chapter provides a functional overview of the components of the KD10 system.

1.1 Functional description

The system described by this spec contains a single-CPU PDP-10 processor capable of supporting 20-32 users. The performance goal is 0.3 to 0.5 times a KL10 model B. Disk storage is provided by an RA60 disk drive with provision for 1 to 3 additional RA60/RA81 drives. Synchronous and asynchronous communication is initially provided by traditional line interfaces connected to a Unibus. This will ultimately be upgraded to the use the NI with additional hardware and software work.

The system is built around a PDP-10 processor with full 30-bit extended addressing support. The processor is an upgraded version of the KS10 (2020) design, which we call the KD10, and it uses as much of the existing KS10 design as possible. The processor also contains a 2K, one-word block size, one-way associative write-through cache and a 1K, one-word block size, one-way associative translation buffer.

The processor fits on two extended hex modules and uses the AMD2901 family and Schottky TTL MSI parts as did the KS10, although the parts are significantly faster than those used on the KS10 design.

The KS10 design was chosen as the starting point for the KD10 design because the internal structure of the machine was very simple. We felt we could exploit this fact in order to minimize time-to-market. The functional differences between the KS10 design and the KD10 design are as follows:

- The KD10 CPU contains full 30-bit extended addressing support.
- The KD10 CPU and memory system can address up to 4 MWords of physical memory.

- The KD10 can support up to 4 I/O adapters and the ultimate I/O interconnects will be to the CI, NI, and SI buses.

- The KD10 design exploits the advances in technology made since the KS10 was designed.

The memory subsystem contains a memory controller module and one to four 1 Mword memory modules for a total memory capacity of 4 Mwords. The memory modules use 256K MOS RAM parts and include parity and ECC bits to allow single error correction and double error detection.

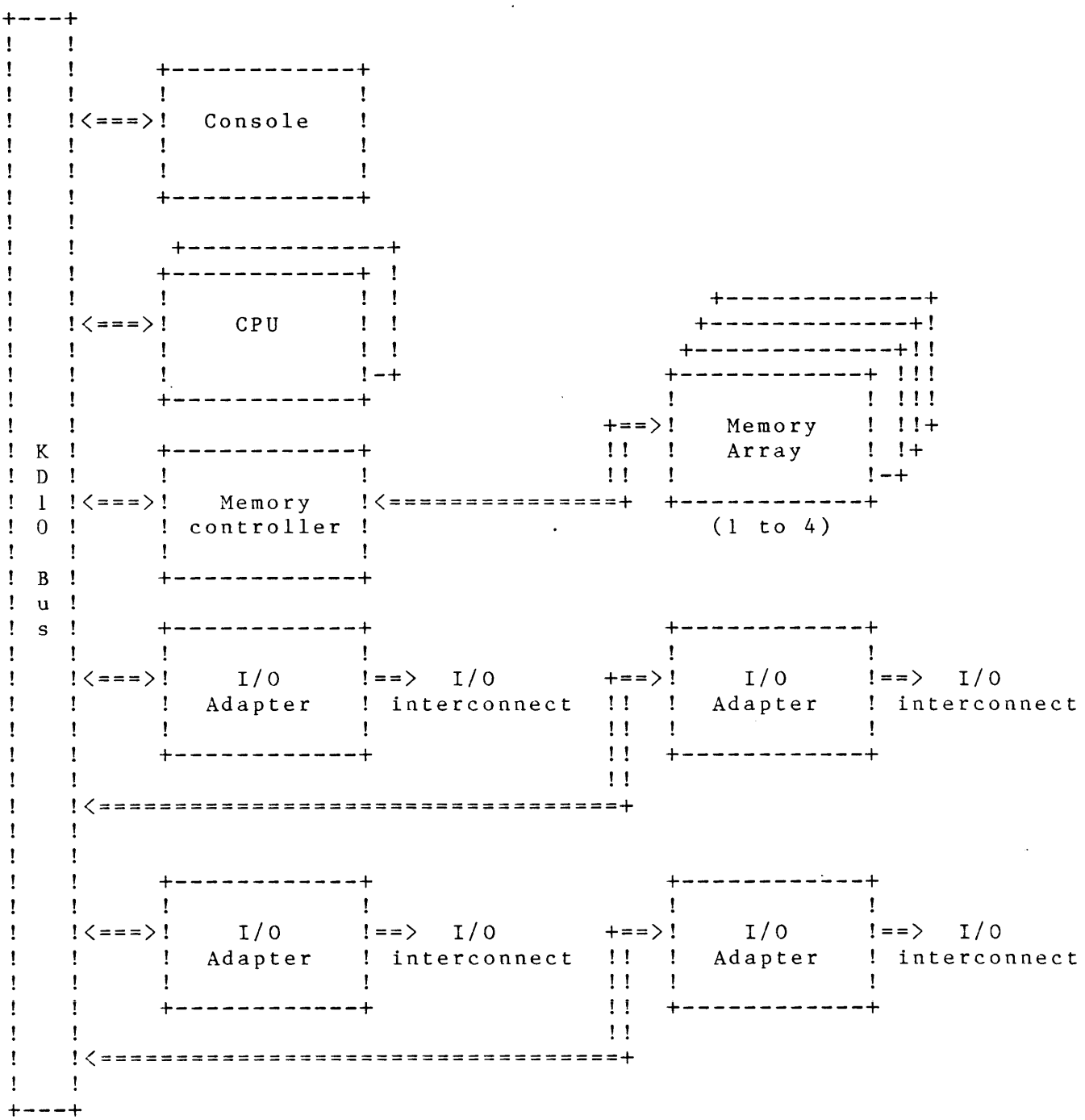
The I/O subsystem consists of 2 to 4 I/O adapters where, in the minimum configuration, one adapter is used for disks and the other is used for all other I/O functions.

The console is a single extended hex module containing an 8080 microprocessor with console code in PROM and RAM. It is an almost exact copy of the KS10 console module, modified only where absolutely necessary.

The main interconnect between the CPU, memory, and the I/O adapters is the KD10 bus which provides a control and data path between the system components. Bus operation is identical to that of the KS10 bus used in the KS10 processor.

1.2 System interconnects

The following diagram shows the major interconnections between the components of the system:



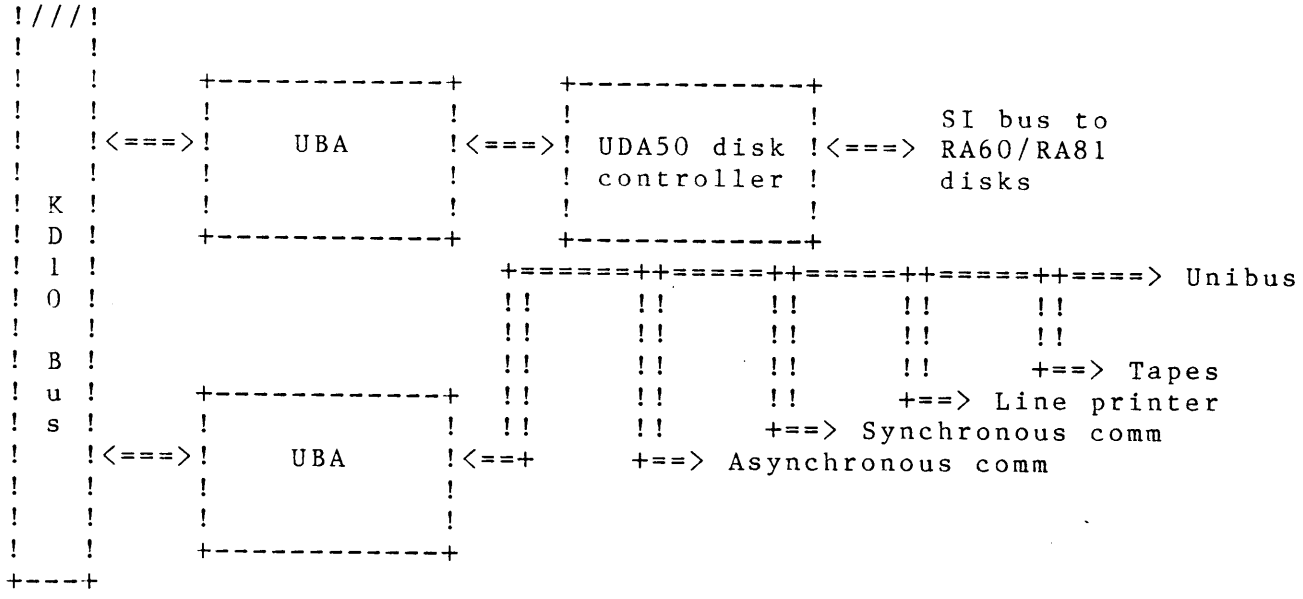
Comparison of this diagram with the equivalent diagram for the KS10 reveals that the basic structure is almost identical. This was done intentionally in an attempt to limit the number and extent of changes required. Most changes are upgrades to the existing KS10 design (with the exception of the addition of extended addressing, which is a bit

more than a simple upgrade).

However, we would expect that the design would be incrementally improved after FCS and ultimately result in a more optimal design. For example, the KD10 bus structure and write-through cache lend themselves well to multi-processing strategies and it might be possible to extend the design to support symmetric multiprocessing quite easily.

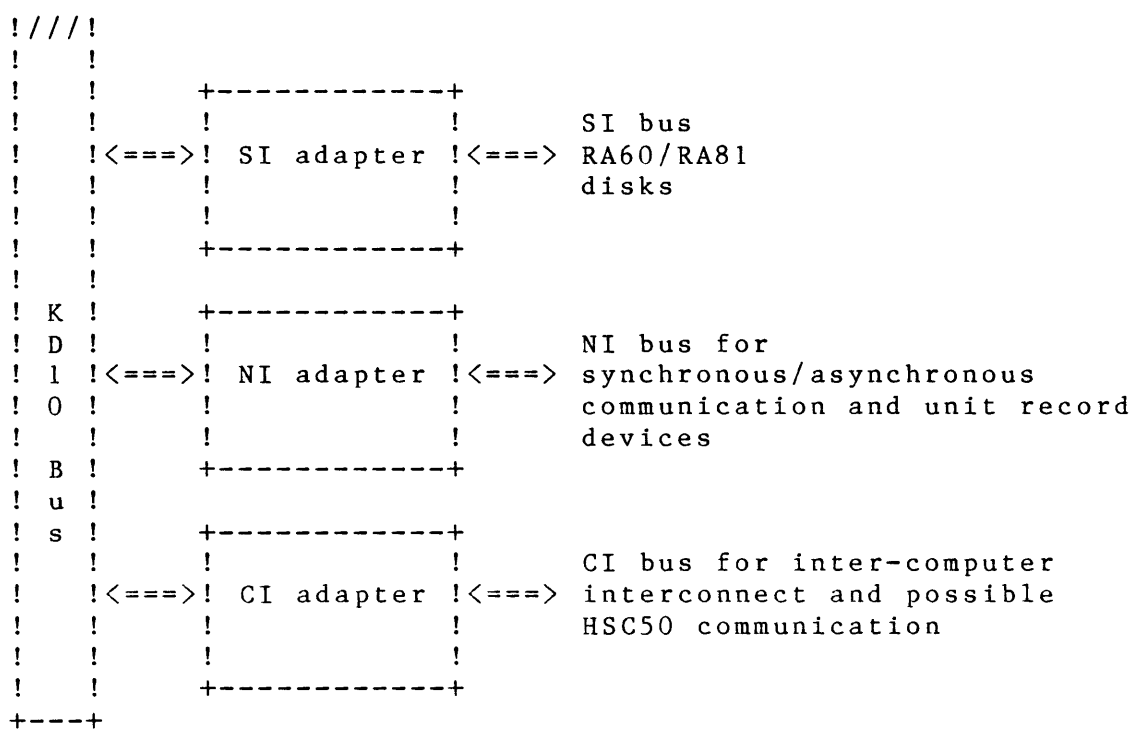
1.3 I/O structure

The FCS I/O structure uses two Unibus Adapters for all I/O. Disk I/O is through a UBA to a UDA50 disk adapter and from there to the new corporate disk products (RA81, RA60, etc). A second UBA will be used for all other I/O including asynchronous and synchronous communication, a line printer, tape drives, etc. The FCS machine will have an I/O structure that looks like:



Using such a structure for the FCS machine avoids having to engineer complex adapters and takes advantage of some of the I/O software already developed for the KS10.

Because this structure isn't particularly attractive in terms of taking advantage of new corporate buses, several new I/O adapter types should be developed after FCS. The ultimate I/O structure might include CI, NI, and SI adapters and look like:



This leaves a spare adapter slot for use in special applications or to increase I/O bandwidth.

CHAPTER 2

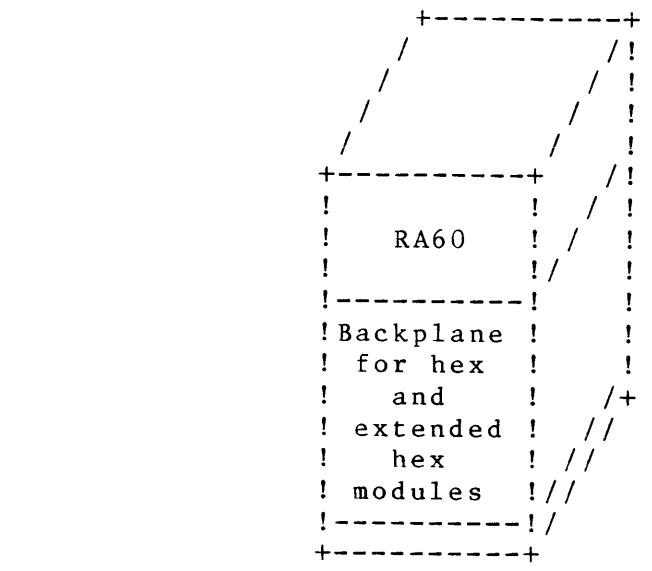
PACKAGING AND MECHANICAL CONSIDERATIONS

This chapter discusses the proposed packaging scheme, including some of the mechanical considerations.

2.1 Packaging

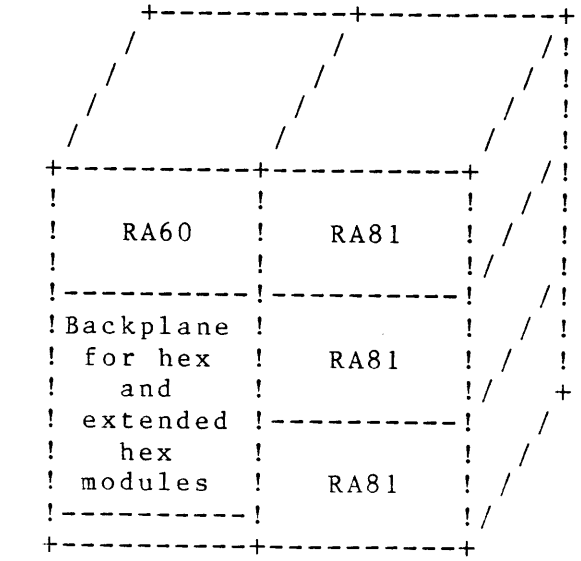
Space efficiency of a system has become increasingly important over the last few years. Because of the size of KL10 systems, customers have discovered that they are limited by the amount of floor space necessary to support their computing needs. In addition to price/performance ratios, today's customers are also using MIPS/square foot as a figure of merit in considering systems. We were very aware of this factor when considering proposed packages for the KD10.

The proposed package for the KD10 system uses an RA81 cabinet to house an RA60 removable media disk, the CPU, memory, I/O adapters, and other I/O-related modules. This package looks like:



Possible package using
RA81 cabinet

Additional disk storage is obtained by placing another RA81 cabinet next to the KD10 cabinet and installing one to three RA81/RA60 drives in the cabinet. This package looks like:



Possible package with
additional RA81 cabinet

Total disk capacity with 3 additional RA81 drives is 1.5 giga-bytes.

2.2 Mechanical considerations

The CPU and memory modules used in the KS10 design are primarily extended Hex modules. To minimize the changes necessary, we've assumed that the KD10 module set would also be primarily extended Hex. Other I/O-related modules are either hex or quad modules.

Preliminary module counts are as follows:

Type	Count	Type
CPU	2	Ext. Hex
Memory controller	1	Ext. Hex
Memory arrays	4	Ext. Hex
I/O adapters	4	Ext. Hex
UDA50	2	Hex
Comm	2-3	Hex

This yields 12 extended Hex modules and 4 to 5 hex modules as a preliminary count.

Our preliminary investigations indicate that a modified BA11K drawer is the best bet for module connections. The BA11K must be modified because it will not handle extended Hex modules. The advantage of the BA11K over a KS10-style backpanel is that the entire module set can be placed in one card cage. In addition, the BA11K contains its own power supply which removes the need for a separate power supply in the cabinet.

CHAPTER 3

SOFTWARE ENVIRONMENT

This chapter describes the software environment provided by the KD10 hardware and microcode.

3.1 The instruction set

The instruction set supported by the KD10 CPU contains most instructions that have been approved by the Architecture Committee. A map of the opcodes follows:

	0	1	2	3	4	5	6	7
000	UUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO
010	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO
020	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO
030	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO
040	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO
050	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO
060	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO
070	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO
	0	1	2	3	4	5	6	7
100	UUO	UUO	GFAD	GFSB	JSYS	ADJSP	GFMP	GFDV
110	DFAD	DFSB	DFMP	DFDV	DADD	DSUB	DMUL	DDIV
120	DMOVE	DMOVN	FIX	EXTEND	DMOVEM	DMOVNM	FIXR	FLTR
130	UUO	UUO	FSC	IBP	ILDB	LDB	IDPB	DPB
140	FAD	UUO	FADM	FADB	FADR	FADRI	FADRM	FADRB
150	FSB	UUO	FSBM	FSBB	FSBR	FSBRI	FSBRM	FSBRB
160	FMP	UUO	FMPM	FMPB	FMPR	FMPRI	FMPRM	FMPRB
170	FDV	UUO	FDVM	FDVB	FDVR	FDVRI	FDVRM	FDVRB
	0	1	2	3	4	5	6	7
200	MOVE	MOVEI	MOVEM	MOVES	MOVS	MOVSI	MOVSM	MOVSS
210	MOVN	MOVNI	MOVNM	MOVNS	MOVM	MOVMI	MOVMM	MOVMS
220	IMUL	IMULI	IMULM	IMULB	MUL	MULI	MULM	MULB
230	IDIV	IDIVI	IDIVM	IDIVB	DIV	DIVI	DIVM	DIVB
240	ASH	ROT	LSH	JFFO	ASHC	ROTC	LSHC	UUO
250	EXCH	BLT	AOBJP	AOBJN	JRST	JFCL	XCT	MAP
260	PUSHJ	PUSH	POP	POPJ	JSR	JSP	JSA	JRA
270	ADD	ADDI	ADDM	ADDB	SUB	SUBI	SUBM	SUBB

	0	1	2	3	4	5	6	7
300	CAI	CAIL	CAIE	CAILE	CAIA	CAIGE	CAIN	CAIG
310	CAM	CAML	CAME	CAMLE	CAMA	CAMGE	CAMN	CAMG
320	JUMP	JUMPL	JUMPE	JUMPLE	JUMPA	JUMPGE	JUMPN	JUMPG
330	SKIP	SKIPL	SKIPE	SKIPLE	SKIPA	SKIPGE	SKIPN	SKIPG
340	AOJ	AOJL	AOJE	AOJLE	AOJA	AOJGE	AOJN	AOJG
350	AOS	AOSL	AOSE	AOSLE	AOSA	AOSGE	AOSN	AOSG
360	SOJ	SOJL	SOJE	SOJLE	SOJA	SOJGE	SOJN	SOJG
370	SOS	SOSL	SOSE	SOSLE	SOSA	SOSGE	SOSN	SOSG
	0	1	2	3	4	5	6	7
400	SETZ	SETZI	SETZM	SETZB	AND	ANDI	ANDM	ANDB
410	ANDCA	ANDCAI	ANDCAM	ANDCAB	SETM	XMOVEI	SETMM	SETMB
420	ANDCM	ANDCMI	ANDCMM	ANDCMB	SETA	SETAI	SETAM	SETAB
430	XOR	XORI	XORM	XORB	IOR	IORI	IORM	IORB
440	ANDCB	ANDCBI	ANDCBM	ANDCBB	EQV	EQVI	EQVM	EQVB
450	SETCA	SETCAI	SETCAM	SETCAB	ORCA	ORCAI	ORCAM	ORCAB
460	SETCM	SETCMI	SETCMM	SETCMB	ORCM	ORCMI	ORCMM	ORCMB
470	ORCB	ORCBI	ORCBM	ORCBB	SETO	SETOI	SETOM	SETOB
	0	1	2	3	4	5	6	7
500	HLL	XHLLI	HLLM	HLLS	HRL	HRLI	HRLM	HRLS
510	HLLZ	HLLZI	HLLZM	HLLZS	HRLZ	HRLZI	HRLZM	HRLZS
520	HLLO	HLLOI	HLLOM	HLLOS	HRLO	HRLOI	HRLOM	HRLOS
530	HLLE	HLLEI	HLLEM	HLLES	HRLE	HRLEI	HRLEM	HRLES
540	HRR	HRRI	HRRM	HRRS	HLR	HLRI	HLRM	HLRS
550	HRRZ	HRRZI	HRRZM	HRRZS	HLRZ	HLRZI	HLRZM	HLRZS
560	HRRO	HRROI	HRROM	HRROS	HLRO	HLROI	HLROM	HLROS
570	HRRE	HRREI	HRREM	HRRES	HLRE	HLREI	HLREM	HLRES
	0	1	2	3	4	5	6	7
600	TRN	TLN	TRNE	TLNE	TRNA	TLNA	TRNN	TLNN
610	TDN	TSN	TDNE	TSNE	TDNA	TSNA	TDNN	TSNN
620	TRZ	TLZ	TRZE	TLZE	TRZA	TLZA	TRZN	TLZN
630	TDZ	TSZ	TDZE	TSZE	TDZA	TSZA	TDZN	TSZN
640	TRC	TLC	TRCE	TLCE	TRCA	TLCA	TRCN	TLCN
650	TDC	TSC	TDCE	TSCE	TDCA	TSCA	TDCN	TSCN
660	TRO	TLO	TROE	TLOE	TROA	TLOA	TRON	TLON
670	TDO	TSO	TDOE	TSOE	TDOA	TSOA	TDON	TSON
	0	1	2	3	4	5	6	7
700	APR0	APR1	APR2	UUO	UMOVE	UMOVEM	PMOVE	PMOVEM
710	UUO	UUO	UUO	UUO	UUO	UUO	LDPAC	STPAC
720	TIOE	TION	RDIO	WRIO	BSIO	BCIO	UUO	UUO
730	TIOEB	TIONB	RDIOB	WRIOB	BSIOB	BCIOB	UUO	UUO
740	PUSHM	POPM	PUSHI	UUO	UUO	UUO	UUO	UUO
750	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO
760	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO
770	UUO	UUO	UUO	UUO	UUO	UUO	UUO	UUO

The opcode map for EXTEND instructions is given below. Note that the missing EXTEND opcodes trap as a UUO.

	0	1	2	3	4	5	6	7
000	UUO	CMPSL	CMPSE	CMPSLE	EDIT	CMPSGE	CMPSN	CMPSG
010	CVTDBO	CVTDBT	CVTBDO	CVTBDT	MOVSO	MOVST	MOVSLJ	MOVSRJ
020	XBLT	GSNGL	GDBLE	GDFIX	GFIX	GDFIXR	GFIXR	DGFLTR
030	GFLTR	GFSC	UUO	UUO	UUO	UUO	UUO	UUO

3.2 The user mode environment

The user mode environment is very similar to that proposed for the KC10 processor. Because the user mode environment is not identical to the KL10, users may notice the following differences between KD10 and KL10 processors:

- One-word global byte pointers are legal in all sections (including section zero).
- XBLT is legal in all sections (including section zero).
- Certain of the privileged JRST functions which were legal with User I/O mode on the KL10 are legal only in kernal mode on the KD10.
- SFM is legal in all sections (including section zero).
- XJRST has been added.
- The G-floating instructions have been added.

NOTE

Due to time-to-market considerations, the G-floating conversion instructions under EXTEND may trap to the monitor for emulation in the FCS machine.

- When rounding is applicable to an instructions, the KD10 uses unbiased rounding instead of biased rounding.

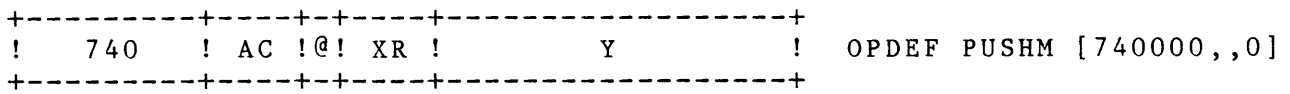
NOTE

Due to time-to-market considerations, the FCS machine may implement biased rounding.

3.2.1 New user mode instructions

This section describes the new user mode instructions that are supported by the KD10.

3.2.1.1 PUSHM - Push multiple ACs



Push ACs onto the stack addressed by the stack pointer in AC as directed by the word in location E. The format of this word is as follows:

0-17	Ignored
18-19	Function code. See below.
20-35	Bit mask of ACs to push. Bit 20 corresponds to AC 0; bit 35 corresponds to AC 17.

Bits 18 and 19 are interpreted as a function code, as follows:

Code	Result
0	Push ACs indicated by bits 20-35 on the stack.
1	Reserved. If this function code is used, the instruction will generate a page fail trap to the monitor.
2	Push ACs indicated by bits 20-35 on the stack. When this is complete, push the full 30-bit effective address of the instruction on the stack.
3	Push ACs indicated by bits 20-35 on the stack. When this is complete, push the full 30-bit effective address of the instruction on the stack.

The ACs corresponding to the bit mask in bits 20-35 are pushed onto the stack beginning with AC 0 and continuing through AC 17. If the stack pointer is designated as one of the ACs to be pushed, the value pushed onto the stack is the contents of the stack pointer at the start of the instruction (before it is incremented).

When all designated ACs are pushed onto the stack, the effective address of the instruction (including section number) is also pushed onto the stack if the function code is 2 or 3.

If the stack pointer overflows during the process of pushing ACs or E, the word which caused the stack pointer to overflow is pushed on the stack (into the location one past the end of stack) and the instruction aborts, setting the trap 2 flag.

If the instruction fails to complete successfully (stack overflow, page fail, interrupt, etc.), the stack pointer in AC is left unchanged (the value that it had at the beginning of the instruction). Note

that in this event, some locations on the stack following the stack pointer may have been modified.

This instruction is identical to the KC10 PUSHM instruction and is not currently implemented by the KS10 microcode.

NOTE

Due to time-to-market considerations, this instruction may not be implemented by the microcode for the FCS machine.

3.2.1.2 POPM - Pop multiple ACs

```
+---------+----+-+----+-----------------+
!   741   ! AC !@! XR !        Y        !  OPDEF POPM [741000,,0]
+---------+----+-+----+-----------------+
```

Pop ACs from the stack addressed by the stack pointer in AC as directed by bits 18-35 of the effective address. The format of these bits is as follows:

18-19	Function code. See below.
20-35	Bit mask of ACs to pop. Bit 20 corresponds to AC 0; bit 35 corresponds to AC 17.

Bits 18 and 19 are interpreted as a function code, as follows:

Code	Result
0	Pop ACs indicated by bits 20-35 from the stack.
1	Reserved. If this function code is used, the instruction will generate a page fail trap to the monitor.
2	Pop ACs indicated by bits 20-35 from the stack. When this is complete, pop another stack location and take the next instruction from the address specified by the contents of the additional stack location. The effect is to perform a POPJ after all ACs have been popped.
3	Pop ACs indicated by bits 20-35 from the stack. When this is complete, pop another stack location and take the next instruction from the address specified by the contents+1 of the additional stack location (i.e., increment the contents). The effect is to perform the instruction sequence: AOS 0(P) POPJ P, after all ACs have been popped.

The ACs corresponding to the bit mask in bits 20-35 are popped from the stack beginning with AC 17 and continuing through AC 0. If the stack pointer is designated as one of the ACs to be popped, the results of the operation are undefined.

When all designated ACs are popped from the stack, a non-skip or skip return may be performed if the function code is 2 or 3.

If the stack pointer underflows during the process of popping ACs or performing the optional return, the instruction is aborted and the

trap 2 flag is set. In the case where the underflow is detected while performing the return, PC is changed to the value from the stack location before the trap occurs.

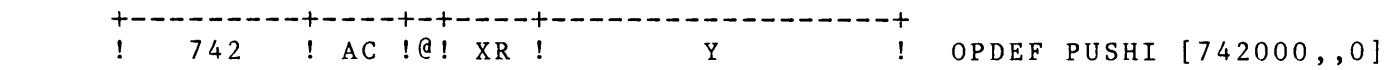
If the instruction fails to complete successfully (stack underflow, page fail, interrupt, etc.), the stack pointer in AC is left unchanged (the value that it had at the beginning of the instruction). Note that in this event, some ACs may have been modified.

This instruction is identical to the KC10 POPM instruction and is not currently implemented by the KS10 microcode.

NOTE

Due to time-to-market considerations, this instruction may not be implemented by the microcode for the FCS machine.

3.2.1.3 PUSHI - Push immediate



If PC section is non-zero, push the full 30-bit effective address onto the stack. If the effective address calculation results in a local reference to an AC in a non-zero section, the effective address is first converted to the global AC address form before being pushed on the stack.

If PC section is zero, 0,,EA is pushed onto the stack.

PUSHI P,E is therefore functionally equivalent to the following sequence:

```
XMOVEI AC,E
PUSH P,AC
```

except that no AC is destroyed.

Note that because of the conversion of local AC references to global AC form, the following instruction, when executed in a non-zero section, pushes 1,,10 on the stack instead of 0,,10:

```
PUSHI P,10
```

This instruction is identical to the KC10 PUSHI instruction and is not currently implemented by the KS10 microcode.

NOTE

Due to time-to-market considerations, this instruction may not be implemented by the microcode for the FCS machine.

3.3 The exec mode environment

Because exec mode software already exists for the proposed KC10 design, the KD10 exec mode environment is as quite similar to the KC10 exec mode environment. The one exception to this is the external I/O instructions. Because the FCS machine uses UBAs for I/O, the external I/O interface is done with the KS10 set of I/O instructions. The set of external I/O instructions will necessarily change as new adapters are added to the system.

The exec mode environment includes not only the privileged instruction set, but also data structures and protocols for processing page fails, traps, MUUOs, LUUOs, and interrupts. This section describes each in detail.

3.3.1 Privileged instructions

The KD10 privileged instructions include instructions to control the processor, manipulate I/O devices, etc. Each instruction is described separately below.

Each privileged instruction is nominally legal only in exec mode. Certain instructions are also legal in user mode if the USER I/O PC flag is set.

3.3.1.1 APR0, APR1, and APR2 instructions

The APR0 (opcode 700), APR1 (opcode 701), and APR2 (opcode 702) instructions control the internal processor devices. The AC field of the instruction is decoded to produce a 1-of-16 sub-instructions.

Most of the instructions described below have functions and bit definitions that are close to the equivalent KC10 instructions. There are a few exceptions to this rule, notably the cache sweep instructions (which don't exist in the KD10), and the timebase and interval timer instructions (which are identical to the KS10).

Note that the AC field value for each sub-instruction is the same as the KS10 value rather than the KC10 value for ease in modifying the microcode. Since the software definitions are in a parameter file, this change should not pose particular problems.

The following table gives the instruction mnemonic for each AC field decode:

AC	APR0	APR1	APR2
00	APRID	SETCU	RDSPB
01	UUO	RDUBR	RDCSB
02	UUO	CLRPT	RDPUR
03	UUO	WRUBR	WRPUR
04	WRAPR	WREBR	RDTIM
05	RDAPR	RDEBR	RDINT
06	SZAPR	UUO	RDHSB
07	SNAPR	UUO	UUO
10	UUO	UUO	WRSPB
11	UUO	UUO	WRCSB
12	UUO	UUO	RDCSTM
13	UUO	UUO	WRCSTM
14	WRPI	UUO	WRTIM
15	RDPI	UUO	WRINT
16	SZPI	UUO	WRHSB
17	SNPI	UUO	UUO

3.3.1.1.1 APRID - APR identification

```
+---------+----+-+----+-----------------+
!   700   ! 00 !@! XR !        Y        !  OPDEF APRID [700000,,0]
+---------+----+-+----+-----------------+
```

Read the microcode version number, the processor serial number, and the microcode and hardware options into location E in the format specified below. At present, there are no microcode or hardware options defined.

The format of the information stored into location E is as follows:

0-5	Microcode options
6-17	Microcode version number
18-20	Hardware options
21-35	Processor serial number

This format is identical to that stored by the KS10 with the exception that the microcode options field is 3 bits narrower and the microcode version number field is 3 bits wider. It differs from the KC10 in that the KC10 stored two words of information.

3.3.1.1.2 WRAPR - Write APR conditions

```
+---------+----+-+----+-----------------+
!   700   ! 04 !@! XR !        Y        !  OPDEF WRAPR [700200,,0]
+---------+----+-+----+-----------------+
```

Perform the specified APR-related functions as specified by bits 18-35 of the effective address. Bits 18 and 20-23 control the operation of the instruction. 1s in these bits produce the indicated results; 0s cause no change. The result of putting 1s in both bits 20 and 21 or 22 and 23 is indeterminate.

The bits in the effective address are as follows:

18 Assign an interrupt level to the processor from bits 33-35.

20 Enable the setting of the flags selected by bits 24-31 to request an interrupt on the level assigned to the processor.

21 Disable the setting of the flags selected by bits 24-31 from requesting an interrupt.

22 Clear the flags selected by bits 24-31

23 Set the flags selected by bits 24-31

24-31 Selected flags. These bits represent individual APR flags, as indicated below, that can be set, cleared, enabled, or disabled with the appropriate combination of bits 20-23. The individual flags are as follows:

- 24 Unassigned APR flag.
- 25 Interrupt console. Setting this flag causes an interrupt to the console. The microcode clears this flag after two clock periods to produce a pulse on the console interrupt line.
- 26 Power failure.
- 27 No memory (NXM).
- 28 Bad memory data (double bit memory error).
- 29 Corrected memory data (single bit memory error).
- 30 Interval done.
- 31 Console interrupt.

33-35 PI level to be assigned to the processor.

With the exception of the addition of bit 18, these bit definitions are identical to those defined by the KS10. It differs from the KC10 in that there is no I/O reset (bit 19), and the APR flags are different.

Note that the operation of this instruction is cumulative over time. For example, a WRAPR 1B20+1B26 enables interrupts on power failures. A subsequent WRAPR 1B20+1B27 enables interrupts on NXMs, but leaves power fail interrupts enabled also. One can think of the operation of this instruction as logically ORing the flag bits for the "enable" and "set" functions with the previous state of the flags. Similarly the "disable" and "clear" functions logically AND the complement of the flag bits with the previous state of the flags.

3.3.1.1.3 RDAPR - Read APR conditions

700	05	@	XR		Y	OPDEF RDAPR [700240,,0]

Read the status of the processor flags into location E in the format specified below. Bits 24-31 represent conditions that can cause interrupts if the condition has been enabled.

The format of the information stored into location E is as follows:

6-13 Flags indicating which APR conditions are enabled to cause interrupts. A 1 in any of these bits indicates that setting the corresponding flag in bits 24-31 will request an interrupt on the level assigned to the processor.

24-31 Flags indicating which APR conditions have occurred. Each flag is discussed separately below.

- 24 Unassigned APR flag. This flag is 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.
- 27 NXM. 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 NXM supplies zero data, on read this error may be accompanied by a 1 in bit 28.
- 28 Double bit error. In a read reference, the ECC logic in the memory controller detected a double bit memory error. This condition also produces a page failure.
- 29 Single bit error. In a read reference, the ECC logic in the memory controller detected and corrected a single bit memory error.
- 30 Interval timer expired. The microcode has completed a count of the interval specified by the program.
- 31 Console interrupt request. The console is requesting a processor interrupt.

32 Some APR condition 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.

33-35 PI level assigned to the processor.

The bits returned by this instruction are identical to those returned by the KS10. It is also identical to the KC10 RDAPR instruction with the exception that the APR flags are different.

3.3.1.1.4 SZAPR - Skip on masked APR conditions all zero

```
+---------+----+-+----+------------------+
!   700   ! 06 !@! XR !        Y         !  OPDEF SZAPR [700300,,0]
+---------+----+-+----+------------------+
```

Test the conditions as returned by RDAPR against the mask produced by bits 18-35 of the effective address. If all masked bits selected by 1s in E are zero, skip the next instruction in sequence.

This instruction is identical to CONSZ APR,E on both KS10 and KC10.

3.3.1.1.5 SNAPR - Skip on any mask APR condition non-zero

```
+---------+----+-+----+------------------+
!   700   ! 07 !@! XR !        Y         !  OPDEF SNAPR [700340,,0]
+---------+----+-+----+------------------+
```

Test the conditions as returned by RDAPR against the mask produced by bits 18-35 of the effective address. If any masked bit selected by 1s in E is one, skip the next instruction in sequence.

This instruction is identical to CONSO APR,E on both KS10 and KC10.

3.3.1.1.6 WRPI - Write PI conditions

```
+---------+----+-+----+-----------------+
!   700   ! 14 !@! XR !        Y        !  OPDEF WRPI [700600,,0]
+---------+----+-+----+-----------------+
```

Perform the PI-related functions as specified by bits 18-35 of the effective address. Bits 22-28 control the operation of the instruction. 1s in these bits produce the indicated results; 0s cause no change. The result of putting 1s in both bits 22 and 24, 25 and 26, or 27 and 28 is indeterminate.

The bits in the effective address are as follows:

22	On the levels selected by bits 29-35, turn off any interrupt requests made in a previous WRPI with bit 24 on.
23	Turn off the PI 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 bits 29-35, but hold the interrupt if the specified level has not been turned on.
25	Turn on the levels selected by bits 29-35.
26	Turn off the levels selected by bits 29-35.
27	Turn off the PI system.
28	Turn on the PI system.
29-35	Selected levels (1-7) to be controlled by bits 22, 24, 25, and 26.

These definitions are identical to those of the KS10 and KC10.

Note that the operation of this instruction is cumulative over time. For example, a WRPI 1B25+1B31 turns on level 3. A subsequent WRPI 1B25+1B33 turns on level 5, but leaves level 3 on also. One can think of the operation of this instruction as logically ORing the levels for the "initiate interrupt" and "turn on" functions with the previous state of the levels. Similarly the "drop request" and "turn off" functions logically AND the complement of the selected level bits with the previous state of the levels.

3.3.1.1.7 RDPI - Read PI conditions

```
+---------+----+-+----+-----------------+
!   700   ! 15 !@! XR !        Y        !  OPDEF RDPI [700640,,0]
+---------+----+-+----+-----------------+
```

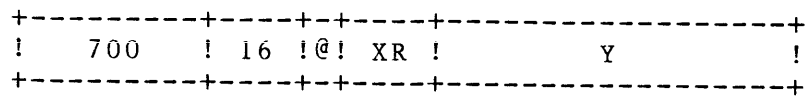
Read the status of the PI system into location E in the format specified below.

The format of the information stored into location E is as follows:

11-17	PI levels (1-7) on which program requests have been made (with a WRPI with a 1 in bit 24).
21-27	PI levels (1-7) on which interrupts are currently in progress.
28	PI system is on.
29-35	PI levels (1-7) which have been turned on (with a WRPI with a 1 in bit 25).

This format is identical to that stored by the KS10 and KC10.

3.3.1.1.8 SZPI - Skip on masked PI conditions all zero

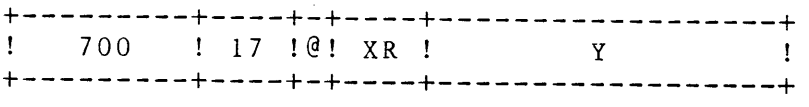


OPDEF SZPI [700700,,0]

Test the conditions as returned by RDPI against the mask produced by bits 18-35 of the effective address. If all masked bits selected by 1s in E are zero, skip the next instruction in sequence.

This instruction is identical to CONSZ PI,E on the KS10 and KC10.

3.3.1.1.9 SNPI - Skip on any masked PI condition non-zero



OPDEF SNPI [700740,,0]

Test the conditions as returned by RDPI against the mask produced by bits 18-35 of the effective address. If any masked bit selected by 1s in E is one, skip the next instruction in sequence.

This instruction is identical to CONSO PI,E on KS10 and KC10.

3.3.1.1.10 SETCU - Set CST-update-needed bits

```
+---------+----+-+----+------------------+
!   701   ! 00 !@! XR !        Y         !  OPDEF SETCU [701000,,0]
+---------+----+-+----+------------------+
```

Sweep the translation buffer and set the "CST update needed" bit for each entry. Setting this bit in a translation buffer entry causes a page fail to occur on the next virtual reference to each page.

The indirect, index register, and Y fields of this instruction are not used and should be zero.

This instruction is identical to the KC10 SETCU instruction and is currently not implemented by the KS10 microcode.

3.3.1.1.11 RDUBR - Read User Base Register

```
+---------+----+-+----+-----------------+
!   701   ! 01 !@! XR !        Y        !  OPDEF RDUBR [701040,,0]
+---------+----+-+----+-----------------+
```

Read the process status of the pager and the current address break information into the locations addressed by E through E+2. The information stored is the same as that supplied by a WRUBR.

The format of the first word (E) is as follows:

0	Returned as a 1 (Load AC blocks in WRUBR).
1	Returned as a 1 (Load PCS in WRUBR).
2	Returned as a 1 (Load UBR in WRUBR).
8	Returned as a 1 (Load address break conditions in WRUBR).
18-35	Physical page number of the UPT.

The format of the second word (E+1) is as follows:

18-20	Current AC block number.
21-23	Previous AC block number.
24-35	Previous context section.

The format of the third word (E+2) is as follows:

0	Address break enabled for a normal fetch of an instruction in the program under control of PC.
1	Address break enabled for any reference that reads except the normal fetch of an instruction.
2	Address break enabled for any reference that writes.
3	Address break enabled for a reference made in user virtual address space (0 implies exec address space).
4	Address break is enabled.
5	Returned as a 0 (Maintain address break conditions in WRUBR).

6-35 Virtual break address.

The format of the information stored by this instruction is very different from that stored by the KS10. It is much closer to the information stored by the KC10 RDCTX instruction, but the address break information requires two fewer words.

NOTE

The ability to implement any address break functionality is subject to question. If it becomes impossible to do so, RDUBR will store only words E and E+1, and bit 8 in word E will become undefined.

3.3.1.1.12 CLRPT - Clear page table entry

701	02	@	XR		Y	OPDEF CLRPT [701100,,0]

Invalidate the translation buffer entry corresponding to the page referenced by the effective address. Clear the appropriate parts of the internal cache of paging information.

In terms of the effect on the translation buffer, this instruction is identical to that on the KS10. However, the KS10 also swept the entire cache because it was virtually addressed. Because the KD10 cache is physically addressed, it need not be swept on a CLRPT.

3.3.1.1.13 WRUBR - Write User Base Register

```
+---------+----+-+----+-----------------+
!   701   ! 03 !@! XR !        Y        !  OPDEF WRUBR [701140,,0]
+---------+----+-+----+-----------------+
```

Set up the process-oriented elements of the pager and address break conditions according to the contents of the locations address by E through E+2.

The first word (E) contains control information and the physical page number of the UPT. A 1 in a control bit causes the specified action to happen; a 0 causes no change. The format is as follows:

0 Load the current and previous AC block numbers from bits 18-23 of word E+1.

1 Load previous context section from bits 24-35 of word E+1.

2 Load bits 18-35 of word E into the user base register. Invalidate the entire translation buffer and cache by clearing the valid bits in all entries. Clear the cache of internal paging information. If paging is on, reinitialize the internal cache of paging information from the EPT and UPT.

8 Load the address break conditions from word E+2. If this bit is a 0, the microcode will not reference word E+2.

18-35 Physical page number of the UPT.

The second word (E+1) contains the CAB, PAB, and PCS fields. The format is as follows:

18-20 Current AC block number.

21-23 Previous AC block number.

24-35 Previous context section.

The third word contains address break conditions and the address break address.

0 Enable address break for a normal fetch of an instruction in the program under control of PC.

1	Enable address break for any reference that reads except the normal fetch of an instruction.
2	Enable address break for any reference that writes.
3	Enable address break for a reference made in user virtual address space (0 implies exec address space).
4	Turn on address break. If this bit is a 0, turn off address break.
5	Ignore bits 0-3 and 6-35 and use the previous condition bits and break address. By setting this bit, the program can turn address break on and off without having to reload the conditions and break address. If this bit is a 0, load new address break conditions and the break address from bits 0-3 and 6-35.
6-35	Virtual break address.

The format of the information supplied to this instruction is very different from that used by the KS10. It is much closer to the information supplied to the KC10 WRCTX instruction, but the address break information requires two fewer words.

NOTE

The ability to implement any address break functionality is subject to question. If it becomes impossible to do so, WRUBR will take only two arguments in words E and E+1, and bit 8 in word E will be ignored.

3.3.1.1.14 WREBR - Write Exec Base Register

```
+---------+----+-+----+-----------------+
!   701   ! 04 !@! XR !        Y        !  OPDEF WREBR [701200,,0]
+---------+----+-+----+-----------------+
```

Set up the system-oriented characteristics of the pager according to the contents of location E.

The format of the argument is as follows:

3 Implement TOPS-20 mode. At present, both TOPS-10 and use the same style of paging so this bit is currently ignored. In the future, it can be used to distinguish TOPS-10 vs. TOPS-20 features.

4 Enable the pager. Invalidate the entire translation buffer and cache by clearing the valid bits in each entry. Clear the internal cache of paging information. If this bit is on, reinitialize the cache from the EPT and UPT.

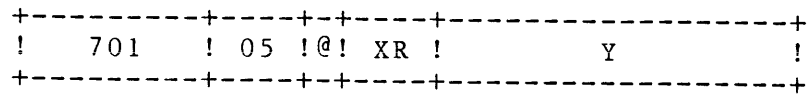
7 Load trap enable from bit 8. If this bit is a 0, do not change the state of trap enable.

8 Enable full processing of traps, LUUOs, MUUOs, and page fails. If bit 7 is a 1 and this bit is a 0, the processing of these conditions is modified. See the section on trap handling for details. If bit 7 is a 1 and this bit is a 1, the processing of these conditions is done normally.

18-35 Physical page number of the EPT.

The information supplied to this instruction is different from the KS10 WREBR instruction. The information is identical to that supplied to the KC10 WREBR instruction with the exception that there are no cache control bits defined.

3.3.1.1.15 RDEBR - Read Exec Base Register



OPDEF RDEBR [701240,,0]

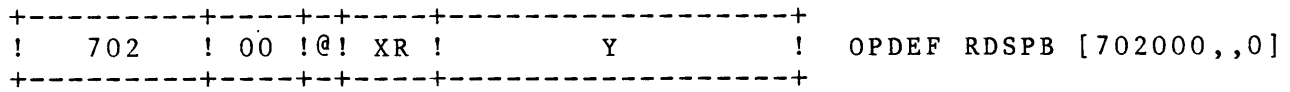
Read the system-oriented characteristics of the pager into location E. The information read is that same as that supplied to to WREBR.

The format of the information stored is as follows:

3	TOPS-20 mode is implemented.
4	Paging is enabled.
7	Returned as a 0 (Load trap enable in WREBR).
8	Full processing of traps, LUUOs, MUUOs, and page fails is enabled.
18-35	Physical page number of the EPT.

The information stored is dissimilar with that stored by the KS10. It is identical to that stored by the KC10 RDEBR instruction with the exception that there are no cache control bits.

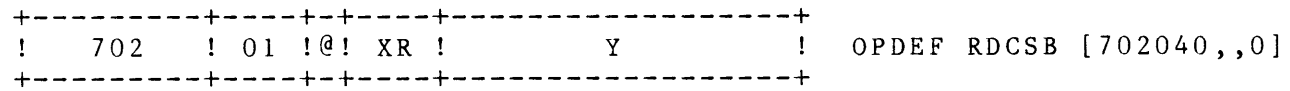
3.3.1.1.16 RDSPB - Read SPT Base Register



Read the contents of the SPT base register into bits 9-35 of location E.

This instruction is identical to the KS10 and KC10 RDSPB instructions.

3.3.1.1.17 RDCSB - Read CST Base Register



Read the contents of the of the CST base register into bits 9-35 of location E.

This instruction is identical to the KS10 and KC10 RDCSB instructions.

3.3.1.1.18 RDPUR - Read Process Use Register

702	02	@	XR	Y		OPDEF RDPUR [702100,,0]

Read the process use register into location E.

This instruction is identical to the KS10 and KC10 RDPUR instructions.

3.3.1.1.19 WRPUR - Write Process Use Register

702	03	@	XR	Y		OPDEF WRPUR [702140,,0]

Load the contents of location E into the process use register for use as the process use word in CST updating.

This instruction is identical to the KS10 and KC10 WRPUR instructions.

3.3.1.1.20 RDTIM - Read timebase conditions

```
+---------+----+-+----+-----------------+
!   702   ! 04 !@! XR !        Y        !  OPDEF RDTIM [702200,,0]
+---------+----+-+----+-----------------+
```

Update the timebase double word kept in internal storage with the hardware counter and store the result into locations E and E+1.

This instruction is identical with the KS10 RDTIM instruction.

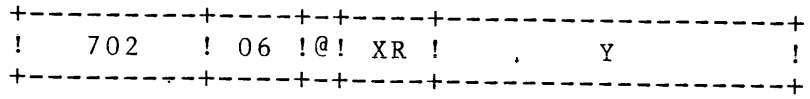
3.3.1.1.21 RDINT - Read interval timer conditions

```
+---------+----+-+----+-----------------+
!   702   ! 05 !@! XR !        Y        !  OPDEF RDINT [702240,,0]
+---------+----+-+----+-----------------+
```

Read the contents of the interval register into location E. The period read is the same as that supplied to WRINT.

This instruction is identical with the KS10 RDINT instruction.

3.3.1.1.22 RDHSB - Read Halt Status Block address

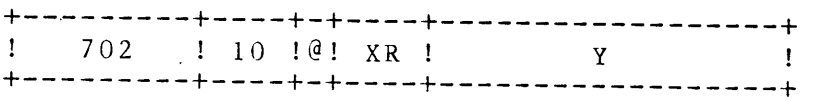


OPDEF RDHSB [702300,,0]

Read the physical address of the halt status block into location E.

This instruction is identical with the KS10 RDHSB instruction.

3.3.1.1.23 WRSPB - Write SPT Base Register

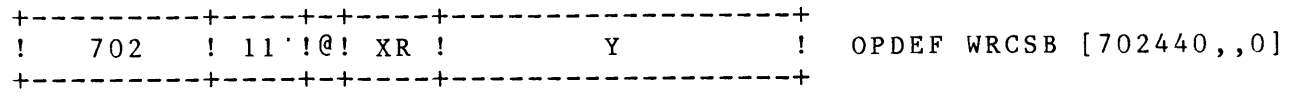


OPDEF WRSPB [702400,,0]

Load the contents of location E into the SPT base register.

This instruction is identical to the KS10 and KC10 WRSPB instructions.

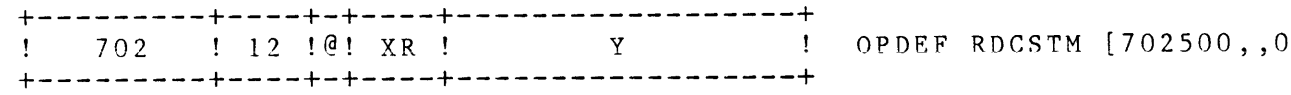
3.3.1.1.24 WRCSB - Write CST Base Register



Load the contents of location E into the CST base register.

this instruction is identical to the KS10 and KC10 WRCSB instructions.

3.3.1.1.25 RDCSTM - Read CST Mask Register



Read the contents of the CST mask register into location E.

This instruction is identical to the KS10 and KC10 RDCSTM instructions.

3.3.1.1.26 WRCSTM - Write CST Mask Register

702	13	@	XR	Y	OPDEF WRCSTM [702540,,0]

Load the contents of location E into the CST mask register for use as the mask in CST updating.

This instruction is identical to the KS10 and KC10 WRCSTM instructions.

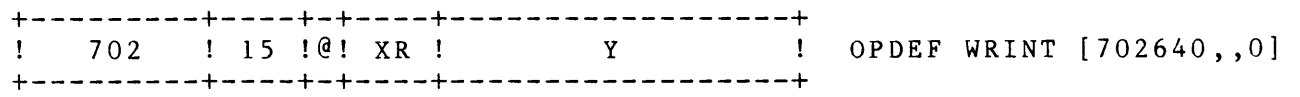
3.3.1.1.27 WRTIM - Write timebase conditions

702	14	@	XR	Y	OPDEF WRTIM [702600,,0]

Read the contents of location E and E+1, clear the right twelve bits of the low order word, and place the result in the timebase registers in internal storage.

This instruction is identical to the KS10 WRTIM instruction.

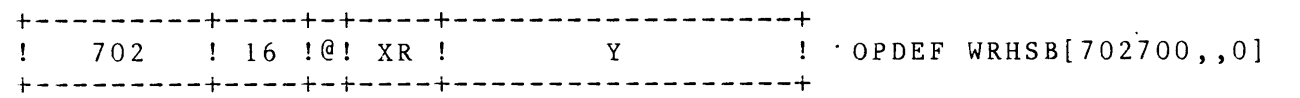
3.3.1.1.28 WRINT - Write interval timer conditions



Load the contents of location E into internal storage.

This instruction is identical to the Ks10 WRINT instruction.

3.3.1.1.29 WRHSB - Write Halt Status Block Address



Load bits 9-35 of location E into the halt status block base register in the workspace. If bit 0 of the word in E is 0, this address will be used as the physical address for storing halt status. If bit 0 is 1, no status will be stored.

This instruction is identical to the KS10 WRHSB instruction.

3.3.1.2 External I/O instructions

I/O on the FCS machine will be done through (modified) Unibus adapters similar to those on the KS10. As a result, the KS10 external I/O instructions have been carried over to the KD10 design unchanged, with the exception that the opcodes have changed.

The external I/O instructions and their opcodes are as follows:

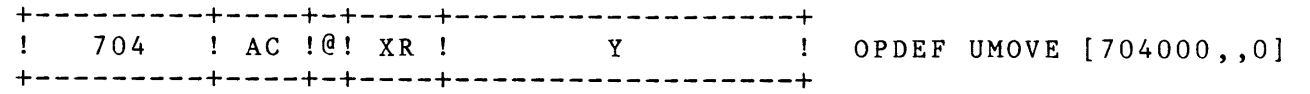
Opcode	Mnemonic
720	TIOE
721	TION
722	RDIO
723	WRIO
724	BSIO
725	BCIO
730	TIOEB
731	TIONB
732	RDIOB
733	WRIOB
734	BSIOB
735	BCIOB

As new adapter types are added, new external I/O instructions may also have to be added. For example, the KC10 queue instructions might be useful with future adapters.

3.3.1.3 Other privileged instructions

Besides the APR0, APR1, and APR2 instructions, and the external I/O instructions, there are several more privilege instructions. Each of these instructions is discussed separately below.

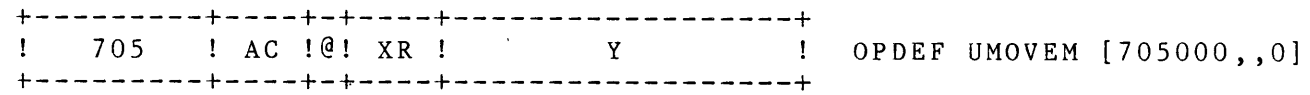
3.3.1.3.1 UMOVE - User move



Perform an effective address calculation in current context, then read the data from the specified location in previous context. Store the result in AC.

This instruction is functionally equivalent to PXCT 4,[MOVE AC,E] and is identical to the KS10 and KC10 UMOVE instructions.

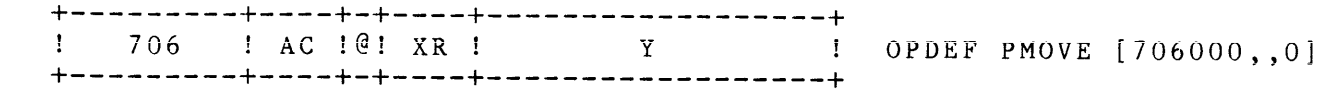
3.3.1.3.2 UMOVEM - User move to memory



Perform an effective address calculation in current context, then store the contents of AC into the specified location in previous context.

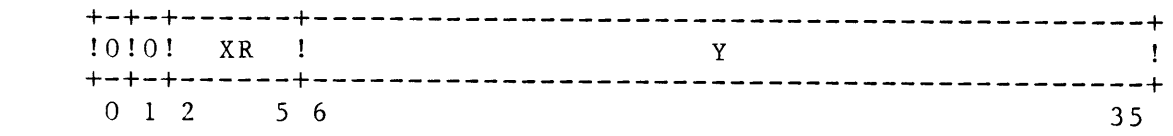
This instruction is functionally equivalent to PXCT 4,[MOVEM AC,E] and is identical to the KS10 and KC10 UMOVEM instructions.

3.3.1.3.3 PMOVE - Physical move



Perform a physical effective address calculation using the word in location E, then read the specified physical memory location and store the result in AC.

A physical effective address calculation evaluates a physical EA-calc word. Such a word is similar to a virtual EFIW word and looks as follows:

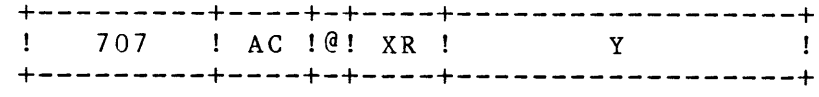


Bits 2-5 of the physical EA-calc word are the index register address, and bits 6-35 are the physical memory address Y. The physical effective address is Y alone if XR is zero. If XR is non-zero, the contents of the specified index register are added to Y to produce a 27-bit physical addresses. Bits 0-1 of the physical EA-calc word must be zero and the microcode will generate a page fail if they are not.

Note that addresses in the range 0-17, inclusive, reference physical memory locations 0-17 and not the ACs. Also, because this instruction makes a physical reference, no CST update is performed by the instruction.

This instruction is identical to the KC10 PMOVE instruction and is not currently implemented by the KS10.

3.3.1.3.4 PMOVEM - Physical move to memory



OPDEF PMOVEM [707000,,0]

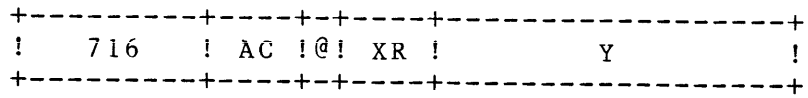
Perform a physical effective address calculation using the word in location E, then store the contents of AC into the specified physical memory location.

The physical effective address calculation algorithm is described under PMOVE above.

Note that addresses in the range 0-17, inclusive, reference physical memory locations 0-17 and not the ACs. Also, because this instruction makes a physical reference, no CST update is performed by the instruction.

This instruction is identical to the KC10 PMOVEM instruction and is not currently implemented by the KS10.

3.3.1.3.5 LDPAC - Load previous AC blocks



OPDEF LDPAC [716000,,0]

Load the previous context ACs (from the AC block specified by the current PAB value) from the block beginning at the location addressed by E. Continue to transfer words from the block until a word has been transferred to the previous context AC specified by the AC field of the instruction.

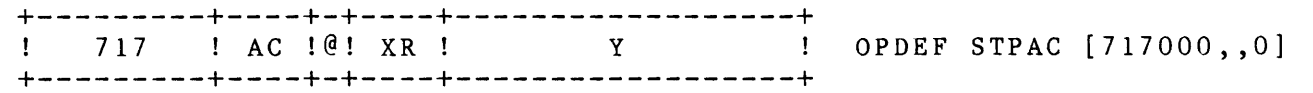
The 16 word block must not cross section boundaries.

To load all previous context ACs from the 16-word current context block beginning at USERAC, one would execute the following instruction:

```
LDPAC 17,USERAC
```

This instruction is identical to the KC10 LDPAC instruction and is currently not implemented by the KS10.

3.3.1.3.6 STPAC - Store previous AC blocks



Store the previous context ACs (as specified by the current PAB value) into the block beginning at the location addressed by E. Continue to transfer words from the previous context ACs to the block until a word has been transferred from the previous context AC specified by the AC field of the instruction.

The 16 word block must not cross section boundaries.

To store all previous context ACs into the 16-word current context block beginning at USERAC, one would execute the following instruction:

```
STPAC 17,USERAC
```

This instruction is identical to the KC10 STPAC instruction and is currently not implemented by the KS10.

3.3.1.3.7 MAP - Map an address

```
+---------+----+-+----+-----------------+
!   257   ! AC !@! XR !        Y        !  OPDEF MAP [257000,,0]
+---------+----+-+----+-----------------+
```

If the pager is on and the processor is in exec mode or user I/O mode, map the page number of the virtual effective address E and place the resulting physical address and other map data in AC.

To do this, the microcode performs a pointer trace as if a reference to the page caused a page fail. The translation buffer is not changed as the result of this pointer trace.

The generic format of the information returned in the AC by the MAP instruction is as follows:

```
!=============================================================!
!U!0!V!M!W!0!0!C!0! 000 !                Physical           !
!S! !L!D!T! ! !H! !     !                Address            !
!=============================================================!
 0 1 2 3 4 5 6 7 8 9   1 1                                   3
                       3 4                                   5
```

The fields are as follows:

0 User. If this bit is a 1, the mapping is in user space. If the bit is a 0, the mapping is in exec space.

2 Valid. If this bit is a 1, the mapping is valid. That is, bits 14-35 contain the physical address corresponding to the virtual effective address.

3 Modified. If this bit is a 1, the page has been modified (according to the W bit in the CST entry for the page).

4 Writable. If this bit is a 1, the page is writable (as indicated by the fact that the logical AND of the W bits in all pointers was a 1).

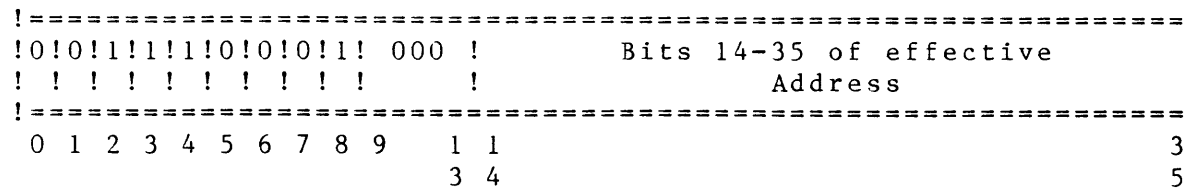
7 Cachable. If this bit is a 1, the page is cachable (as indicated by the fact that the logical AND of the C bits in all pointers was a 1).

14-35 Physical address.

The information returned in the AC by the MAP instruction can be in one of three different formats, as follows:

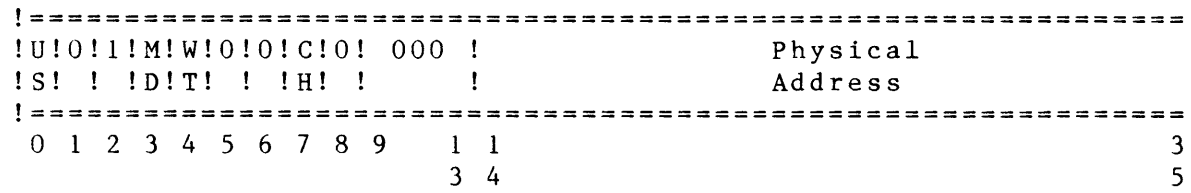
If the pager is off, bits 14-35 of the effective address are returned in bits 14-35 of AC. In addition, the valid, modified, and writable

bits are also set. This looks as follows:



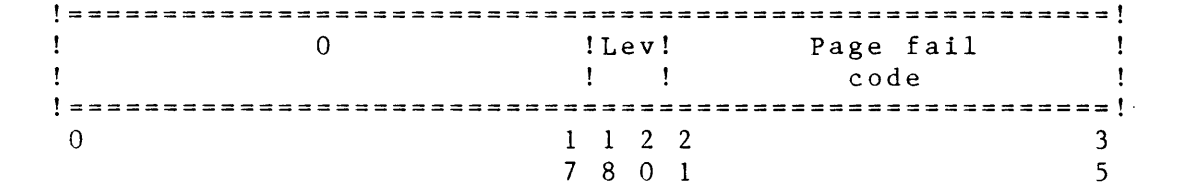
Format of AC if paging is off

If the pager is on and there is a valid mapping for the specified virtual address, the format of the information returned in the AC is as follows:



Format of AC if a valid mapping was found

If the pager is on and there was no valid mapping for the effective address, the entire left half o AC is zero (including the valid bit) and the right half has the same format as the right half of the page fail word that would be returned if a reference was made to the specified page. This looks as follows:



Format of AC if no valid mapping exists

This format is a combination of the KS10 and KC10 formats. The main difference between this format and the KC10 format is the position of the valid bit in the AC (the KC10 valid bit was bit 3).

3.3.2 Changes to JRST

The KD10 implementation of JRST is identical to the KC10 implementation. It is also very similar to the KS10 and extended KL10 implementation with several exceptions. The exceptions are as follows:

F	Mnemonic	Function
01		Previously PORTAL. Because the KD10 processor does not implement public mode, this is treated as a normal JRST 0,.
05	XJRSTF	Restore the program flags (as appropriate for the mode of the processor) and PC from the flag-PC double word in locations E and E+1 and continue performing instructions in normal sequence beginning at the location then addressed by PC. If the instruction is executed in exec mode, also restore CAB, PAB, and PCS from the first word of the flag-PC double word.
06	XJEN	Restore the level on which the highest priority interrupt is currently being held and then perform an XJRSTF.
07	XPCW	Save the program flags, CAB, PAB, PCS, and PC in a flag PC double word in locations E and E+1. Then restore the program flags, CAB, and PC from the flag-PC double word in locations E+2 and E+3 and continue performing instructions in normal sequence beginning at the location then addressed by PC. Do not restore PAB or PCS from E+2.
10		Always execute as an MUUO through the I/O undefined opcode new PC words in the UPT.
12		Previously JEN. Always execute as an MUUO through the I/O undefined opcode new PC words in the UPT. Since the KD10 always stores flag-PC double words in XJEN format, there is no need for JEN.
14	SFM	Save the program flags in bits 0-12 of the word addressed by E and clear bits 13-17. If the instruction is executed in exec mode, store CAB, PAB, and PCS in bits 18-20, 21-23, and 24-35, respectively, of the same word. If the instruction is executed in user mode, clear bits 18-35. This instruction is legal in any section.
15	XJRST	Restore the PC from bits 6-35 of the word addressed by E and continue performing instructions in normal sequence beginning at the location then addressed by PC. Do not change the program flags, CAB, PAB, or PCS.

For each of the 16 possible JRST functions, the table given below indicates where each form of the instruction is legal. 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 zero
K	Legal only in kernel (executive) mode
No	Legal nowhere
-H	Legal where indicated by first symbol but causes a halt

If the JRST function is illegal in the mode or context in which it is executed, the instruction traps as an MUUO through the I/O undefined opcode new PC words in the UPT.

Function	Mnemonic	Legal domain
JRST 0,	JRST	Yes
JRST 1,	JRSTCI	Yes
JRST 2,	JRSTF	Z
JRST 3,		No
JRST 4,	HALT	K-H
JRST 5,	XJRSTF	Yes
JRST 6,	XJEN	K
JRST 7,	XPCW	K
JRST 10,		No
JRST 11,		No
JRST 12,		No
JRST 13,		No
JRST 14,	SFM	Yes
JRST 15,	XJRST	Yes
JRST 16,		No
JRST 17,		No

3.3.3 Cache hardware and control

The KS10 cache used the top 512 locations in the workspace. It was virtually addressed, one-way associative, and had a one-word block size. Because it was virtually addressed, the microcode swept the cache whenever an event caused a significant change in the paging structure (such as WREBR, WRUBR, and CLRPT).

The KD10 cache has been increased in size to 2K. It uses the top 2K locations of the 4K workspace. Because the cache is larger than the size of a page (512 words), it has been changed to a physically addressed cache addressed by translated bits 25-26 of the virtual address and the untranslated bits 27-35. It continues to be one-way associative, and has a one-word block size.

Each entry in the cache directory contains a parity bit, a valid bit, and bits 14-24 of the physical memory address corresponding to the word in the data cache.

Although a physically addressed cache need not be swept on a context switch, the KD10 cache is swept by the microcode on a WREBR or a WRUBR that changes the UPT (but not on a CLRPT). The reason for this is that the I/O adapters don't invalidate the cache on memory writes (and probably can't, given the structure of the machine). As result, the cache sweeps are necessary to insure that the cache doesn't contain stale data after an I/O write.

3.3.4 Paging hardware and data structures

This section describes the KD10 paging hardware, microcode, and data structures.

3.3.4.1 Paging hardware

The KD10 translation buffer is a 1K, one-way associative, one-word block size paging cache. The translation buffer is addressed by VMA<17>.XOR.VMA USER and VMA<18:26>. The XOR function on bit 17 separates equivalent user and exec entries by half the TB as on the KL10.

Each entry in the translation buffer contains a parity bit, a valid bit, a writable bit, a cachable bit, a user bit, a CST update needed bit, bits 6-15 of the virtual address mapped by the entry (TB VMA), and bits 14-26 of the physical memory address corresponding to the mapping (TB PMA).

An entry in the translation buffer maps a virtual address into a physical address if all of the following conditions are met:

1. The valid bit is on.
2. The parity is good.
3. The CST update needed bit is off.
4. If the references is a write or write-test, the writable bit is on.
5. The state of the user bit is the same as the user bit in the request.
6. TB VMA<6:16> matches bits 6-16 of the request VMA.

If these conditions are true, the physical memory address corresponding to the requested VMA is TB PMA<14:26> concatenated with the untranslated VMA bits 27-35.

3.3.4.2 Caching of paging information other than the TB

In an attempt to make the virtual-to-physical translation process faster in the case where there is a miss in the translation buffer, the KD10 microcode maintains a cache in the workspace of recent transactions.

All caching of paging information implies that the program must tell the hardware and the microcode when the information is changed. In previous machines, this meant that the program did a CLRPT to clear a single entry, and a WREBR or WRUBR to clear the entire paging cache.

The same concept holds for the KD10, except that the microcode transaction cache must be cleared in addition to the appropriate translation buffer entries. This process should be transparent to the program since the microcode does the appropriate information clearing at the appropriate time.

There is one exception to the "do the right thing and the right time" rule which the monitor must be aware of. The KD10 microcode maintains a cache of the exec and user super section pointers from the EPT and UPT in internal store. This cache is not cleared when a CLRPT is done; it is only cleared as the result of a WREBR or WRUBR. As a result, the monitor must issue one of these two instructions if any of the super section pointers are changed.

3.3.4.3 Pager data structure

The KL10's implementation of extended sections was to allow a maximum of 32 section pointers to be placed in EPT/UPT locations 540-577. A single page full of section pointers can only reference 512 sections. 8 pages of section pointers will be required to address 4096 sections. Since we are going to create some new data items and structure, let us define some terms:

1. A page containing section pointers will be called a "Section Table" or ST. The pointer types found herein are identical to those already found in EPT/UPT locs 540-577 on a KL10.

2. A page containing map pointers will be called a page map.

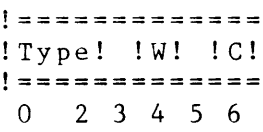
3. VMA<6:8> will be called the "Super Section Number" and will be used to determine which of the 8 Section Tables to look in.

4. EPT/UPT locations 520-527 will be a "Super Section Table" or SST, and will be indexed by VMA<6:8>.

5. The Super Section Table will contain new pointer types called "Super Section Pointers" defined below.

3.3.4.3.1 Pointers

The microcode evaluates three kinds of pointers: super section pointers, section pointers, and map pointers. These are used in super section tables, section tables, and page maps, respectively. There are 5 types of pointers distinguished by a type code in bits 0-2 of the pointer; of these, three are access pointers that allow access to the given super section, section, or page and are identical in the format of the left 9 bits. This format is as follows:



Bits 3, 5, and 8 are ignored by the microcode and may be used by the software.

Every access pointer of this type must have "use" bits for the super section, section, or page it represents. The W and C bits indicate whether the super section, section, or page is writable or cachable. The K bit is reserved for future expansion, but is not currently used by the KD10 hardware.

Throughout the evaluation procedure the microcode logically 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,

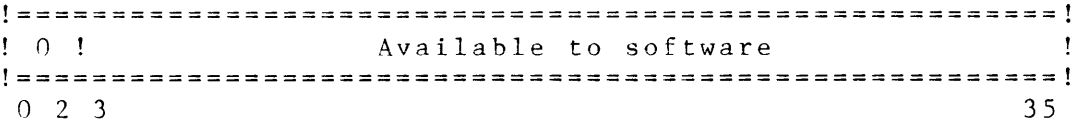
if W is 1 in the final pointer for the mapping, the page is writable provided the super section and the section were also specified as writable by the original super section and section pointers, and "writable" has been specified by every other pointer encountered along the way.

Note that the W bit is also ANDed with the W bit in the CST entry for the final data page to determine the state of the translation buffer W state bit. This final operation is not done if the CST base address is zero.

3.3.4.3.1.1 Super Section Pointers

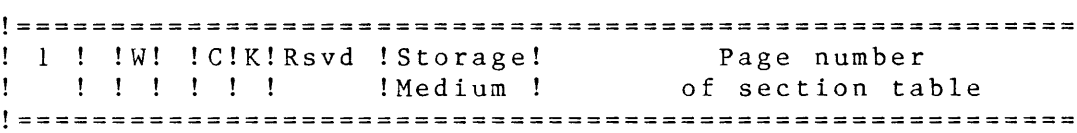
Entries in the Super Section Table in EPT/UPT locations 520-527 are of the following five types. All other types are reserved and will cause a page fail if the microcode encounters them on a refill.

No access



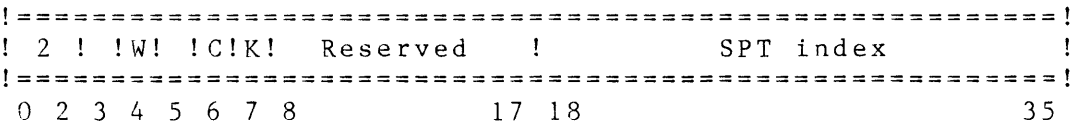
The super section is inaccessible.

Immediate



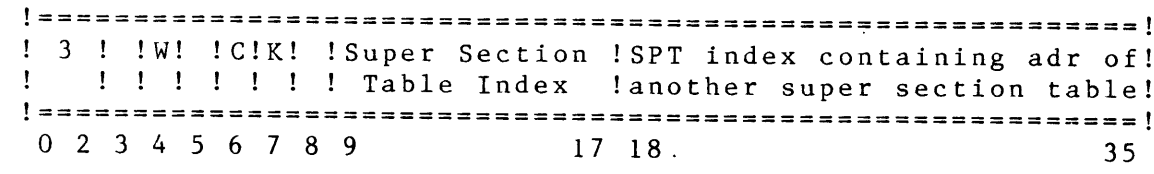
If bits 12-17 are are zero, the section table is in the page specified by bits 18-35. Otherwise, the page is not in memory.

Shared



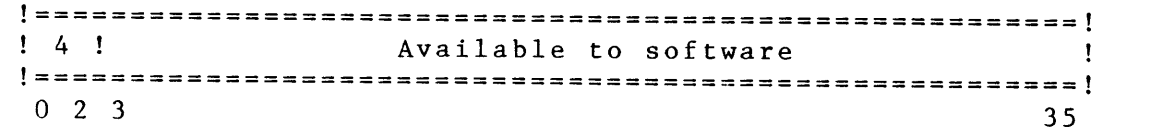
The page address of the section table is in the SPT at the offset specified by bits 18-35

Indirect



In the SPT offset specified by bits 18-35 is the page address of a secondary super section table. The next super section pointer to be evaluated is in that table at the offset specified by bits 9-17.

KL compatible

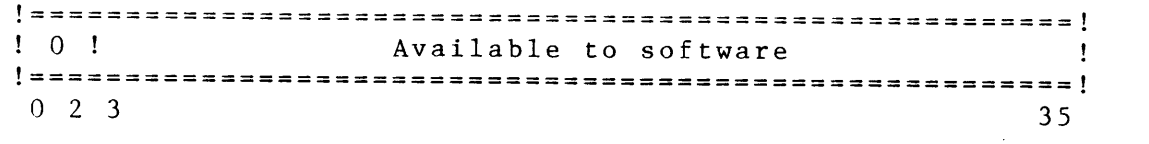


This type of pointer may ONLY appear in EPT/UPT offset 520 and indicates that KL compatible paging is to be used. If VMA<6:12> is zero, use VMA<13:17> as an index into the KL compatible section table starting at EPT/UPT offset 540 and perform the pointer evaluation exactly as a KL10 would. If VMA<6:12> is non-zero or if this type of pointer appears in a super section table entry other than that at EPT/UPT offset 520, a page fail trap will occur. See the section on page fail conditions for the page fail codes.

3.3.4.3.1.2 Section Pointers

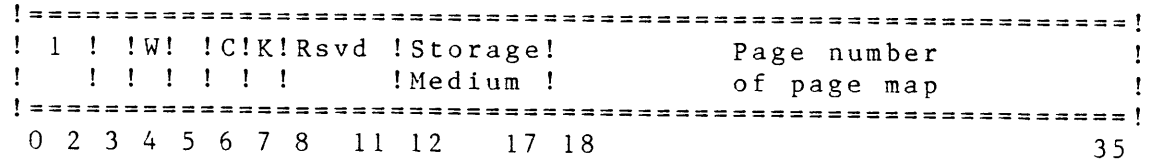
Entries in a section table are of the following four types. All other types are reserved and will cause a page fail if the microcode encounters them on a refill.

No access



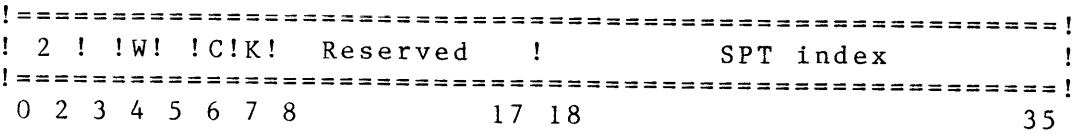
The section is inaccessible.

Immediate



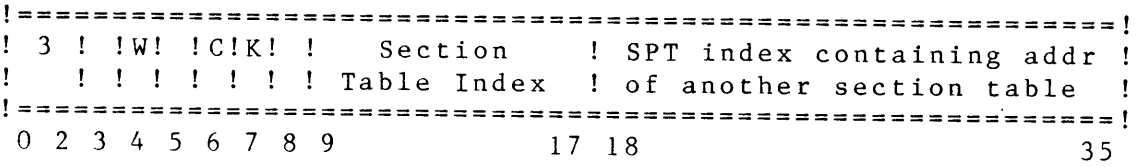
If bits 12-17 are zero, the page map is in the page specified by bits 18-35. Otherwise, the page is not in memory.

Shared



The page address of the page map is in the SPT at the offset specified by bits 18-35

Indirect

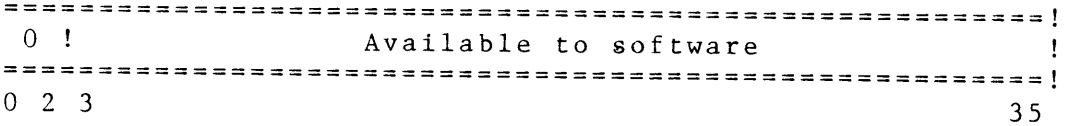


In the SPT offset specified by bits 18-35 is the page address of a secondary section table. The next section pointer to be evaluated is in that table at the offset specified by bits 9-17.

3.3.4.3.1.3 Map pointers

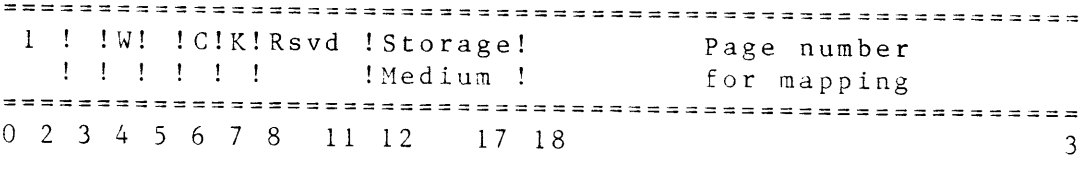
Entries in a page map are of these four types. All other types are reserved and will cause a page fail if the microcode encounters them on a refill.

No access



The page is inaccessible.

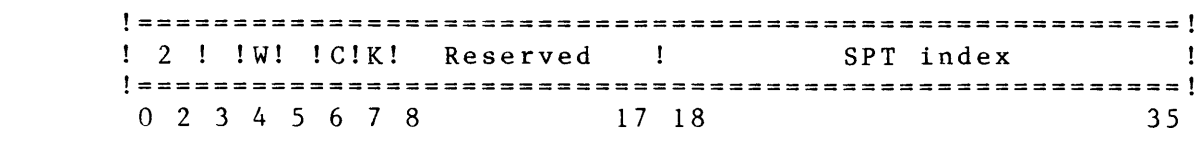
Immediate



If bits 12-17 are zero, the physical page specified by bits 18-35 corresponds to the referenced virtual page. Otherwise, the page is

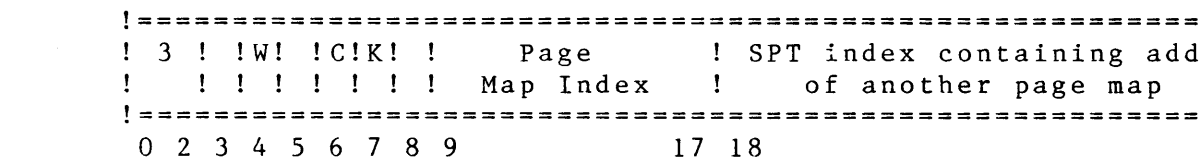
not in memory.

Shared



The page address for the mapping for the referenced virtual page is in the SPT at the offset specified by bits 18-35.

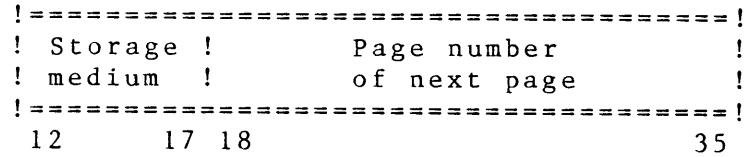
Indirect



In the SPT offset 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 offset specified by bits 9-17.

3.3.4.3.2 Page address words

The translation buffer refill process causes the microcode to follow pointers in memory to finally determine the physical page number of the data page that should be mapped by the virtual page that caused the page fault. In order to do this, the microcode must evaluate 3 different kinds of pointer levels, super section, section, and page pointers. At each level, the microcode must encounter a "page address word" that gives the page number of the page for the next level. For the page pointer evaluation, the page address word actually gives the page number of the final data page. This page address word has the following format:

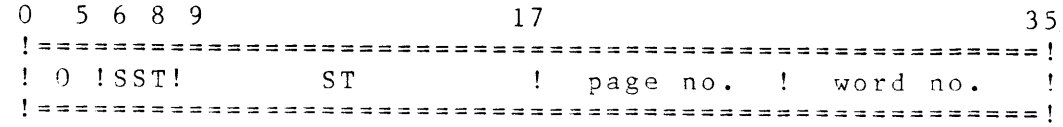


If bits 12-17 are zero, the storage medium is memory, i.e., bits 18-35 supply the number of a page that is in memory. If bits 12-17 are nonzero, the page exists but is stored on some other medium and the microcode traps to the monitor to bring the page into memory. The page address word may be extracted from bits 12-35 of an immediate pointer, or from bits 12-35 of the SPT for share or indirect pointers. For indirect pointers, the microcode will actually encounter more than one page address word.

3.3.4.3.3 Conversion of Virtual to Physical Addresses

An address is converted to a physical page number as follows:

VMA<6:8> is used to index into the Super Section Table. One of the 5 pointer types (Super Section Pointers) can occur here: No Access, immediate, shared, indirect, or KL compatible. Immediate, shared, or indirect pointers yield the physical page number of a Section Table page. VMA<9:17> is used to index into the Section Table to obtain a Section Pointer. Address translation then proceeds as on the KL10 after the section pointer fetch. (See DECsystem10/20 Processor Reference Manual, AA-H391A-TK for a complete description). The VMA can be thought of as follows:



3.3.4.3.4 Page refill

3.3.4.3.4.1 CST updates

The microcode performs an operation called a "CST update" at several points during the processing of a page fault detected by the translation buffer. The operations performed by a CST update are as follows:

1. If the CST base address is zero, skip the rest of the steps.
2. Read the CST entry for the physical page in question from the word addressed by the sum of the CST base register and the physical page number.
3. If the age in the entry (bits 0-5) is zero, start an age-too-small page fail trap to the monitor and skip the rest of the steps.
4. AND the entry with the contents of the CST mask register (set by the WRCSTM instruction).
5. OR the masked entry with the contents of the process use register (set by the WRPUR instruction).
6. Set the modified (M) bit in the entry, if necessary.
7. Write the entry back into the CST in memory, if necessary.

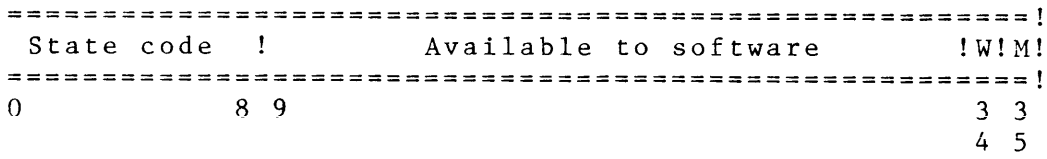
The cases under which a CST update is performed are as follows:

1. A page fault caused by a write reference to a page that is writable but not yet modified. This case sets the modified bit in the entry and writes it back into the CST.
2. A page fault caused by the CST-update-needed bit set in the translation buffer entry for the referenced page. This case writes the entry back into the CST.
3. A pointer trace evaluates the address of a new physical page. This case performs only steps 1-3 as described above for the intermediate pages in the pointer trace. For the final data page that is evaluated by the pointer trace, the full update is performed and the updated entry is written back into the CST.

3.3.4.3.4.2 CST entry format

The CST is a table indexed by physical page number and checked whenever a new memory page is referenced by the microcode. In addition, it is updated for the final data page obtained in a page fail pointer trace and for writable-but-not-yet-modified and

CST-update-needed page fails. The CST format is as follows:



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 non-zero for the page to be usable. A zero page age results in an age-too-small page fail trap to the monitor. The "W" bit is the master write-enable bit for the physical page and is ANDed with the "W" bits in the page pointers when a data page address is written into the translation buffer. The "M" bit indicates that the page has been modified since being brought into memory and is set by the microcode on a writable-but-not-yet-modified page fail trap.

3.3.4.3.4.3 CST mask register format

The CST mask register is ANDed with the CST entry during the CST update process. It should contain a one in every bit position that must be preserved during the update procedure and a zero in every bit position that must be cleared during the update. Therefore, the CST mask register should always contain ones in bits 34 and 35 (the W and M bits) and zeros in bits 0-5 (the page age).

3.3.4.3.4.4 Process Use Register format

The Process Use Register is ORed with the masked CST entry during the CST update process. It should contain a zero in every bit position that must be preserved during the update procedure and a one in every bit position that should be set. Therefore, the Process Use Register should always contain zeros in bits 34 and 35 (the W and M bits) and the new page age in bits 0-5.

3.3.4.3.4.5 Translation buffer state bits

A refill sets the translation buffer state bits as a function of the logical AND of all the pointer use bits that it evaluated in the pointer chase. The relationship is as follows:

State bit	Set if the following condition is met
User	1 if this mapping is for user space.

Valid	Always set to a 1.
Writable	1 if the logical AND of the W pointer use bits of all pointers evaluated was a 1 and the page has been modified according to the CST entry for the page
Cachable	1 if the logical AND of the C pointer use bits of all pointers evaluated was a 1.
CST update	Always set to a 0

3.3.4.3.4.6 Write references

When a virtual write reference is made, the result is a function of the translation buffer entry corresponding to the virtual address specified by the microcode. Write references are particularly interesting because they can succeed or fail based on the exact state of the page in question. A page can be in one of the following three states:

1. Not writable. In this case, a write-failure page fail trap will be given to the monitor.

2. Writable, but not yet modified. In this case, the microcode will update the CST entry for the page, mark the page as modified, and restart the reference.

3. Writable and modified. In this case the write will succeed.

Since the translation buffer has only one bit (the writable bit), the microcode sets the bit if and only if the page is both writable and modified. In the case where it is writable, but not yet modified, state information external to the translation buffer distinguishes this state from the not-writable state.

3.3.4.3.5 Page fail conditions and formats

A page failure occurs when the pager is unable to make a desired memory reference, the microcode detects an illegal condition while executing an instruction (e.g., incorrectly formated indirect word, illegal one-word-global byte pointer, etc.), or the hardware detects a failure while processing a memory request. When such a condition occurs, the microcode stores information about the page fail in UPT locations 451-455, stores the current flag-PC double word in UPT locations 456-457 and loads the new flags, CAB, and PC from the new flag-PC double word in UPT locations 460-461. The format of each of these words is described below.

UPT location 451 contains the page fail word that describes the condition that caused the page fail. The format is as follows:

```
      !=============================================================!
451:  !U!0!I!M!M!W!0!C!P!P!I!W!V!B!Rsvd !Lev!     Page fail code    !
      !S!0!F!R!T!R!0!I!H!R!0!C!C!C!     !   !                       !
      !=============================================================!
       0 1 2 3 4 5 6 7 8 9 1 1 1 1 1   1 1 2 2                      3
                           0 1 2 3 4   7 8 0 1                      5
```

Bits 0-13 come from the VMA flags register and contain information about the reference that caused the page fail. The definition of the bits in the page fail word is as follows:

0 User reference. This bit is returned as a 1 if the reference was to user space. If the reference was to exec space, this bit is returned as a 0.

2 VMA FETCH bit from VMA flags. See the prints.

3 MEM READ bit from VMA flags. See the prints.

4 MEM WR-TEST from VMA flags. See the prints.

5 Write reference. If this bit is a 1, the page fail was caused by a reference that write-failed because of the state of the translation buffer writable and modified state bits. Such a reference may either be a write or a write test.

7 MEM CACHE INH from VMA flags. See the prints.

8 Physical reference. If this bit is a 1, the page fail was caused by a physical reference. If this bit is a 0 (the normal case), the reference was a virtual reference.

9 VMA PREVIOUS VMA flag bit. See the prints.

10 VMA I/O VMA flag bit. See the prints.

11 WRU CYCLE VMA flag bit. See the prints.

12 VECTOR CYCLE VMA flag bit. See the prints.

13 I/O BYTE CYCLE VMA flag bit. See the prints.

14-17 Reserved

18-20 This field gives the level at which this page fail was detected. The level is primarily used to tell the monitor where a translation buffer refill pointer trace stopped and is used in conjunction with the additional data words described below. This field can contain one of four values as follows:

0 This page fault was not the result of a pointer trace, or the page fail condition was detected before the first pointer was fetched.

1 This page fault was detected while processing a super section pointer.

2 This page fault was detected while processing a section pointer.

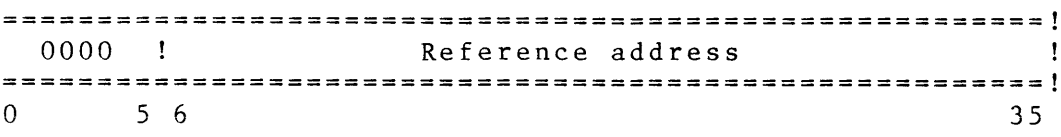
3 This page fault was detected while processing a page pointer.

NOTE

Due to time-to-market considerations, the microcode for the FCS machine may not store the level field in the page fail word.

21-35 This field gives a code that describes the cause of the page fail. The monitor should never have to look at anything other than bits 0 (user), 18-20 (level), and this code to determine the exact cause of the page fail. The rest of the bits in this word are returned only as additional information to be used to debug problems. Each possible page fail code is described below.

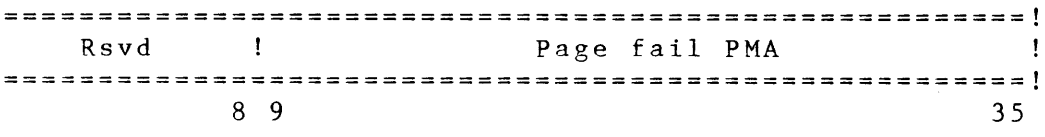
UPT location 452 contains the reference address (if any) for the request that page failed. This address is the virtual memory address for virtual requests and the physical memory address for physical requests. It is only valid for those page fail conditions that resulted from a virtual reference. The table at the end of this section describes under which page fail conditions it is valid.



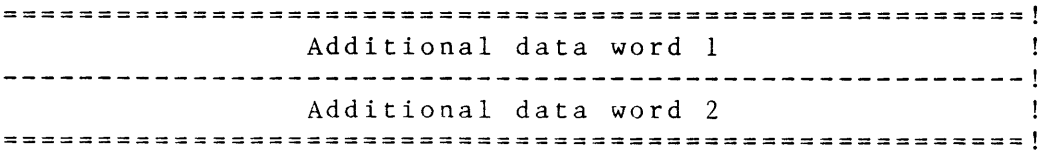
NOTE

Due to time-to-market considerations, the microcode for the FCS machine may not store the information indicated for words 453-455.

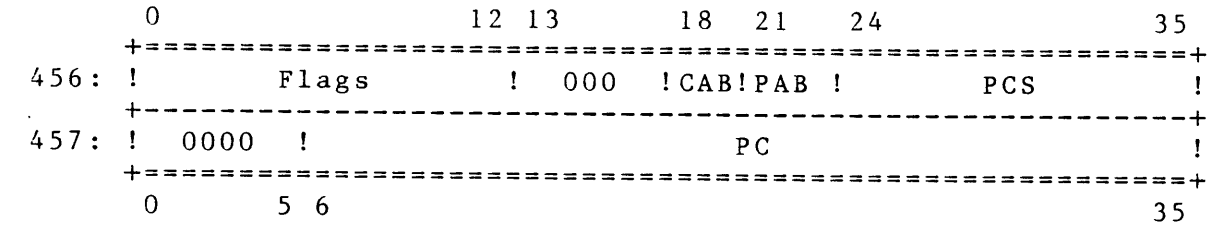
UPT location 453 contains the physical memory address (if any) for the request that page failed. It is only valid for those page fail conditions that have a valid PMA. The table at the end of this section describes under which page fail conditions it is valid.



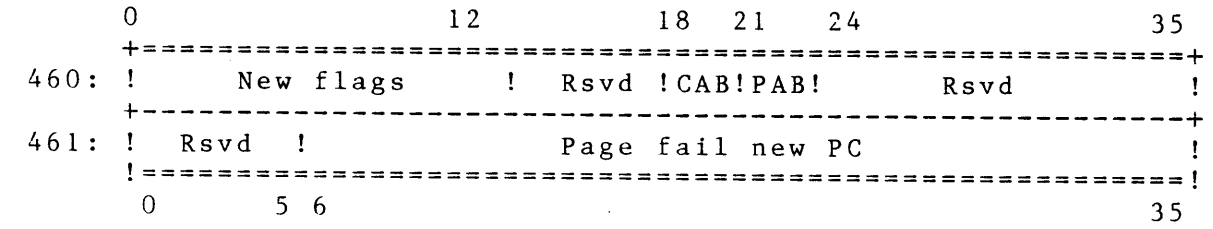
UPT locations 454 and 455 contain additional data that is different for each type of page fail. The contents of these words are given for each page fail at the end of this section. The format of these words is as follows:



UPT locations 456-457 contain the flags, CAB, PAB, PCS, and PC at the time of the page fail in the following format:



UPT locations 460-461 are setup by the monitor and contain the flags, CAB, PAB, and new PC to be loaded when a page fail occurs. The words are in the following format:



3.3.4.3.5.1 Page fail codes and additional data

This section defines the page fail codes that may appear in bits 21-35 of the page fail word and the additional data words returned for each code. For each code below, "RAD", "PMA", "AD1", and "AD2" represent the data returned in words 452-455 of the UPT.

CAUTION

> The page fail codes described below are generated by the microcode and can be easily changed. These page fail codes are a first-pass attempt at assigning values. They may very well change as we add or delete codes. It is strongly suggested that you not make assumptions about the numeric value of any particular code.

The codes that may appear in bits 21-35 of the page fail word are as follows. Note that those code which represent a potential hardware error (memory error, NXM, etc.) have bit 21 on (i.e., they have the form 4xxxx).

1 Write failure - A write reference was attempted to a write-protected page (W bit off in the translation buffer).

RAD Reference address that caused the page fail.

PMA Physical address corresponding to the reference address.

AD1 Undefined.

AD2 Undefined

2 Illegal age - An Illegal CST age was detected for a page during the processing of one of the following page fails:

1. CST update needed.

2. Write reference to a writable but not yet modified page.

RAD Reference address that caused the page fail.

PMA Physical address corresponding to the reference address

AD1 Undefined.

AD2 Undefined.

3 Address break - An address break occurred.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 Undefined.

AD2 Undefined.

4 Illegal super section pointer 0 - A pointer with type 5, 6, or 7 was found in super section table offset 0.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 Zero

AD2 The illegal super section pointer.

5 Section greater than 37 - In KL compatible mode, a virtual reference was made to a section greater than 37.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 -1,,offset in EPT/UPT of super section pointer.

AD2 Super section pointer.

6 Illegal pointer - A pointer with type 4, 5, 6, or 7 was found in the super section table, section table, or page table.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 Source of last word processed (see below).

AD2 The illegal pointer.

7 No access pointer - A no-access pointer was discovered during a pointer trace.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 Source of last word processed (see below).

AD2 The no-access pointer

10 Page not in core - A page-address word was discovered whose storage medium field (bits 12-17) was non-zero.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 Source of last word processed (see below).

AD2 Last pointer processed.

11 Illegal age - An Illegal CST age was detected for a page during a pointer trace.

RAD Reference address that caused the page fail.

PMA Undefined.

AD1 Source of last word processed (see below).

AD2 Last pointer processed

12 Must-be-zero bits non-zero - The microcode discovered bits that were declared "must be zero" to be non-zero.

RAD Address of word containing the MBZ bits.

PMA Undefined.

AD1 Undefined.

AD2 Undefined.

13 Illegal indirect - An extended effective address calculation has encountered an indirect word with 11 (binary) in bits 0 and 1.

RAD Address of word containing the illegal indirect.

PMA Undefined.

AD1 The illegal indirect word.

AD2 Undefined.

14 Illegal physical effective address word - A physical effective address word was discovered with a 1 in bit 0 or 1.

RAD Address of illegal physical effective address word.

PMA Undefined.

AD1 The illegal physical effective address word.

AD2 Undefined.

15 Illegal one-word-global byte pointer - A one-word-global byte pointer was discovered with a code of 77 (octal)

RAD Address of the illegal one-word-global byte pointer.

PMA Undefined.

AD1 Undefined.

AD2 The illegal one-word-global byte pointer.

40001 Uncorrectable memory error. In a processor reference to memory, the 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).

RAD Address of the word that caused the error.

PMA Undefined.

AD1 Undefined.

AD2 Undefined.

40002 NXM. 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).

RAD Address of the word that caused the NXM.

PMA Undefined.

AD1 Undefined.

AD2 Undefined.

40003 Nonexistent I/O register. The processor gave an I/O address to which there was no response.

RAD I/O address.

PMA Undefined.

AD1 Undefined.

AD2 Undefined.

3.3.4.3.5.1.1 Additional data words for a pointer trace -

NOTE

Due to time-to-market considerations, this information may not be stored by the microcode in the FCS machine. If it is not stored, the contents of the words are undefined.

When the microcode detects a page fail condition during a pointer trace, it stores the source of the last word processed in additional data word 1 (454) and the last pointer fetched in additional data word 2 (455). Additional data word 2 is simply the last pointer processed by the microcode and may be a super section, section, or page pointer. Additional data word 1 specifies the source of the last word processed and may have one of the following forms:

0,,0 If the page fail code is "illegal super section 0 pointer", this word indicates that the pointer trace failed immediately after initialization. If the page fail code is anything else, it is really the following case.

0,,offset The last word examined was fetched from SPT+offset.

-1,,offset The last word examined was fetched from UPT+offset or EPT+offset. The user reference bit in the page fail word determines which.

page,,offset The last word examined was fetched from physical page "page", offset "offset".

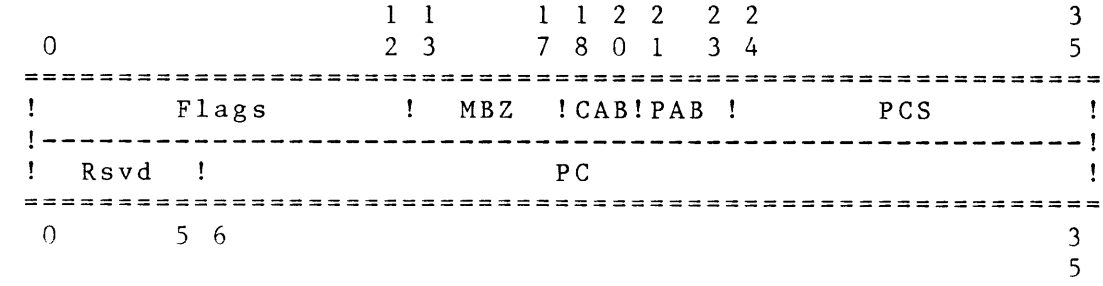
3.3.5 Process context variables

3.3.5.1 Introduction

In order to take advantage of the full 4096 section virtual address space implemented by the KD10 processor, the flag-PC double word format has been changed to allow for a larger section number. In addition, the PAB and CAB fields have been added.

3.3.5.1.1 New flag-PC double word

The format of the double word is as follows:



Where:

Flags — PC flags. The operation is identical to the PC flags on the KS10 with the exception that the Address Failure Inhibit flag occupies bit 8 as on the KL10.

MBZ — Must be zero

CAB — Current AC Block Number (0-7)

PAB — Previous Context AC Block Number (0-7)

PCS — Previous Context Section Number

PC — PC of the program

1. In kernel mode (XPCW/SFM), or when stored on a page fail or MUUO, all of the above fields will be stored as defined. In kernel mode, XJRSTF and XJEN will restore all fields.

2. In user mode, PCS, PAB, and CAB will always be stored as 0. An XJRSTF in user mode will treat these fields as it does the user mode and user I/O flag now (i.e. ignore them).

3.3.5.1.2 Context changing

Returning to a previous context may be done with an XJRSTF or XJEN instruction which restores the context variables stored in the previously saved PC double word.

Entering a new context will be done as follows: All of the "previous" context variables in the old PC flag word will be set to their corresponding values in the "current" context. If the "current" context is not user-mode, then set the "previous" context from the new PC flag word. The following operations are defined as entering a new context:

1. Monitor call (MUUO).
2. Page fail trap.
3. Priority interrupt initiation.
4. I/O page fail trap.

Each of these operations will store a PC double-word containing the "current" context variables and then load a new PC double-word to set new values for those variables not set automatically.

The following chart summarizes what variables are saved, and what new values are set. It includes for comparison what is currently implemented on the KL10 processor.

Key:

Store Save in appropriate block (old)

Load Set from appropriate block (new)

Set Set "previous" to old "current"

* In process context word

** Ucode sets PCS; XPCW stores flags, PC, PCS, and loads flags and PC

		Flags	PC	CAB	PAB	PCS/PCU
XPCW	KL	Store	Store	No	No	Store
		Load	Load	No	No	No
	KD	Store	Store	Store	Store	Store
		Load	Load	Load	No	No
Interrupt	KL	Store	Store	No	No	Store
		Load	Load	No	No	Set(PCS)
	KD	Store	Store	Store	Store	Store
		Load	Load	Load	No	No
MUUO	KL	Store	Store	* Store	* Store	Store
		Clear	Load	No	No	Set (PCS)
	KD	Store	Store	Store	Store	Store
		Load	Load	Load	Load	Set
Page Fail	KL	Store	Store	No	No	Store
		Clear	Load	No	No	Set(PCS)
	KD	Store	Store	Store	Store	Store
		Load	Load	Load	Load	Set
LUUO	KL	Store	Store	No	No	No
		No	Load	No	No	No
	KD	Store	Store	No	No	No
		No	Load	No	No	No
XJRSTF	KL	No	No	No	No	No
		Load	Load	No	No	Load
XJEN	KD	No	No	No	No	No
		Load	Load	Load	Load	Load

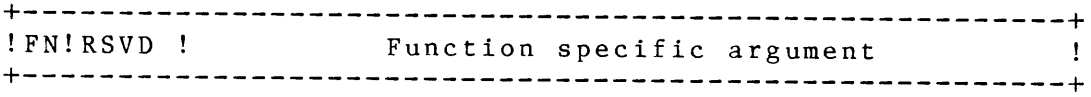
3.3.6 Trap handling

Trap handling has been changed considerably from the KL10 in that traps on the KD10 are processed via a trap function word rather than the execution of an instruction. The trap function word indicates how the trap is to be processed and provides the address of a function-specific block to be used as part of the processing.

3.3.6.1 Trap Function Word

EPT/UPT locations 421-423 contain a trap function word that determines the action of the processor when it detects an arithmetic overflow, stack overflow, or trap 3 condition.

The format of each word is as follows:



The format of this word is as follows:

0-1 Function code. This field is interpreted as follows:

- 00 Do nothing on trap condition (ignore)
- 01 Execute MUUO (take new PC from function specific argument)
- 10 Transfer control to exec/user depending on the mode in which the trap occurred. This function uses a LUUO-like block as described in the function specific argument below.
- 11 Reserved.

2-5 Available to software

6-35 Function specific argument. This field is used in a manner specific to the function performed as follows:

- 0 Ignored for this function.
- 1 New PC for the MUUO.

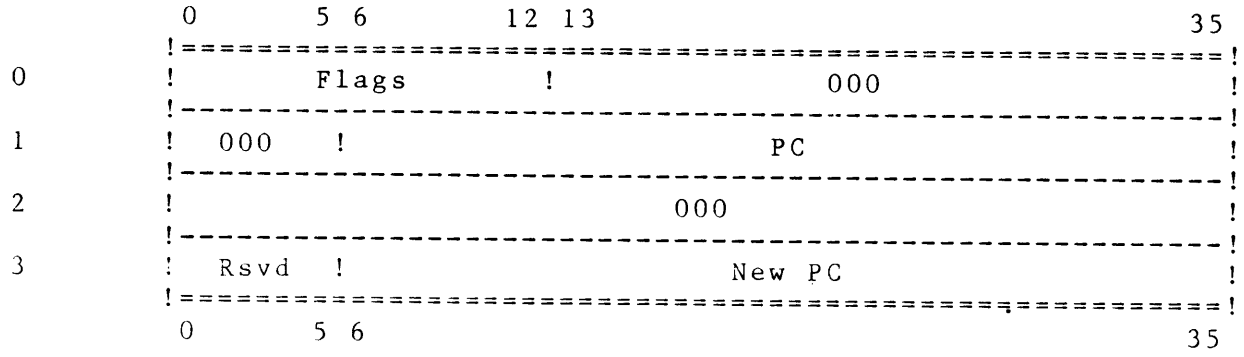
  This function stores only the program flags, CAB, PAB, PCS and the PC in UPT locations 424-425. The opcode, AC, and effective address of the instruction are NOT stored in UPT locations 426-427. The new program flags, CAB, and PAB are loaded from UPT location 430 as in a normal MUUO.

2 Virtual address in the current context (exec/user) of a 4 word LUUO-like block.

This function stores only the program flags and the PC in words 0-1 of the block. The opcode, AC, and effective address of the instruction are NOT stored in words 0 and 2 of the block. The new PC is then taken from the fourth word of the block.

3 Reserved.

The format of the LUUO-like block used in function 2 is as follows:



Notes

1. The trap 1 and trap 2 flags are never stored in the MUUO (function code 1) or LUUO-like (function code 2) blocks when a trap is processed. It is the responsibility of the program to determine which trap condition occurred by supplying different new PCs for each possible condition.

2. An instruction that causes a trap and also jumps (e.g., AOJA) stores the PC of the destination of the jump, not PC+1 of the jump instruction.

3.3.6.2 Trap enable

WREBR bits 7 and 8 affect how the processor handles traps, LUUOs, MUUOs, and page fails. If the monitor enables full processing of these conditions (by setting WREBR argument bits 7 and 8), the microcode will process these conditions as described above. If the monitor disables full processing of these conditions (the default power-up state of the machine), the microcode will process them differently as described below:

1. Traps. The microcode will treat trap 1, 2, and 3 conditions as if the trap function word had specified "ignore trap".

2. LUUOs. LUUOs executed in section zero (or in the low 256K with paging off) will be treated exactly as they are now, i.e., they will store the LUUO in location 40 and execute the instruction in location 41. Note that LINK stores a HALT instruction in location 41 when it loads programs.

   LUUOs executed in non-zero sections will halt the machine.

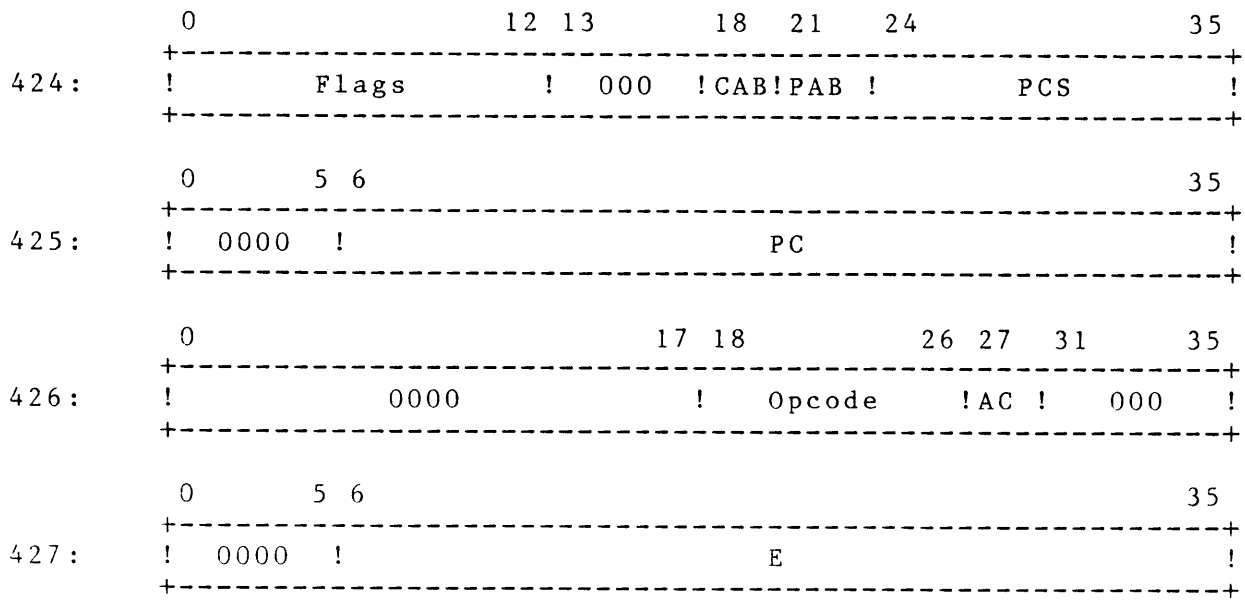
3. MUUOs. MUUOs will halt the machine.

4. Page fails. Page fails that must be processed by the monitor will halt the machine. Page fails that can be resolved entirely by the EBOX microcode will continue to be processed normally.

This special handling will cause the machine to halt when a condition for which the program is unprepared occurs instead of doing something unexpected. As a result, conditions for which the monitor is unprepared to handle will be detected early as the result of the condition instead of as a by-product of the condition.

3.3.7 MUUO handling

MUUO handling on the KD10 is radically different from that on the KS10, but identical to that on the KC10. Instead of the previous format of UPT locations 424-427, the following format is used to store the program flags, CAB, PAB, PCS, PC, Opcode, AC, and effective address of the MUUO:



The new current and previous AC blocks, and the new program flags are loaded from the word at UPT location 430. The new PC is taken from one of the words of the dispatch vector beginning at UPT location 432, based on the MUUO opcode and whether the MUUO was executed in user or executive mode. The dispatch vector consists of pairs of words, one for user and one for exec, and contains 5 separate MUUO dispatches plus words reserved for future expansion. The dispatches are as follows:

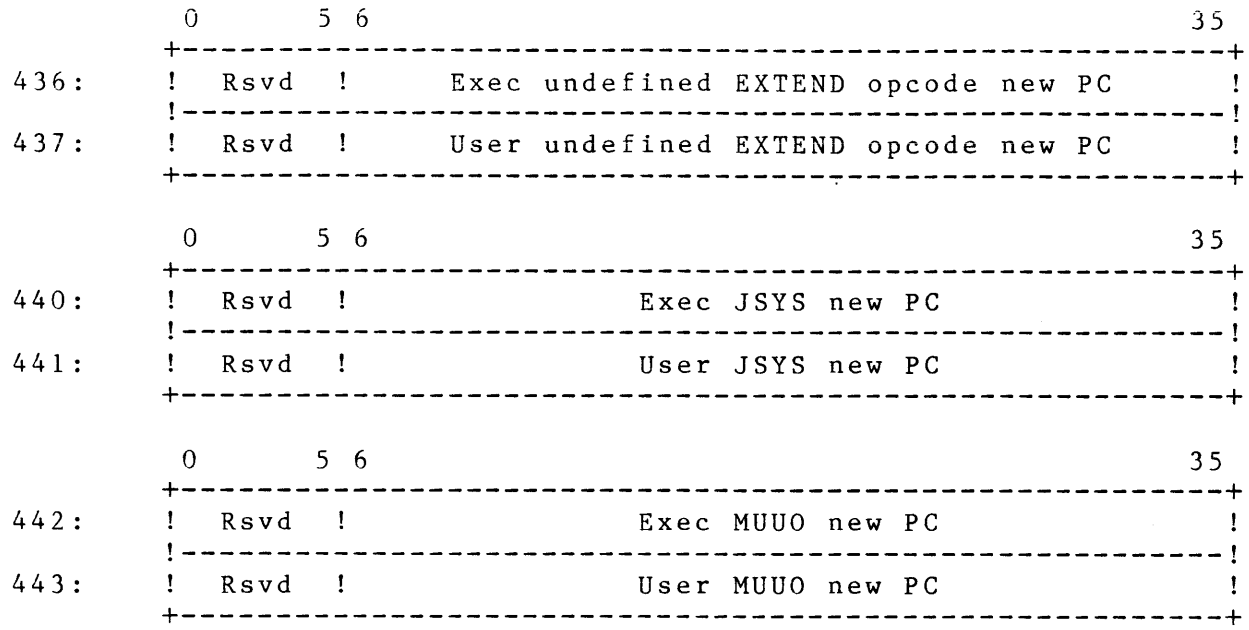
Offset	Use
432-433	Opcode 0 and all unassigned opcodes less than 700.
434-435	Unassigned opcodes in the range 700-777 plus any instruction that is executed in user mode without user I/O enabled that requires user I/O. This includes all internal and external I/O instructions, MAP, JRSTF executed in a non-zero section, JRST 3, HALT, XJEN, XPCW, JRST 10, JRST 11, JEN executed in a non-zero section or in user mode, JRST 13, JRST 16, and JRST 17,.
436-437	Undefined EXTEND opcodes
440-441	JSYS (opcode 104)
442-443	All other MUUO opcodes

The format of these words is as follows:

```
          0                  12          18  21  24                  35
         +-------------------------------------------------------------+
430:     !     New flags      !  Rsvd !CAB!PAB!        Rsvd           !
         +-------------------------------------------------------------+

          0      5 6                                                  35
         +-------------------------------------------------------------+
432:     !  Rsvd  !          Exec undefined opcode new PC              !
         !-------------------------------------------------------------!
433:     !  Rsvd  !          User undefined opcode new PC              !
         +-------------------------------------------------------------+

          0      5 6                                                  35
         +-------------------------------------------------------------+
434:     !  Rsvd  !        Exec undefined I/O opcode new PC            !
         !-------------------------------------------------------------!
435:     !  Rsvd  !        User undefined I/O opcode new PC            !
         +-------------------------------------------------------------+
```

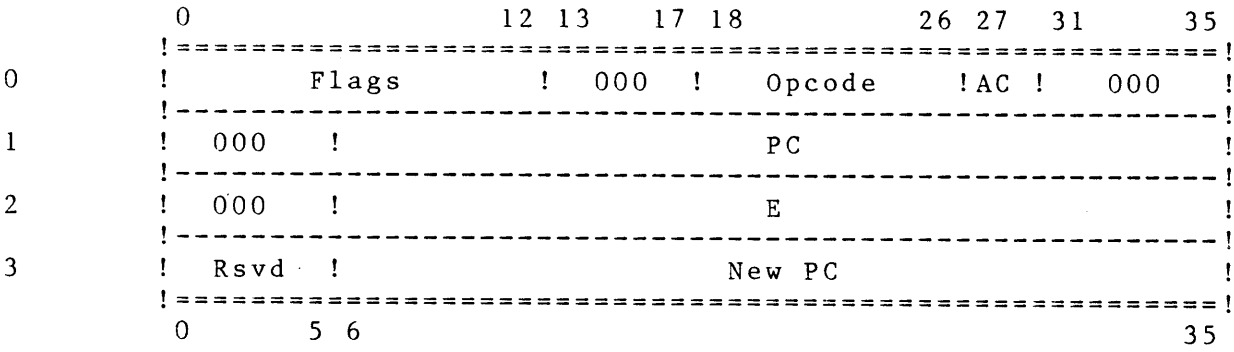

0 5 6 35
436: Rsvd Exec undefined EXTEND opcode new PC
437: Rsvd User undefined EXTEND opcode new PC
0 5 6 35
440: Rsvd Exec JSYS new PC
441: Rsvd User JSYS new PC
0 5 6 35
442: Rsvd Exec MUUO new PC
443: Rsvd User MUUO new PC

3.3.8 LUUO handling

If the program is running in section 0, store the opcode, AC, and the effective address in bits 0-8, 9-12, and 18-35 respectively of location 40; clear bits 13-17. Then execute the instruction contained in location 41. An LUUO executed in user mode uses virtual locations 40 and 41 in the user program. An LUUO executed in executive mode uses locations 40 and 41 in executive virtual address space. This action is identical to the KL10 implementation.

If the program is running in a nonzero section, use bits 6-35 of UPT location 420 if the program is running in user mode, or EPT location 420 if the program is running in exec mode, as the address of a block of four words. In the first three locations of the block, store the program flags, opcode, AC, effective address, and PC of the LUUO. Then take the next instruction from the location specified by bits 6-35 of the fourth word of the block. In user mode, this action is identical to the KL10 implementation. In executive mode, this action is different from what is currently documented, but identical to what the KL10 actually implements.

The format of the block is as follows:



3.3.9 Interrupt handling

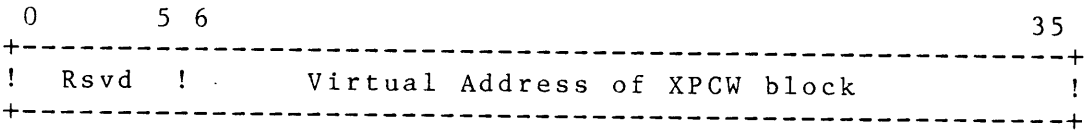
Interrupt handling on the KD10 is quite similar to interrupt handling on the KS10. The difference is that the processor doesn't execute an instruction to start an interrupt. Rather, it fetches an interrupt vector (from the same location that the KS10 would have executed) and uses that to start the interrupt.

If the interrupt was requested by an adapter, the microcode fetches a table address from EPT+100+adapter number, i.e., EPT+100 for adapter 0, EPT+101 for adapter 1, etc. The vector address returned by the device is divided by 4 and added to the table address to obtain the address of an interrupt vector word described below.

If the interrupt was generated by the processor (PI software request, APR condition, timer done, etc.), the microcode fetches the interrupt vector word from EPT+40+2n, where n is the level on which the internal device is requesting an interrupt. Therefore, a device requesting an interrupt on PI 1 would fetch the interrupt vector word from EPT+42, a device requesting an interrupt on level 2 would fetch the interrupt vector word from EPT+44, etc.

An interrupt vector is a 30-bit Exec Virtual Address pointing to a 4-word block that is similiar to a XPCW control block. Return from an interrupt should be made by an XJEN instruction that addresses the same block. The saving and restoring of the "previous" context is described in the section on process context variables. The new context will be set up from the XPCW control block. The action of an interrupt cycle will be as if an actual XPCW was executed with its EA taken from the appropriate location in the I/O page.

An interrupt vector has the following format:

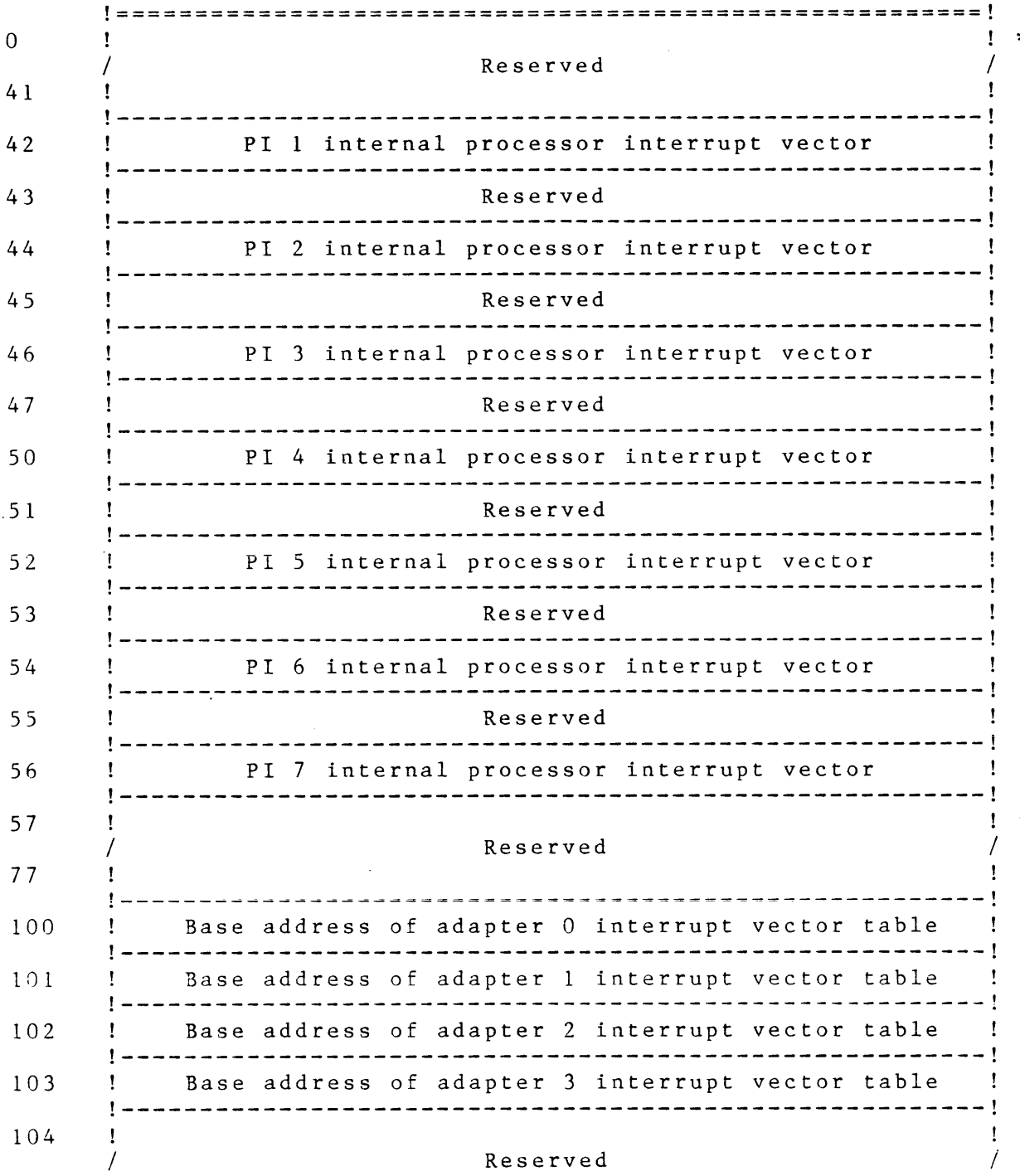


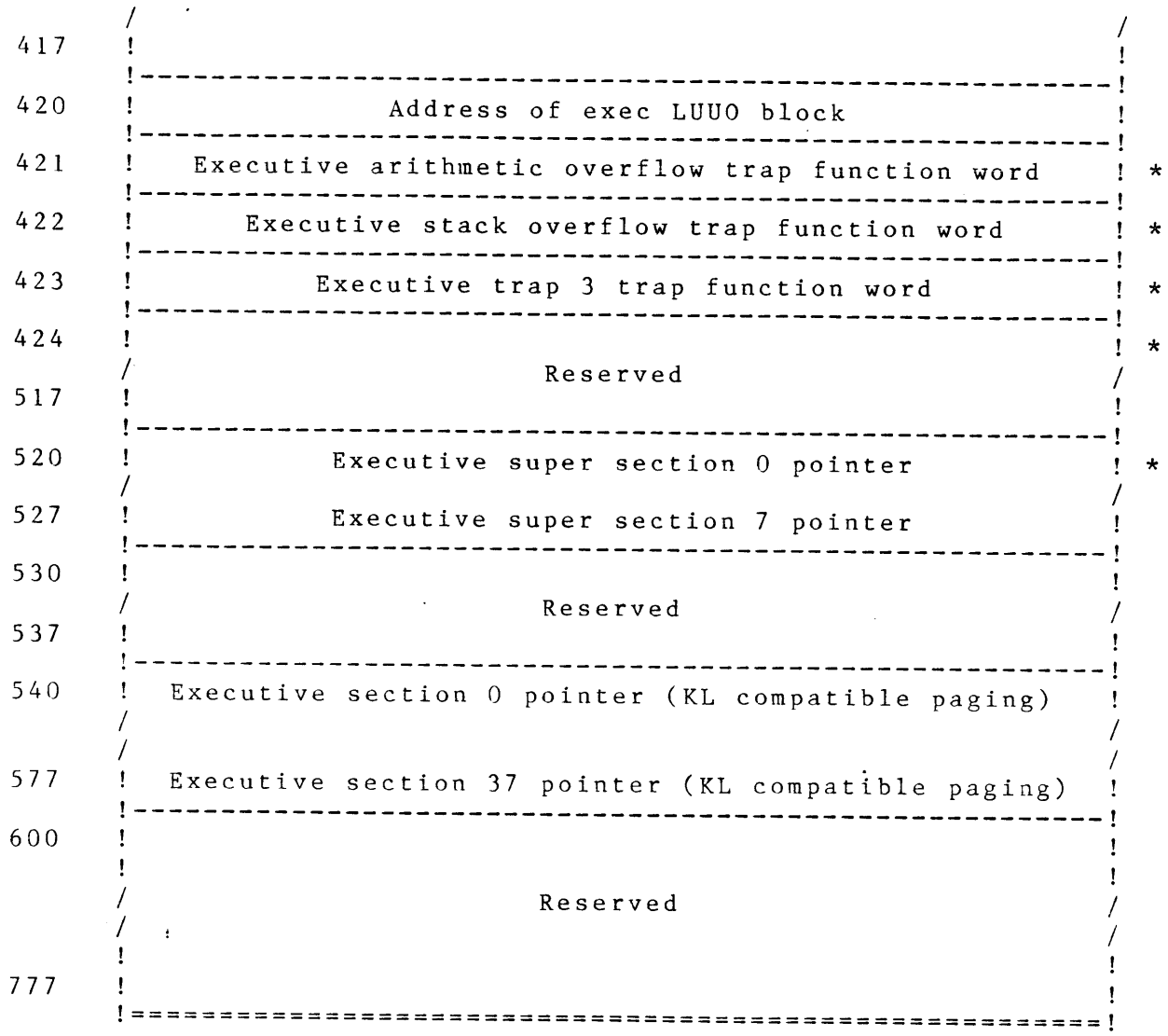
where bits 6-35 specify the exec virtual address of the XPCW block.

3.3.10 Summary of EPT and UPT formats

The following diagrams summarize the format of the EPT and UPT used on the KD10. The format is identical to those used on the KC10. All areas that differ from the KL10 are marked with an asterisk.

Executive process table configuration





These locations are described in more detail on the following page.

42-56 Internal interrupt vectors. The even locations in this range contain the interrupt vector words for each PI level. These locations are used for any internal processor interrupt conditions. for more information on the function of these words, see the section on interrupt handling.

100-103 I/O adapter interrupt table pointers. These locations contain the base address of a table of interrupt vector words for each adapter. For more information on the function of these words, see the section on interrupt handling.

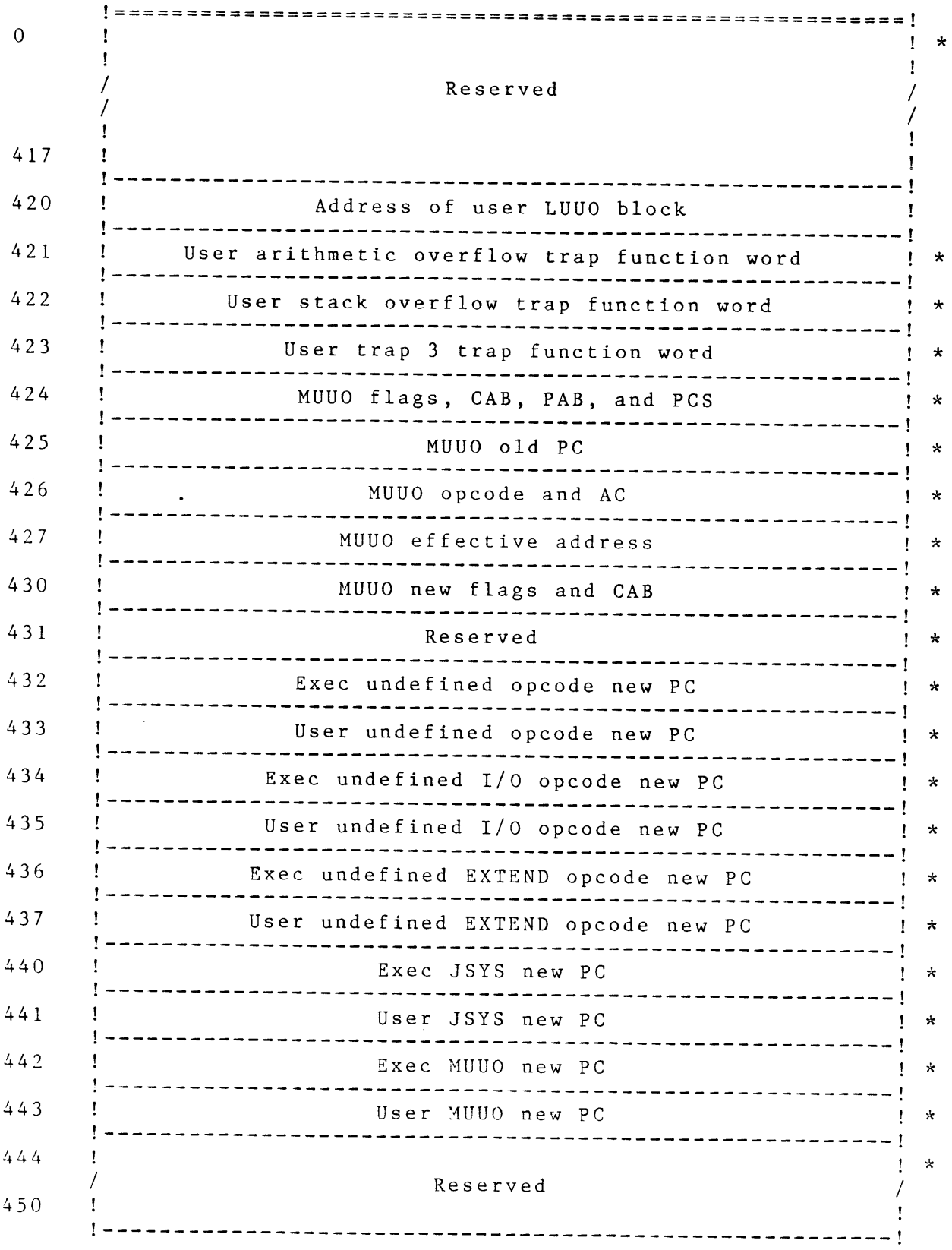
420 Address of exec LUUO block. Exec LUUOs executed with PC section non-zero are processed through the four-word LUUO block whose 30-bit virtual address is contained in this word. For more information on the format of the four-word block, see the section on LUUO handling.

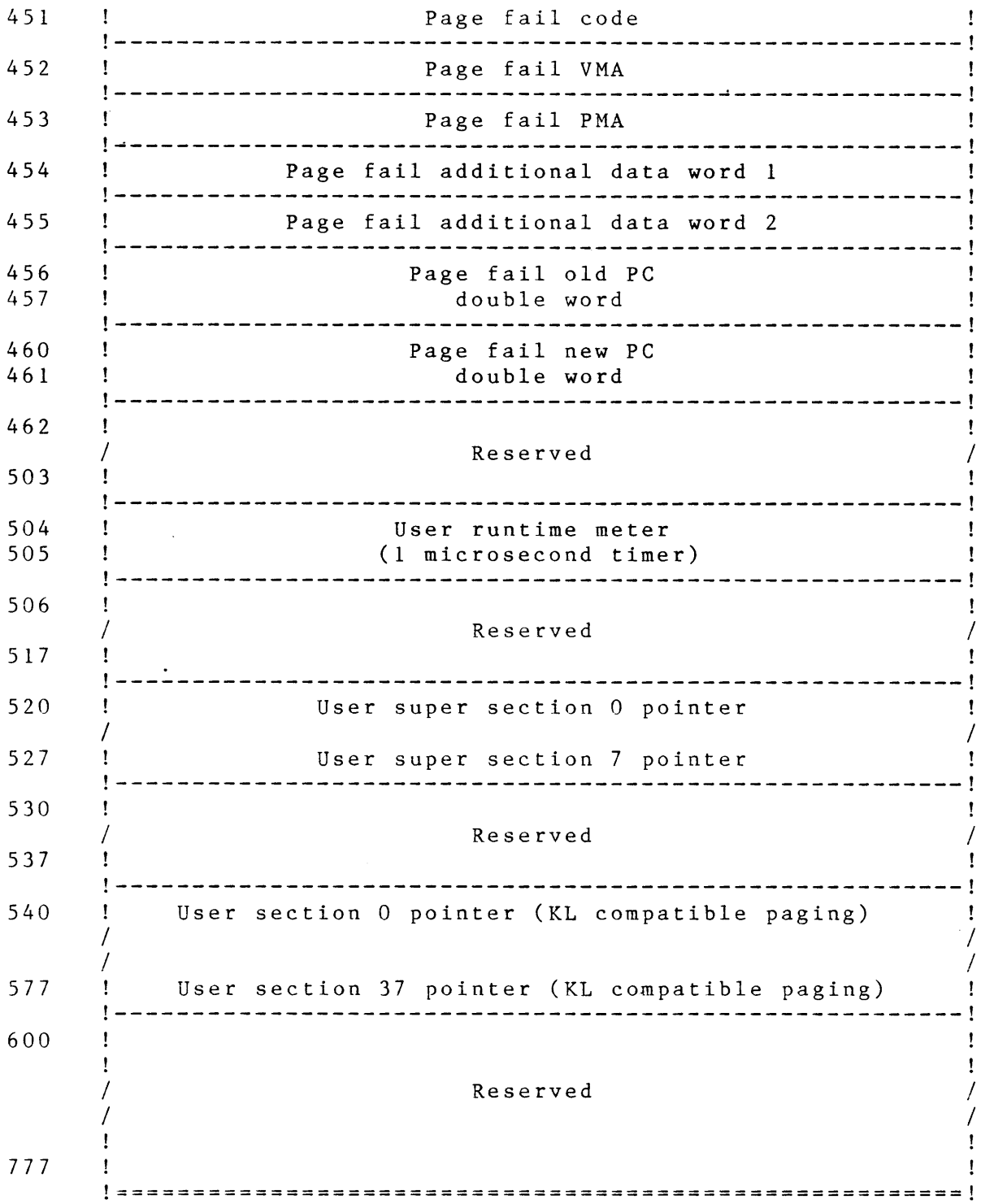
421-423 Exec trap function words for trap 1, 2, and 3. These function words are interpreted to process exec trap 1 (arithmetic overflow), trap 2 (pushdown list overflow), and trap 3 exceptions. For more information on the format of a trap function word, see the section on trap handling.

520-527 Exec super section pointers. These words contain the super section pointers for exec super sections 0-7. For more information on the format of a super section pointer, see the section on Paging.

540-577 Exec section pointers. These words contain the section pointers for exec sections 0-37 when the processor is running with KL compatible paging enabled. For more information on the format of a section pointer, see the section on Paging.

User process table configuration





These locations are described in more detail on the following page.

420 Address of user LUUO block. User LUUOs executed with PC section non-zero are processed through the four word LUUO block whose 30-bit virtual address is contained in this word. For more information on the format of the four word block, see the section on LUUO handling.

421-423 User trap function words for trap 1, 2, and 3. These function words are interpreted to process user trap 1 (arithmetic overflow), trap 2 (pushdown list overflow), and trap 3 exceptions. For more information on the format of a trap function word, see the section on trap handling.

424-443 MUUO processing locations. These locations are used to process user and exec MUUOs. For the format of each word, see the section on UUO handling.

451-461 Page fail processing locations. These locations are used to process user and exec page fails. For the format of each word, see the section on paging.

504-505 User runtime meter. These locations contain the current value of the user runtime meter counter for this process. The user runtime meter is a one microsecond counter maintained by the hardware and microcode. For more information on the format of these words, see the KS10 section of the Processor Reference Manual.

520-527 User super section pointers. These words contain the super section pointers for user super sections 0-7. For more information on the format of a super section pointer, see the section on Paging.

540-577 User section pointers. These words contain the section pointers for user sections 0-37 when the processor is running with KL compatible paging enabled. For more information on the format of a section pointer, see the section on Paging.

3.3.11 Halt status

As in the KS10, the KD10 processor stores a halt status block when it halts. In addition, it stores the halt code in physical memory location 0 and the PC in physical memory location 1.

At machine power up, the microcode supplies a default address of 376000 for the start of the halt status block. The program can change this address with the WRHSB instruction. Note that no halt status will be stored if bit 0 of the WRHSB argument is a 1.

The contents of the halt status block are identical to that for the KS10, with one exception. Because VMA and VMA flags will no longer fit in a single word, the KD10 stores the VMA flags in an additional word. The format of the block is:

```
     !=================================================!
00:  !                        MAG                      !
     !-------------------------------------------------!
01:  !                         PC                      !
     !-------------------------------------------------!
02:  !                         HR                      !
     !-------------------------------------------------!
03:  !                         AR                      !
     !-------------------------------------------------!
04:  !                        ARX                      !
     !-------------------------------------------------!
05:  !                         BR                      !
     !-------------------------------------------------!
06:  !                        BRX                      !
     !-------------------------------------------------!
07:  !                        ONE                      !
     !-------------------------------------------------!
10:  !                        EBR                      !
     !-------------------------------------------------!
11:  !                        UBR                      !
     !-------------------------------------------------!
12:  !                        MASK                     !
     !-------------------------------------------------!
13:  !                        FLG                      !
     !-------------------------------------------------!
14:  !                         PI                      !
     !-------------------------------------------------!
15:  !                        XWD1                     !
     !-------------------------------------------------!
16:  !                         T0                      !
     !-------------------------------------------------!
17:  !                         T1                      !
     !=================================================!
20:  !                        VMA                      !
     !-------------------------------------------------!
21:  !                  PC/VMA flags                   !
     !=================================================!
```

Where offset 20 (VMA) contains the local/global flag in bit 0 and the

VMA in bits 6-35.

Offset 21 (PC/VMA flags) contains the PC flags in bits 0-17, PI NEW 04, 02, and 01 in bits 19-21, and the VMA flags in bits 22-35.

CHAPTER 4

EXTENDED ADDRESSING

This chapter provides a description of extended addressing as defined by the PDP-10 architecture. This material really belongs in the Processor Reference Manual, and every attempt will be made to get it included in the next release of the manual. Note that certain implementations of the PDP-10 architecture don't always conform to the descriptions given in the memo. These are descriptions of what SHOULD be implemented, not necessarily what IS implemented. However, all future PDP-10 processors should conform to these descriptions.

In order to make it easier for the reader, I've also added a lot of background, definitions, and descriptions of extended addressing that are found in other references. This additional discussion should make the overall structure of extended addressing more clear.

In order to avoid swamping the reader with too much detail at any point, I sometimes intentionally ignore or understate certain important aspects of the examples that I use. These items are generally covered later in the memo. I also occasionally forward reference topics. Because of this organization, it may be best to make a quick first pass through the memo to pick out the highlights and then go back and make a more detailed pass.

This memo assumes that the reader has at least a basic knowledge of the PDP-10 instruction set, the notation used to describe instructions, and the format of an instruction word. Readers who do not have this knowledge are referred to sections 1.4 through 1.6 of the Processor Reference Manual and to the Macro Assembler Reference Manual.

4.1 Reference materials

The primary source of information about the instruction set is the Processor Reference Manual. Unfortunately, there are some inaccuracies and some omissions in the sections related to extended addressing. The "Extended Effective Address Calculation" flow chart on page 1-30 of the PRM is the best "description" of the effective address calculation algorithms and it is attached to this memo for the convenience of the reader.

The KL10 Engineering Functional Spec contains several chapters related to this topic and has some interesting insights. Especially interesting are chapters 2.2, "User Interface to Extended Addressing", and 2.3, "Monitor Calling (MUUO, PXCT)". Along with these chapters is a hand-drawn flow chart by Tom Hastings entitled "Flow for Extended Addressing" that clears up several questions about EA-calc algorithms, especially in the area of PXCT. A copy of this flow chart is attached.

Old memos describing the design of extended addressing and the implementation of extended addressing in TOPS-20 are also somewhat helpful.

Finally, the KL10 microcode contains a few helpful comments about exception conditions in that implementation of extended addressing. It is in no sense "light reading", however.

4.2 Historical summary of extended addressing

PDP-10 processors prior to the model B KL10 implemented a virtual address space of 256K words. As programs and the operating systems grew, it became apparent that a virtual address space that was limited to 256K was insufficient for future expansion. Sometime in late 1973, an Extended Addressing Design Group was formed to evaluate proposals for increasing the virtual address space of the PDP-10. By early 1975, this group had agreed upon one proposal, and this proposal was documented in chapter 2.2 of the KL10 Engineering Functional Spec.

This proposal increased the size of the virtual address space from 256K words to 1 billion words by expanding the size of a virtual address from 18 bits to 30 bits. The virtual address space is logically divided into 4096 sections of 256K words each. The program may use these sections as separate logical entities or treat them as one large contiguous address space. Instructions, however, must explicitly transfer control between sections; they may not "fall" into the next section.

The increase in the size of the virtual address space was accompanied by an increase in the size of PC, from 18 to 30 bits. This increase allowed a program to execute in any of the extended sections. The contents of bits 6-17 of PC were termed the "PC section".

In order to allow an instruction to specify a full 30-bit virtual address, the rules for indexing and indirection were modified when PC section was non-zero. In addition, new instructions were defined to allow a program to jump to other sections.

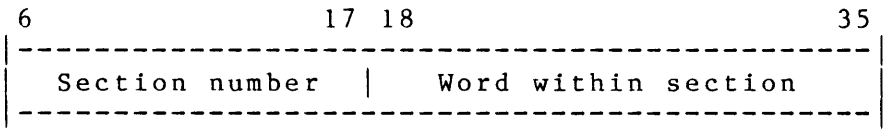
To insure compatibility with programs written for non-extended processors, section zero is treated exactly as it is on non-extended processors. This means that if a program is executing in section zero, nearly all instructions behave exactly as they would if the program were executed on a non-extended machine. Programs running in section zero cannot reference data in any other section (with one exception) and entry into another section is possible only with a few instructions (e.g., XJRSTF, XJRST, etc.).

The first processor to implement extended addressing was the model B KL10. Due to hardware restrictions, this processor implemented only 32 of the 4096 sections of virtual address space. References to virtual sections above the implemented range cause a page fail trap to the monitor. The KC10 implements the full 30-bit virtual address space.

4.3 Definition of terms

Before we start looking at extended addressing, let's define some terms:

- A **virtual address** is a 30-bit address used to reference a word in an address space. Although the address space can be considered to be one large, contiguous space, it is probably easier to consider it to be broken into sections of 256K words each. Bits 6-17 of the virtual address then specify the section number and bits 18-35 specify the word within the section. A virtual address looks like:



Virtual address format

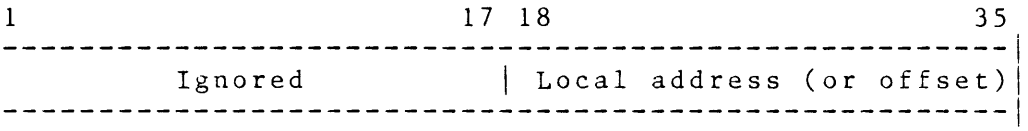
PC has the same format as a virtual address.

- An **address word** is a word containing I, X, and Y fields (see the PRM for definitions for these fields) in either IFIW or EFIW (see below) format. An effective address calculation takes such a word as input. Thus, instructions, indirect words, and byte pointers are all examples of address words.

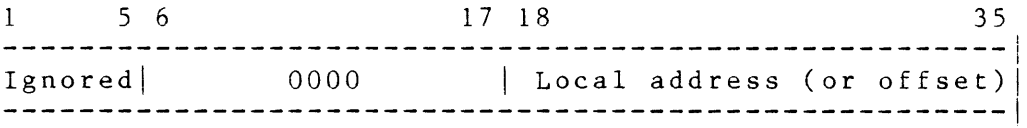
- A **local address** is an 18-bit in-section address that, when combined with a default section number, specifies a full 30-bit address. The section number is supplied by something other than the address word or index register.

- A **global address** is a 30-bit address that supplies its own section number. Therefore, no default section need be applied.

- A **local index** is an 18-bit displacement or address obtained from an index register used in an effective address calculation in section zero, or from an index register used in a non-zero section that has bit 0=1 or bits 6-17 equal zero. In a non-zero section, an index register containing a local index has one of the following formats:

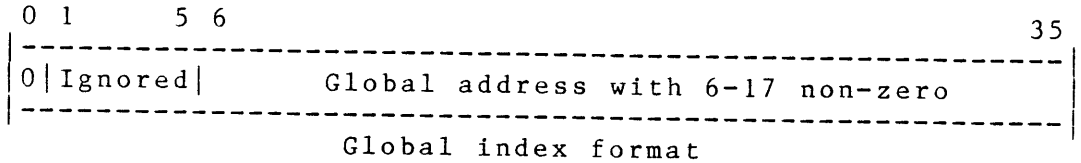


Local index format (bit 0 = 1)



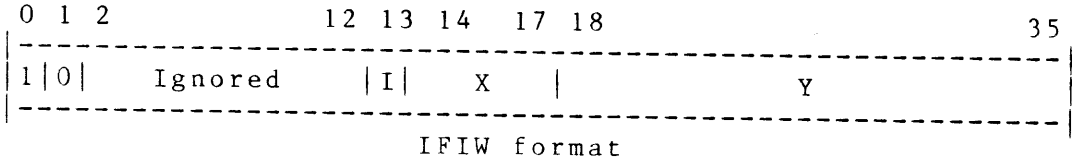
Local index format (bits 6-17 = 0)

- A **global index** is a 30-bit displacement or address obtained from an index register used in an effective address calculation in a non-zero section, that has bit 0=0 and bits 6-17 non-zero. An index register containing a global index looks like:



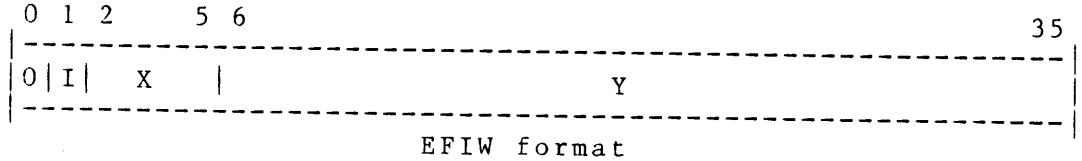
Global index format

- An **instruction format indirect word (IFIW)** is any indirect word in section zero, or an indirect word in a non-zero section that has bit 0=1 and bit 1=0 (instructions being executed are always interpreted in IFIW format). In this format, bit 13 is the indirect bit, bits 14-17 are the index register address, and bits 18-35 are the local memory address. An IFIW in a non-zero section looks like:



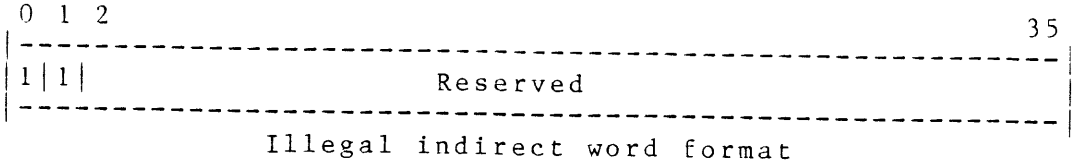
IFIW format

- An **extended format indirect word (EFIW)** is any indirect word in a non-zero section that has bit 0=0. In this format, bit 1 is the indirect bit, bits 2-5 are the index register address, and bits 6-35 are the global memory address. An EFIW looks like:



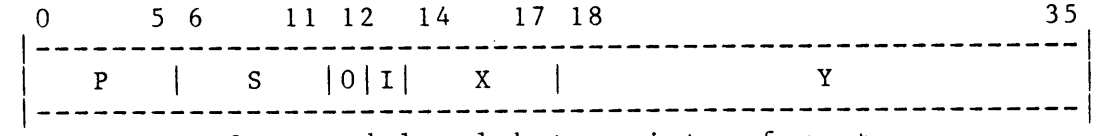
EFIW format

- An **illegal indirect word** is any indirect word in a non-zero section that has both bits 0 and 1 set to a 1. This type of indirect word is reserved for use by future hardware. If an EA-calc encounters this type of indirect word in a non-zero section, it will generate a page fail. The monitor cannot perform any user service as a result of this trap, including trapping to the user, since this would cause possible compatibility problems with future machines. An illegal indirect word looks like:



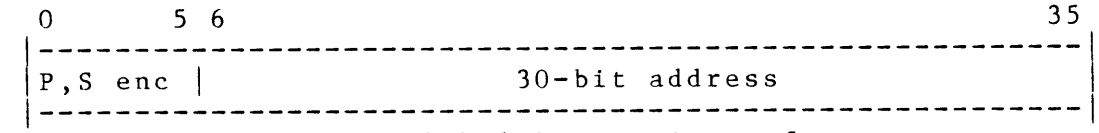
Illegal indirect word format

- A **one-word local byte pointer** is any byte pointer whose P field is less than or equal to 36 and that has bit 12=0. In this type of byte pointer, bits 13-35 have the same format as an IFIW, and bits 0-11 specify the size and position of the byte. A one-word local byte pointer looks like:



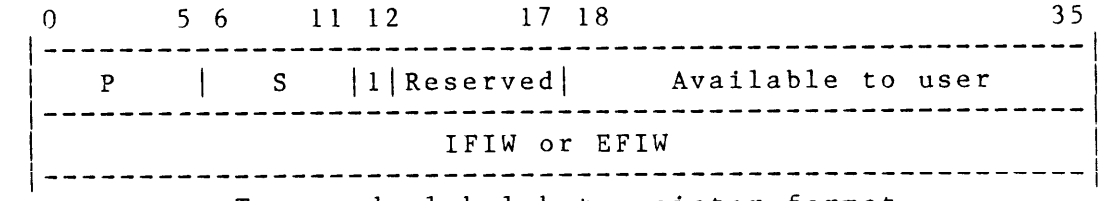
One-word local byte pointer format

- A **one-word global byte pointer** is any byte pointer whose P field is greater than 36. In this type of byte pointer, bits 0-5 are an encoded representation of the size and position of the byte and bits 6-35 supply a full 30-bit address of the word containing the byte. A one-word global byte pointer looks like:



One-word global byte pointer format

- A **two-word global byte pointer** is any byte pointer in a non-zero section whose P field is less than or equal to 36 and which has bit 12=1. As its name implies, this type of byte pointer consists of two words where bits 0-11 of the first word give the size and position of the byte and bit 12 must be a 1. The second word is either an IFIW or an EFIW and, when EA-calc'ed, supplies the address of the word containing the byte. A two-word global byte pointer looks like:



Two-word global byte pointer format

- A **local stack pointer** is any stack pointer in section zero, or a stack pointer in a non-zero section that has bit 0=1 or bits 6-17 equal zero before incrementing or decrementing (exactly like a local index). Incrementing or decrementing such a stack pointer will operate on both halves of the pointer independently, suppressing carries out of bit 18.

- A global stack pointer is a stack pointer in a non-zero section that has bit 0=0 and bits 6-17 non-zero before incrementing (exactly like a global index). Incrementing or decrementing such a stack pointer will treat the entire word as a 30-bit quantity.

4.4 Effective Address Calculations

No discussion of extended addressing is complete without talking about EA-calc's. An effective address calculation is performed on every instruction before it is executed. In addition, some instructions perform additional EA-calc's during the processing of the instruction (e.g. byte instruction EA-calc of the byte pointer).

4.4.1 Description of the EA-calc algorithm

The basic operation of an EA-calc is to process a so-called address word by adding the Y field of the word to the contents of the optional index register to compute a modified address. If the indirect bit is set in the address word, another word is fetched from the memory location addressed by the computed address and the entire process repeats until a word is found with the indirect bit not set. Sound simple? Well, let's look at the operation in a bit more detail.

The address word can be of two different formats, IFIW or EFIW (an instruction is treated as an IFIW when it is EA-calc'ed). In addition, an index can be of two different formats, local or global. Note that in section zero, all address words are IFIWs and all indices are local by definition. The complexity involved in the EA-calc algorithm is the result of these multiple formats.

Since the indirect bit simply causes another address word to be fetched and the EA-calc process to be repeated, we can fully characterize an EA-calc by looking at the combinations of IFIW, EFIW, and indices in local and global format. Let's look at these combinations one at a time.

4.4.1.1 No indexing

If no index register is specified in the address word, the EA-calc is strictly a function of the Y field in the address word. For an IFIW, the result is a local address. For example, both

```
1,,100/ MOVE 1,200
```

and

```
1,,100/ MOVE 1,@150
1,,150/ 400000,,200
```

compute a local effective address of 200. In the first case, the only address word is the instruction itself, which is treated as an implicit IFIW. In the second case, there are two address words, the instruction and the indirect word, and the indirect word is in the IFIW format.

For an EFIW, the result is a full 30-bit global address. For example,

```
1,,100/ MOVE 1,@[1,,200]
```

computes a global effective address of 1,,200 because the indirect word has a global format.

4.4.1.2 IFIW with local index

If the address word is an IFIW and the index is local, the result is a local address. The 18-bit address is computed by adding the Y field to the right half of the contents of the index register. For example:

```
1,,100/ MOVE 1,[-1,,10]
1,,101/ MOVE 2,@[400001,,200]
```

The indirect word has an IFIW format, so bits 14-17 specify the index register address. Since the contents of the index register are negative, it is a local index and the EA-calc is performed by adding the Y field (200) to the right half of the index register (10) to produce a local effective address of 210.

4.4.1.3 IFIW with global index

If the address word is an IFIW and the index is global, the result is a 30-bit global address. The address is computed by adding bits 6-35 of the contents of the index register to the value of the Y field, that has been sign-extended from bit 18 into bits 6-17. For example:

```
1,,100/ MOVE 1,[2,,10]
1,,101/ MOVE 2,-2(1)
```

The second instruction word has an implicit IFIW format, so bits 14-17 specify the index register address. Since the left half of the index register is positive non-zero, it is a global index and the EA-calc is computed by adding the Y field, after sign-extending it from bit 18 into bits 6-17 (7777,,-2), to bits 6-35 of the contents of the index register (2,,10), producing a global effective address of 2,,6.

Note that the sign extension allows Y to be used as a positive or negative constant offset to the global address in an index register. This offset is limited to +/- 128K.

4.4.1.4 EFIW with global index

If the address word is an EFIW, the index is always assumed to have the global format and the result is a 30-bit global address. The address is computed by adding bits 6-35 of the contents of the index register to bits 6-35 of the Y field. For example:

```
1,,100/ MOVE 1,[2,,10]
1,,101/ MOVE 2,@[010002,,200]
```

The indirect word has an EFIW format, so bits 2-5 specify the index register address. The index is always global, so the EA-calc is computed by adding the Y field (2,,200) to bits 6-35 of the contents of the index register (2,,10) to produce a global effective address of 4,,210.

4.4.1.5 References to section zero

Note that the only way to reference section zero from a non-zero section is via an EFIW format indirect word with bits 6-17 equal zero. Indexing alone cannot be used to reference section zero, because an index with bits 6-17 equal zero is treated as a local address to the section from which the last address word was fetched.

4.4.1.6 Summary of EA-calc rules

The preceding sections can be summarized by the table that follows. This table gives the computation done for all combinations of address words and index registers formats plus an indication as to whether the result is local or global.

Index Reg Type \ Address Word Type	IFIW	EFIW
None	Y[18:35] Local	Y[6:35] Global
Local	Y[18:35]+(XR)[18:35] Local	Not Defined (Actually the case below)
Global	Y[18]*7777,,Y[18:35]+(XR)[6:35] Global	Y[6:35]+(XR)[6:35] Global

4.4.2 Results of an EA-calc

When the microcode performs an EA-calc, it is simply following the rules described above and shown graphically in the EA-calc flow chart from the PRM. The result of this EA-calc is a 30-bit address and a

1-bit flag that indicates the address is local or global. These two pieces of information must be considered together whenever the results of the EA-calc are used; it is seldom, if ever, correct to consider the address without also considering the local/global bit.

Every EA-calc carries a default section along during the calculation of the effective address. The initial default section for an EA-calc of an instruction is PC section. More generally, the default section is initially that from which the first address word was fetched. This default section is changed from the initial value if the EA-calc follows a global address into another section. In fact, the default section is always the section from which the last address word was fetched.

If a local address is calculated using the rules given above, the default section is applied to complete the 30-bit address. If a global address is calculated, the default section is not used.

The last iteration of the EA-calc (the computation done on the last address word that doesn't have the indirect bit set) determines whether or not the result of the EA-calc is local or global. If the result of the last iteration is a local address, the result of the EA-calc is local. Similarly, if the result of the last iteration is global, so is the entire EA-calc. The transitions of the local/global flag are indicated on the PRM flow chart by notations such as "E Global".

The significant thing to remember is that a local EA-calc still results in a 30-bit address, even though 12 bits (the section number) were not explicitly supplied to the EA-calc routines as part of an address word or an index register.

- **An effective address calculation always computes 31 bits of information: a 30-bit address, and a 1-bit local/global flag.**

4.4.3 Simple EA-calc examples

In the examples above, we ignored the fact that EA-calc's always produce a 30-bit address when we said that the result was a local address n. In the following examples, we emphasize that a full 30-bit address is produced. Consider the following instruction:

```
0,,200/ MOVE 1,100
```

The EA-calc for this instruction results in a local EA. Therefore, the EA-calc computes the 30-bit address as 0,,100 and the 1-bit local/global flag as local. Since the EA is local, we know that the section number was defaulted from something, in this case, the PC section. We say that the effective address is 0,,100 LOCAL (this notation is used throughout the rest of this discussion to specify all 31 bits of information).

Let's consider a slightly more complex example:

```
1,,200/ MOVE 1,@300

1,,300/400000,,100
```

As in the previous example, the effective address calculation computes a local address of 100. Since the address word was fetched from section 1, the result of the EA-calc is 1,,100 LOCAL.

Let's look at a global EA-calc:

```
1,,100/ MOVE 1,@[2,,200]
```

In this case, the effective address calculation produces a global address of 2,,200 GLOBAL and no default section need be applied.

4.5 Use of the local/global flag

There are two uses for the local/global flag. First, it is used to determine if the address is actually an AC. If the address is local, and bits 18-35 are in the range 0 to 17, inclusive, the address references an AC, independent of bits 6-17. This means that a program can reference the ACs while running in any section, as long as the reference is local.

Second, the local/global flag determines how to increment or decrement the address. If the address is local, incrementing or decrementing it suppresses carries from bit 17 to bit 18 and vice versa. That is, the address always wraps around in the current section if the right half is incremented past 2^18-1 or decremented past 0. A global address is handled as a full 30-bit quantity and overflow or underflow of the right half can affect the left half section number.

4.5.1 AC references

Let's look at several examples that make use of the local/global flag. First, let's compare what happens to AC references for local and global effective addresses.

```
2,,100/ MOVE 1,@[400000,,5]
```

The EA-calc for this instruction yields 2,,5 LOCAL, where the section number was defaulted to 2. Is this memory location 2,,5 or AC 5? Because the EA-calc is local, the rule says that it is an AC reference and not a memory reference. On the other hand, the EA-calc for

```
2,,100/ MOVE 1,@[2,,5]
```

results in an EA of 2,,5 GLOBAL. Since the EA is global, this is a memory reference and not an AC reference.

- EA-calc's which yield local addresses, where bits 18-35 of EA are in the range 0-17, inclusive, always refer to the ACs independent of the section number.

Finally, there is the concept of "global AC address". This concept allows a program running in any non-zero section to make a global reference to the ACs by computing a global address in the first 16 (decimal) locations of section 1. Consider the following example:

```
2,,100/ MOVE 1,@[1,,5]
```

The EA-calc yields 1,,5 GLOBAL and because of the "global AC address" rule, the reference is to AC 5 instead of memory location 1,,5.

- An EA-calc which yields a global address to locations 0-17, inclusive, of section 1, refers to the ACs and not to memory. Such an address is called a global AC address.

4.5.2 Incrementing EA

Another use for the local/global flag computed as the result of an EA-calc is to determine how to increment the effective address. Let's look at two examples using DMOVE, one computing a local EA and one computing a global EA.

```
2,,100/ DMOVE 1,@[400000,,777777]
```

The EA-calc for this instruction results in an effective address of 2,,777777 LOCAL. The DMOVE instruction fetches two contiguous words from E and E+1, but what is E+1 in this case? Since the EA-calc resulted in a local address, incrementing E is done section-local, resulting in 2,,0 LOCAL for E+1. But this is a local reference to the ACs, so the two references for E and E+1 go to 2,,777777 (memory) and 2,,0 (AC). Note that the state of the local/global flag is maintained during the incrementing of EA.

- Incrementing or decrementing a local address is always done relative to the original section, i.e., the addresses "wrap around" in section.

- Incrementing a local address whose in-section part is 777777 causes the address to wrap around into the ACs.

Let's look at the corresponding global case:

```
2,,100/ DMOVE 1,@[2,,777777]
```

In this case, the EA-calc yields 2,,777777 GLOBAL. Because this is a global address, incrementing E to get the second word results in a reference to 3,,0 GLOBAL. Since this isn't a local address, the reference is made to memory location 3,,0 and not to AC 0.

- Incrementing or decrementing a global address affects the entire address; i.e., section boundaries are ignored.

- The process of incrementing or decrementing an address, whether the address is local or global, preserves the state of the local/global flag.

4.6 Multi-section EA-calc's

So far we have considered only EA-calc's that remain in one section. If the program is running in a non-zero section, a global quantity encountered during the EA-calc (from either an index register or indirect word) can cause the EA-calc to "change sections". An example will make this more clear:

```
3,,100/ MOVE 1,@[200002,,100]
2,,100/ 3,,200
```

The EA-calc for this instruction computes a global address of 2,,100 from the indirect word in the literal. Since the indirect bit is set in this word (bit 1 is the indirect bit in an EFIW), the EA-calc routine fetches the word at 2,,100 and continues the EA-calc. The final result of the EA-calc yields 3,,200 GLOBAL. This isn't a very interesting example, because it doesn't demonstrate the significance of the section change, so let's look at a slightly different example:

```
3,,100/ MOVE 1,@[200002,,100]
2,,100/ 400000,,200
```

In this example, the first part of the EA-calc remains the same and the routine fetches the word at 2,,100. In this case, however, the result of the EA-calc yields a local address instead of a global one. But what section is the address local to? The rule says that a local address is always local to the section from which the address word was fetched. Since the EA-calc changed from section 3 to section 2 when the last address word was fetched, the EA-calc is relative to section 2 and the EA-calc yields 2,,200 LOCAL.

- **The default section for a local address is always that from which the address word was fetched.**

Now that we've seen what happens to EA-calc's that cross section boundaries, let's see what happens if the EA-calc enters section zero:

```
3,,077/ MOVEI 3,1
3,,100/ MOVE 1,@[200000,,100]
0,,100/ 3,,200
```

As with the example above, the EA-calc for this instruction fetches the word at 0,,100 and continues. But since the EA-calc entered section zero, this word is treated as an IFIW instead of an EFIW. Therefore, the 3 in the left half of 0,,100 is interpreted as the index register field instead of a global section number. Since AC 3 contains a 1, the EA-calc yields 0,,201. In addition, the last address word was fetched from section zero, so the result is a local address.

- An effective address calculation which "falls" into section zero always results in an effective address that is local (to section zero). Furthermore, the effective address calculation can never "get out" of section zero once it enters it because all addresses in section zero are treated as local. Further operations obey section zero rules.

4.7 Special case instructions

Other than modifications to the EA-calc algorithms when the PC is in a non-zero section, most instructions are unaffected by the addition of extended addressing. However, there are a few classes of instructions that behave differently on an extended machine from the way they would on a non-extended machine. This section describes the behavior of each class of instruction that has this characteristic.

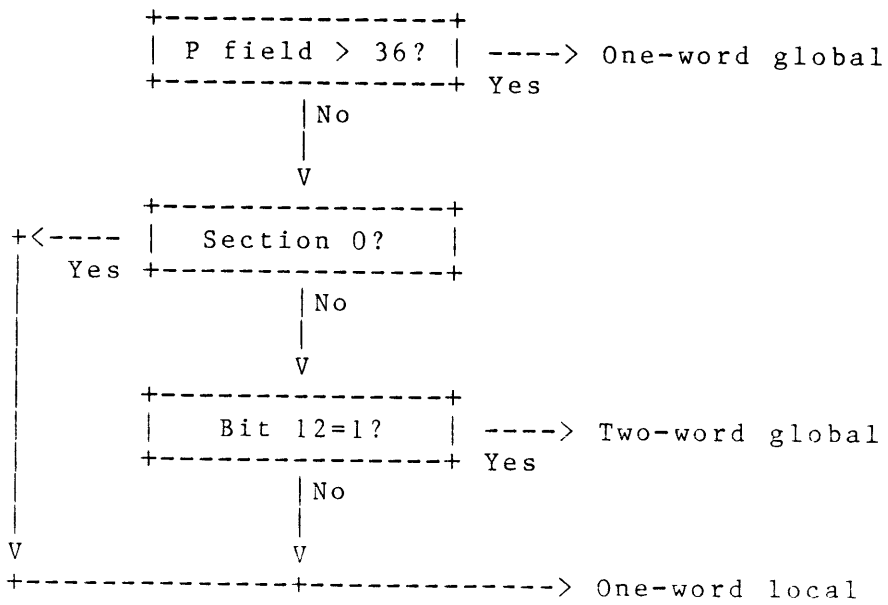
Examples in this section sometimes use the POINT pseudo-op to describe a byte pointer. For those readers who do not know what this pseudo-op generates, a description can be found in the Macro manual.

4.7.1 Byte instructions

The effective address calculation for a byte instruction addresses the byte pointer word(s). The instruction then does another EA-calc on the byte pointer after determining which one of the three possible byte pointer formats was supplied.

4.7.1.1 Byte pointer interpretation

The algorithm for determining the type of the byte pointer is as follows:



Byte pointer decode algorithm

The "Section 0?" test in the flow chart is based on where the first word of the two-word global byte pointer was fetched from and not on PC section. This is an important distinction if the byte instruction and the byte pointer are not in the same section.

- For byte instructions, the test for the possibility of a two-word global byte pointer is done based on the section from which the first word of the byte pointer was fetched. That is, if the section from which the first word of the byte pointer was fetched is non-zero, the byte pointer may be global.

4.7.1.2 Byte pointer EA-calc

The default section for the byte pointer EA-calc is initially that from which the byte pointer was fetched. Once again, this may be different from PC section if the instruction and byte pointer are in different sections. If we realize that the byte pointer is really an address word, this is an extension of the rule that says local addresses are local to the section from which the address word was fetched. For example:

```
3,,100/ LDB 1,@[2,,100]
2,,100/ POINT 6,200,0
```

In this example, the byte instruction is fetched from section 3. The EA-calc for the instruction follows an EFIW into section 2 and the byte pointer is fetched. The byte pointer is in one-word local format, so the EA-calc of the byte pointer results in a local address. But is the address local to section 3 (section containing the byte instruction) or 2 (section containing the byte pointer)? The rule says that byte pointer EA-calc's start off local to the section from which the byte pointer was fetched, so the EA-calc is local to section 2. The result of the EA-calc is therefore 2,,200 LOCAL.

Note that, while the initial default section may be that containing the byte pointer, the default section may change if the EA-calc encounters a global quantity. For example:

```
3,,100/ LDB 1,@[2,,100]
2,,100/ POINT 6,@[200004,,100],0
4,,100/ 400000,,200
```

As in the previous example, the byte pointer is fetched from section 2. The byte pointer has the indirect bit set, so the byte pointer EA-calc follows the EFIW in the literal (which also has the indirect bit set) into section 4, where the final address word is fetched from location 4,,100. This final address word is an IFIW, so the result of the EA-calc is a local address. Even though the byte pointer EA-calc started in section 2, the result of the EA-calc is local to section 4, because that's where the last address word was fetched from. The byte pointer EA-calc results in an effective address of 4,,200 LOCAL.

- For byte instructions, the initial default section for the byte pointer EA-calc is the section from which the byte pointer was fetched, which may not be the same section as that containing the byte instruction. Further, if the EA-calc results in a local address, the address is local to the section from which the last address word in the effective address calculation was fetched.

4.7.2 EXTEND instructions

Like the byte instructions, certain EXTEND instructions perform another EA-calc for the byte pointer (MOVSxx, CMPSxx, CVTBDx, CVTDBx, and EDIT). The AC field of the EXTEND instruction addresses a block of ACs, that contain the byte pointers. In addition, some EXTEND instructions perform an EA-calc on the extended opcode word, which is interpreted in IFIW format. The extended opcode word is addressed by the effective address of the EXTEND instruction.

4.7.2.1 Byte pointer interpretation

The algorithm for determining the byte pointer format is the same as that described for byte instructions with one exception. For EXTEND instructions, the "Section 0?" test in the flow chart is based on PC section.

- For EXTEND instructions, the test for the possibility of a two-word global byte pointer is done based on PC section. That is, if PC section is non-zero, the byte pointers may be global.

4.7.2.2 Byte pointer EA-calc

The default section for the byte pointer EA-calc is initially PC section even if other parts of the EXTEND instruction are in other sections. For example:

```
3,,100/ MOVEI 1,5                ;Source length
3,,101/ MOVE 2,[POINT 7,200]     ;Source byte pointer
3,,102/ MOVEI 4,5                ;Destination length
3,,103/ MOVE 5,[POINT 7,300]     ;Destination byte pointer
3,,104/ SETZB 3,6                ;Clear 2nd word of BPs
3,,105/ EXTEND 1,@[2,,100]

2,,100/ MOVSLJ                   ;Extended opcode is MOVSLJ
2,,101/ 0                        ;Fill character is 0
```

In this example, the EXTEND instruction is in section 3 and the EA-calc of the instruction follows an EFIW into section 2. The EA-calc's for the one-word local byte pointers in ACs 2 and 5 generate local addresses of 200 and 300 respectively. But are they local to section 3 (PC section) or to section 2 (section containing the extended opcode)? Because the byte pointers are fetched from the ACs, which are implicitly in PC section, the EA-calc is relative to PC section. Once again, this is a conceptual extension to the rule that local addresses are local to the section from which the address word (in this case, the byte pointer) was fetched.

As with byte instructions, the default section of the EA-calc may change if the EA-calc encounters a global quantity. An example of this for the EXTEND instruction would be analogous to that for byte instructions given above.

- For EXTEND instructions, the initial default section for the byte pointer EA-calc is PC section.

One interesting aspect of this rule is demonstrated by the following example:

```
3,,100/ MOVEI 1,5                ;Source length
3,,101/ MOVE 2,[POINT 7,200]     ;Source byte pointer
3,,102/ MOVEI 4,5                ;Destination length
3,,103/ MOVE 5,[POINT 7,300]     ;Destination byte pointer
3,,104/ SETZB 3,6                ;Clear 2nd word of BPs
3,,105/ EXTEND 1,@[0,,100]

0,,100/ MOVSLJ                   ;Extended opcode is MOVSLJ
0,,101/ 0                        ;Fill character is 0
```

In this example, the EXTEND instruction is in a non-zero section (3) and the extended opcode is in section zero. Even though part of the processing of the instruction fell into section zero, the EA-calc of the byte pointers is still done relative to PC section. Hence, the result is the same as in the previous example.

4.7.2.3 Extended opcode EA-calc

Some EXTEND instructions also perform an EA-calc on the extended opcode word. In this case, the default section for the EA-calc is initially the section from which the extended opcode word was fetched. For example:

```
3,,100/ MOVEI 1,5               ;Source length
3,,101/ MOVE 2,[POINT 7,200]    ;Source byte pointer
3,,102/ MOVEI 4,5               ;Destination length
3,,103/ MOVE 5,[POINT 7,300]    ;Destination byte pointer
3,,104/ SETZB 3,6               ;Clear 2nd word of BPs
3,,105/ EXTEND 1,@[2,,100]

2,,100/ MOVST 200               ;Extended opcode is MOVST
2,,101/ 0                       ;Fill character is 0
```

As in the last example, the EXTEND instruction EA-calc follows an EFIW into section 2 to fetch the extended opcode word from location 2,,100. In this example, the extended opcode turns out to be a MOVST which addresses a translation table with the result of the EA-calc of the word. This EA-calc results in a local address which is local to the section from which the address word was fetched. Therefore, the table is read from locations starting at 2,,200 LOCAL.

- The initial default section for the EA-calc of the extended opcode word under an EXTEND instruction is that from which the extended opcode word was fetched.

4.7.2.4 EDIT pattern and mark addresses

In addition to byte pointer type determination, the EDIT instruction under EXTEND interprets the pattern string and mark addresses differently based on PC section. If PC section is zero, both addresses are limited to 18-bit addresses in section zero and the result of setting bits 6-17 non-zero is undefined. Conversely, if PC section is non-zero, both addresses are treated as full 30-bit global addresses and no default sections are applied. An example of this is too complex to be given here and will be left as an exercise to the reader.

4.7.3 JSP and JSR

In a non-extended machine, these two instructions store the flags and an 18 bit PC before jumping to the effective address. This is also true if the instructions are executed in section zero of an extended machine. Because this format is insufficient to store a full 30-bit address, the operation of the instructions is modified when the PC is in a non-zero section. Instead of storing the flags and PC, these instructions store the full 30-bit PC (actually PC+1), omitting the

flags. For example:

```
2,,100/ JSP 1,200
```

stores 2,,101 in AC 1 before jumping to location 2,,200. Similarly,

```
2,,100/ JSR 200
```

stores 2,,101 in 2,,200 before jumping to location 2,,201. Note that for JSR, the PC is stored in the word addressed by the effective address even if that address is in another section, e.g.,

```
2,,100/ JSR @[3,,200]
```

In this case, the EA-calc for the JSR results in an effective address of 3,,200 GLOBAL. Therefore, 2,,101 (PC+1) is stored in 3,,200 (EA) before jumping to 3,,201 (EA+1).

An interesting aspect of this is demonstrated by the following example:

```
2,,100/ JSP 1,@[0,,100]
```

Because the PC is in a non-zero section, the instruction stores 2,,101 in AC 1 and then jumps to location 0,,100. But an attempt to return to the caller in section 2 via the usual JRST (1) instruction would fail, because the EA-calc of the return instruction, done in section zero, would fail to produce a 30-bit global address. As a result, it is difficult to write a subroutine in section zero that can be called via JSP or JSR from an arbitrary section.

A final example illustrates the difference between a local and global EA for JSR:

```
2,,200/ JSR 777777
```

The EA-calc for this case results in a value of 2,,777777 LOCAL. Therefore, 2,,201 (PC+1) is stored in 2,,777777 (EA) and the destination of the jump is 2,,0 (EA+1 local). This is consistent with the rule that local addresses always wrap around in section when incremented.

The global analogy is as follows:

```
2,,200/ JSR @[2,,777777]
```

In this case, the result of the EA-calc is 2,,777777 GLOBAL so the instruction stores 2,,201 (PC+1) into location 2,,777777 (EA) as in the last example. The difference is in the destination of the jump. Because the effective address is global, incrementing it produces 3,,0 GLOBAL (EA+1 global) as the destination of the jump. See the section on instruction fetches below for additional information on these two cases.

- If PC is in a non-zero section, the JSP and JSR instructions store a full 30-bit PC in the appropriate place instead of storing flags and PC. This is true even if the destination of the jump is in section zero.

4.7.4 Stack instructions

In a non-extended machine (and an extended machine in section zero), the stack pointer typically contains a negative control count in the left half and an 18-bit address in the right half. Such a stack pointer is called a local stack pointer. Because this format is insufficient to hold a full 30-bit stack address, an additional format for stack pointers is allowable when the PC is in a non-zero section. In this format (called a global stack pointer), the stack pointer is positive, bits 6-17 are non-zero, and bits 6-35 of the word are interpreted as the global address of the stack.

If the stack pointer is in local format, the stack address is local to PC section. For example:

```
2,,100/ MOVE 17,[-100,,200]
2,,101/ PUSH 17,300
```

Because the left half of the stack pointer is negative, it is in local format. Therefore, the stack address is 2,,200 LOCAL, because the stack is local to PC section.

- Local stack pointers are always local to PC section.
- The test for the possibility of a global stack pointer is done based on PC section. That is, if PC section is non-zero, the stack pointer may be global.

Note that a PUSH-type stack operation done on a local stack pointer that has overflowed (i.e., the left half of the pointer has gone to zero) changes the stack pointer to global format.

The type of stack pointer also determines how the stack address is incremented or decremented. For example, consider the following:

```
2,,100/ MOVE 17,[-100,,777777]
2,,101/ PUSH 17,200
```

The stack pointer in this example is local, so the stack address is 2,,777777 LOCAL. When the PUSH instruction increments the pointer, it does so section-local, resulting in an incremented stack address of 2,,0 LOCAL (which actually references AC 0). The stack pointer would then look like -77,,0.

Let's look at the same example with a global stack pointer:

```
2,,100/ MOVE 17,[2,,777777]
2,,101/ PUSH 17,200
```

With a global stack pointer, the increment is done globally, resulting in an incremented stack address of 3,,0 GLOBAL (which is memory location 0 in section 3). The stack pointer would then look like 3,,0.

- Incrementing or decrementing a local stack pointer wraps around in section. Conversely, the same operation on a global stack pointer may cross section boundaries.

In addition to the requirement for a global stack pointer to specify a full 30-bit stack address, the operation of the PUSHJ and POPJ instructions is modified when the PC is in a non-zero section. Like JSP and JSR, PUSHJ stores a full 30-bit PC (again, actually PC+1) on the stack, omitting the flags. Similarly, POPJ restores a full 30-bit PC from the stack instead of an 18-bit PC local to PC section. Let's look at some examples:

```
2,,100/ MOVE 17,[-100,,200]
2,,101/ PUSHJ 17,400
```

Because PC section is non-zero, the PUSHJ stores 2,,102 on the stack at location 2,,201, which was addressed by a local stack pointer, and then jumps to location 2,,400. An updated stack pointer of -77,,201 is stored back into AC 17. Similarly:

```
2,,400/ MOVE 17,[-77,,201]
2,,401/ POPJ 17,
```

restores the full 30-bit PC from stack location 2,,201 (addressed by the local stack pointer) and then stores an updated stack pointer of -100,,200 back into AC 17.

This behavior has some interesting aspects, as the next example demonstrates:

```
2,,100/ MOVE 17,[2,,200]
2,,101/ PUSHJ @[0,,300]
```

Because PC is in a non-zero section, the PUSHJ instruction stores a full 30-bit PC (2,,102) on the stack at location 2,,201 (addressed by the global stack pointer). The jump is then made into section zero. But an attempt to return to the caller with a POPJ instruction will result in bedlam. In the first place, the global stack pointer will be interpreted as a local one in section zero. In addition, POPJ will assume that the stack word contains flags and PC and restore an 18-bit PC, local to section zero.

As this example demonstrates, it isn't very practical to call subroutines in section zero, from a non-zero section, using the normal call/return conventions.

- If PC is in a non-zero section, the PUSHJ instruction stores a full 30 bit PC on the stack. This is true even if the destination of the jump is in section zero and regardless of the format of the stack pointer.

- If PC is in a non-zero section, the POPJ instruction always restores a full 30-bit PC from the stack.

4.7.5 JSA and JRA

These instructions use a format that is incompatible with extended addressing. Because they are also considered an obsolete method for subroutine call/return, no attempt has been made to find an alternate format for these instructions when executed in a non-zero section.

For compatibility with section zero programs, these two instructions continue to work in non-zero sections. However, their use is restricted to intra-section operation, and all inter-section use is undefined.

In the case of JSA, the effective address is calculated in the normal manner. However, if the EA-calc results in an address outside of PC section, the action of the instruction is undefined. For example, the results of:

```
2,,100/ JSA 1,@[3,,200]
```

are undefined because the effective address is in section 3 and PC section is section 2. Note that a JSA which computes a global effective address which addresses the last word of PC section is also undefined. Let's look at an example of why this is true:

```
2,,100/ JSA 1,@[2,,777777]
```

In this case, the microcode would store the contents of AC into 2,,777777 and attempt to jump to E+1. But because EA is global, the computation of E+1 would result in 3,,0 GLOBAL which is outside of PC section.

The normal usage of JRA is of the form JRA AC,(AC) and the operation of the instruction is defined to take this into account. After the normal effective address calculation is performed, PC section is appended to the in-section addresses in AC to form the address of where the old contents of AC were stored and the new PC address. This forces all references to be in PC section. For example,

```
2,,201/ MOVE 1,[200,,101]
2,,202/ JRA 1,(1)
```

restores AC from location 2,,200 (PC section plus contents of AC left) and then jumps to 2,,101 (EA in PC section).

These definitions for JSA and JRA are consistent with the operation of the instructions in section zero.

- The use of JSA and JRA in a non-zero section is restricted to the case where the EA-calc results in an address in PC section. All inter-section usage is undefined.

4.7.6 LUUOs

In a non-extended machine, LUUOs trap via a pair of locations (40 and 41) in exec or user virtual memory. Because this scheme is insufficient to support extended addressing, the operation of LUUOs is modified if the PC is in a non-zero section. In this circumstance, the LUUO is processed through a four-word block which is addressed by a word in the exec or user process tables. See the PRM for more details.

- If PC is in a non-zero section, LUUOs trap through a four-word block addressed by a location in the EPT (exec LUUO) or UPT (user LUUO).

4.7.7 BLT

The format used for source and destination addresses by BLT is insufficient to represent two 30-bit addresses. As a result, the XBLT instruction was added to the instruction set to allow block transfers from one arbitrary 30-bit address to another. Despite this, BLT is still useful for intra-section block transfers, and the operation of the instruction has been changed slightly.

The initial source address is constructed by taking the 18-bit address in the left half of the AC and appending it to the section number and local/global flag from the effective address. Similarly, the initial destination address is constructed from the 18-bit address in the right half of the AC and the section number and local/global flag from the effective address. This means that transfers are always to and from the same section as that specified by the effective address, which need not necessarily be the same as PC section. Source and destination addresses are then incremented, section-local (even if EA is global) until the destination address is equal to EA. For example:

```
2,,100/ MOVE 1,[200,,300]
2,,101/ BLT 1,@[3,,302]
```

In this example, the EA-calc for the BLT results in 3,,302 GLOBAL. Using the rules above, the initial source and destination addresses would be 3,,200 GLOBAL and 3,,300 GLOBAL. Therefore, the following transfer would take place:

```
3,,200 => 3,,300
3,,201 => 3,,301
3,,202 => 3,,302
```

Let's look at an example that demonstrates the significance of incrementing the addresses section-local:

```
2,,100/ MOVE 1,[777776,,300]
2,,101/ BLT 1,@[3,,302]
```

As in the previous example, EA is 3,,302 GLOBAL and the initial destination address is 3,,300 GLOBAL. In this case, the initial source address is 3,,777776 GLOBAL and the following transfer takes place:

```
3,,777776 => 3,,300
3,,777777 => 3,,301
3,,0      => 3,,302
```

Note that the source address was incremented section-local even though it was a global address.

It is important to note that the local/global flag must be included in constructing the initial source and destination addresses even though the addresses are always incremented section-local. This is because the check for an AC reference is done by including this flag. Let's look at two examples, one whose EA is local and one whose EA is global:

```
2,,100/ MOVE 17,[1,,200]
2,,101/ BLT 17,201
```

In this case, the result of the EA-calc for the BLT is 2,,201 LOCAL. Therefore, the initial source and destination addresses are 2,,1 LOCAL and 2,,200 LOCAL, respectively. Because the source is a local address whose in-section part is in the range 0-17, it references AC 1. Now let's look at the global case:

```
2,,100/ MOVE 17,[1,,200]
2,,101/ BLT 17,@[2,,201]
```

In this case, the result of the EA-calc for the BLT is 2,,201 GLOBAL. Therefore, the initial source and destination addresses are 2,,1 GLOBAL and 2,,200 GLOBAL, respectively. In this case, the source address references memory location 2,,1 instead of the ACs because the effective address is global. In both cases, however, the addresses

are incremented section-local.

- The initial source and destination addresses for BLT are constructed by appending the appropriate half of the AC to the section number and local/global flag from the effective address. Incrementing of source and destination addresses is always done section-local independent of the state of the local/global flag. However, the determination of AC reference is done via the normal rules by including the local/global flag.

4.7.8 XBLT

The XBLT instruction is the one exception to the rule that a section zero program cannot reference data in non-zero sections. In this one case, the contents of AC+1 (source pointer) and AC+2 (destination pointer) are always treated as 30-bit global addresses, even if the PC is in section zero. This means that a program running in section zero can allocate a non-zero section and XBLT code or data into it without having to jump into a non-zero section to do it.

- The source and destination addresses for XBLT are always interpreted as full 30-bit global addresses, even if the PC is in section zero.

This means that the final addresses left in AC+2 and AC+3 at the end of the XBLT may be inaccessible by other instructions in section zero. For example:

```
0,,100/ MOVEI 1,777777          ;Word count
0,,101/ MOVEI 2,20              ;Source address
0,,102/ MOVE 3,[2,,100]         ;Destination address
0,,103/ EXTEND 1,[XBLT]
```

In this example, the transfer is from 0,,20 to 2,,100, and the number of words transferred is 256K-1. The final source and destination addresses left in ACs 2 and 3 are 1,,17 and 3,,77 respectively.

- For XBLT, the final values stored in AC+2 and AC+3 for source and destination addresses are computed by adding the initial word count to the initial source and destination addresses. This computation is the same in all sections, including section zero.

4.7.9 JRSTF

If the PC is in a non-zero section, JRSTF traps as an MUUO. This is because JRSTF is usually used with an indirect word or index register with PC flags in the left half. It is quite likely that these flags would be mistaken for a global section number.

- If PC is in a non-zero section, JRSTF traps as an MUUO. XJRSTF should be used in a non-zero section.

4.7.10 XMOVEI and XHLLI

Unlike other immediate instructions that use only 18 bits of the effective address, these two instructions operate on all 30 bits of EA. XMOVEI returns the full 30-bit effective address in AC. XHLLI stores the section number of the effective address in the left half of AC, leaving the right half unchanged.

One important implication of these two instructions is that they convert a local reference to an AC in any non-zero section into the global form. For example:

```
2,,100/ XMOVEI 1,6
```

The EA-calc of the XMOVEI results in 2,,6 LOCAL, which is a local reference to AC 6. This result is then converted to the global AC address of 1,,6 before being loaded into AC 1.

This conversion is not done if the AC reference is local to section zero. For example:

```
2,,100/ XMOVEI 1,@[200000,,6]
```

In this example, the EA-calc follows an indirect EFIW into section zero. The result of the EA-calc is therefore 0,,6 LOCAL, which is a local reference to AC 6. Because the effective address is in section zero, it is not converted to the global form and 0,,6 is stored in AC 1.

- If the effective address of an XMOVEI or XHLLI is a local reference to an AC in a non-zero section, the AC address is converted to a global AC address before being loaded into AC.

4.7.11 XCT

With the exception of the modification of the EA-calc rules in a non-zero section, the XCT instruction operates in the same manner as on a non-extended machine. The operation of the instruction being executed, however, may be affected. This section describes these

cases and gives examples to demonstrate them.

4.7.11.1 Default section for EA-calc

If an instruction is executed by an XCT, the initial default section for the EA-calc of that instruction is the section from which the instruction was fetched. This may be different from PC section if the XCT and the executed instruction are in different sections. For example:

```
3,,100/ XCT @[2,,100]

2,,100/ MOVE 1,200
```

In this example, the XCT instruction is in section 3 and the executed instruction is in section 2. The Ea-calc for the MOVE yields a local address, which is local to the section from which the MOVE was fetched. Therefore, the result of the EA-calc is 2,,200 LOCAL. This rule allows one to XCT an instruction in another section and have local references generated by the executed instruction be local to the section containing the instruction.

- The initial default section for the EA-calc of an instruction executed by XCT is that from which the instruction was fetched.

4.7.11.2 Relationship with skip and jump instructions

When a skip instruction is XCTed, the skip is always relative to PC section, i.e., the section containing the XCT (first XCT if there is a chain of XCTs). This is true even if the skip instruction is in another section. For example:

```
3,,100/ XCT @[2,,300]

2,,300/ SKIPA 1,200
```

In this example, an XCT in section 3 executes a skip instruction in section 2. Because this instruction always skips, the next instruction is taken from location 3,,102 (PC+2), not 2,,302 (instruction+2). However, the EA-calc of the SKIPA instruction results in 2,,200 LOCAL, so the contents of location 200 in section 2 are stored in AC.

- If an XCT executes a skip instruction, the skip is always relative to PC section, even if the skip instruction is in another section.

The following example demonstrates the effect of XCTing a jump instruction:

```
3,,100/ XCT @[2,,100]

2,,100/ JRST 200
```

In this example, an XCT in section 3 executes a jump instruction in section 2. The EA-calc for the JRST results in an address local to section 2, so the next instruction is taken from 2,,200, not 3,,200.

- If an XCT executes a jump instruction that jumps, the next instruction is fetched from the effective address of the jump. This is true even if the XCT and the jump are in different sections and the EA-calc of the jump results in a local address whose section is different from PC section.

4.7.11.3 PC storing instructions

When an XCT executes an instruction that stores PC as part of the operation of the instruction (e.g., PUSHJ, JSP, etc.), the value stored is relative to PC section (i.e., the XCT) and not the section of the executed instruction. For example:

```
3,,100/ XCT @[2,,200]

2,,200/ JSP 1,300
```

In this example, an XCT in section 3 executes a JSP in section 2. The next instruction is fetched from location 2,,300 because the EA-calc of the JSP is local to section 2. However, the PC stored in AC 1 is 3,,101 (XCT+1), not 2,,201 (JSP+1).

- If an XCT executes an instruction that stores PC as part of its execution, the value stored is relative to the XCT and not the executed instruction.

4.7.11.4 Local stack references

When an XCT executes a stack instruction that uses a local stack pointer, the stack pointer is local to PC section and not to that containing the stack instruction. For example:

```
3,,077/ MOVE 17,[-100,,300]
3,,100/ XCT @[2,,200]

2,,200/ PUSH 17,400
```

In this example, an XCT in section 3 executes a PUSH in section 2. Since the EA-calc for the PUSH results in a local address, the datum to be pushed is in the same section as the PUSH instruction (at location 2,,400). However, the stack pointer is local to PC section, not the section containing the PUSH. Therefore, the datum is stored on the stack at location 3,,301.

- **If an XCT executes a stack instruction whose stack pointer is local, the stack is local to PC section, not the section containing the stack instruction.**

4.7.11.5 Generalizations for XCT

The examples above cover specific relationships between XCT and the executed instruction. There are really two generalizations (one of which was given above) that can be made about XCT, as follows:

1. The initial default section for the EA-calc of an XCTed instruction is that from which the instruction was fetched, and not the section from which the XCT was fetched.

2. Any test of PC section for determining whether section zero rules or non-zero section rules apply is done based on the section from which the XCT instruction was fetched (the first one if there is a chain of XCTs). That is, PC section doesn't change because an XCT executes an instruction in another section.

4.8 Summary of default sections for EA-calc

After covering all the special case instructions, it is worthwhile to summarize the rules regarding the initial default section number for EA-calc's. The initial default section for any EA-calc is that from which the address word was fetched. This is true for the simple cases as well as the more complex cases. The following table gives the initial default section for the various kinds of EA-calc:

EA-calc class	Initial default section
Instruction	PC section
XCTed instruction	Section containing the executed instruction
Byte instruction byte pointer	Section containing the byte pointer
EXTEND instruction byte pointer	PC section
EXTEND instruction opcode word	Section containing the opcode word
Local stack pointer	PC section

4.9 Section zero vs. non-zero section rules

As the previous discussion of special case instructions indicates, some instructions do different things based on a test for section zero. However, this test isn't always on PC section. We have intentionally left out examples that demonstrate some of the boundary conditions that make extended addressing hard to document to avoid confusing the reader before the simple cases are understood. This section includes examples of these boundary conditions, and summarizes the rules for testing to see if section zero rules apply.

The first example illustrates the test for the possibility of a global byte pointer:

```
3,,100/ LDB 1,@[0,,200]
0,,200/ 000640,,300
0,,201/ 400000,,400
```

In this example, the byte instruction is in section 3 and the byte pointer is in section 0. Note that bit 12 is set in the byte pointer which, if global byte pointers are allowed, would indicate a two-word global byte pointer. Is this byte pointer interpreted as a one-word local or two a word global byte pointer? The rule given in a previous section says that the test is made based on the section from which the byte pointer was fetched. Therefore, bit 12 is ignored, the byte pointer is interpreted in one-word local format, and the byte is fetched from the word at location 0,,300.

Let's look at a similar case involving both XCT and EXTEND:

```
3,,100/ MOVEI 1,5                ;Source length
3,,101/ MOVE 2,[440740,,500]     ;Source b.p. (1st wd)
3,,102/ MOVE 3,[5,,100]          ;Source b.p. (2nd wd)
3,,103/ MOVEI 4,5                ;Destination length
3,,104/ MOVE 5,[440740,,300]     ;Destination b.p. (1st wd)
3,,105/ MOVE 6,[5,,200]          ;Destination b.p. (2nd wd)
3,,106/ XCT @[0,,100]            ;Execute EXTEND in section 0

0,,100/ EXTEND 1,200

0,,200/ MOVSLJ                   ;Extended opcode is MOVSLJ
0,,201/ 0                        ;Fill character is 0
```

In this example, the XCT is in section 3 and the entire EXTEND instruction is in section zero. Both the source and destination byte pointers have bit 12 set, which means they may be interpreted as two-word global pointers. But are they? The rule given in a previous section says that the test is made based on PC section, which is non-zero. Therefore, the byte pointers are two-word global and the string is moved from 5,,100 to 5,,200. If this seems like an anomaly, remember that the test is based on PC section because the byte pointers are fetched from the ACs. References to ACs addressed by the AC field of the instruction are always made in PC section.

A final example combines an XCT with a JSR:

```
3,,100/ XCT @[0,,200]

0,,200/ JSR 300
```

In this example, the XCT is in section 3 and the JSR is in section zero. The EA-calc of the JSR is local to section zero, so the destination of the jump is 0,,301. But what is stored in 0,,300? The rule given in a previous section says that the test is based on PC section. Therefore, we store a full 30-bit PC (3,,101) into location 0,,300.

- The test for section zero rules vs. non-zero section rules is done based on PC section for all cases except byte instructions. This is true even if the instruction is an XCT which executes an instruction in another section (including section zero).

- The test for section zero rules vs. non-zero section rules for a byte instruction is done based on the section from which the byte pointer was fetched.

It is important to realize that PC section may be different from that containing the instruction being executed if an XCT (or chain of XCTs) is involved. PC section is always that from which the original instruction (the XCT if that instruction is involved) was fetched. This is a subtle distinction, but it is important in testing for section zero rules.

4.10 Special consideration for ACs

On the PDP-10, the ACs are both general purpose registers and also part of the virtual address space of every program. This dual use is convenient but also confusing when one is attempting to understand the rules of extended addressing. This section describes some of the aspects of the relationship between extended addressing and the use of the ACs.

4.10.1 AC references

An AC can be referenced in one of four ways as follows:

1. As a general purpose register through the AC field of an instruction.

2. As an index register through the index register field of an instruction or indirect word.

3. As a local memory reference to the first 16 (decimal) locations of any section.

4. As a global memory reference to the first 16 (decimal) locations of section 1.

In this discussion, we are concerned with the last two uses.

The rules for extended addressing say that memory references in section zero are always local. Therefore, a section zero memory reference can reference the ACs only if it is to the first 16 (decimal) locations in section zero. On the other hand, a memory reference in a non-zero section can reference the ACs in two different ways. If the memory reference is local, the ACs appear in the virtual address space of every section as the first 16 locations. For example, both

```
2,,100/ MOVE 1,2
```

and

```
5,,100/ MOVE 1,2
```

reference AC 2 even though the addresses are local to different sections.

In addition, the ACs may be referenced in a section-independent way via a reference to global address 1,,n, where n is in the range 0-17, inclusive. This means that an AC address can be passed between two routines running in a non-zero section, even if the routines are in different sections. For example:

```
5,,100/ MOVE 16,[1,,6]      ;Get global AC address for AC
5,,101/ PUSHJ 17,@[3,,200] ; 6 and call routine
          :
          :
3,,200/ MOVE 1,(16)         ;Use global XR to fetch data
```

In this example, the calling routine in section 5 places the global AC address for AC 6 into AC 16 and calls a routine in section 3. Because 1,,6 is a global AC address, the called routine interprets the index in global format and the data is fetched from AC 6.

Note that an address of the form 1,,n, where n is in the range 0-17, will always reference the ACs, whether the address is local or global. If the address is local, the reference is a local reference to the ACs in section 1. If the address is global, it is a global AC reference to the ACs.

- An address of the form 1,,n, where n is in the range 0-17, inclusive, refers to the ACs whether it is a local or global address. Therefore, such an address can be used to refer to the ACs even if the state of the local/global bit is not known.

4.10.2 Instruction fetches

All instruction fetches are made as local references, even though the PC is a full 30-bit address. Therefore, an instruction is fetched from the ACs whenever bits 18-35 of PC are in the range 0-17, inclusive, independent of the section number. Consider the following example:

```
1,,100/ XJRST [3,,2]
```

This instruction sets the PC to 3,,2. However, the next instruction fetch will come from AC 2 because it is made as a local reference.

This behavior can have some implications for instructions that also store information before changing PC. Consider the following example:

```
1,,100/ JSR @[3,,2]
```

The JSR stores the current PC into memory location 3,,2 and then changes the PC to 3,,3. The next instruction is then fetched from AC 3 because of the local reference, but the old PC is in memory and must be fetched with a global reference.

- Instruction fetches from C(PC) are always made as local references even if PC was previously set to a global address. This means that instruction fetches from the first 16 (decimal) locations of any section cause the instruction to be fetched from the ACs.

4.10.3 Storing PC

If an instruction that stores PC as part of its execution is fetched from the ACs, the PC is stored as a full 30-bit address if PC is in a non-zero section. For example:

```
3,,100/ MOVE 4,[JSP 2,200]
3,,101/ JRST 4
```

In this example, the MOVE instruction stores a JSP into AC 4, and the JRST instruction computes a local effective address that references the ACs. PC is set to 3,,4, but the next instruction is fetched from AC 4 because instruction fetches are always made as local references. Therefore, the next instruction to be executed is the JSP. Because PC section is non-zero (it is still 3), the JSP must store a full 30-bit PC into AC 2. The important thing to realize is that PC is 3,,4 and is not 0,,4 (a section zero AC address) or 1,,4 (a global AC address). Therefore the JSP stores 3,,5 (remember, it stores PC+1) into AC 2 and jumps to 3,,200.

- **If an instruction that is fetched from AC stores PC as part of its execution, the PC stored is a full 30-bit address including PC section, if PC section is non-zero.**

4.10.4 Storing EA for LUUO, MUUO and page fails

When an LUUO or MUUO is executed or an instruction page fails, the microcode stores some information about the exception in a block addressed by a word fetched from the UPT or EPT. The information stored includes the effective address (or reference address in the case of page fail) for the instruction that caused the exception. If the resulting effective address is a local reference to an AC in a non-zero section, the microcode converts this address to a global AC reference before storing it in the block. This is the same rule used for XMOVEI and XHLLI.

- **If the effective address of an LUUO or MUUO, or an instruction that causes a page fail results in a local reference to the ACs in a non-zero section, the microcode converts the local AC reference to a global AC address before storing the result.**

4.10.5 An example

Consider the following example that brings together all of these rules:

```
3,,100/ MOVE 6,[001000,,10]
3,,101/ JRST 6
```

In this example, the MOVE stores an LUUO (opcode 001) into AC 6 and the JRST sets PC to 3,,6. The following list indicates the significant actions that are performed to process the LUUO:

1. The EA-calc for the LUUO is performed and the result is 3,,10 LOCAL.

2. Because PC section is non-zero, the LUUO must be processed through a four-word block addressed by a location in the UPT.

3. PC+1 must be stored as a full 30-bit address, including section number. The value stored is 3,,7.

4. Because the EA-calc of the LUUO resulted in a local reference to AC 10, it must be converted to a global AC address before it is stored in the block. The value stored is therefore 1,,10.

4.11 PXCT

When the monitor is invoked by an MUUO, page fail, etc., the address space of the process that caused the invocation is potentially different from that of the monitor. In order to provide a communications mechanism between the monitor and the so-called "previous context", the PXCT (for Previous context XCT) instruction was defined. Although PXCT is normally considered as a separate topic from extended addressing, there are interactions between the two that make it desirable to talk about them together.

Because PXCT is legal only in exec mode, there is no need to define a new opcode for the instruction. Rather, the normal XCT opcode is used, and a non-zero AC field distinguishes a PXCT from a normal XCT. The opcode name PXCT is simply a notational convenience to emphasize that the executed instruction is making previous context references.

4.11.1 Previous context

For the purposes of this discussion, "previous context" is defined by three processor state variables: Previous Context Section (PCS), Previous Context User (PCU), and Previous AC Block (PAB). PCS is a 12-bit state register (5 on the KL10) that gives the value of PC section in the previous context at the time of the event that invoked the monitor. PCU is a 1-bit register that indicates that the previous context was user mode (as opposed to exec mode). PAB is a 3-bit register that gives the AC block number used by the previous context (there are typically multiple AC blocks implemented by a machine, 8 in both KL10 and KC10. The so-called "current ac block" is addressed by another 3-bit state register called Current AC Block, or CAB). Therefore, the previous context includes both the address space and ACs that were in use at the time of the event that invoked the monitor.

When a context change occurs as the result of an MUUO, page fail, interrupt, etc., the previous context state variables are set according to a set of rules that are defined for each type of context change. The specific rules aren't important for the purpose of this discussion and the reader is referred to other sources for more information. The important point is that the state variable are set as the result of the context change.

In addition to being set on a context change, the monitor may also set the state variables explicitly when it desires to make an asynchronous reference to previous context.

These previous context state registers then direct references to the previous context as described below. Note that the previous context need not always be user mode. It is exec mode in cases where the monitor makes a request of itself, such as the execution of an MUUO by the monitor.

4.11.2 Use of the previous context state variables

The state registers PCS, PCU, and PAB hold information necessary to make a previous context memory or AC (as memory or index register) reference. This section describes the use for each register.

PCS is a 12-bit state variable that gives the value of PC section in the previous context. It is used in the PXCT EA-calc algorithm as described below to provide a default section number for a local EA-calc. It is also used as the basis for the test for section zero in some instructions that behave differently in non-zero sections as described below. (For most instructions, the effect is as if the instruction were executed in previous context.)

PCU is a 1-bit state variable that indicates that the previous context was user mode. PCU is used to select the address space for a previous context memory reference. That is, if the reference is to previous context and PCU is set, the reference is made to the user address space as mapped through the UPT. Conversely, if the reference is to previous context and PCU is not set, the reference is to the exec address space as mapped through the EPT.

PAB is a 3-bit state variable that gives the AC block number for the previous AC block. If an index register or AC is referenced in previous context, PAB gives the number of the AC block containing the data.

4.11.3 References to previous context

The PXCT mechanism allows the monitor to execute an instruction such that certain references of the executed instruction are made to the previous context. Conceptually, these references are made as if the PXCTed instruction were being executed in the previous context.

It is important to understand exactly which operations are modified by PXCT. The instruction fetch and EA-calc of the PXCT instruction and the fetch of the executed instruction are always done in current context. In addition, all AC references (as the result of bits 9-12 of the executed instruction) are made to the current context ACs. The only difference between an instruction executed under PXCT and one that is not is the way certain memory and index register references are made. In particular, the EA-calc of the executed instruction may reference indirect words and index registers in previous context. Also, memory and AC references made as the result of the EA-calc may be to previous context. Exactly which references are made in previous context is determined by the type of instruction that is being executed and by the bits set in the AC field of the PXCT instruction.

4.11.4 Applicable instructions

Not all instructions may be executed via PXCT. The use of PXCT is limited to instructions that are useful to the monitor, and no attempt is made to trap those cases that aren't applicable. The instructions that may be executed are as follows:

MOVE class instructions
Halfword class instructions
EXCH
XMOVEI, XHLLI
BLT (with restrictions), XBLT
Arithmetic (integer and floating point) instructions
Boolean instructions
DMOVE class instructions
CAI and CAM class instructions
SKIP, AOS, and SOS class instructions
Logical test instructions
PUSH and POP (with restrictions)
Byte class instructions
MOVSLJ (with restrictions)
MAP

All other instructions are inapplicable, and the results of executing an inapplicable instruction are undefined. Note that this list explicitly excludes all instructions that jump.

4.11.5 Interpretation of the AC field bits

The four bits of the AC field of the PXCT instruction determine which memory references of the executed instruction are made to previous context. For most PXCTed instructions, the AC field bits are logically grouped into two pairs (9-10 and 11-12) to control how EA-calc and data references are performed. Within each pair, the first bit (the generic "E control bit") causes index register and address word references to come from previous context during an EA-calc. The second bit (the generic "D control bit") causes data fetches as the result of instruction execution to come from previous context. When considered as a whole, bits 9-12 of the AC field are named "E1", "D1", "E2", and "D2" but the generic names ("E" and "D") may be used when it is clear which bits control the reference in question.

Not all executed instructions use both pairs of bits. In fact, the great majority of applicable instructions use only bits 9 and 10; bit 9 for the EA-calc of the PXCTed instruction and bit 10 for the data reference made as the result of that EA-calc. A notable example of the use of bits 11 and 12 to control previous context references is the byte instructions. In this case, bit 11 controls the EA-calc of the byte pointer and bit 12 controls the data reference to the word containing the byte. Some instructions use other combinations of bits, e.g., BLT, EXTEND (MOVSLJ and XBLT), and stack instructions.

The previous context memory references controlled by each AC field bit may be summarized by the following table:

Bit	References made in previous context if bit is 1
9 (E1)	Effective address calculation of instruction (index registers, indirect words).
10 (D1)	Memory operands specified by EA, whether fetch or store (e.g, PUSH source, POP or BLT destination); byte pointer.
11 (E2)	Effective address calculation of byte pointer; source in EXTEND (e.g., XBLT or MOVSLJ source); effective address calculation of source byte pointer in EXTEND (MOVSLJ).
12 (D2)	Byte data; source in BLT; destination in EXTEND (e.g., XBLT or MOVSLJ destination); effective address calculation of destination byte pointer in EXTEND (MOVSLJ).

There are obviously a limited number of valid combinations of AC field bits for those instructions that may be PXCTed. The following table gives the legal combinations. The "AC" column gives the AC field value for the equivalent bits, e.g., the AC column would contain a 4 for a 0 1 0 0 bit string.

Instructions	AC	E1 9	D1 10	E2 11	D2 12	References
General	4	0	1	0	0	Data
	14	1	1	0	0	E, data
PUSH, POP	4	0	1	0	0	Data
	14	1	1	0	0	E, data
Immediate	10	1	-	0	0	E (no data reference)
BLT	5	0	1	0	1	Source data, destination data
	15	1	1	0	1	E, source data, destination data
XBLT	2	0	0	1	0	Source data
	1	0	0	0	1	Destination data
	3	0	0	1	1	Source data, destination data
Byte	1	0	0	0	1	Byte data
	3	0	0	1	1	Pointer E, byte data
	7	0	1	1	1	Pointer, pointer E, byte data
	17	1	1	1	1	E, pointer, pointer E, byte data

MOVSLJ	1	0	0	0	1	Destination pointer E, destination data
	2	0	0	1	0	Source pointer E, source data
	3	0	0	1	1	Source pointer E, destination pointer E, source data, destination data

Note that BLT, PUSH, POP, and MOVSLJ have restrictions on what memory references can be PXCTed. For BLT, all references, optionally including the EA-calc, must be done in previous context. The results of PXCTing a BLT where source but not destination or destination but not source is in previous context are undefined. The LDPAC and STPAC instructions should be used to transfer the previous ACs to and from current context. In all other cases, XBLT must be used to transfer data between current and previous context.

For PUSH and POP, the stack must always be in current context. This means that previous context references for PUSH and POP are limited to the EA-calc and data reference made to the location addressed by the EA-calc. PUSH and POP therefore reduce to the "general" case.

For MOVSLJ, if source or destination data is in previous context, the source or destination byte pointer EA-calc must be done in previous context also. If the monitor wishes to force a current context EA-calc for a previous context data reference, it can compute the effective address of the byte word and use a one- or two-word global byte pointer. The microcode will still do the EA-calc in previous context, but no previous context defaults will be applied.

4.11.6 Modifications to the EA-calc algorithm

The appropriate "E" and "D" control bits from the AC field of the PXCT instruction are used to modify an EA-calc done on the executed instruction or a subsequent EA-calc done by the instruction (e.g., byte pointer). This modification involves pre- and post-processing the normal effective address calculation algorithms to conditionally include PCS at two points.

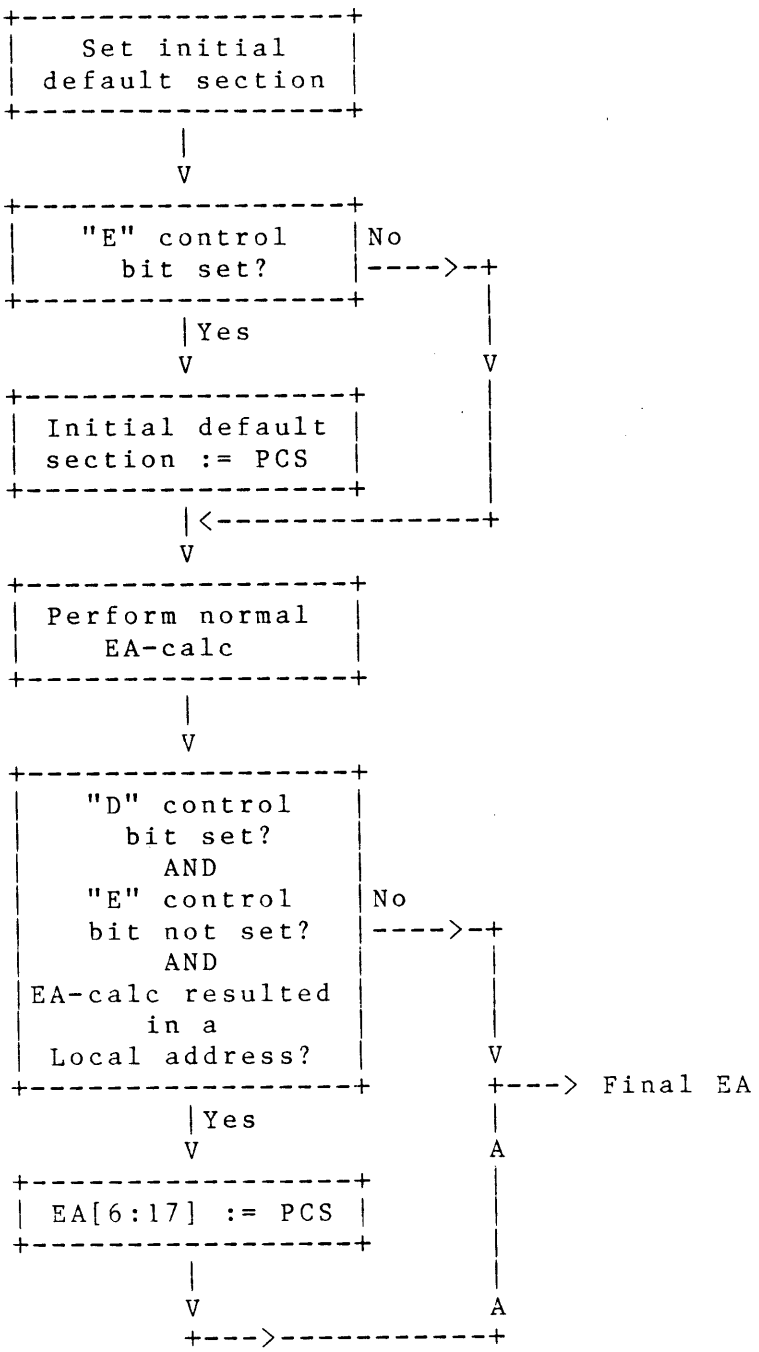
If the appropriate "E" control bit is set, the initial default section for the EA-calc is set to PCS. Since the "E" control bit also controls previous context indirect word and index register references, this means that the entire EA-calc is done in previous context. If the "E" control bit is not set, the initial default section for the EA-calc is that from which the address word was fetched, and the EA-calc is done in current context.

When the normal EA-calc is completed, the resulting value is post-processed. If the result of the EA-calc was a local address AND the "E" control bit was not set AND the "D" control bit was set, the section number of the EA-calc is replaced by PCS. Note that the local/global flag remains local if this is done.

The application of PCS at the end of the EA-calc may seem to make no

sense at first glance, so let's take a closer look at it. Remember that the purpose of PXCT is to allow the monitor to reference data in the previous context as if the user had supplied it. If the user supplies a local address in, for example, a JSYS argument, the monitor should make the data reference local to the section in which the user was running. By applying PCS at the end of the EA-calc as indicated above, the microcode automatically makes the reference to the correct section.

This algorithm may be described by the following flow chart:



PXCT EA-calc algorithm

Assume that PCS is 1 and consider the following example:

```
2,,100/ PXCT 4,[MOVE 1,100]
```

MOVE is one of the "general" class of opcodes, so bits 9 and 10 of the PXCT AC field control the previous context references. In this example, bit 9 (The "E1" bit) is off and bit 10 (the "D1" bit) is on. Therefore, the EA-calc is done in current context with a result of 2,,100 LOCAL. Because the "D1" bit is on, the "E1" bit is off, and the result of the EA-calc is local, the PXCT EA-calc algorithm applies PCS to bits 6-17 of the EA-calc. The final effective address is therefore 1,,100 LOCAL and the data reference is made to that location in previous context.

Let's look at another example. Assume that PCS is 2 and that the following locations exist in previous context:

```
2,,200/ 200003,,300

3,,300/ 400000,,400
```

In current context, the following instruction is executed:

```
1,,100/ PXCT 14,[MOVE 1,@200]
```

In this example, both the "E1" and "D1" bits are on in the PXCT AC field. Therefore, the EA-calc is done in previous context and the initial default section for the EA-calc is set to 2 (PCS). Location 2,,200 in previous context contains an indirect EFIW that the EA-calc follows into section 3. The final address word fetched from previous context location 3,,300 is in IFIW format, so the result of the EA-calc is local to the section from which the address word was fetched. The result of the EA-calc is 3,,400 LOCAL. Because the "D1" bit is also set, the MOVE fetches data from previous context location 3,,400.

A final example demonstrates the result of an EA-calc that references an AC. Assume that PCS is 3.

```
2,,100/ PXCT 4,[MOVE 1,2]
```

As with the first example, the EA-calc is done in current context and PCS is applied to bits 6-17 of the result to produce an effective address of 3,,2 LOCAL. Just as in the non-PXCT case, this is a local reference to AC 2. Because the "D1" bit is set, the reference is made to previous context AC 2 in the AC block specified by PAB.

- The EA-calc of a PXCTed instruction may be pre- or post-processed as directed by the AC field control bits of the PXCT instruction. Except for this additional processing, the EA-calc algorithms and results are exactly the same as for the non-PXCT case. This includes the uses for the local/global flag.

4.11.7 Section zero vs. non-zero section rules

Of the instructions that may be PXCTed, there are three types (stack, byte, and MOVSLJ) that operate differently in non-zero sections and section zero. When one of these instructions is PXCTed, the test for zero/non-zero rules may not be the same as the test when there is no PXCT involved. The interaction of PXCT with each of the instruction types is covered separately below.

4.11.7.1 Stack instructions

When no PXCT is involved, the test for the possibility of a global stack pointer is done based on PC section. When a PUSH or POP instruction is PXCTed, the previous context references are limited to the EA-calc and the datum addressed by the EA-calc, and the stack reference is always made in current context. Because the stack is in current context, the interpretation of the stack pointer type is made based on the current context PC section and is not dependent on PCS. For example, assume that PCS is 0.

```
2,,100/ MOVE 1,[3,,1000]
2,,101/ PXCT 4,[PUSH 1,200]
```

In this example, PC section is non-zero and the stack pointer in AC 1 has a global format. The test to determine whether the stack pointer is allowed to be global is still made based on PC section (even though there is a PXCT involved), and not on PCS. Therefore, the stack pointer is indeed global and previous context location 0,,200 is pushed onto the stack in current context location 3,,1001.

- When a stack instruction (PUSH, POP) is PXCTed, the test for the possibility of a global stack pointer is done based on PC section.

- When a stack instruction is PXCTed, local stack pointers are always local to PC section.

4.11.7.2 Byte instructions

Normally, the byte instruction test for the possibility of global byte pointers is done based on the section from which the byte pointer was fetched. When a byte instruction is PXCTed, this rule continues to apply, with extensions to include the possibility that the byte pointer may be fetched from previous context. This is best explained with several examples.

Assume that PCS is 0 and that the following locations exist in previous context:

```
0,,100/ 400000,,200

0,,200/ 12
```

In current context, the following instruction is executed:

```
2,,300/ PXCT 3,[LDB 1,400]

2,,400/ 000640,,0
2,,401/ 400020,,100
```

For PXCT of byte instructions, bits 9 (E1) and 10 (D1) direct the EA-calc of the byte instruction and the fetch of the byte pointer. Bits 11 (E2) and 12 (D2) direct the EA-calc of the byte pointer and the fetch of the word containing the byte. In this example, the "D1" bit is off, so the byte pointer is fetched from current context location 2,,400. Bit 12 is on in the byte pointer, and a test must be made to see if it may be global. The byte pointer is global because it was fetched from current context section 2, and the fact that PCS is zero is not considered.

The "E2" bit and the "D2" bit of the PXCT AC field are both on, so the byte pointer EA-calc is done in previous context. The second word of the two-word global byte pointer has the indirect bit set, and the next address word is fetched from previous context location 0,,100. The final result of the EA-calc is 0,,200 LOCAL in previous context and bits 30-35 of that word are extracted and placed in current context AC 1.

Let's look at a similar example in which the byte pointer is also fetched from previous context. Once again assume that PCS is 0 and the previous context contains the following locations:

```
0,,400/ 000640,,100
0,,401/ 400000,,200

0,,100/ 10
0,,200/ 20
```

In current context, the following instruction is executed:

```
2,,300/ PXCT 7,[LDB 1,400]
```

In this case, the "D1" bit of the PXCT AC field is set, so the byte pointer is fetched from previous context location 0,,400. As in the last example, bit 12 is set in the byte pointer. But because the byte pointer was fetched from previous context section 0, bit 12 is ignored and the byte pointer is interpreted in one-word local format. The EA-calc is done in previous context and results in an effective address of 0,,100 LOCAL. The byte is then fetched from bits 30-35 of previous context location 100.

- When a byte instruction is PXCTed, the test for the possibility of a global byte pointer is done based on the section from which the byte pointer was fetched. This is true independent of whether the byte pointer is fetched from current or previous context.

This interpretation, while correct architecturally, causes some problems for TOPS-20 as it is implemented today because TOPS-20 copies byte pointers from the previous context into current context. Ideally, when a JSYS does a byte instruction on behalf of the user, the byte pointer would be interpreted exactly as if the user had executed the byte instruction. Thus, if the byte pointer were fetched from section 0, it would be interpreted as a local pointer; if it were fetched from any other section, it would be interpreted as possibly being global. This can be accomplished by using PXCT 7, as indicated in the example above.

Because TOPS-20 copies the byte pointer from the previous context into current context, one that looks like a global byte pointer will be interpreted as a global byte pointer even if it is fetched from previous context section zero. This is because the monitor typically runs in a non-zero section and the PXCTed byte instruction fetches the byte pointer from current context. Hence the test for the possibility of a global byte pointer is made based on current context section rather than previous context section.

4.11.7.3 EXTENDed MOVSLJ instruction

If no PXCT is involved, the MOVSLJ test for the possibility of a global byte pointer is made based on PC section. If a PXCT is involved, the test is more complex because it is based on PC section if the PXCT control bit for the byte pointer is off and on PCS if the PXCT control bit is on. For example, assume that PCS is zero and that previous context contains the following locations:

```
0,,200/ ASCII|ABCDE|

0,,300/ ASCII|FGHIJ|
```

In current context, the following instruction sequence is executed:

```
3,,100/ MOVEI 1,5                   ;Source length
3,,101/ DMOVE 2,[440740,,200        ;Source BP (word 1)
                 400000,,300]       ;Source BP (word 2)
3,,102/ MOVEI 4,5                   ;Destination length
3,,103/ DMOVE 5,[440740,,400        ;Destination BP (word 1)
                 400000,,500]       ;Destination BP (word 2)
3,,104/ PXCT 2,[EXTEND 1,600]       ;PXCT the MOVSLJ

3,,600/ MOVSLJ                      ;Extended opcode is MOVSLJ
3,,601/ 0                           ;Fill character is 0
```

In this example, the "E2" bit is set in the PXCT AC field, which indicates that the source EA-calc and string reference are to be made to previous context. Conversely, the "D2" bit is off, which indicates that the destination EA-calc and string references are to be made to current context.

Because the source-in-previous control bit is set in the PXCT AC field, the test for the possibility of a global source byte pointer is made based on PCS. In this case, PCS is zero, so bit 12 is ignored in the byte pointer and it is interpreted in one-word local format. The byte pointer EA-calc results in 0,,200 LOCAL in previous context.

On the other hand, the destination-in-previous control bit is not set, so the test for the possibility of a global destination byte pointer is made based on PC section. Since PC section is non-zero and bit 12 is set, the byte pointer is interpreted in two-word global format, and the byte pointer EA-calc results in 3,,500 LOCAL in current context.

The result is to transfer the string "ABCDE" from previous context location 0,,200 to current context location 3,,500.

- When a **MOVSLJ** instruction is PXCTed, the test for the possibility of a global byte pointer is done based on PC section if the appropriate PXCT control bit is off. If the bit is on, the test is done based on PCS.

CHAPTER 5

MICROCODE CHANGES

This chapter discusses the changes that must be made to the KS10 microcode in order to convert it to the KD10 microcode.

5.1 Microcode assemblers

Because the existing KS10 microcode is assembled using the unsupported microassembler MICRO, the first step should probably be to convert it to use MICRO2, the corporate microassembler. Fortunately, the statement syntax is identical and the changes should, for the most part, be isolated to the field/value definitions. The largest potential problem is the field defaulting mechanism which is quite different between MICRO and MICRO2.

5.2 New functionality

This section describes the additions to the microcode to support new functionality.

5.2.1 Extended addressing effective address calculation

Because an EA-calc is performed on every instruction, changes in this area are critical to the performance of the machine. The first general step is to use HR as a full 30-bit effective address rather than using the left half for the instruction. To do this introduces some problems. Some instructions assume that the opcode and AC field of the instruction are in HR bits 0-15 (e.g., MUUO). To store them in some other reg file location would slow down the EA-calc for every instruction; not just those that need it. The best solution appears to be to allow the IR and AC registers to be read back into the data path so that these fields need not be stored in the register file.

The next step is to augment the 30-bit EA with a 1-bit local/global flag. There are two changes required here. First, the microcode must be able to force bit 0 of AD to either zero or one (if bits 1-5 are

also affected, that's fine). In addition, the microcode must be able to control the write enable for bits 6-17 of the register file similar to what the HOLD LEFT and HOLD RIGHT macros do now. Armed with these changes, the modifications to the EA-calc can now be described.

Any effective address calculation starts out in some default section. For an instruction EA-calc, that section is PC section. For a byte EA-calc, it is the section from which the byte pointer was fetched (which should already be in the left half of HR). To initialize an instruction EA-calc with PC section, an additional microinstruction must be added at XCTGO (to be followed by the one there now). This microinstruction copies PC to HR. Since this microinstruction is overlapped with the memory cycle, it may be free.

The normal EA MODE DISP is done at INCPC to the table starting at EACALC. For the non-indexed cases, the microcode should do the appropriate manipulation of HR, but force the local flag on and block stores to the section number field. For indexed references, the action is a bit harder since there are two possible formats of index register. The best bet seems to be adding a new dispatch that allows the microcode to check the sign of Y (that's bit 18 of HR) and the format of the index register (that's bits 0 and 6-17 of the XR) at one time. The indexed cases would then call a routine using that dispatch which would sign-extend Y as appropriate and then add the contents of XR. The local/global flag and the section number in HR should be manipulated appropriately.

The rest of the changes necessary for EA-calc appear if control reaches FETIND to fetch an indirect word. Since the indirect word can be either an IFIW or an EFIW, an additional code must be added (note than an analogous change must be made for byte and EXTEND EA-calc). The indirect word cannot be written directly into HR because it may have one of two different formats.

The decoding of the indirect word is a two step processes. The first step uses a new dispatch to check whether it is an IFIW, an EFIW, or an illegal indirect word with one dispatch (see the section on hardware changes for details). For the case where the indirect word is an IFIW, the right half is loaded into HR and the XR and indirect fields are loaded into the hardware registers. For the case where the indirect word is illegal, a page fail trap is started. For the case where the indirect word is an EFIW, bits 1-5 are shifted to bits 13-17 and loaded into the XR and indirect bit registers. The normal XR and indirect dispatch can then be used to separate out the four cases.

The result is that HR contains the full 30-bit effective address in bits 6-35. Bit 0 contains the local/global flag and bits 1-5 contain junk. Note that bits 6-17 always contain the correct section number even if it is defaulted. This is important because many instructions depend on the fact that the section number of EA is correct.

As with HR, PC is extended to a full 30-bits. Since the hardware will ignore bits 1-5, we don't care what if they are junk. These bits must be cleared before the microcode stores PC into memory for things like MUUO, etc. This restriction also applies to storing EA.

Note that all stores into PC should be changed to force on the local bit. This will cause all PC references to be done section-local. In addition, all updates of PC to +1 should inhibit carry between the halves.

5.2.1.1 PXCT and the effective address calculation

As noted in the chapter on extended addressing, the normal EA-calc may be pre- or post-processed when PXCT is involved. For an instruction EA-calc, the pre-processing should be done in the XCT code itself since that code jumps into the middle of EA-calc in any event. The post-processing must be done in the instruction EA-calc code. To avoid wasting another cycle to check for this case, the need for PXCT post-processing should be built into the EA MODE DISP (see section on hardware changes for details).

For byte and EXTEND EA-calc, the pre-processing (initializing EA section with PCS) must be done in-line. With the appropriate dispatch on previous EA-calc, this can be done at entry to the byte and EXTEND EA-calc subroutines.

5.2.2 G-floating instructions

The addition of the four G-floating instructions (GFAD, GFSB, GFMP, and GFDV) and the G-floating conversion instructions under EXTEND should be viewed as somewhat high risk. In order to add these instructions, the entire SCAD path, including SC, FE, the SCAD input muxes, and the SCAD ALU would have to be widened from 10 to 13 bits to accommodate the larger exponent. While this in itself isn't difficult, there are certain microcode changes that make it risky.

Besides floating point operations, the SCAD path is also used for byte manipulation and counting. The existing hardware and microcode doesn't right-justify the P and the S fields from the byte pointer in FE and SC because the merge back into the pointer is done with the BYTE1 select through DBM. As a result, there are numerous places in the microcode that add 2, 4, or 8 to FE and SC when the logical intent is to add 1. As a result, making the SCAD path three bits wider would force a careful look at the microcode to find all uses of the SCAD path to make sure that the constants were correct. Also be wary of breaking the EXP merge for D-floating if the only available merge is 12 bits.

It might be better to ignore G-floating for the FCS machine and take the time to look at it more closely and upgrade in the field. If this is impossible, there should be a concentrated effort to simulate all uses of the SCAD path to make sure the microcode was changed in all the necessary places.

In any event, the actual addition of G-floating should be a nearly one-for-one copy from D floating with the exception that the exponent

is three bits wider and the mantissa is three bits narrower. Note that the KC10 data path is very similar to the changes necessary for the KD10, so the KC10 microcode may be of help in determining algorithms.

5.2.3 Unbiased rounding

The change from biased rounding to unbiased rounding is being done because unbiased rounding produces more accurate answers. However, this change should not cause a slip in the FCS date since it can be installed in a future microcode release. The techniques for doing unbiased rounding were worked out and installed with conditional assembly in the KC10 microcode, so that might be a good place to start in trying to define algorithms.

5.2.4 PUSHM, POPM, and PUSHI

These instructions were added to the instruction set to improve the performance of certain very common instruction sequences (saving and restoring AC during subroutine calls and passing arguments to subroutines). The implementation is fairly straight forward from the functional description of the instruction. The one tricky point is determining the format of the stack pointer and how to update it, but this is identical to that described for the other stack instructions below.

Note that the addition of these instructions for the FCS machine would be nice, but they should not delay the FCS date.

5.3 Changes to existing instructions

This section describes the changes that must be made to instructions currently implemented by the KS10 microcode.

5.3.1 Double word instructions

For any instruction which references two consecutive memory locations, there is a potential problem in computing the correct address for the second word. See the section on address computations for a description of the problem and a suggested solution.

5.3.2 LUUO

The changes to LUUO processing are limited to the addition of the case where the LUUO is performed in a non-zero section. The section zero processing is identical to the existing KS10 code. A check for PC section non-zero should be inserted at LUUO1 and jump to a new section of code if it is indeed non-zero.

The new code should check for user or exec mode and fetch the LUUO block pointer from UPT or EPT location 420. This pointer is the VIRTUAL address in the current address space of the first word of the four-word LUUO block as described in the section on LUUO handling.

The microcode should store the indicated information into the first three words of the LUUO block and then start the processor at the location specified by the fourth word of the block. Note that the processor flags and mode remain unchanged; only the PC is changed. Also note that if the effective address is a local reference to an AC that it must be converted to the corresponding global AC address before it is stored.

Note that trap enable (from WREBR) must be checked before a non-zero section LUUO is processed. If trap enable is off, the microcode should halt the machine instead of processing the LUUO as indicated.

5.3.3 MUUOs

The MUUO processing routine must be rewritten to conform to the new MUUO block format. It stores PC flags, CAB, PAB, PCS, PC, the opcode and AC fields of the MUUO instruction, and the effective address into the four-word block starting at UPT+424. Note that if the effective address is a local reference to an AC that it must be converted to the corresponding global AC address before it is stored. CAB, PAB, and PCS are kept in a location in the workspace. Also see the section above concerning the problems of obtaining the opcode and AC.

The largest problem with MUUO processing is determining where to fetch the new PC word from. Unlike the current KS10 microcode, the new PC word is a function of the kind of MUUO executed and the mode of the processor rather than whether a trap occurred or not. This means that there should be multiple entry points into the MUUO handler, one for each kind of MUUO. This was the scheme in the KC10 microcode which had an identical MUUO block format.

The new PC flags, and CAB and PAB are loaded from the word at UPT+430 and the new PC is taken from the appropriate word starting at UPT+432. Note that PCS is set from the value of PC section before loading the new PC.

Note that trap enable (from WREBR) must be checked before the MUUO is processed. If trap enable is off, the microcode should halt the machine instead of processing the MUUO as indicated.

5.3.4 Byte instructions

The changes to the byte instructions require major work on the microcode. The possibility of one- or two-word global byte pointers must be checked for, and the byte pointer EA-calc must be considered. Note that the check for two-word global byte pointers and the legality of extended byte pointer EA-calc is based on the section from which the byte pointer was fetched and not on PC section.

A major problem is converting one-word global byte pointers to the equivalent P and S format and back again. The KL10 used a translation table in the EPT to perform this transformation with a resulting performance problem. Since there is sufficient microcode space in the KD10, the translations can probably be done with microcode dispatches similar to that used by the KC10 microcode. Note that both the KL10 and the KC10 microcodes are a good source for algorithms for performing these instructions.

Note that the microcode must check for previous context references during the byte EA-calc and supply PCS if necessary as the section number of the byte EA. This point is discussed in more detail in the section on instruction EA-calc.

The byte instructions are critical to the performance of the machine. In addition, the one-word pointer formats are the most important and the code should be optimized for those formats. Additional microcode dispatches may be considered to improve the performance of these instructions.

5.3.5 Stack instructions

In all stack instructions, a check must be added for the type of stack pointer. If PC section is non-zero, the microcode must check to see if bit 0 is 0 and bits 6-17 are non-zero. If this is true, the stack pointer is global and the pointer should be incremented or decremented in 30 bits instead of as two 18-bit quantities.

In addition, PUSHJ and POPJ must change to check for PC section non-zero and manipulate the full 30-bit PC instead of PC flags and the in-section part of PC.

The KC10 microcode performed these instructions with little hardware assistance. It may be worth looking at that code.

5.3.6 JSR and JSP

The primary change necessary in these instructions is determining whether to store PC flags,,PC or a full 30-bit PC. This requires that the microcode check the current PC section and store PC flags,,PC if it is zero and a 30-bit PC (clearing bits 0-5) if it is not.

5.3.7 JSA and JRA

For JSA, the microcode must store AC into location E and the in-section part of E and PC into AC. Thus, it ignores the section number of E and PC. It then jumps to E+1, which must be calculated based on the local/global flag.

For JRA, the microcode must append the current PC section to the two halves of the AC to form the 30-bit addresses of the location in which AC was stored and the new PC. As a result, the section number of the EA-calc is ignored.

5.3.8 BLT

The largest change for BLT involves constructing the 30-bit source and destination addresses, and incrementing them properly. The source and destination addresses must be constructed by appending the section number and local/global flag from EA to the halves of the AC. It is important that the local/global flag by copied because this plays a part in the determination of a local AC reference.

Independent of the state of the local/global flags, the source and destination addresses must be incremented section-local. That is, the microcode must always suppress carries between bit 18 and bit 17 when the addresses are incremented.

5.3.9 XBLT

Because the definition of XBLT at the time the KS10 microcode was written made it illegal in section zero, the KS10 microcode has no support for it. The main transfer loop for XBLT is fundamentally that of BLT with the exception that the addresses are always incremented or decremented as full 30-bit quantities. There is also the backward BLT case if the count is negative. Finally, there is the question of cleanup and storing the correct values into the ACs at the end of the instruction which is different from the normal BLT code.

5.3.10 JRST

The changes to JRST include the addition of the new JRST decodes and the code necessary to handle certain new functionality.

In JRSTF (JRST 2,), the microcode must check for PC section zero. If the PC section is non-zero, the instruction traps as an MUUO. Otherwise, it does what it does now.

In XJRSTF (JRST 5,), the microcode must check for exec mode and load CAB, PAB, and PCS from bits 18-35 of the first word of the argument block. In both modes, the second word contains a full 30-bit PC

rather than an 18 bit PC. The same thing applies to the XJRSTF part of XJEN (JRST 6,).

In XPCW (JRST 7,), the microcode must store CAB, PAB, and PCS into bits 18-35 of the first word of the block and the full 30-bit PC into the second word. The loading of new values into these registers is the same as for XJRSTF above.

JRST 10, is now illegal and should be removed from the JRST decode table and replaced by an MUUO.

JEN (JRST 12,) is now illegal and should be removed from the JRST decode table and replaced by an MUUO.

SFM is now legal in all modes and in all sections, so the check for kernal mode should be removed. However, another check for kernal mode should be added and SFM should store CAB, PAB, and PCS into bits 18-35 of the word stored if the instruction is executed in kernal mode. In user mode, these bits should be set to zero.

The XJRST instruction (JRST 15,) has been added to the instruction set to set PC to the contents of the word at E without changing any flags.

5.3.11 XMOVEI and XHLLI

The SETMI and HLLI instructions must be converted to full XMOVEI and XHLLI functionality. This means that they store the full 30-bit effective address (or just section number for XHLLI) into AC. Since the only time that EA can have a non-zero section is if the instruction was executed in a non-zero section, there is no test necessary for section zero (i.e., EA always contains 0,,E if the instruction is executed in section 0).

There is one additional test required to convert local AC references to the equivalent global AC address. If the local/global flag indicates a local reference (i.e., bit 0 is on), and bits 6-16 and 18-31 are all zero, the address is a local reference to an AC. To convert to the equivalent global AC address, the microcode must simply insert a 1 into bits 6-17. Note that a section zero reference to an AC is never converted to the global AC form.

5.3.12 EXTEND string instructions

Most of the points mentioned for byte instructions apply here as well. However, the check for two-word global pointers and the legality of extended byte EA-calc is based on PC section.

The KL10 converted all one-word global pointers to two-word global when it stored them back in the ACs of the string instruction. This caused nothing but problems architecturally and should not be done here. This problem was solved in the KC10 microcode and that should

be used an example.

The PXCT relationship with string instructions is different from byte instructions. As a result, the need to add PCS at the appropriate point of the EA-calc is done based on different PXCT bits.

In addition to these points, the EDIT microcode must be changed to check for PC section zero and ignore bits 6-17 of the pattern string address and the mark address in that case. If PC section is non-zero, these bits are used.

5.3.13 The privileged instructions

This section discusses the changes necessary to convert the existing KS10 privileged instructions.

5.3.13.1 APRID

The only change necessary for this instruction is to adjust the field widths as appropriate.

5.3.13.2 WRAPR

Bit 18 has been added as a PI enable bit for WRAPR. The microcode must check bit 18 before loading a new APR PIA from bits 33-35

5.3.13.3 SETCU

This instruction may be difficult to implement without some amount of hardware help. The idea is to set the CST update needed bit in the translation buffer without altering the state of any other bits. Conceptually, the microcode would read each entry in the TB, set the bit, and write it back. The fact that the microcode can't read a TB entry back into the data path causes some problems with this algorithm.

The solution may be to take advantage of one of the performance gains mentioned in the description of the page fail algorithms below. In that scheme, 1K in the workspace is dedicated to a copy of the bits in the translation buffer. Therefore, the microcode simply reads the appropriate location in the workspace, sets the CST update bit, and writes that entry back into the TB. If this scheme isn't used, there could be a problem implementing SETCU.

5.3.13.4 RDUBR and WRUBR

Because of the additional information required, the WRUBR instruction now has three words of argument. The first word contains control bits which the microcode uses to determine what to do. The changes to WRUBR are straight forward based on the functional description and are similar to the existing KS10 instruction flow. If the "load AC blocks" bit is on, the microcode must load the values into the hardware from bits 18-23 of E+1. If the "load PCS" bit is on, the microcode should replace the current value in the workspace. Since PCS is manipulated totally by the microcode, there appears to be no reason to add a hardware register to hold the value. If the "load UBR" bit is on, the microcode does what the KS10 microcode does. In addition to clearing the translation buffer and data cache, it must clear and reinitialize the cache of paging information in the EBOX. A major change is the manipulation of the address break information if the "load address break" bit is on. Since this hardware doesn't exist on the KS10, the exact microcode action is subject to the addition of the required hardware.

For RDUBR, the microcode must reconstruct the information passed to it in the last WRUBR, setting the specified control bits to the correct state. The UPT address is contained in the register file and must be converted back to a page number. CAB, PAB, and PCS are stored in the workspace, as are the current address break conditions.

5.3.13.5 RDEBR and WREBR

The WREBR instruction must change from an immediate instruction to an instruction which reads its argument from location E. The instruction flow is similar to that for the KS10 WREBR except there are separate trap enable and pager enable bits, and the trap enable bits has a load bit. The result is that the microcode must do more checking before setting bits 22 and 23 in the APR enables.

Since there is no longer room in the register file location EBR to store pager enable and trap enable, the RDEBR instruction will have to be changed to extract the state of those bits from workspace location APR.

5.3.13.6 CLRPT

for CLRPT, the microcode should load the entire 30-bit effective address into VMA and clear the appropriate page table entry. Because the cache is physically addressed, it is no longer to sweep the cache, so that code may be removed from CLRPT. Finally, the internal cache of paging information must be cleared from the workspace.

5.3.13.7 PMOVE and PMOVEM

The hardest part of these instructions is doing the physical EA-calc computation on the argument. This computation should be fairly straight forward from the functional description. After that, the physical memory reference is done as any other physical memory reference would be.

5.3.13.8 LDPAC and STPAC

The LDPAC and STPAC instructions are primarily a core BLT-like transfer loop for the number of words-1 specified by the AC number in the instruction. The KC10 microcode implemented these instructions and used a single transfer loop for both with checks in the loop to see if the memory reference was a load or store. One of the two references in each pass through the loop is to current context memory (i.e., normal memory read or store) and the other is to a previous context AC. Note that this instruction does not have to be interruptable.

5.3.13.9 MAP

The changes required for MAP fall out from the microcode changes to the page fail handler. Since the valid, modified, writable, and cachable bits are in the same position as on the KS10, this all stays the same.

If there is no valid mapping, the page fail handler returns the page fail word to MAP, and MAP should clear the left half of the word. If there is a valid mapping, the KS10 code at PF130 cleans things up for MAP. Note that the AND of #/3 at PF130 must be changed to #/17 to account for the larger physical address.

There is one optimization possible if 1K of the workspace is allocated for a readable copy of the translation buffer (see the section below). In this case, the MAP instruction should see if there is a valid translation in the translation buffer before going off to do the pointer trace. This could be a performance improvement.

5.4 Other functional changes

This section describes other functional changes.

5.4.1 Processing page fails

This section discusses the changes that must be made in the page fail handling microcode.

5.4.1.1 Classifying page fails

Page fail processing on the KD10 is a bit different from the existing KS10 code. The modifications to the translation buffer and page fail logic described in the chapter on CPU hardware changes result in 5 major classes of page fail conditions. Each condition is covered separately below.

5.4.1.1.1 Interrupt, NXM, or memory error

These conditions take precedence over other conditions and are handled exactly the way the KS10 microcode handles them.

5.4.1.1.2 Invalid translation

An invalid translation fault forces the microcode to do a pointer trace in an attempt to find a valid translation for the virtual address. The pointer trace logic is quite similar to the existing KS10 microcode with the exception that it should keep more information about the source of the last pointer fetched to store in the additional data words if the page fault is given to the monitor. Note that this change should not be made if it will delay FCS.

Note that the translation buffer entry is written from different data path bits than on the KS10 in order to keep the microcode from having to juggle bits around. A translation buffer entry written as the result of a pointer trace should always clear the CST update needed bit in the entry since the pointer trace performed a CST update along the way.

5.4.1.1.3 Address break

An address break fault causes a page fail trap to the monitor. Therefore, processing this condition consists of jumping to the page fail trap microcode.

5.4.1.1.4 Write violation

A write violation fault can occur for one of two reasons. The page can be marked as not writable in which case the microcode should start a page fail trap to the monitor.

It can also be writable but not yet modified. When the microcode originally wrote the TB entry for the page, it turned the writable bit in the TB off if the page was writable, but not yet modified (according to the M bit in the CST entry for the page). This page fault is necessary so that the microcode can perform a CST update on the page and set the M bit in the CST entry. When the CST update is completed, the microcode should rewrite the TB entry with the writable bit on. Because a CST entry has just be performed, the microcode should also clear the CST update needed bit.

Note that in order to determine whether the page is not writable or writable but not yet modified, the microcode must keep the writable and modified bits for each entry. See the discussion below for more information.

5.4.1.1.5 CST update needed

A CST update needed fault is generated when the CST update needed bit is found set in a TB entry. The microcode processing consists of doing a CST update on the physical page corresponding to that entry and then rewriting the entry with the CST update needed bit turned off. See below for a description of the problems in obtaining the physical page number.

5.4.1.2 Reading a translation buffer entry

The KS10 microcode performed a pointer trace on just about every type of page fail. This was primarily because it couldn't read the translation buffer entry that caused the page fail, nor could it obtain the PMA even if there were a valid translation in the TB.

To get around this problem in the KD10, we suggest using 1K of the workspace to store a copy of the translation buffer entry that is contained in the TB RAMs. Doing so means that the microcode can read a TB entry (from the workspace), manipulate one or more bits, and write the entry back into the translation buffer (both the RAMs and the workspace). This function is useful in processing both a write violation and a CST update needed fault.

In addition, keeping the entry around means that the microcode can construct the PMA if there was a valid translation. This is necessary in order to find the correct CST entry for a page.

Using the workspace in this manner means that the microcode need not always do a pointer trace on a page fail that was the result of

something other than an invalid translation. This could have a significant performance advantage. Note that this requires that the workspace locations be written at the same time as the TB RAMs, including during TB sweeps.

5.4.1.3 Trap enable

If any page fail is given to the monitor for processing, the microcode must check the state of trap enable. If trap enable is off, the microcode should halt the machine rather than start the page fail in the normal manner.

5.4.2 Processing traps

Trap processing on the KD10 is more complex than simply executing an instruction as on the KS10. Instead of executing the instruction stored in EPT/UPT locations 421-423, the KD10 treats this word as a function code and a function specific argument as described in the chapter on trap handling.

If the trap function is a no-op, the microcode simply clears the trap flags and jumps back to the next-instruction logic.

If the trap function is an MUUO, the MUUO new PC is taken from the function specific argument rather than from the MUUO block in the UPT. The new flags are still taken from UPT location 430.

If the trap function is an LUUO, the function-specific argument contains the virtual address of the 4-word LUUO block in the current processor mode. Other than clearing the trap flags, the PC flags are unchanged.

The latter two functions can most likely simply jump into the MUUO and LUUO code after proper setup. The one problem that may exist is the interaction between the trap processing and a page fail or interrupt detected during that processing. If an interrupt or page fail trap goes to the monitor during the processing of a trap function, the trap bits must be stored correctly in the interrupt XPCW or page fail block so that the trap processing gets restarted after the interrupt or page fail is processed. The KS10 hardware and microcode seem to use TRAP CYCLE for detecting this case, but it may have to change since the KD10 doesn't XCT an instruction any more. This needs to be looked at in more detail.

5.4.3 Processing interrupts

Interrupt handling on the KD10 is quite similar to that on the KS10. The difference is that there is no instruction XCTed by the processor on the KD10 as there was on the KS10. Instead, the word that formerly

contained an instruction contains a 30-bit exec mode virtual address of an XPCW block. The microcode then simulates an XPCW instruction using this address as the effective address of the simulated XPCW.

Since the KS10 only allowed XPCW and JSR as interrupt instructions, the changes mostly involve removing code. The code starting at PI50 which checks for an XPCW or a JSR can be removed, as can the code at PIJSR. The microcode simply stores the 30-bit address of the XPCW block into AR and then jumps to PIXPCW as it does now.

5.4.4 Changes for a VMA and VMA flags

Since the VMA address has been expanded from bits 14-35 to 6-35 plus the local/global flag, the way the microcode operates on the VMA and VMA flags must change. In particular, the hardware will allow the VMA address and VMA flags to be loaded separately. In addition, the VMA and VMA flags are now read back into the data path separately, rather than together as in the KS10.

In most instances, the VMA flags are set from microcode # field bits when VMA is loaded. In this case, the function of loading VMA and the VMA flags remains unchanged with the exception that the microcode must assert the additional enable to get the VMA flags loaded. There is one change, however. Since the local/global flag is carried around in bit 0 of the data path, the hardware loads the VMA local/global flag from that bit when the VMA address bits are loaded. As a result, the microcode must insure that bit 0 is in the correct state to represent the local/global characteristics of the VMA being loaded.

In the case where the VMA flags are loaded from the data path rather than from the microword # field (DP FUNC asserted in the microcode), things change a bit. Because there are insufficient bits to hold the VMA address, the local/global flag, and the other VMA flags in one 36-bit word, the microcode must perform separate operations to load VMA flags and VMA address. This type of operation is done rarely in the microcode (interrupt, IORD, IOWR, and page fail) and the impact of doing this should be nil. Typically, the microcode would load the VMA flags first, followed by the VMA address and local/global flag using the independent enables provided by the hardware.

An analogous situation exists for reading VMA and VMA flags. The local/global flag and the VMA address bits are now available via DBM mixer bits 0, and 6-35. The VMA flags are available via DBUS bits 22-35. To get both the VMA and VMA flags, the microcode must read them independently and store them independently. Once again, this is done rarely (and most of those cases only want one or the other). It would appear that page fail is the only processing routine which needs to read and save both VMA and the VMA flags.

5.4.5 Getting address computations correct

As indicated in the chapter on extended addressing, the carry between bit 18 and bit 17 (i.e., into the section bits) during an address computation is a function of the local/global flag associated with the address. That's one of the reasons that the local/global flag is necessary in the register file. To get these address computations correct, the microcode needs to enable conditional carry control based on the local/global flag whenever address computations are done. The exact implementation of this is unclear, but there should really be some hardware support for this function so that the microcode isn't always checking bit 0 to see if a full carry should be done or not. See the chapter on required hardware changes for some suggestions about the hardware support required.

5.4.6 Storing PC and EA

Because both PC and EA use bit 0 as the local/global flag, and both may contain junk in bits 1-5, bits 0-5 must be cleared before PC and EA are stored into memory for things like MUUO, stack instructions, etc. If PC or EA are being stored in 18 bits, there's no problem, since the microcode can simply do half-word manipulations.

Also note that an EA that is a local reference to an AC must be converted to the equivalent global AC address before being stored for XMOVEI, XHLLI, MUUO, non-zero section LUUO, and page fail.

CHAPTER 6

CPU HARDWARE CHANGES

This section discusses the changes that must be made to the CPU hardware to upgrade it from the KS10 design to the KD10 design.

6.1 Changes necessary for EA-calc

The following changes are necessary to support the new EA-calc microcode algorithms:

1. Because EA has been expanded into the left half of HR, there is no longer room to store the opcode and AC field of the instruction. Because certain instructions require this information, the opcode and AC hardware registers must be able to be read back into the data path. At present, it's not clear what the best way to do this is.

2. The microcode must have the ability to force set/clear bit 0 of AD. Note that this is the output of the ALU and not the output of the 2901s which makes it harder to do. We may have to dink the function and the D inputs to the 2901 to get the desired affect. Also note that any bits to the left of bit 6 are fair game and can be treated as indeterminate in the operation. Therefore, if those bits are modified along with bit 0, that's fine.

3. The microcode must have the ability to block the write enable to bits 6-17 of the 2901 register file. This is analogous to the HOLD LEFT and HOLD RIGHT macros, but must be independent of both.

4. The microcode must have the ability to control the AD carry from bit 18 to bit 17 based on the state of bit 0. This is probably some modification to the existing SPEC decode. Where the bit 0 control comes from is undefined at this point.

5. The microcode needs a new dispatch that will allow it to check the sign bit of the Y field (bit 18) and the format of the index register in one dispatch. Since the Y field is in

the register file, DP<18> seems to be the logical candidate for that dispatch bit. The other dispatch bit comes from the output of the RAM file (or DBUS) and computes the function (RAM<0>==1.OR.RAM<6:17>==0).

6. The microcode needs a new dispatch to check for IFIW and illegal indirect in one dispatch. The two bits involved are as follows:

```
IFIW := (VMA<6:17>==0 .OR. (DBUS<0>==1 .AND. DBUS<0>==1))
ILLIW := (DBUS<0>==1 .AND. DBUS<1>==1)
```

The combinations of these bits yield the following table:

IFIW	ILLIW	Result
0	0	EFIW
0	1	Illegal indirect
1	0	IFIW
1	1	IFIW

7. The microcode needs a new dispatch to determine if an EA-calc must be pre-processed for PXCT. This dispatch would allow the microcode to check for the case that PXCT is enabled and bit 9 or 11 is on. Note that the microcode must be able to select which bit because one is used for normal instruction EA-calc and the other is used for byte EA-calc. This should probably be a SKIP condition to make is most useful for entry into the byte EA-calc subroutines.

8. The microcode needs a new dispatch to determine if an EA-calc must be post-processed for PXCT. Because this check is at the end of every EA-calc, the best bet appears to use (the currently unused) bit 11 of the EA MODE DISP dispatch. To limit the number of microcode locations used, the bit should compute the function:

(-JRST .AND. -INDIRECT .AND. -E BIT .AND. D BIT .AND. PXCT ENABLED).

where "E BIT" and "D BIT" are bits 9 and 10 or 11 and 12 of the PXCT AC register. Which pair of bits is used must be selected by the microcode since one is used for instruction EA-calc and the other is used for byte EA-calc.

6.2 Dispatches for stack instructions

In order to make the check for the type of stack pointer fast, the microcode could use a single dispatch that let it check PC section non-zero and the format of the stack pointer with a single dispatch. Because PC is contained in the register file, the easiest thing to do

may be to take advantage of the dispatch being added to check for the format of XR (see point 5 under Changes necessary for EA-calc above) and add AD left.NE.0 as another bit in that dispatch. This lets the microcode read PC from the register file and mask the section bits at the same time it reads AC from the AC blocks. The dispatch is then to 1-of-4 locations.

6.3 Hardware support for G-floating

The addition of G-floating instructions requires that the SCAD path be widened from 10 to 13 bits. See the comments in the chapter on microcode changes for a discussion of the risk involved in making this change.

The actual addition of hardware is fairly straight forward. Three additional bits should be added to the right of bit 9 (making them 10, 11, and 12) for the FE and SC registers, the SCADA and SCADB input mixers, and the SCAD ALU. At first glance, it would appear that the inputs to the new SCADA and SCADB mixers would be analogous to the bit 9 of the existing mixers.

6.4 Workspace

The KS10 workspace is implemented with 1Kx1 RAMs addressed by 10 bits selected from the RAMFILE ADR mixer. To allow room for a bigger cache and more scratchpad area, the KD10 workspace uses 4Kx4 RAMs. Because the RAM is four times larger, two additional bits are needed for the address, so two additional address mixers are required.

The current RAM address bits are RAMFILE ADR 00 through RAMFILE ADR 09. Let us call the two additional bits RAMFILE ADR -2 and RAMFILE ADR -1. The changes to the RAM file addresses are limited to the addition of the two new bits, plus changes to RAMFILE ADR 00 for one case. The selects and enable for both new bits is identical to that for RAMFILE ADR 00. For each select, the inputs to the three mixers in question should be:

Select	RAMFILE ADR -2	RAMFILE ADR -1	RAMFILE ADR 00
0	+3V	TB PMA 25	TB PMA 26 (changed)
1	GND	GND	GND
2	VMA 24	VMA 25	VMA 26
3	DBM 24	DBM 25	DBM 26
4	GND	GND	GND
5	GND	GND	GND
6	VMA 24	VMA 25	VMA 26
7	DBM 24	DBM 25	DBM 26

Where TB PMA 25 and TB PMA 26 are PMA bits 25 and 26 after translation through the TB.

6.5 Extension of VMA

Since the virtual address has been extended to 30 bits, the size of the address portion of VMA must be expanded from bits 14-35 to 6-35. This requires another two D flops on DPM4. There are other implications of making this change, however.

At present, the microcode can load VMA and VMA flags at one time and read both into the data path through select 5 of the DBM mixer. Because 8 additional bits have been added to the address portion of VMA, it is no longer possible to operate on both VMA and VMA flags at one time. This is normally not a problem for loading the VMA flags (see the section on microcode changes for details). However, the ability to read and write both VMA and VMA flags is still required.

To retain the existing functionality, there are several additional changes that must be made, as follows:

1. The microcode must have independent control over VMA and VMA flags. At present, there is a single control (VMA EN) which causes both to be loaded. This could be done by using another microword bit to enable loading the VMA flags.

2. The local/global flag carried in bit 0 of the data path should be latched along with the VMA address bits from bit 0 of the data path. This will be used as part of the determination of an AC reference.

3. The VMA address bits should be read into data path bits 6-35 via select 5 of the DBM mixer. The local/global flag should be read into bit 0 via the same mixer. Bits 1-5 of the mixer should be tied low so that those bits appear as zeros.

4. The VMA flags should be read into data path bits 22-35 of the DBUS mixer on DPE4, replacing the VMA bits. The VMA flags and the appropriate bit connections are:

Bit	VMA flag signal
22	VMA USER
23	NC
24	VMA FETCH
25	MEM READ
26	MEM WR-TEST
27	MEM WRITE
28	NC
29	MEM CACHE INH
30	VMA PHYSICAL
31	VMA PREVIOUS
32	VMA I/O
33	VMA WRU CYCLE
34	VECTOR CYCLE
35	I/O BYTE CYCLE

5. When the VMA flags are loaded from the data path (as opposed to from # bits; see DPM4), the input bits should be changed to correspond to the bit assignments above. This makes it easier and faster for the microcode to save and restore the VMA flags.

There is one additional change to the VMA logic. Because the AC REF rules change with extended addressing, there is more logic involved in the determination of AC REF (see DPM4). Besides the existing logic, an address references an AC instead of memory if bits 6-16 are zero or if the local flag is on (this assumes that the check for VMA 18-31 equal zero remains).

6.6 PI changes

The current KS10 design allows a software interrupt to be started even if the level is off. The KD10 should be changed to start the software interrupt only if the level is on. This change can be made by reversing the order of the AND and OR gates whose inputs are PI ACTIVE n L and PI SOFT REQ n L on DPEB.

6.7 Translation buffer

The KS10 translation buffer used pairs of chip-selected 256x4 RAMs to construct a 512 word translation buffer. In order to improve the hit rate in the translation buffer, the KD10 uses 1Kx4 RAMs which double the size of the translation buffer. As with the KS10, the TB is one-way associative and has a one-word block size.

The increased size of the data cache requires PMA bits 25 and 26 for the address and these bits have to be translated through the TB before the cache can use them. The use of 1Kx4 RAMs for bits 25 and 26 may not work because these bits may have to be translated quickly in order to supply the cache address early. If this is so, we may have to continue to use 256x4s for these two bits. The following discussion assumes that 1Kx4s are used throughout but the changes to use 256x4s for two bits should be straight forward.

The contents of the translation buffer RAMs is considerably different from the KS10 since the virtual address space is larger than the size of the TB. In addition, the data cache uses physical addresses, so the translation buffer must hold the translated PMA. The bits in the translation buffer are as follows:

Signal	No. Bits	Use
PAGE VALID	1	Indicates entry is valid
PAGE WRITEABLE	1	Indicates page is writable
PAGE CACHEABLE	1	Indicates page is cachable
PAGE USER	1	Indicates entry is for user space
PAGE PARITY	1	Parity bit for entry
PAGE CST UPDATE	1	Indicates CST update is needed
PAGE VMA 6:16	11	Rest of VMA for this entry
PAGE PMA 14:26	13	PMA page bits for this entry
Total	30 = 8 1Kx4 RAMs	

The translation buffer is addressed by VMA bits 17:26 with bit 17 XORed with VMA USER to offset equivalent exec and user entries by half the translation buffer.

The check for a page fail is quite similar to the existing KS10 logic with a few changes to the priority encoder (E409) on DPM6 and the addition of several comparitors. Three comparitors should be added to compare PAGE VMA<6:16> with VMA<6:16> and PAGE USER with VMA USER. The cascaded result of these three comparitors replaces the XOR gate PAGE FAIL 5 EN on DPM6. These comparitors compare the rest of the virtual address of the entry in the TB with the requested address plus the requested address space with that stored in the entry.

The inputs to the priority encoder should then be arranged as follows:

Input	Signal
3	- PAGE VALID
4	- comparitor match (from above)
5	Address break
6	PAGE FAIL 6 EN (existing write-test check)
7	PAGE CST UPDATE (new bit in TB)

The inputs have to change around from what they are in the KS10 design because there are two new conditions (address break and CST update needed) which can cause a page fail. Inputs 3 and 4 check for a valid match, i.e., the entry is valid, the upper VMA bits match the requested VMA, and the address space is correct. Conditions 5-7 all require that the entry be valid in order for them to be valid. Input 5 comes from the address break logic described elsewhere. Input 6 comes from the existing write-test check, and input 7 comes from new CST update bit in the TB.

The data inputs to the TB RAMs should also be moved around to make is easier and faster to reload a TB entry. The data inputs should be as follows:

RAM signal	Data input comes from
PAGE VALID	DP 2
PAGE WRITEABLE	DP 4
PAGE CACHEABLE	DP 6
PAGE USER	VMA USER
PAGE PARITY	Parity generator
PAGE CST UPDATE	DP 3
PAGE VMA 6:16	VMA 6:16
PAGE PMA 14:26	DP 23:35

The inputs to PAGE WRITEABLE and PAGE CACHEABLE should be changed so that the microcode can load the result of the AND of the W and C bits from the pointer trace (which happen to be bits 4 and 6) directly into the TB.

6.8 Cache

The KS10 data cache occupied the top 512 locations of the workspace. The cache directory was constructed from pairs of chip-selected 256x4s. The cache was also virtually addressed.

The KD10 cache has been expanded to 2K and occupies the top 2K locations in the 4K workspace. Like the KS10 cache, it is one-way associative, and has a one-word block size. The addressing of the data cache (in the workspace) is described in the section on the workspace above.

The cache directory is constructed of pairs of chip-selected 1Kx4s. Although it could be constructed of 4Kx4s which only use half the RAM, there is a performance advantage in using the 1Kx4s. Not only are the 1Kx4s faster than the 4Kx4s, it also takes half the time to sweep the cache with 1Kx4s because both RAMs in the pair can be enabled during the sweep. This scheme is used in the KS10 design to speed up cache sweeps. Making cache sweep fast is important because the cache is swept on every WREBR and WRUBR that changes the UPT to insure that there is no stale data in the cache after an I/O write.

The cache directory is addressed by TB PMA<25:26> and VMA<27:35>. It contains CACHE PMA<14:24>, 1 valid bit, and 1 parity bit. The data input lines to the directory come from TB PMA<14:24>.

The cache directory comparitors compare CACHE PMA<14:24> with TB PMA<14:24> as part of the check for a cache hit. The rest of the cache hit logic is similar to that of the KS10.

6.9 Microcode dispatches

There is a distinct lack of available dispatches in the KS10 microcode. Since so much new functionality is being added, it might be a good idea to add another bit of enable to the DISP field to allow

an additional 8 dispatches.

6.10 Miscellaneous

To make it easier for the microprogrammer, the inputs to the CURRENT BLOCK and PREVIOUS BLOCK registers on DPE5 should be changed from DP<6:11> to either DP<0:5> or DP<18:23>, whichever is easier for the microcode. This change is due to the change in format between the KS10 WRUBR and that on the KD10.

The trap enable bit on DPEB should be loaded from DP<8> instead of DP<22> due to the change in the format of WREBR.

The address break logic needs to be added if it will fit. The style of address break is similar to that on the KL10. That is, there is one virtual break address and the ability to break on exec/user reference for a read, write, or instruction fetch. The address and qualifier compare can be driven directly from VMA. There also needs to be a way for the microcode to load the address, qualifiers, and the enable into the hardware from the data path. Since the microcode would probably keep the last setting of address break in the workspace, there is not need for the address break registers to be read back into the data path. Note that, in addition to the comparitors, the inhibit address failure PC flag must be added and cleared at the right time. Copying the KL10 implementation might be a good idea since the KC10 was radically different.

CHAPTER 7

MEMORY CONTROLLER AND MEMORY ARRAY CHANGES

This chapter discusses the hardware changes necessary to the KS10 memory controller and memory array modules.

7.1 Memory controller

Because the KD10 bus contains 22 bits of address lines, support for a full 4 MWords of physical memory (which happens to require 22 bits) is easier than it might be. The main changes seem to be the inclusion of two additional row/column bits on MMC3 and the appropriate changes to the address match logic on MMC8.

At present, the address match logic is designed with the knowledge that the KS10 would support only 512K of memory and the "MOS BOARD IN" mixer and the XOR gates comparing bits 14-16 of the address will have to change to reflect the larger size of the array boards.

7.2 Memory array modules

The KD10 memory array modules use 256K MOS RAM chips to supply 1 MWord by 44 bits per module. To convert the KS10 array boards (which use 16K RAMs), the main change seems to be the addition of n copies of two additional bits worth of MOS drivers (see MMA2).

CHAPTER 8

I/O ADAPTER HARDWARE CHANGES

This chapter discusses the changes to the I/O adapter hardware to support the KD10 I/O structure.

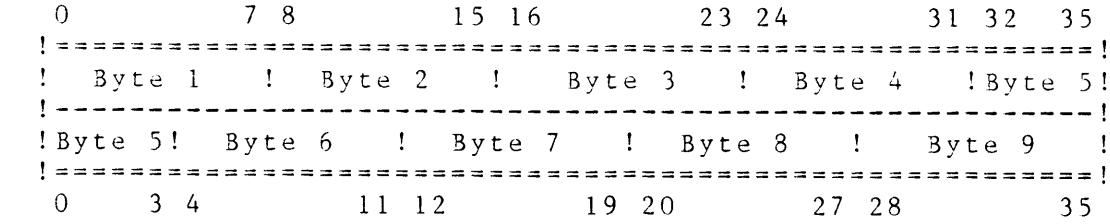
8.1 Changes to the KS10 UBA

There are two obvious changes required to the KS10 UBA. The first requires adding two additional bits to the UBA paging RAM to support the full 22-bits of physical addressing. The two additional bits go into the mixer on UBA8.

In addition to this, there is a much larger change required to support the UDA50. At present, the UBA supports PDP-11 byte and word transfers, and Massbus transfers by using the two parity lines and the 16 data lines to provide a path for the 18 bits of data. The UDA50 only supports 16-bit PDP-11 word transfers and the chance of getting this changed to use the parity bits for 18 bit transfers seems low.

As a result, the UBA must be changed to do some rudimentary data packing and unpacking to convert 36-bit memory words to and from 16-bit PDP-11 words on the Unibus. In order to add this kind of logic, something else will probably have to be removed. As a result, we'll probably need two kinds of UBAs; one for the UDA50 and one for other Unibus devices.

In order to perform the required data packing, the UBA needs a 72-bit assembly register on the memory side into which it can pack or unpack 9 8-bit PDP-11 bytes. The data packing used is commonly called high density mode, which can be described pictorially as follows:



Note that byte 5 is split across the two words. Because these two

words contain an odd number of bytes, it may need a holding register on the Unibus side also.

8.2 Changes to the UDA50

At present, the UDA50 supports only 512 byte sectors. A PDP-10 disk sector contains 128 36-bit words, or 576 bytes. The HSC50 supports both 512-byte and 576-byte sectors and for performance and convenience reasons, it would be nice if the UDA50 did the same thing.

A brief conversation with one of the UDA50 engineers in Colorado led us to believe that the change required to support 576-byte sectors was a simple change to the UDA50 microcode. This should be pursued to determine what the cost of the change would be.

The alternative is to store one PDP-10 disk sector on two 512-byte sectors on the RA60/RA81. This will cost in both performance and in disk capacity of the disks and will also make the disks unreadable on a non-KD10 TOPS-20 machine.

CHAPTER 9

CONSOLE HARDWARE AND MICROCODE CHANGES

This chapter discusses the console hardware and microcode changes that are necessary to upgrade the KS10 design to the KD10 design.

9.1 Console hardware changes

The obvious console hardware changes fall into three categories; there may be others.

In order to get the KD10 started the first time, we probably need some sort of boot device. The cheapest seems to be the TU58 (used on the HSC50, etc.). If it turns out that we do indeed need a boot device, the console hardware must be changed to provide some sort of interface to that device.

The ability to obtain a performance of 0.3 to 0.5 times a KL10 requires that the KS10 clocks be increased in frequency. Our very preliminary investigations indicate that it might be possible to double the speed of the clocks from 150 ns to 75 ns. To do this, the clock logic on the console module must be changed. This might be as simple as replacing the crystal oscillator.

One performance optimization that was suggested for the KS10 was increasing the resolution of the microcode T field. With the KS10's 150 ns cycle time, each additional number in the microcode T field increased the microcycle time by 150 ns. In some cases, this was considerably in excess of what was really required. It was suggested that each additional value in the T field extend the microcycle time by half the cycle time. Therefore, with the proposed KD10 cycle time of 75 ns, each T field value would increase the microcycle time by 37.5 ns. To do this will require changes to the T field logic on CSL5 as well as increasing the size of the microcode T field by 1 bit.

9.2 Console microcode changes

We haven't looked closely at the changes required to the console microcode. As a result, there are bound to be more changes than are listed here. A close examination of the console microcode will be necessary to determine what additional changes are necessary.

There are two areas that obviously require change. The first is the need to add code to support the boot device. If we use a TU58 as the boot device, a driver will have to be written to allow the console to read and write the TU58. The same is true of any other boot device.

In addition to this, the console microcode must be changed to add RA60/RA81 disk support. To do this, the console microcode must have the code necessary to position and transfer to an RA60/RA81 through a UBA and a UDA50 disk controller. This may be non-trivial both in complexity and the amount of code involved. If additional code space is needed, the tape driver support may have to be removed.

CHAPTER 10

SOFTWARE CHANGES

This chapter discusses the monitor and diagnostic software changes that are required to support the KD10 design. The KD10 design is very similar to the KC10 design, at least from a functional view point. As such, the recommended approach is to start with the KC10 monitor and diagnostic work and upgrade that to minimize the amount of work necessary for FCS.

It should be noted that this list of changes is the result of our talking to various people in the software groups and not the result of exhaustive investigations by these groups. A more detailed investigation needs to be done in order to insure that this list is complete.

10.1 Monitor software changes

The changes described in this section assume that the existing KC10 monitor code is used as a starting point for all processor-related code. Because I/O is much different from the KC10 design, it is also assumed that the old KS10 I/O code is used as a starting point for I/O-related code.

The first step is to make the changes to the existing code to reflect the fact that the KD10 processor-related design isn't quite the same as the KC10 design. The changes related to the processor are as follows:

1. Update parameter files for new opcode and bit definitions.

2. APRID stores one word at location E instead of two words at locations E and E+1.

3. The SWPUA and SWPIA instructions have been removed and should not be used.

4. The APR flags defined by WRAPR/RDAPR are different from the KC10 and the monitor code should change accordingly. Note that most of these changes are in the error handling code.

5. The SETCU instruction has been added but the monitor shouldn't be changed to use this if it will delay FCS.

6. The format of WRUBR/RDUBR has changed and replaces the KC10 WRCTX/RDCTX.

7. The interval timer and timebase instructions are significantly different. Since they are identical to the KS10, that code could be lifted from release 4. Note that there is no user runtime meter with which to do accounting.

8. The halt status block address must be setup with WRHSB during monitor initialization.

9. There is no WRIOP instruction. The console communication must be done via physical page 0 because that's where the 8080 looks.

10. The format of MAP has changed in that the valid bit has moved from bit 3 to bit 2. In addition, the number of bits of physical address has changed.

11. The page fail word format is very different. If the monitor page fail handler was looking at any bits other than the user bit, the level field, and the code field, there may not be equivalent bits in the KD10 page fail word. Also, the FCS machine probably won't store the additional data words in the page fail block. The monitor shouldn't depend on these words for FCS.

12. Interrupts are handled with interrupt vector words as on the KC10, but there is no I/O page in which the words are stored. Rather, the words are either stored in the EPT or in tables pointed to by the EPT.

Note that BOOT and DDT must also be changed as appropriate.

The largest changes to the monitor involve the I/O-related code. Because the I/O structure is very similar to that on the KS10, it bears little resemblance to the KC10 structure. Ideally, the monitor code would start with the KS10 code. The problem is that KS10 support was removed from TOPS-10 in release 5. As a result, it may be difficult to merge the KS10 code back into the new monitor in any mechanical way.

In any event, the I/O-related changes are as follows:

1. A new disk driver must be written for support of the RA60/RA81 disks through the UDA50 controller. The UDA50 talks a subset of MSCP, but does no transfer optimizations. As a result, the code requires pieces from the old PHYSIO code (which did its own seek and transfer optimizations) and the new PHYKLP code which talks MSCP, but lets the HSC do all the optimizations.

2. A KS10-like terminal line driver must be written to handle terminals and the KLINIK lines. This may differ from the KS10 driver because the current plan is to use DMZ32 and DMF32 line interfaces which may use different protocols than the KS10 DZ11s.

3. Some sort of DECnet driver must be implemented. This could be a serious problem because the KS10 DECnet code was all phase II and upgrading the existing phase III code to support the KS10 could be a major undertaking. The ultimate solution is obviously phase III over the NI, but that won't be supported in TOPS-20 until release 6.1. As a short-term expedient to getting the system out the door, we may have to live with phase II support until 6.1 is ready. Note that such a decision means that a KD10 wouldn't be able to talk directly to a VAX (since the VAX phase IV won't talk to phase II).

4. A new line printer driver similar to that on the KS10 will be required to drive line printers through the DMF32.

5. If tapes are supported on the KD10, a new tape driver may be required.

10.2 Diagnostic software changes

Because the KD10 hardware design is similar to the KS10 and the architectural design is similar to that of the KC10, a large number of diagnostics can be converted from the KS10 and KC10 efforts.

The KD10 architectural design is very similar to that of the KC10 from a functional viewpoint. As such, it seems most profitable to convert the KC10 functional diagnostics (DCKAA-DCKAO, DCKBA, etc.) to be compatible with the KD10 design. The first step in this process is to convert the KC10 diagnostic monitor and functional diagnostics to take into account the differences between the KC10 and KD10 designs (a process similar to the monitor software effort). Beyond that, the KC10 diagnostic monitor must be converted to the KD10-style I/O scheme, which can probably be lifted from the KS10 diagnostic monitor.

In addition to this, some KC10 functional were never completed (the extended addressing and PXCT test, for example). These tests will have to be completed to provide a complete test set.

The hardware diagnostics can probably be converted from the existing KS10 hardware diagnostics since the basic structure of the KD10 and KS10 is similar. Additional tests will probably have to be written to test the new hardware functionality.

Since there is a reasonable chance that the microword format will change, at least slightly, it will probably be necessary to make changes to the microcode conversion and checking utilities that

currently exist for the KS10 microcode.

Finally, additional peripheral diagnostics will probably have to be developed. Since the KD10 uses Unibus I/O for everything but disks, the existing diagnostics may be adequate. They are certainly not adequate for diagnosing the disk subsystem. A diagnostic for the UDA50/RA60/RA81 will have to be developed. This includes a disk formatter.

CHAPTER 11

ADDITIONAL INVESTIGATIONS

This chapter discusses additional investigations that are required. It also makes recommendations for possible future performance enhancements.

11.1 Possible performance enhancements

11.1.1 Cache sweeps

The existing design causes the translation buffer and cache to be swept on WREBR and WRUBR that changes the UPT and it is quite costly in time to perform the sweep. Although the cache is physically addressed, it must be swept because the I/O adapters don't invalidate words in the cache on writes to memory. Since the monitor appears to do a WRUBR before it makes the I/O data available to the user, a cache sweep on WRUBR keeps the data consistent.

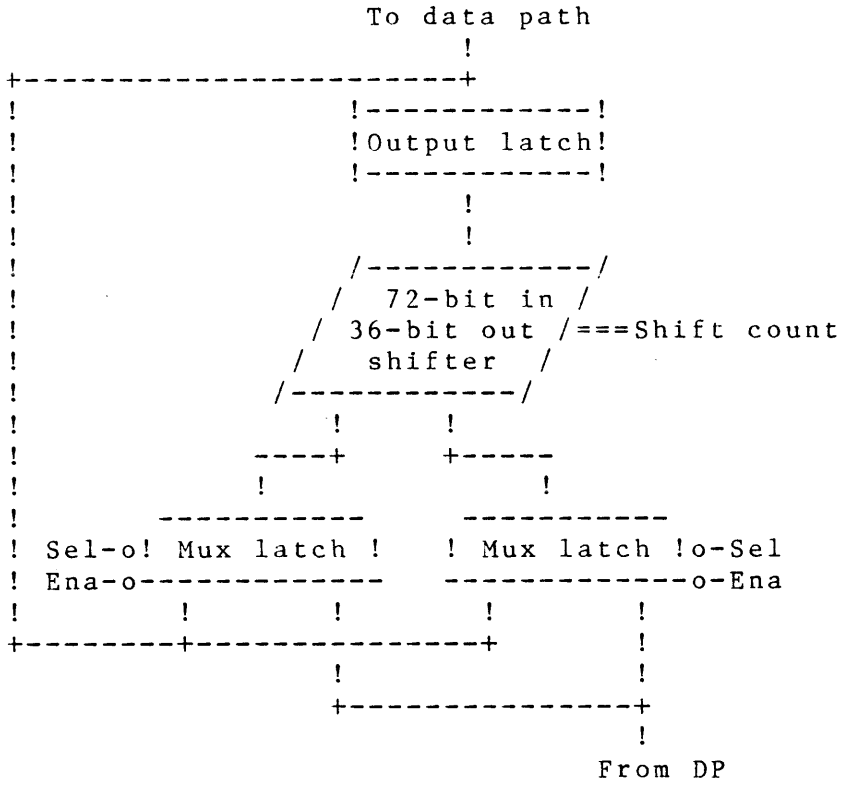
Because of the structure of the machine, it appears to be nearly impossible to cause the I/O adapters to invalidate the cache on writes to memory. There appear to be two possibilities for improving the performance of cache sweep. The cache directory RAMs could be implemented using a RAM part that has a single master reset line. In this way, the entire cache directory could be cleared in one cycle. The second possibility is to put explicit cache sweep instructions back into the instruction set and to remove the cache sweep from the WRUBR. It would then be up to the monitor to do a cache sweep when appropriate. There is only an advantage in doing this if multiple WRUBRs are being done between an I/O adapter write and the time the data is given to the user. If this is true, the monitor could, through software means, perform exactly one cache sweep. This obviously has the potential for causing software bugs, and this must be part of the evaluation.

11.1.2 Cache and TB organizations

The current design uses a one-way associative, one-word block size cache and TB. Some performance gain could result from increasing the associativity and/or the block size of each. This would probably require that the data cache be removed from the workspace. If the translation buffer is made larger or the associativity increased, the addition of the "Keep me" bit might also help. The K bit has been reserved as bit 7 of paging pointers for this purpose.

11.1.3 Barrel shifter

At present, the design uses a 1-bit shifter. This appears to cause performance problems for shift instructions, byte instruction, string instructions, and possibly floating point instructions. One possibility for improving this situation is to build a custom CMOS shifter that would provide a 72-bit in, 36 bit out shifter. The inputs to the chip would come from DP and the outputs would be connected to some (unspecified) input mixer. The chip might look like:



The input mux latch selects and enables can be individually selected by the microcode. The internal feedback paths provide the output-into-input function that is common in byte instructions.