For each illegal op code, shade offset in UPT
Move pushm to 770 to allow exec mode growth
Should this be called a PDP10 arch. spec
Can we list KL exceptions?

d i g i t a l — interoffice memorandum

Any support for 9-bit bytes? 9-to-8 copy

To: List

Date: 19 Apr 83
From: G. M. Uhler
Dept: Jupiter Engineering
Loc: MR1-2/E85
DTN: (8-)231-6448
File: KC10.RNO

Copy-on-write

Distributed: 19 Apr 83
Revision: 8

Trap on section ∅ (EC)

Address traces, page refill traces/counts

Subject: KC10 Functional Description

Changes made to revision 7 of KC10 to create revision 8 on 01-Apr-83

1. Add EXARCH,MEM as a chapter.

2. Add the opcode maps for the instruction set and clean up the document a bit.

3. Redefine some of the microcode option bits in APRID to reflect the new product goals.

4. Re-order and re-format some of the chapters to make the content more logical.

5. Remove the chapter on Address Break and include the material as a section in the chapter on paging.

Changes made to revision 6 of KC10 to create revision 7 on 09-Jan-83.

1. Expand on the legal PXCT cases.

2. Remove most of the legal PXCT [BLT] cases.

3. Add the LDPAC and STPAC instructions to save and restore previous context ACs.

4. Move the description of the physical EA-calc algorithm from the chapter on queue instructions to the chapter on Miscellany.

5. Make it clear that bit 29 in RDAPR and WRAPR actually exists in the hardware.

6. Fix some typos in WRCTX.

7. Remove support for TOPS-10 paging and turn the WREBR bit into something that indicates which features we support.

8. Describe the algorithm used to decide if there is a valid mapping for a page using the result of a MAP instruction.

9. Move the trap enable bits in WREBR and RDEBR from 8 and 9 to 7 and 8.

10. Fix some types in the list of additional data for page fail codes.

11. Declare bit 8 in each pointer to be available to software.

12. Define the JRSTCI instruction as JRST 1, to flush the instruction cache (read IBOX here) if the instruction stream is modified.

13. Use the AC field of LDPAC and STPAC to specify the highest previous AC to transfer to/from.

14. Add an IBOX flush for PMOVEM, the queue instructions, WRITTB, and DUMPTB.

15. Add the IOPMOV and IOPMVM instructions.

16. Add the description for the RDTRAX instruction.

17. Improve the documentation of RNGB and RNGBW and indicate that they now store an error code in AC if the AC field is non-zero and the instruction times out the I/O bus.

18. Add documentation for more microcode option bits in APRID.

19. Clarify the AC field assignment table for APR0, APR1, and APR2 a bit.

20. Add a microcode option bit to APRID to indicate that unbiased rounding is enabled.

21. Allocate words 260-267 in the IO page for use by the ports during the self-checking process.

22. Clean things up a bit and expand on things that are unclear.

23. Add PUSHM, POPM, and PUSHJ.

24. Add brief descriptions for each word in the EPT, UPT, and IO page to the pictures in the chapter on Special System Pages.

25. Add a description of the interrupt request protocol used by the ports.

Changes made to revision 5 of KC10 to create revision 6 on 08-Mar-82.

1. Declare XBLT to be legal in section 0.

2. Document the differences between the KL and KC implementation of JRST n,. Allow SFM in any section. Add the XJRST (JRST 15,) instruction.

3. Document the occurrence of an IBOX flush on WRCTX, WREBR, CLRPT, and SWPIA.

4. Redesign the WRTMB and RDTMB instructions. Add the WRACT, RDACT instructions.

5. Remove the 18 bit EA restriction from PMOVE and PMOVEM in section 0.

6. Document the changes to the legal PXCT bits.

7. Change SFTCU from 700100,,0 to 701000,,0.

8. Change RDUBR from 700040,,0 to 701400,,0.

9. Change the format of E+1 of the WRCTX, RDCTX, and RDUBP blocks to match that of the first word of the flags/PC double word.

10. Make UMOVE/UMOVEM generate an illegal instruction trap if executed in user mode.

11. Add functional descriptions for WRCTX, WREBR, WRTMB, and WRACT.

12. Change the polarity of bit 10 returned in the MAP instruction AC.

13. Remove function code 3 from the trap function word.

14. Remove the "do not load current AC block values" bit from the interrupt vector word.

15. Move the control bits around in word E of the WRCTX (and RDCTX and RDUBR) block to make it easier to implement.

16. Remove the "do not load AC block values" bit from the WRCTX block.

17. Move the mapping for exec pages 0-337 with TOPS-10 paging from words 600-757 in the EPT to 0-157 in the EPT.

18. Remove the "FPA present" from the hardware options field in APRID and put it in the microcode options field.

19. Remove the JEN instruction.

20. Redefine the algorithm used to compute a physical address for PMOVE, PMOVEM, and the queue instructions.

21. Redefine words 453-454 in the UPT (page fail block) to be "additional data" returned on a page fail. These words will be different for each type of page fail code and will be documented as such.

22. Modify MUUO, page fail, I/O page fail, and traps (MUUO function) to "load" PAB instead of "setting" it.

23. Make super section pointer types 1 (immediate) and 3 (indirect) legal.

24. Start the page fail block in the UPT at offset 451 instead of 450.

25. Change the page number field in an immediate pointer from bits 20-35 to 18-35.

26. Document the format of the "page address word" encountered in pointer traces.

27. Add a description of the debugging instruction implemented by the microcode.

28. Describe the paging information cache.

29. Change the format of the page fail word to include the "level" at which the page fail was detected.

30. Describe the CST update operation.

31. List the processor flags in the description of the Flags/PC double word.

32. Define the page fail codes for TOPS-20 paging.

33. Describe bits 0-10 of the page fail PMA as "reserved" instead of zeros. They come back undefined from the MBOX and it didn't seem particularly necessary to mask them since the monitor probably won't use the word anyway.

34. Note that the trap 1 and trap 2 flags are never stored in the LUUO/MUUO block when a trap that specifies function code 1 (MUUO) or 2 (LUUO) is processed.

35. Note that an instruction that references multiple words (e.g., BLT) will not cause an address break for every word in the address break range if the monitor restarts the instruction with the "inhibit address break" flag set.

36. Add the trap enable bit to WREBR and RDEBR and explain how it works.

37. Add descriptions for the debugging instructions RDTRAX and WRTRAX.

38. Add a new chapter for Miscellany and start it by describing the halt status codes.

39. Change the format of the I/O page fail block to make it more consistent with the normal page fail block.

40. Document the effect of a write reference for the various combination of written and modified bits in the translation buffer.

Changes made to revision 4 of KCIO to create revision 5 on 18-Aug-81.

1. Add the definition of the bits returned by the MAP instruction.

2. Define bit 0 of the microcode options field in APRID as "diagnostic microcode loaded".

3. Move the remaining APR flag bits in WRAPR and RDAPR to bits 30 and 31 to right justify them in the field and make them contiguous

4. Add the "VM mode" state bit as bit 10 in word E for WRCTX, RDCTX, and RDUBR. Also use UPT location 431 as the VM mode new PC MUUO dispatch.

5. Add a statement that pager clears caused by CLRPT and WREBR ignore the keep me bit.

6. Declare the results of executing a UMOVE or UMOVEM in user mode as undefined.

7. Define bit 19 in the RNGB? instructions to perform a port init for the port specified by bits 33-35.

8. Put all the accounting meter stuff under an "available with the accounting meter option only" disclaimer.

9. Remove the "Port interrupt logout word" from the I/O page and add 8 "Port interrupt PI status words".

10. Remove the SWPVA instruction from the instruction set.

11. Change the spec to indicate that no flags are loaded on a LUUO.

12. Move the address break condition bits from 0-7 to 10-17 in the WRCTX/RDCTX argument.

13. Require that paging be turned on to load the address break conditions with WRCTX or to read them with RDCTX.

14. Rearrange the WRCTX control bits.

15. Rearrange the bits in RNGB and RNGBW one more time.

16. Remove the cache look and load bits from WREBR and RDEBR and the TB cachable bit from the page table pointer and add cache on/off controls bits to WREBR and RDEBR.

17. Do not cause an IBOX flush as the result of a cache sweep instruction.

18. Remove time base 1 completely and remove the time base 2 words in the FPT (they're now kept strictly in EBOX scratchpad locations).

19. Redesign the time base and interval timer instructions.

20. Add the RDURTM instruction to read the user runtime meter.

21. Add the page fail formats.

22. Remove the discussion of the UBA from the interrupt vector definitions.

Changes made to revision 3 of KC10 to create revision 4 on 30-Apr-81.

1. Reserve bits 0-10 of the link words of physical queues.

2. Change the console reload bit in RNGB and RNGBW from bit 19 to bit 18.

3. Describe the relationship between the effective address calculation and the reference address for those instructions that use the EA as a physical address.

4. Reserve bits 0-10 of AC for the queue instructions and define the action on bit 0 in the AC on an empty/non-empty queue.

5. Document the MUUO block and the new PC dispatch vectors. Change the spec to reflect the new dispatch algorithm approved by the architecture committee. Move the page fail locations from location starting at 440 to locations starting at 450 to make room for the new MUUO dispatch.

6. Document the LUUO block format and the action of the processor to an LUUO.

7. Clean up the trap function word definitions and change functions 1 and 2 to agree with the decisions of the architecture committee.

8. Clean up the interrupt vector description and document the format of the I/O page fail block in the I/O page.

9. Clean up the queue instruction explanations. Thank you Judy Hall.

10. Redo the entire chapter on virtual addressing (and rename it "paging") to add much more information.

11. Describe the changes to the MAP instruction.

Changes made to revision 2 of KC1U to create revision 3 on 19-Mar-81:

1. Add the SETCU instruction to set the CST update needed bit in each page table entry.

2. Note that PMOVE, PMOVEM and the physical queue instructions do not cause the CST to be updated.

3. Remove the INSQUE and REMQUE instructions and the references to virtual queues.

4. Add pictures for the EPT, UPT and I/O page. Also provide separate picture for TOPS-10 and TOPS-20 for the first two.

5. Change the format of the trap function word that simulates a LUUO to specify the opcode of the LUUO to be used in the function word.

6. Change the definition of the queue manipulation instructions to skip return if the entry has been successfully added to the queue with bit 0 in the AC set if the entry was added to an empty queue. The instruction will not skip if the secondary interlock was timed out.

Changes made to revision 1 of KC10 to create revision 2 on 9-Mar-81:

1. Add definition of "reserved" fields of instruction operands and data.

2. Define all bits of instruction operands and data.

3. Add "KL/KS compatibility" section to each instruction description.

4. Add an enable bit to load the CPU PIA in WRAPR.

5. Change the page number field for WRCTX, WREBR and WRIOP from bits 18-35 to bits 20-35

6. Remove the commitment to make a PXCT of a CLRPT work.

7. Define what "TOPS-10" paging really means.

8. Change the "interrupt 2080 console" bit in RNGB and RNGBW from bit 18 to bit 32.

9. Change the description of the operation of PMOVE and PMOVEM to do a normal effective address calculation and use the result as a physical address.

10. Change the definition of WRIOP from an immediate mode instruction to one which takes its data from the word addressed by E.

CHAPTER 1 INTRODUCTION

CHAPTER 2 APR0, APR1, AND APR2 INSTRUCTIONS

2,1 APRID 2-2
2,2 WRAPR 2-3
2,3 RDAPR 2-5
2,4 SZAPR 2-6
2,5 SNAPR 2-7
2,6 WRPI 2-8
2,7 RDPI 2-10
2,8 SZPI 2-11
2,9 SNPI 2-12
2,10 SETCU 2-13
2,11 RDCTX 2-14
2,12 CLRPT 2-17
2,13 WRCTX 2-18
2,14 WREBR 2-23
2,15 RDEBR 2-26
2,16 WRIOP 2-27
2,17 RDIOP 2-28
2,18 RDUBR 2-29
2,19 SWPIA 2-31
2,20 SWPUA 2-32
2,21 RDSPB 2-33
2,22 RDCSB 2-34
2,23 RDPUR 2-35
2,24 RDCSTM 2-36
2,25 RDTMB 2-37
2,26 RDINT 2-38
2,27 RDTIME 2-39
2,28 RDUBTM 2-40
2,29 WRSPB 2-41
2,30 WRCSB 2-42
2,31 WRPUR 2-43
2,32 WRCSTM 2-44
2,33 WRTMB 2-45
2,34 WRINT 2-47
2,35 WRACT 2-48
2,36 RDACT 2-50

CHAPTER 3 EXTERNAL I/O INSTRUCTIONS

3,1 RNGB 3-2
3,2 RNGBW 3-3
3,3 SNBSY 3-4

CHAPTER 4 I/O INSTRUCTIONS THAT MOVE DATA

4,1 UMOVE 4-2
4,2 UMOVEP 4-3

4.3 PMOVE 4-4
4.4 PMOVEM 4-5
4.5 IOPMOV 4-6
4.6 IOPMVM 4-7
4.7 LDPAC 4-8
4.8 STPAC 4-9

CHAPTER 5 SPECIAL DEBUGGING INSTRUCTIONS

5.1 RDTRAX 5-2
5.2 WRTRAX 5-3
5.3 READTB 5-4
5.4 WRITTB 5-6
5.5 DUMPTB 5-8

CHAPTER 6 QUEUES AND QUEUE MANIPULATION INSTRUCTIONS

6.1 Introduction 6-1
6.2 Data structures 6-1
6.2.1 The queues 6-1
6.2.2 Formats 6-2
6.3 Operations 6-3
6.3.1 Insertion 6-3
6.3.2 Removal 6-5
6.4 Interlocks 6-5
6.5 The instructions 6-6
6.6 Errors 6-7
6.7 INSQHI 6-8
6.8 INSQTI 6-9
6.9 REMQHI 6-10
6.10 REMQTI 6-11

CHAPTER 7 FUNCTIONAL CHANGES FROM PREVIOUS MACHINES

7.1 MAP 7-2
7.2 JRST 7-6
7.3 PUSHM 7-8
7.4 POPM 7-10
7.5 PUSHJ 7-12
7.6 Other functional changes 7-13
7.6.1 Changes to privileged instructions 7-13
7.6.2 Elimination of public and supervisor modes . . 7-13
7.6.3 Overflow in exec mode 7-13
7.6.4 Cachable page status 7-13
7.6.5 XBLT in section zero 7-13
7.6.6 JRST changes 7-14
7.6.7 PXCT 7-14
7.6.8 Paging 7-14
7.6.9 Flags-PC double word 7-15
7.6.10 Process context variables 7-15
7.6.11 System timers 7-15

CHAPTER 8 PAGING

8.1 Introduction 8-1
8.2 Paging hardware and microcode 8-1
8.3 Caching of paging information 8-2
8.4 TOPS-20 paging 8-4
8.4.1 Pager Data Structure 8-4
8.4.2 Pointers 8-4
8.4.2.1 Super Section Pointers 8-5
8.4.2.2 Section Pointers 8-6
8.4.2.3 Map pointers 8-7
8.4.3 Page address words 8-8
8.4.4 Conversion of Virtual to Physical Addresses . . 8-9
8.4.5 Page refill 8-9
8.4.5.1 CST updates 8-9
8.4.5.2 CST entry format 8-10
8.4.5.3 CST mask register format 8-11
8.4.5.4 Process Use Register format 8-11
8.4.5.5 Translation buffer state bits 8-11
8.4.5.6 Write references 8-12
8.4.6 Page fail conditions and formats 8-12
8.4.6.1 Tops-20 page fail codes and additional data . 8-16
8.4.6.1.1 Additional data words for a pointer trace . . 8-20
8.5 Address Break 8-22

CHAPTER 9 PROCESS CONTEXT VARIABLES

9.1 Introduction 9-1
9.1.1 New flag-PC double word 9-1
9.1.2 Context changing 9-3

CHAPTER 10 EXTENDED ADDRESSING

10.1 Reference materials 10-2
10.2 Historical summary of extended addressing . . . 10-3
10.3 Definition of terms 10-4
10.4 Effective Address Calculations 10-8
10.4.1 Description of the EA-calc algorithm 10-8
10.4.1.1 No indexing 10-8
10.4.1.2 IFIW with local index 10-9
10.4.1.3 IFIW with global index 10-9
10.4.1.4 EFIW with global index 10-10
10.4.1.5 References to section zero 10-10
10.4.1.6 Summary of EA-calc rules 10-10
10.4.2 Results of an EA-calc 10-11
10.4.3 Simple EA-calc examples 10-11
10.5 Use of the local/global flag 10-13
10.5.1 AC references 10-13
10.5.2 Incrementing EA 10-14
10.6 Multi-section EA-calc's 10-15
10.7 Special case instructions 10-17
10.7.1 Byte instructions 10-17

10.7.1.1 Byte pointer interpretation 10-17
10.7.1.2 Byte pointer EA-calc 10-18
10.7.2 EXTEND instructions 10-19
10.7.2.1 Byte pointer interpretation 10-19
10.7.2.2 Byte pointer EA-calc 10-19
10.7.2.3 Extended opcode EA-calc 10-21
10.7.2.4 EDIT pattern and mark addresses 10-21
10.7.3 JSP and JSR 10-21
10.7.4 Stack instructions 10-23
10.7.5 JSA and JRA 10-25
10.7.6 LUUOs 10-26
10.7.7 BLT 10-26
10.7.8 XBLT 10-28
10.7.9 JPSTF 10-29
10.7.10 XMOVEI and XHLLI 10-29
10.7.11 XCT 10-30
10.7.11.1 Default section for EA-calc 10-30
10.7.11.2 Relationship with skip and jump instructions . 10-30
10.7.11.3 PC storing instructions 10-31
10.7.11.4 Local stack references 10-31
10.7.11.5 Generalizations for XCT 10-32
10.8 Summary of default sections for EA-calc 10-33
10.9 Section zero vs, non-zero section rules 10-34
10.10 Special consideration for ACs 10-36
10.10.1 AC references 10-36
10.10.2 Instruction fetches 10-37
10.10.3 Storing PC 10-38
10.10.4 Storing EA for LUUO, MUUO and page fails . . . 10-38
10.10.5 An example 10-39
10.11 PXCT 10-40
10.11.1 Previous context 10-40
10.11.2 Use of the previous context state variables . 10-41
10.11.3 References to previous context 10-41
10.11.4 Applicable instructions 10-42
10.11.5 Interpretation of the AC field bits 10-42
10.11.6 Modifications to the EA-calc algorithm 10-44
10.11.7 Section zero vs, non-zero section rules . . . 10-48
10.11.7.1 Stack instructions 10-48
10.11.7.2 Byte instructions 10-49
10.11.7.3 EXTENDed MOVSLJ instruction 10-50

CHAPTER 11 SYSTEM TIMERS

11.1 Summary 11-1
11.2 Time clocks 11-1
11.2.1 Time Base 11-2
11.2.2 User Runtime Meter 11-2
11.3 Interval Timer 11-2

CHAPTER 12 TRAP, UUO AND INTERRUPT HANDLING

12.1 Introduction 12-1

12.2 Trap Function Word 12-2
12.3 Virtual Machine Simulation Mode 12-4
12.4 MUUO handling 12-5
12.5 LUUO handling 12-8
12.6 Trap enable 12-9
12.7 Interrupt vectors 12-10
12.8 I/O page failure 12-10
12.9 Interrupt request protocol 12-11

CHAPTER 13 MISCELLANY

13.1 Halt status codes 13-1
13.2 Physical EA-calc 13-2

CHAPTER 14 SPECIAL SYSTEM PAGES (EPT / UPT / IOP)

Index

CHAPTER 1

INTRODUCTION

is it the PDP-10 architecture?

This document describes the functional operation of the KC10 CPU. This information includes descriptions of the privileged I/O instructions, functional changes between previous implementations of the PDP-10 architecture and this one, discussions of paging, UUO, trap, and interrupt processing, and a lengthy chapter on extended addressing.

Opcodes in the range 700 through 737, inclusive, are privileged opcodes which are generally used to control internal or external devices related to the hardware itself.

Opcodes in the range 740 through 777, inclusive, are legal in both exec and user mode, but only a few instructions are currently defined in this range.

Does it replace, or enhance, the PDP10 ref. manual? That gives more functional info.

Please read this

In some instances, fields of the operands of an instruction or fields of the values returned by an instruction are described as "reserved". This means simply that no guarantee is made of the correct operation of an instruction whose "reserved" fields are set non-zero by the program or of the state of the bits in the "reserved" fields returned by an instruction. If you wish to experiment and find a result to your liking, you are hereby warned that your program may well not be compatible with any other processor, with any other model of your processor, with the same model of your processor at some other installation, or even with your own processor running at some other time with a different version of the microcode or Monitor.

When the definition of a bit is given in this document, that definition applies when the bit is set to a 1 (unless explicitly stated otherwise). If the bit is set to a zero, the logical complement of the definition applies. For example, bit 8 in word E in APRID below is described as "TRACKS (PC trace) microcode present". This means that the TRACKS feature is present in the microcode if the bit is a 1 and not present if the bit is a 0.

Opcode assignment map

	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	APRO	APR1	APR2	UUO	UMOVE	UMOVEM	PMOVE	PMOVEM
710	KNGB	KNGBW	SNBSY	UUO	IOpMOV	IOpMVM	LDPAC	STPAC
720	INSOHI	INSOTI	REMOHI	REMOTI	UUO	UUO	UUO	UUO
730	UUO	UUO	RDTRAX*	WRTPAX*	READTB*	WRITTB*	DUMPTB*	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

* = Debug instructions, UUO in production machine

EXTENDed opcode map

	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

For opcodes 700-702 (APR0, APR1, and APR2), the AC field is decoded to produce 16 possible instructions. The following table gives the instruction mnemonic for each AC decode.

AC Field Assignments

AC	APR0	APR1	APR2
00	APRID	SETCU	RDSPB
01	UUO	RDCTX	RDCSB
02	UUO	CLRPT	RDPUR
03	UUO	WRCTX	RDCSTM
04	WRAPR	WREBR	RDTMB
05	RDAPR	RDEBR	RDINT
06	SZAPR	WRIOP	RDTIME
07	SNAPR	RDIOP	RDURTM
10	UUO	RDUBR	WRSPB
11	UUO	SWPIA	WRCSB
12	UUO	UUO	WRPUR
13	UUO	SWPUA	WRCSTM
14	WRPI	UUO	WRTMB
15	RDPI	UUO	WRINT
16	SZPI	UUO	WRACT
17	SNPI	UUO	RDACT

CHAPTER 2

APR0, APR1, AND APR2 INSTRUCTIONS

This chapter describes the APR0, APR1, and APR2 instructions (opcodes 700-702) which control the internal processor devices. The AC field of these instructions is decoded to produce 1-of-16 instructions.

All instructions in this set are privileged instructions which are typically executed in exec mode. Some instructions are legal in user mode if the USER I/O PC flag is also set.

2.1 APRID

```
+---------+----+-+----+-----------------+
|   700   | 00 |@| XR |        Y        |
+---------+----+-+----+-----------------+
```

This instruction stores the microcode version number, CPU serial number, and processor options in the words addressed by E and E+1, The format of the first word (E) is:

0-8 Reserved for microcode options,

- 0 Diagnostic microcode loaded,
- 1 Unbiased rounding microcode enabled,
- 6 XJRSTF debugging microcode present, This microcode is a debugging aid that generates a page fail to the monitor if bits 13-17 of the flags word are non-zero or if the CAB and PAB field are both zero,
- 7 Debugging instructions present (READTB, WRITTB, etc,), See the chapter on Special Debugging Instructions for a description of each instruction,
- 8 TRACKS (PC trace) microcode present, See the description of the WRTRAX and RDTRAX instructions in the chapter on Special Debugging Instructions for more information concerning the control of this feature,

9-17 Hardware options bits?

18-35 Processor serial number field?

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

0-35 Microcode version number guaranteed leading zeroes?

KL/KS compatibility

The KL and KS returned microcode options, microcode version number, hardware options and processor serial number in one word, The KC returns two words and defines different hardware and microcode option bits,

Question on format: how can I tell whether 0, 1, 6, 7, 8 are possible values in a field, or possible bits?

2.2 WRAPR

700	04	@	XR	Y

This immediate mode instruction decodes its effective address to control the processor. The effective address bits are used as follows:

18	Load the PI assignment for the CPU from bits 33-35.
19	I/O reset. When this bit is set, "reset" is asserted on the KC10 I/O bus. This will reset all the port micromachines (but affects no internal devices, such as the pager and processor flags).
20	Enable the APR conditions selected by bits 24 thru 31 to cause interrupts.
21	Disable the APR interrupts for conditions selected by bits 24 thru 31.
22	Clear the APR flags indicated by bits 24 thru 31.
23	Set the APR flags indicated by bits 24 thru 31.
24-31	Selected flags. These bits represent individual APR flags that can be set, cleared, enabled, or disabled with the appropriate combination of bits 20-23.
24-28	Reserved
29	Unassigned APR flag
30	Console attention
31	Power failure
32	Reserved
33-35	PI assignment for CPU

CAUTION

The results of executing a WRAPR instruction are undefined if the WRAPR argument has both bits 20 and 21 or bits 22 and 23 set. There is no logical meaning in attempting to both enable and

disable the same conditions or in attempting to both set and clear the same flags in one instruction.

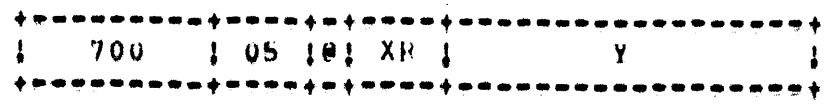
0 21 & 23 OK?

KL/KS compatibility

The KL and KS unconditionally set the CPU PIA. The KC ~~only~~ sets the PIA if bit 18 is on. The KC also defines different flags in bits 24-31. only

What was that debate w/ Pomfret about I/O reset?

2,3 RDAPR



This instruction stores the APR status in the word addressed by E. The status is as follows:

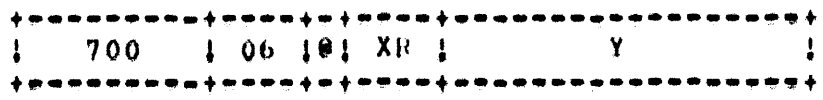
0-5	Reserved
6-13	Interrupts enabled. These bits indicate which of the APR flags are enabled to cause interrupts.
6-10	Reserved
11	Unassigned APR flag
12	Console attention enabled
13	Power failure enabled
14-23	Reserved
24-31	Interrupts pending. These bits indicate which of the APR flags are are requesting an interrupt.
24-28	Reserved
29	Unassigned APR flag — what do I do w/ this?
30	Console attention
31	Power failure
32	Interrupt requested (IOR of bits 24-31).
33-35	PI assignment for the CPU.

KL/KS compatibility

The KC defines different flags in bits 24-31.

KL?

2.4 SZAPR



This instruction tests bits 18-35 of the APR status (as indicated under RDAPR) against bits 18-35 of E. If all status bits selected by 1s in E are 0s, the next instruction in sequence is skipped.

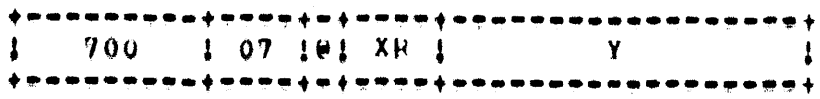
KL/KS compatibility

Functionally identical to CONSZ APR,E.

How are bits 24-31 cleared? Must software do it, or will XJEN?

Is it really 18-35? Why not make it immediate, then?

2,5 SNAPR

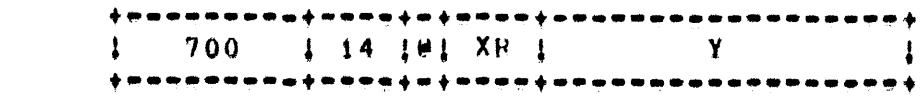


This instruction tests bits 18-35 of the APR status (as indicated under RDAPR) against bits 18-35 of E. If any status bits selected by is in E is 1, the next instruction in sequence is skipped.

KL/KS compatibility

Functionally identical to CONSO APR,E.

2.6 WRPI



This immediate mode instruction decodes its effective address to control the priority interrupt system. The effective address bits are used as follows:

18-21	Reserved
22	Turn off program (software) requests on the PI levels selected by 1s in bits 29-35.
23	Clear PI system.
24	Initiate program (software) interrupts on the PI levels selected by 1s in bits 29-35. Such interrupts vector through the software interrupt vector words in the I/O page. An interrupt is not initiated on a level unless the PI system and the requested level are on.
25	Turn on the PI levels selected by 1s in bits 29-35.
26	Turn off the PI levels selected by 1s in bits 29-35.
27	Turn off PI system
28	Turn on PI system
29-35	Selected PI levels to be affected by the control of bits 22, 24, 25, and 26. where 29 - ? 30 - ? etc ?

CAUTION

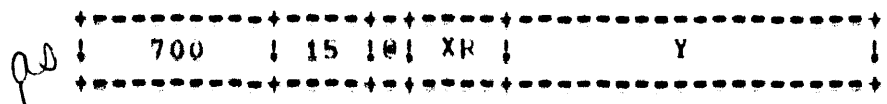
The results of executing a WRPI instruction are undefined if the WRPI argument has both bits 22 and 24 or bits 25 and 26 set. There is no logical meaning in attempting to both clear and initiate software requests or in attempting to both turn on and turn off PI levels.

Where do you define legal levels & their relationship to each other?

KL/KS compatibility

The KL used bits 18-20 to force parity errors, The KL initiated an interrupt on a level as the result of a 1 in bit 24 even if the specified level was off, Although the KS is documented to act in the same manner, it did not initiate the interrupt unless the level was on,

2.7 RDPI



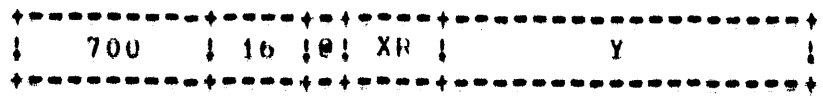
This instruction stores the PI status in the word addressed by E. The status is as follows:

0-10	Reserved
11-17	PI levels on which program (software) requests have been made.
18-20	Returned as zeroes
21-27	PI levels on which interrupts are in progress.
28	PI system on.
29-35	PI levels which have been turned on.

KL/KS compatibility

The KL returned the state of the forced parity error bits in bits 18-20. Otherwise, it is functionally equivalent to CONI PI,E on the KL and RDPI E on the KS.

2,8 SZPI

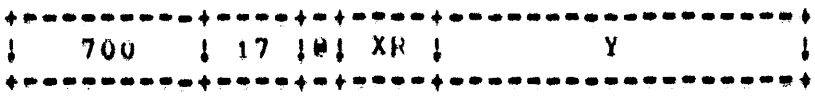


This instruction tests bits 18-35 of the PI status (as indicated under RDPI) against bits 18-35 of E. If all status bits selected by 1s in E are 0s, the next instruction in sequence is skipped.

KL/KS compatibility

Functionally equivalent to CONSZ PI,E.

2.9 SNPI

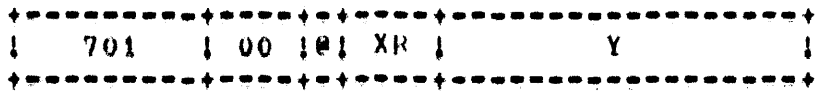


This instruction tests bits 18-35 of the PI status (as indicated under RDPI) against bits 18-35 of E. If any status bits selected by is in E is 1, the next instruction in sequence is skipped.

KI/KS compatibility

Functionally equivalent to CONSO PI,E.

2.10 SETCU



This instruction causes the "CST update needed" bit to be set in each entry in the hardware translation buffer so that the first virtual reference to each page will cause the EBOX to update the CST entry for the page.

This instruction blocks further CPU activity until all bits are set in the translation buffer.

The actions performed by this instruction are as follows:

1. Set the "CST update needed" bit in each entry in the hardware translation buffer.

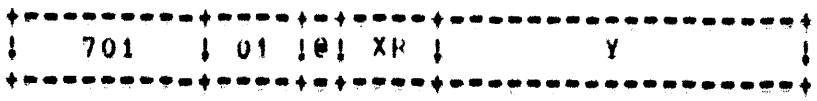
2. Flush and restart the IBOX.

The indirect, index register, and Y fields of this instruction are not used and are reserved.

KL/KS compatibility

No functional equivalent on the KL or KS.

2.11 RDCTX



This instruction stores the user process context in the five words addressed by E through E+4 in exactly the same format as used by WRCTX. In order to allow these words to be used directly in a WRCTX instruction, bits 0-2, 5, 8, and 9 are set to 1 and bits 3, 4, and 7 are set to 0 in (E), the 1st returned word

CAUTION

Paging must be enabled (with WREBR bit 4) to read the address break conditions. If paging is not enabled, the values returned in words E+2 through E+4 (address break conditions, lower and upper address bounds) are undefined.

The format of the first word (E) is:

0 Returned as a 1 (load CAB/PAB in WRCTX).

1 Returned as a 1 (load PCS in WRCTX).

2 Returned as a 1 (load UBR in WRCTX).

3 Returned as a 0 (unconditional pager clear in WRCTX).

4 Returned as a 0 (inhibit meter update in WRCTX).

5 should be 0 → Returned as a 1 (load VM-mode in WRCTX).

6 Virtual machine mode (VM-mode) enabled for this user context.

7 Returned as a 0 (Inhibit address break in WRCTX).

8 → Returned as a 1 (Load address break conditions in WRCTX).

9 → Returned as a 1 (Load address break enable in WRCTX).

10 Address break enabled.

11-17 Reserved

18-35 Physical page number of UPT.

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

0-17 Reserved

18-20 Current AC block

21-23 Previous AC block

24-35 Previous Context Section

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

0-9 Reserved

10 Address break enabled for a normal fetch of an instruction in the program under control of PC.

11 Address break enabled for any reference that reads except the normal fetch of an instruction.

12 Address break enabled for any reference that writes.

13 Address break enabled for a reference made in user virtual address space. (0 implies executive space).

14 Address break enabled for a reference made by the CPU to memory.

15 Address break enabled for a reference made by a port to memory.

16 Address break enabled for a physical memory reference. (0 implies virtual memory references).

17 Compare only bits 18 through 35 of the reference address with the address range when doing address compares.

18-35 Reserved

The format of the fourth word (E+3) is:

0-5 Reserved

6-35 The lower bound break address,

The format of the fifth word (E+4) is:

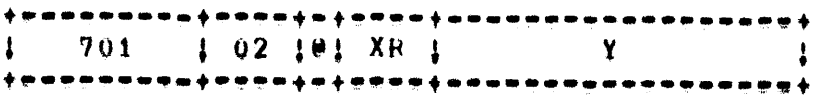
0-5 Reserved

6-35 The upper bound break address,

KL/KS compatibility

The KL returned current and previous AC blocks, previous context section, and the physical page number of the UPT with a DATAI PAG,E, Address break conditions were returned with a DATAI APR,E, The KS returned current and previous AC blocks, and the physical page number of the UPT with a RDUBR E, The KC RDCTX instruction combines the functions of these instructions,

2.12 CLRPT



701	02	@	XR	Y

This immediate mode instruction clears the hardware translation buffer entry for the virtual page addressed by E so that the next virtual reference to a word in that page will cause an EBOX page fail trap to occur.

The translation buffer "keep me" bit is ignored by this instruction and an unconditional clear is done.

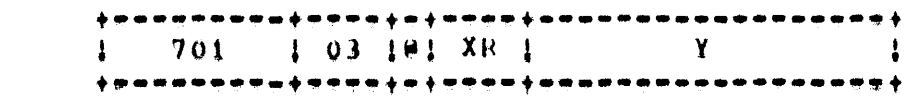
The actions performed by this instruction are as follows:

1. Clear the translation buffer entry for the virtual page addressed by E.
2. Clear the internal cache of paging information kept by the EBOX (but do not clear the first exec and user super section pointer). ——— why
3. Flush and restart the IBOX.

KL/KS compatibility

Functionally equivalent to CLRPT E on the KL.

2.13 WRCTX



This instruction loads user process context from either 2 or 5 words addressed by E through E+4 depending on the state of bit 8 in the first word. The process context includes previous and current AC blocks, previous context section, user base register, virtual machine simulation mode enable, and address break enable and conditions.

The format of the first word (E) is:

0 Load current and previous AC block numbers (CAB, PAB) from bits 18-23 of word E+1.

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

2 Load the user base register (UBR) from bits 20-35 of this word. The function of this bit is more complex that indicated here. See the complete description below.

3 Clear all pages from the hardware translation buffer (include "kept" pages) if a pager clear is done as the result of bit 2 being set.

4 Do not change the state of the user runtime meter (neither update it into the old UPT nor load it from the new UPT).

5 Load VM-mode enable from bit 6 of this word.

6 Enable virtual machine (VM-mode) simulation mode for this user context.

7 Inhibit all address break conditions for the next instruction executed. The effect of setting this bit is to set the "inhibit address break" PC flag for the next instruction.

8 Load address break conditions from the words at E+2 through E+4. If this bit is not on, the words at E+2, E+3, and E+4 are not referenced by the microcode.

9 Load address break enable from bit 10 of this word.

10	Enable address break using the existing conditions (the conditions may be also be changed with the same instruction).
11-17	Reserved
18-35	Physical page number of UPT.

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

0-17	Reserved
18-20	Current AC block
21-23	Previous AC block
24-35	Previous Context Section .

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

0-9	Reserved
10	Enable address break for a normal fetch of an instruction in the program under control of PC.
11	Enable address break for any reference that reads except the normal fetch of an instruction.
12	Enable address break for any reference that writes.
13	Enable address break for a reference made in user virtual address space. (0 selects executive space).
14	Enable address break for a reference made by the CPU to memory.
15	Enable address break for a reference made by a port to memory.
16	Enable address break for a physical memory reference. (0 selects virtual memory references).
17	Compare only bits 18 through 35 of the reference address with the address range when doing address compares.
18-35	Reserved

The format of the fourth word (E+3) is:

0-5 Reserved

6-35 The lower bound break address.

The format of the fifth word (E+4) is:

0-5 Reserved

6-35 The upper bound break address.

In word E, bits 0-2, 5, and 7-9 control the action of this instruction; when a bit is 0, the corresponding action is ignored. The actions are as follows:

1. If bit 0 is on, load CAB and PAB from bits 18-20 and 21-23, respectively, from word E+1. If bit 0 is off, do not change CAB and PAB.

2. If bit 1 is on, load PCS from bits 24-35 of word E+1. If bit 1 is off, do not change PCS.

3. If bit 2 is on, perform the following functions:

    1. If bit 3 in E is 0, clear all pages except those marked "kept" in the page table. If bit 3 is 1, clear all entries.

    2. Load bits 18-35 of E into the User Base Register

    3. If bit 4 in (E) offset Φ is 0, perform the following functions:

        1. Update the user runtime meter into the previous UPT by simulating a RDURTM instruction and storing the resulting doubleword in the previous UPT in locations 504-505.

        2. Load the user runtime meter kept in the EBOX internal registers from locations 504-505 of the new UPT.

        If bit 4 is a 1, do not update the user runtime meter into the previous UPT and do not reload it from the new UPT.

    4. Clear the internal cache of paging information kept by the EBOX. If paging is on, re-initialize it with the first super section pointer from the EPT and the UPT

(offset 520).

4. If bit 5 is on, perform the following functions:

   1. If bit 6 is on, enable virtual machine simulation mode (VM-mode) for this user context. If bit 6 is off, disable VM-mode for this user context.

   If bit 5 is off, do not change the state of VM-mode.

5. If bit 7 is on, inhibit all address break traps for the next instruction executed after the WRCTX. If bit 7 is off, do not inhibit address break traps.

6. If bit 8 is on, load the address break conditions from the words at E+2 through E+4. If bit 8 is off, the address break conditions remain unchanged.

CAUTION

Paging must be enabled (with WREBR bit 4) to load the address break conditions. If paging is not enabled, the result of loading the address break conditions is undefined.

7. If bit 9 is on, perform the following functions:

   1. If bit 10 is on, turn on address break using the existing break conditions (which may be set in the same instruction by setting bit 8 to a one). If bit 10 is off, turn off address break but leave the break conditions unchanged.

   If bit 9 is off, do not change the state of address break enable.

8. Flush and restart the IBOX.

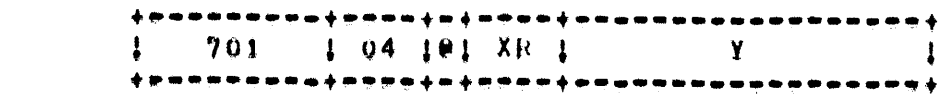
See the chapter on paging for a functional description of address break.

KL/KS compatibility

The KL set current and previous AC blocks, previous context section, and the physical page number of the UPT with a DATAO PAG,E. Address break conditions and address were set with a DATAO APR,E. The KS set current and previous AC blocks, and the physical page number of the

UPT with a WRUBR E. The KC WPCTX instruction combines the functions of these instructions.

2.14 WREBR



This instruction loads the exec mode context from the word addressed by E. The exec mode context includes cache enable, monitor type, pager enable, trap enable, and the exec base register. The format of the word is:

0	Load the cache enable bit from bit 1. This bit should never be set by the monitor. The cache will only be turned off as the result of a serious error and should remain off until the problem is fixed. The ability to enable and disable the use of cache is provided strictly for diagnostics.
1	Enable use of the cache for all references. Enabling the use of the cache with this bit does not enable the use of all four cache quadrants if one has been turned off because of an error. It simply causes the cache to be used for any quadrants that are on.
2	Reserved
3	TOPS-20 mode (see below)
4	Pager enable. ?
5-6	Reserved
7	Load trap enable from bit 8
8	Enable full processing of traps, LUUOs, MUUOs, and page fails
9-17	Reserved
18-35	Physical page number of EPT.

The actions performed by this instruction are as follows:

1. If bit 7 is a 1, perform the following functions:
    1. If bit 8 is a 1, enable full processing of traps, LUUOs, MUUOs, and page fails by the monitor as described in the appropriate sections below.

2. If bit 8 is a 0, change the processing of certain processor conditions as follows:

   1. Traps, Treat trap 1, 2, and 3 conditions as if the trap function word had specified "ignore trap",

   2. LUUOs, Process section 0 LUUOs in the normal manner, Halt the machine on LUUOs executed in non-zero sections,

   3. MUUOs, Halt the machine,

   4. Page fails, Process page fails that can be resolved by the EBOX microcode alone in the normal manner, Halt the machine on page fails that must be processed by the monitor,

If bit 7 is a 0, do not change the state of trap enable

2. If bit 0 is a 1, perform the following functions:

   1. If bit 1 is a 1, enable the use of the cache as described above, If bit 1 is a 0, disable the use of the cache,

If bit 0 is a 0, do not change the state of the cache enable,

3. If bit 4 is a 1, perform the following functions:

   1. Enable the use of the paging hardware for virtual-to-physical translations for memory references and select the type of paging to be used,

   2. If bit 3 is a 1, select TOPS-20 mode, If bit 3 is a 0, select TOPS-10 mode, This bit allows the monitor to select the features appropriate for the operating system being run, At present there are no differences between TOPS-10 and TOPS-20 mode, The bit is here for future expansion,

If bit 4 is a 0, disable paging so that all memory references are to physical locations unpaged, Note that disabling the pager does not mean there can be no page failures, as these can be caused by conditions that have nothing to do with paging,

CAUTION

Paging can be disabled only for kernel mode, A user mode process will not run correctly unless the pager is turned on,

4. Load bits 18-35 into the Exec Base Register.

5. Clear the internal cache of paging information kept by the EBOX. If paging is on, re-initialize it with the first super section pointer from the EPT and UPT (offset 520).

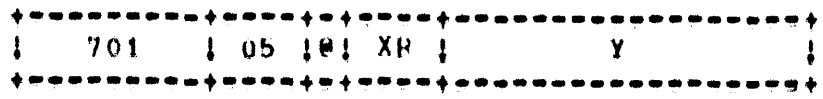
6. Invalidate all entries in the MBOX translation buffer, ignoring the state of the "keep-me" bits.

7. Flush the IBOX and restart it.

KL/KS compatibility

The KL set exec mode context with the IMMEDIATE MODE instruction CONO PAG,E. The KS set exec mode context with the IMMEDIATE MODE instruction WREBR E.

2,15 RDEBR



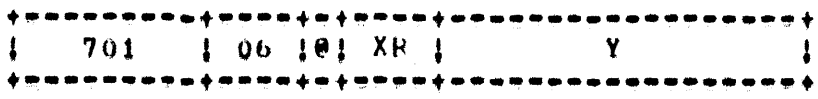
This instruction stores the exec mode context in the word addressed by E. The format of the word is:

0	Returned as zero
1	The use of cache is enabled for all references. This bit being set does not imply that all four cache quadrants are in use. It simply means that the cache will be used for any quadrants that are on.
2	Reserved
3	TOPS-20 mode (see WREBR above)
4	Paging is enabled
5-6	Reserved
7	Returned as 0
8	Full processing of traps, LUUOs, MUUOs, and page fails is enabled
9-19	Reserved
18-35	Physical page number of EPT.

KL/KS compatibility

The KL and KS returned the exec mode context with CONI PAG,E and RDEBR E, respectively. The KC returns the same fields as the KL CONI PAG,E but the bit positions have changed because of the increased size of the EPT page number.

2.16 WRIOP



This instruction loads the I/O page base register from the word addressed by E. The format of the word is:

0-17 Reserved

18-35 Physical page number of the I/O page

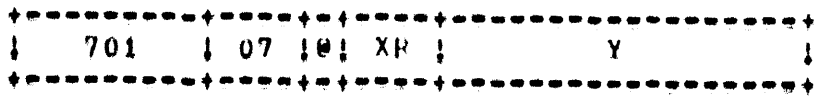
The I/O page base register is set to physical page 1 by the console during a cold-start or as the result of the INITIALIZE command.

See the chapter on Special System Pages for a description of the layout of this page.

KL/KS compatibility

No functional equivalent on the KL or KS.

2,17 RDIOP



This instruction returns the value of the I/O page base register and stores it in the word addressed by E. The format of the word is:

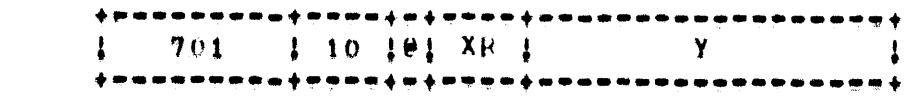
0-17 Reserved

18-35 Physical page number of the I/O page

KL/KS compatibility

No functional equivalent on the KL or KS.

2.18 RDUBR



This instruction stores the user process context in the two words addressed by E and E+1 in exactly the same format as the first two words used by WRCTX. In order to allow these words to be used directly in a WRCTX instruction, bits 0-2, 5, and 9 are set to 1 and bits 3,4, 7, and 8 are set to 0 in E. The format of the first word (E) is:

0	Returned as a 1 (load CAB/PAB in WRCTX).
1	Returned as a 1 (load PCS in WRCTX).
2	Returned as a 1 (load UBR in WRCTX).
3	Returned as a 0 (unconditional pager clear in WRCTX).
4	Returned as a 0 (inhibit meter update in WRCTX).
5	Returned as a 1 (load VM-mode in WRCTX).
6	Virtual machine mode (VM-mode) enabled for this user context.
7	Returned as a 0 (inhibit address break in WRCTX).
8	Returned as a 0 (Load address break conditions in WRCTX).
9	Returned as a 1 (Load address break enable in WRCTX).
10	Address break enabled.
11-17	Reserved
18-35	Physical page number of UPT.

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

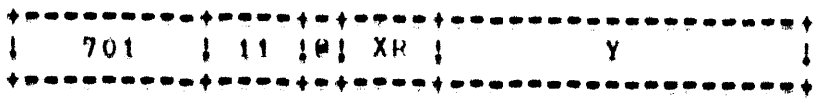
0-17	Reserved
18-20	Current AC block
21-23	Previous AC block

24-35 Previous Context Section

KL/KS compatibility

The KL returned current and previous AC blocks, previous context section, and the physical page number of the UPT with a DATAI PAG,E, The KS returned current and previous AC blocks, and the physical page number of the UPT with a RDUBR E,

2.19 SWPIA



Sweep Cache, Invalidate All Pages

Clear the valid and written state in all cache entries and do not write any words that are written in the cache back into memory.

This instruction blocks further CPU activity until the cache sweep is complete.

The actions performed by this instruction are as follows:

1. For each directory entry in the cache, clear the valid and written bits. Do not write any written words back to main memory.

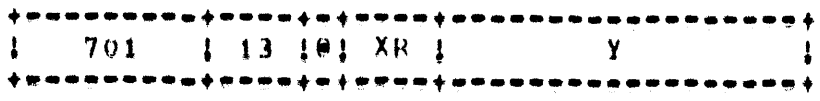
2. Flush and restart the IBOX.

The indirect, index register, and Y fields of this instruction are not used and are reserved.

KL/KS compatibility

The KS had no cache sweep instructions. The KL allowed other requests to happen in parallel with the sweep, setting sweep busy and sweep done in the APR status to indicate the sweep-in-progress interval.

2.20 SWPUA



Sweep Cache, Unload All Pages

Write all written words in the cache back into memory, Invalidate all entries (i.e, clear valid and written state),

This instruction blocks further CPU activity until the cache sweep is complete,

The actions performed by this instruction are as follows:

1. For each directory entry in the cache, clear the valid and written bits, If the written bit was on, write any valid words from the corresponding data cache entry back into main memory,

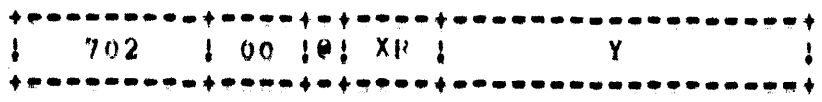
2. Flush and restart the IBOX,

The indirect, index register, and Y fields of this instruction are not used and are reserved,

KL/KS compatibility

The KS had no cache sweep instructions, The KL allowed other requests to happen in parallel with the sweep, setting sweep busy and sweep done in the APR status to indicate the sweep-in-progress interval,

2.21 RDSPB



Read SPT Base Register

This instruction stores the SPT base register in the word addressed by E. The format of the word is:

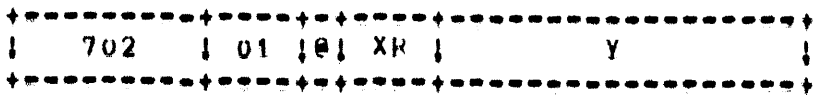
0-10 Reserved

11-35 Physical address of the SPT

KL/KS compatibility

Functionally identical to the KS RDSPB E instruction. The KL kept the address of the SPT in AC block 6.

2.22 RDCSB



Read Core Status Table Base Register

This instruction stores the CST base register in the word addressed by E. The format of the word is:

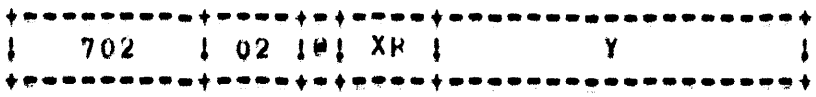
0-10 Reserved

11-35 Physical address of the CST

KL/KS compatibility

Functionally identical to the KS RDCSB E instruction. The KL kept the address of the CST in AC block 6.

2.23 RDPUR



Read Process Use Register

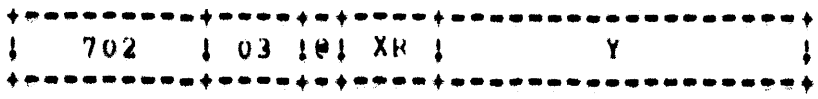
This instruction stores the process use register into the word addressed by E.

See the chapter on Paging for a description of the process use register.

KL/KS compatibility

Functionally identical to the KS PDPUR E instruction. The KL kept the process use register in AC block 6.

2.24 RDCSTM



Read CST Mask Register

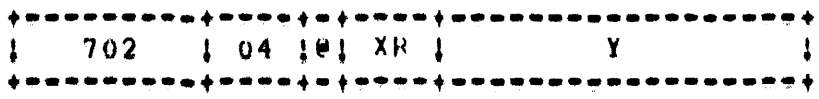
This instruction stores the CST mask register into the word addressed by E.

See the chapter on Paging for a description of the CST mask register.

KL/KS compatibility

Functionally identical to the KS RDCSTM E instruction. The KL kept the CST mask register in AC block 6.

2,25 RDTMB



Read Time Base Enables

This instruction reads the status of the time base and the interrupt level assigned to the interval timer into the word addressed by E. The status is as follows:

0-22	Reserved
23	Time base on.
24-32	Reserved
33-35	PIA for interval timer.

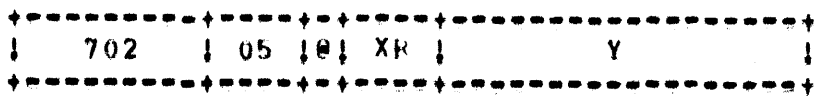
KL/KS compatibility

The KL CONI MTR, instruction returned the time base enable and interval timer PIA in a manner very analogous to this instruction. The KS had no direct equivalent.

how about using words?

Where is this described?

2.26 RDINT



Read The Interval Register

Read the status of the interval timer into the word addressed by E. The status is as follows:

0-5	Reserved
6-17	Interval count (current contents of the counter).
18-20	Reserved
21	Interval timer on.
22	Interval timer done (causes interrupt).
23	Overflow (implies bit 22).
24-35	Interval period.

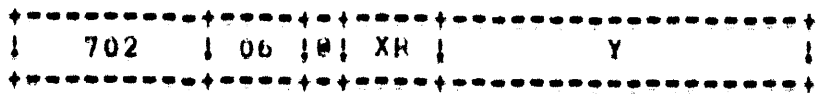
Bits 22 and 23 are the counter flags; note that Interval timer done can be set alone, but a 1 in bit 23 implies a 1 in bit 22 as well. Bits 24-35 are the period supplied by WRINT, and bits 6-17 are the current contents of the counter.

KL/KS compatibility

This instruction is functionally equivalent to the KL CONI TIM, instruction

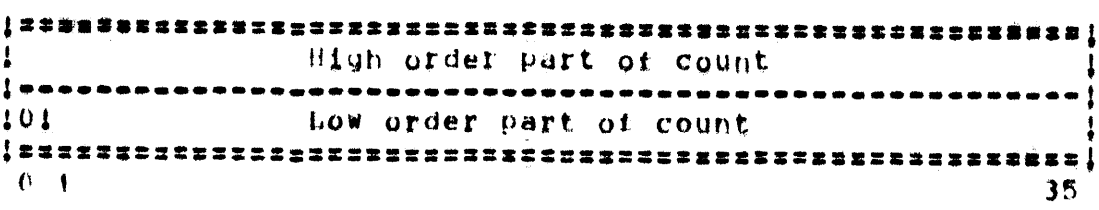
What are the units?

2.27 RDTIME



Read Time Base Value

This instruction updates the time base double word kept in internal EBOX storage from the hardware counter and returns the updated double word in the words addressed by E and E+1. The double word is a double precision integer in 1 microsecond units with the following format:



KL/KS compatibility

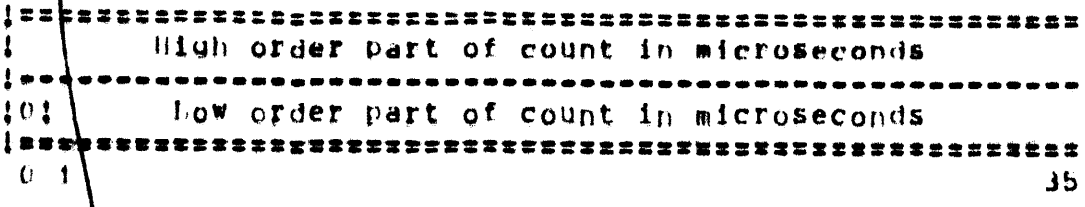
The KL RDTIME instruction also updated the time base double word kept in EPT locations 510 and 511. The KS RDTIM instruction returned the double word in the same manner as this instruction.

2.28 RDURTM

702	07	@	XR	Y

Read User Runtime Meter

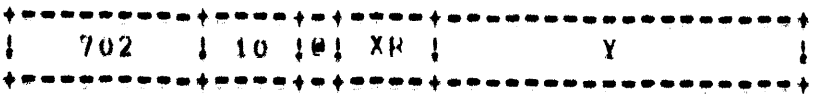
This instruction updates the user runtime meter double word kept in internal FBUX storage from the hardware counter and returns the updated double word in the words addressed by E and E+1. The double word is double precision integer in 1 microsecond units with the following format:



KL/KS compatibility

The KL and KS had no comparable instructions.

2.29 WRSPB



Write SPT Base Register

This instruction loads the SPT base register from the word addressed by E. The word format is:

0-10 Reserved

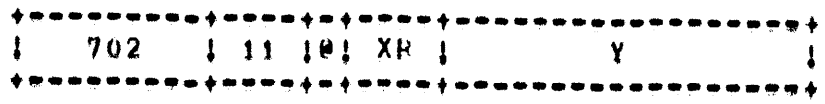
11-35 Physical address of the start of the SPT.

The SPT base register is loaded with a physical word address. The address need not be on a page boundary and may be any location in physical memory. There is no range check on SPT offsets. The monitor is assumed to always put correct data into the SPT base register.

KL/KS compatibility

Functionally identical to the KS WRSPB E instruction. The KL kept the SPT address in AC block 6.

2.30 WRCSB



Write Core Status Table Base Register

This instruction loads the CST base register from the word addressed by E. The word format is:

0-10 Reserved

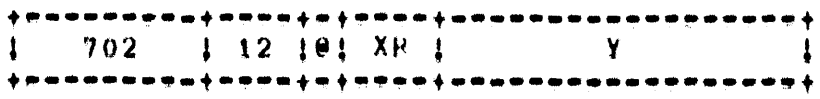
11-35 Physical address of the start of the CST. If this address is zero, the microcode will make no CST references.

The CST base register is loaded with a physical word address. The address need not be on a page boundary and may be any place in physical memory. There is no range check on CST offsets. The monitor is assumed to always put correct data into the CST base register.

KL/KS compatibility

Functionally identical to the KS WRCSB E instruction. The KL kept the CST address in AC block 6.

2.31 WRPUR



Write Process Use Register

This instruction loads the process use register from the word addressed by E.

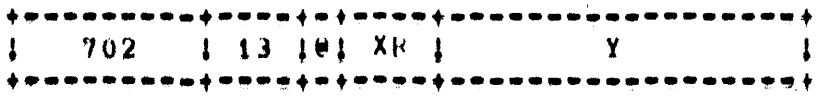
The microcode updates a CST entry by ANDing the CST mask word (see below) with the entry and ORing the process use register into the entry.

See the chapter on Paging for the format of the process use register.

KL/KS compatibility

Functionally identical to the KS WRPUR E instruction. The KL kept the process use register in AC block 6.

2.32 WRCSTM



Write CST Mask Register

This instruction loads the CST mask register from the word addressed by E.

The microcode updates a CST entry by ANDing the CST mask word with the entry and ORing the process use register into the entry.

See the chapter on Paging for the format of the CST mask register.

KL/KS compatibility

Functionally identical to the KS WRCSTM E instruction. The KL kept the CST mask register in AC block 6.

2.33 WRTMB

702	14	0	XR	Y

Write Time Base Controls

This immediate-mode instruction decodes its effective address to control the time base and the interval timer. The effective address bits are used as follows:

18	Load PI assignment for interval timer from bits 33-35.
19	Load time base controls from bits 20 and 23.
20	Clear time base.
21-22	Reserved
23	Turn on time base.
24-32	Reserved
33-35	PIA for interval timer.

PI level

The actions of this instruction are as follows:

1. If bit 18 is a 1, load the interval timer PI assignment from bits 33-35. If bit 18 is a 0, do not change the interval timer PIA.

2. If bit 19 is a 1, perform the following operations:

   1. If bit 23 is a 1, turn on the time base. If bit 23 is a 0, turn off the time base.

   2. If bit 20 is a 1, clear the time base. If bit 20 is a 0, it is ignored.

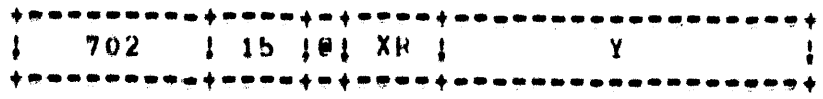
what it

   If bit 19 is a 0, do not change the state of the time base.

KL/KS compatibility

The KL CONO MTR, instruction controls the time base and the interval timer PIA in a manner very analogous to this instruction. The KS had no equivalent instruction.

2.34 WRINT



Write Interval Timer

This immediate-mode instruction decodes its effective address to setup the interval timer. The effective address bits are used as follows:

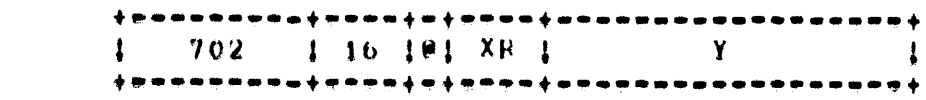
18	Clear interval timer.
19-20	Reserved
21	Turn interval timer on.
22	Clear interval flags.
23	Reserved
24-35	Interval period.

A 1 in bit 18 clears the counter and can be given simultaneously with a 1 or 0 in bit 21 to turn the counter on or off. A 1 in bit 22 clears both Interval Done and Interval Overflow. If the counter is on, Interval Done will set when the count reaches the value specified by bits 24-35.

KL/KS compatibility

This instruction is functionally equivalent to the KL CONO TIM, instruction.

2,35 WRACT



Write accounting information

This immediate instruction decodes its effective address to control the user runtime meter, The effective address bits are used as follows:

18	Load user runtime meter controls from bits 19-21,
19	Enable user runtime meter count during exec PI time,
20	Enable user runtime meter count during exec non-PI time,
21	Turn on user runtime meter
22-35	Reserved

The actions of this instruction are as follows:

1, If bit 18 is a 1, perform the following operations:

   1, If bit 19 is a 1, enable the user runtime meter to count during exec PI processing, If bit 19 is a 0, disable the user runtime meter from counting during exec PI processing,

   2, If bit 20 is a 1, enable the user runtime meter to count during exec non-PI processing, If bit 20 is a 0, disable the user runtime meter from counting during non-PI processing,

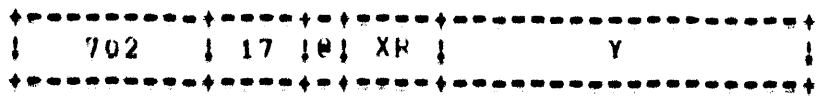
   3, If bit 21 is a 1, turn on the user runtime meter, If bit 21 is a 0, turn off the user runtime meter,

   If bit 18 is a 0, do not change the state of the user runtime meter,

KL/KS compatibility

The KL CONO MTR, instruction controls the accounting meters in a manner very analogous to this instruction, The KS had no equivalent instruction,

2.36 RDACT



Read accounting information

This instruction reads the status of the user runtime meter into the word addressed by E. The status is as follows:

0-18	Reserved
19	The user runtime meter has been enabled to count during exec PI time.
20	The user runtime meter has been enabled to count during exec non-PI time.
21	User runtime meter on.
22-35	Reserved

KL/KS compatibility

The KL CONI MTR, instruction returned the accounting meter controls in a manner very analogous to this instruction. The KS had no direct equivalent.

CHAPTER 3

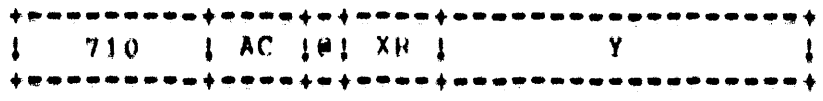
EXTERNAL I/O INSTRUCTIONS

The external I/O instructions on the KC10 allow a program to communicate with the I/O ports and the console. In particular, they will manipulate the I/O Command/Response Queues and Port Doorbell mechanism. See the "I/O Bus Spec," in the 2080 EFS for a complete description of the Queue and Doorbell features. The interface to the KC10 ports is primarily data areas called "mailboxes" and a doorbell. It is the doorbell mechanism that the following instructions manipulate. The Command/Response Queues will be covered by the queue instructions in the next section.

In general the BUSY and RING signals work as follows; The CPU can assert RING on the I/O Bus if BUSY is clear. Upon setting RING and a port number, the CPU must observe BUSY setting and then clearing before it can assume that the I/O Port has seen its command. The following 2 instructions (RNGB and RNGBW) will skip if no bus timeouts occur.

The Console does not use this protocol and therefore RING and BUSY signals are ignored if any console functions are requested by RNGB or RNGBW. The (port reset function) also ignores the BUSY signal.

3.1 RNGB



Ring Doorbell

This immediate-mode instruction makes requests of the console and ports based on the bits in E. The bits in E are interpreted as follows:

18	Cause console to reload (electronic boot finger)
19	Initialize the port specified by bits 33-35 to the power-up state.
20	Interrupt KC10 Console
21-32	Reserved
33-35	Port number (Ignored if a console function)

If none of bits 18-20 are on, the the microcode rings the doorbell of the port indicated by bits 33-35. Only one of bits 18-20 may be on in E. If more than one bit is on, a page fail is generated.

If during the process of ringing a port doorbell, a bus timeout occurs, the microcode stores an error code indicating which timeout failed in AC (the AC field of the instruction is non-zero) and takes the next instruction from PC+1. The possible error codes are as follows:

-1	The microcode was unable to ring the bell because busy never cleared. The most likely explanation for this timeout is that the port previously activated with RNGB has failed to drop "busy."
1	The microcode was able to ring the bell but busy did not set as the result of the doorbell. The most likely explanation for this timeout is that the port currently being activated is not responding to bus commands.

If no bus timeout occurs, the next instruction is taken from PC+2, i.e., the instruction skips, and AC remains unchanged.

This instruction differs from RNGBW in that it does not wait for busy to clear after the doorbell is rung. As such, it is useful for starting asynchronous operations that cause an interrupt when they complete.

3.2 RNGBW

Ring Doorbell and Wait (for BUSY to clear).

This immediate-mode instruction makes requests of the console and ports based on the bits in E. The bits in E are interpreted as follows:

18	Cause console to reload (electronic boot finger)
19	Initialize the port specified by bits 33-35 to the power-up state.
20	Interrupt KC10 Console
21-32	Reserved
33-35	Port number (ignored if a console function)

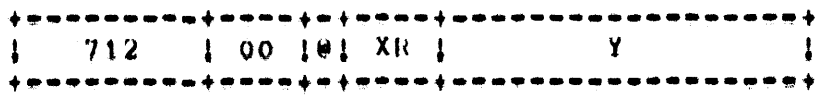
If none of bits 18-20 are on, the the microcode rings the doorbell of the port indicated by bits 33-35. Only one of bits 18-20 may be on in E. If more than one bit is on, a page fail is generated.

If during the process of ringing a port doorbell, a bus timeout occurs, the microcode stores an error code indicating which timeout failed in AC if the AC field of the instruction is non-zero, and takes the next instruction from PC+1. The possible error codes are as follows:

-1	The microcode was unable to ring the bell because busy never cleared. The most likely explanation for this timeout is that the port previously activated with RNGB has failed to drop busy.
0	The microcode was able to ring the bell and detect that busy set but it did not clear again. The most likely explanation for this timeout is that the port currently being activated failed to drop busy.
1	The microcode was able to ring the bell but busy did not set as the result of the doorbell. The most likely explanation for this timeout is that the port currently being activated is not responding to bus commands.

If no bus timeout occurs, the next instruction is taken from PC+2, i.e., the instruction skips, and AC remains unchanged.

3.3 SNBSY



Skip if BUSY not set

This instruction skips to PC+2 if the BUSY line of the KC10 I/O bus is not set. When used in combination with the RNGB instruction, one can achieve the identical effect of RNGBW as follows:

```
RNGB    pn              ; Assert RING to port "pn"
SNBSY                   ; Busy set?
 JRST   .-1             ; Yes, wait.
...                     ; No - proceed
```

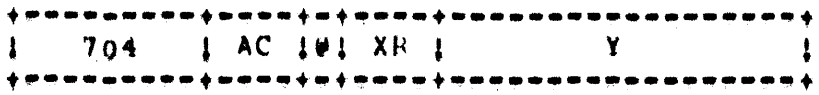
CHAPTER 4

I/O INSTRUCTIONS THAT MOVE DATA

This chapter describes the I/O instructions in the range 700-737 that move data to and from virtual or physical memory.

Like all instructions whose opcode is in the range 700-737, inclusive, these instructions may only be executed in user mode if user I/O is set. If these instructions are executed in user mode without user I/O, they execute as an MUUO, trapping through the user undefined I/O opcode dispatch in location 435 of the UPT.

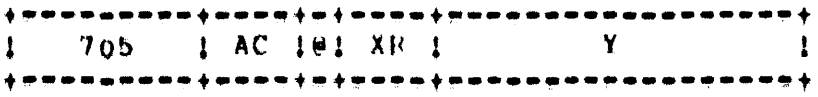
4.1 UMOVE



User Move from Memory

Load the contents of the previous context memory location addressed by E into AC. This is a (faster) replacement for PXCT 4,[MOVE AC,E]. As such, the effective address calculation for the instruction is done in current context and the word is fetched from that location in previous context.

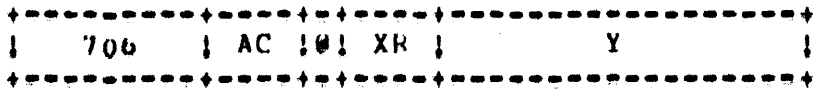
4.2 UMOVEM



User Move to Memory

Store the contents of AC into the previous context memory location addressed by E. This is a (faster) replacement for PXCT 1,[MOVEM AC,E]. As such, the effective address calculation for the instruction is done in current context and the word is stored into that location in previous context.

4.3 PMOVE



Physical Move from Memory

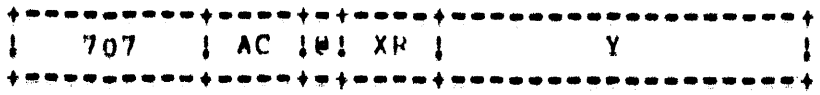
Perform a physical EA-calc using the word addressed by E, then load the physical memory location addressed by the result of the EA-calc into the AC.

See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

Because the data reference is to physical memory, an effective address in the range 0-17, inclusive, will reference physical memory locations 0-17, and not the ACs.

No CST update will be performed to indicate that this instruction referenced the physical page specified by the effective address. It is the responsibility of the monitor to perform such an update if this is required.

4.4 PMOVEM



Physical Move to Memory

Perform a physical EA-calc using the word addressed by E, then store AC into the physical memory location addressed by the result of the EA-calc.

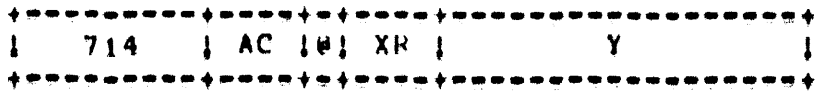
When the store is complete, flush and restart the IBUX.

See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

Because the data reference is to physical memory, an effective address in the range 0-17, inclusive, will reference physical memory locations 0-17, and not the ACs.

No CST update will be performed to indicate that this instruction referenced the physical page specified by the effective address, it is the responsibility of the monitor to perform such an update if this is required.

4.5 IOPMOV

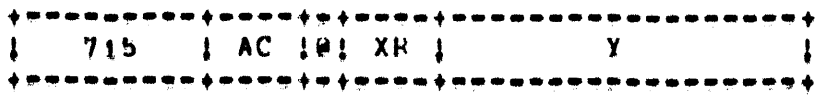


I/O page relative MOVE

Read a word from the I/O page offset specified by bits 27:35 of E and load the result in AC. The I/O page address is that specified by the last WRIOP instruction, or the initial default if no WRIOP has been done. If bits 0:26 of the effective address are non-zero, generate a page fail trap.

No CST update will be performed to indicate that this instruction referenced the physical page specified by the effective address. It is the responsibility of the monitor to perform such an update if this is required.

4.6 IOPMVM



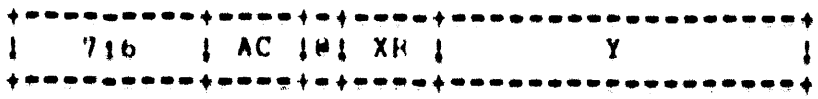
I/O page relative MOVEM

Store AC into the I/O page offset specified by bits 27:35 of E. The I/O page address is that specified by the last WRIOP instruction, or the initial default if no WRIOP has been done. If bits 6:26 of the effective address are non-zero, generate a page fail trap.

When the store is complete, flush and restart the IBOX.

No CST update will be performed to indicate that this instruction referenced the physical page specified by the effective address. It is the responsibility of the monitor to perform such an update if this is required.

4.7 LDPAC



Load previous context ACs

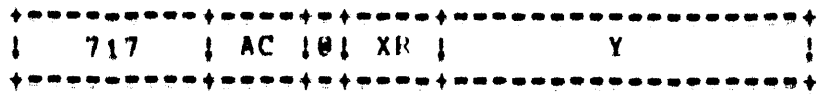
Load the previous context ACs (from the AC block specified by the current PAB value) from the block beginning at the location addressed by E. Continue to transfer words from the block until a word has been transferred to the previous context AC specified by the AC field of the instruction.

The 16-word block must not cross section boundaries. why

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
```

4.8 STPAC



Store previous context ACs

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
```

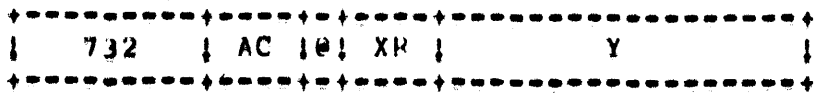
CHAPTER 5

SPECIAL DEBUGGING INSTRUCTIONS

This chapter describes several instructions that have been added to the instruction set to aid in debugging the hardware and microcode. They will not appear in the final production microcode and are documented here only for completeness.

Like all instructions whose opcode is in the range 700-717, inclusive, these instructions may only be executed in user mode if user I/O is set. If these instructions are executed in user mode without user I/O, they execute as an MUUO, trapping through the user undefined I/O opcode dispatch in location 435 of the UPT.

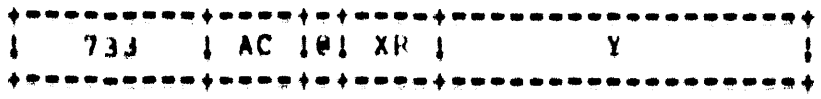
5.1 RDTRAX



Read tracks buffer information

This instruction returns the current tracks buffer address (not page number). This address is the next location into which the microcode will write a PC.

5.2 WRTRAX



Write tracks buffer address/enable

This instruction sets the tracks buffer address, length, and enables or disables the microcode tracks processing. The word addressed by E controls the operation of the instruction and has the following format:

0 Enable the microcode tracks processing. When this feature is enabled, the microcode stores the PC of each instruction executed in a circular buffer in physical memory. If this bit is off, the tracks processing is disabled.

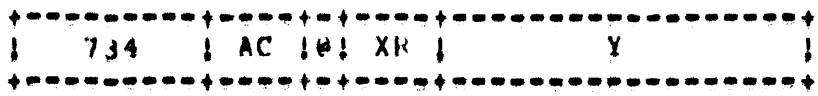
1-17 Two's complement length of the tracks buffer in words. Note that if tracks processing is being enabled, this makes the entire left half of this word be the two's complement length of the buffer.

18-35 Physical page number of the start of the buffer. The microcode will begin storing PCs starting at this physical page and continuing for the length of the buffer. When the buffer limit is reached, the microcode will reset its pointers and start at the beginning of the buffer again.

CAUTION

Enabling tracks processing will significantly degrade the speed of the machine. Besides the overhead of one memory write for each instruction executed, the implementation of this feature also causes the IBUX to be flushed at the end of every instruction, thereby completely defeating the pipeline mechanism.

5.3 READTB



Read translation buffer entry

This instruction allows the monitor to directly read an MBOX translation buffer entry. The word addressed by E contains the index into the translation buffer and may be thought of as a virtual address. In addition to the normal 30 bit VMA in bits 6-35 of the word, bit 5 is used to specify whether the reference is for exec (bit=0) or user (bit=1) translation. Bits 16-26 are the index into the translation buffer, but bit 16 is complemented if bit 5 is on. To simply read a specified translation buffer location, bit 5 should be zero and bits 16-26 should give the desired index into the translation buffer. Bits 5-15 only need be specified if the read is also to do a valid translation check as indicated by bit 7 returned in the AC (see below). This instruction does not modify the contents of the translation buffer entry (except as the possible result of a translation buffer refill that occurs as the result of the instruction or data fetch). The translation buffer entry is returned in the AC and has the following format;

0 Hardware error. If this bit is on, the translation buffer access caused a hardware error.

1 TB KEEP. If this bit is on, the translation buffer entry has the "keep" bit set.

2 VMA 05. This bit is a copy of bit 5 of the address that was specified to the MBOX for the read.

3 TB VALID. If this bit is on, the translation buffer entry contains a valid translation.

4 TB CST UPDATE. If this bit is on, the translation buffer "CST update needed" bit is set.

5 TB WRITABLE. If this bit is on, the translation buffer "writable" bit is on.

6 TB MODIFIED. If this bit is on, the translation buffer "modified" bit is on.

7 -VALID TRANSL. If this bit is on, there was no valid translation for the requested address. This bit should normally be ignored since the READTB instruction specifies

an index into the translation buffer and not a full address.

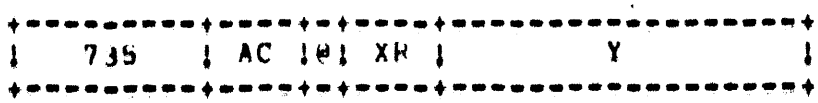
8 Returned as zero.

9 TB USER. If this bit is on, the mapping in the entry is for a user page. If this bit is off, the mapping is for an exec page.

10-19 TB DIR<6:15> This field contains the translation buffer directory entry for the mapping. This is bits 6-15 of the VMA for the mapping (bits 16-26 are implicitly specified by the offset in the translation buffer).

20-35 TB PPN<11:26> This field contains bits 11-26 of the physical address for the mapping.

5.4 WRITTB



Write translation buffer entry

This instruction allows the monitor to directly write an MBOX translation buffer entry. The word addressed by E contains the index into the translation buffer and may be thought of as a virtual address. In addition to the normal 30 bit VMA in bits 6-35 of the word, bit 5 is used to specify whether the reference is for exec (bit=0) or user (bit=1) translation. Bits 16-26 are the index into the translation buffer, but bit 16 is complemented if bit 5 is on. The data to be written into the translation buffer entry is taken from the AC specified by the instruction and is written into the translation buffer entry specified by bits 5 and 16-26 of the VMA. The format of the data is as follows:

0-2 Ignored.

3 TB VALID. If this bit is on, the translation buffer will contain a valid translation.

4 TB CST UPDATE. If this bit is on, the translation buffer "CST update needed" bit will be set.

5 TB WRITABLE. If this bit is on, the translation buffer "writable" bit will be set.

6 TB MODIFIED. If this bit is on, the translation buffer "modified" bit will be set.

7 Ignored.

8 TB KEEP. If this bit is on, the translation buffer keep bit will be set.

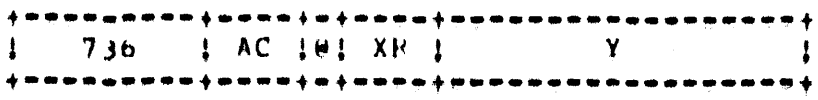
9 TB USER. If this bit is on, the mapping in the entry is for a user page. If this bit is off, the mapping is for an exec page.

10-19 TB DIR<6:15> This field contains the translation buffer directory entry for the mapping. This is bits 6-15 of the VMA for the mapping (bits 16-26 are implicitly specified by the offset in the translation buffer).

20-35 TB PPN<11:26> This field contains bits 11-26 of the physical address for the mapping.

At the completion of the instruction, the IBOX is flushed and restarted

5.5 DUMPTB



Dump translation buffer

This instruction allows the monitor to dump the entire MBOX translation buffer into 2048 contiguous physical memory locations. AC contains the physical memory address of the first translation buffer entry to be stored. The address need not be on a page boundary but the locations must be contiguous in physical memory. Each entry dumped has the format described for the READTB instruction described above. The indirect, index and Y fields of the instruction are not used and are ignored.

At the completion of the instruction, the IBOX is flushed and restarted.

CHAPTER 6

QUEUES AND QUEUE MANIPULATION INSTRUCTIONS

6.1 Introduction

The KC10 provides instructions to manipulate queues. These instructions are available in EXEC mode only, and are intended to allow sharing of queues among any combination of the following:

1. One or more processes running in the CPU.
2. One or more ports.

6.2 Data structures

6.2.1 The queues

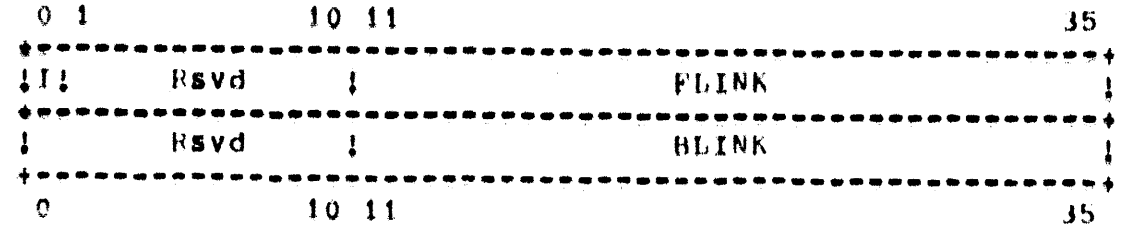
Queues that are manipulated by these instructions must

1. Be doubly-linked.
2. Contain a forward pointer in offset 0 of each entry.
3. Contain a backward pointer in offset 1 of each entry.
4. Be pointed to by a pair of header words.
5. Be referenced by physical addresses.

6.2.2 Formats

A queue header consists of a pair of words. Offset 0 points to the first entry in the queue; offset 1 points to the last entry. If a queue is empty, both header words point to offset 0.

The format of the header words is as follows:



The format of word H is as follows:

0 The secondary queue interlock bit

1-10 Reserved

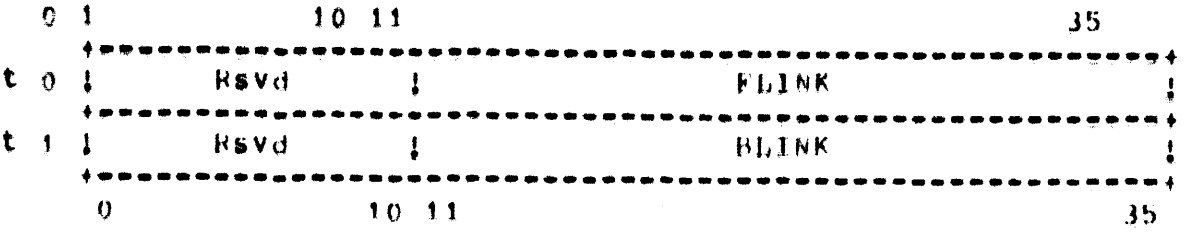
11-35 Physical address of first entry in queue (FLINK)

The format of word H+1 is as follows:

0-10 Reserved

11-35 Physical address of last entry in queue (BLINK)

Each entry contains forward and backward pointers in the following format:



The format of the first link word in a queue entry is as follows:

0-10 Reserved

11-35 Physical address of next entry in queue (FLINK)

The format of the second link word in a queue entry is as follows:

0-10 Reserved

11-35 Physical address of previous entry in queue (BLINK)

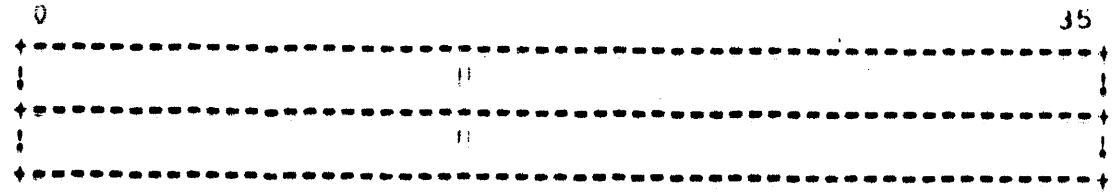
6.3 Operations

The instructions provide four functions:

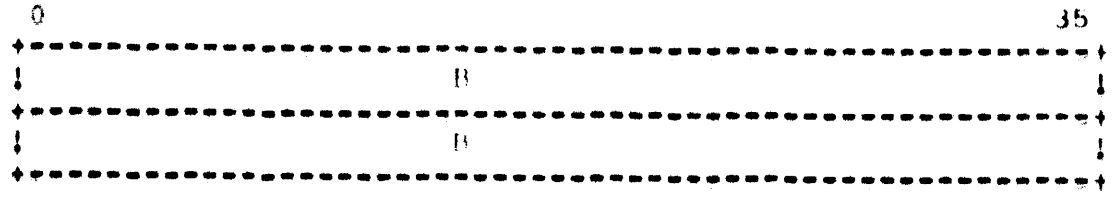
1. Insertion of an entry at the head of a queue,
2. Insertion of an entry at the tail of a queue,
3. Removal of an entry from the head of a queue,
4. Removal of an entry from the tail of a queue,

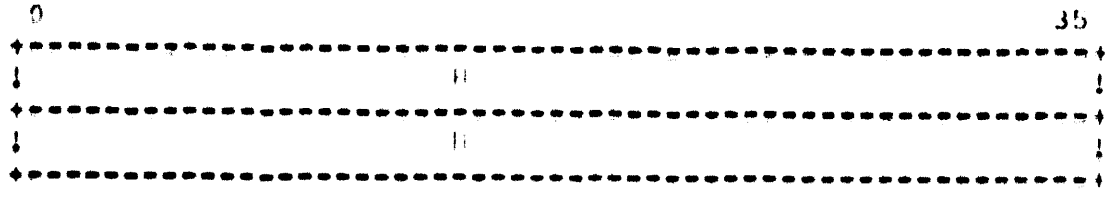
6.3.1 Insertion

An empty queue is specified by its header at address H:



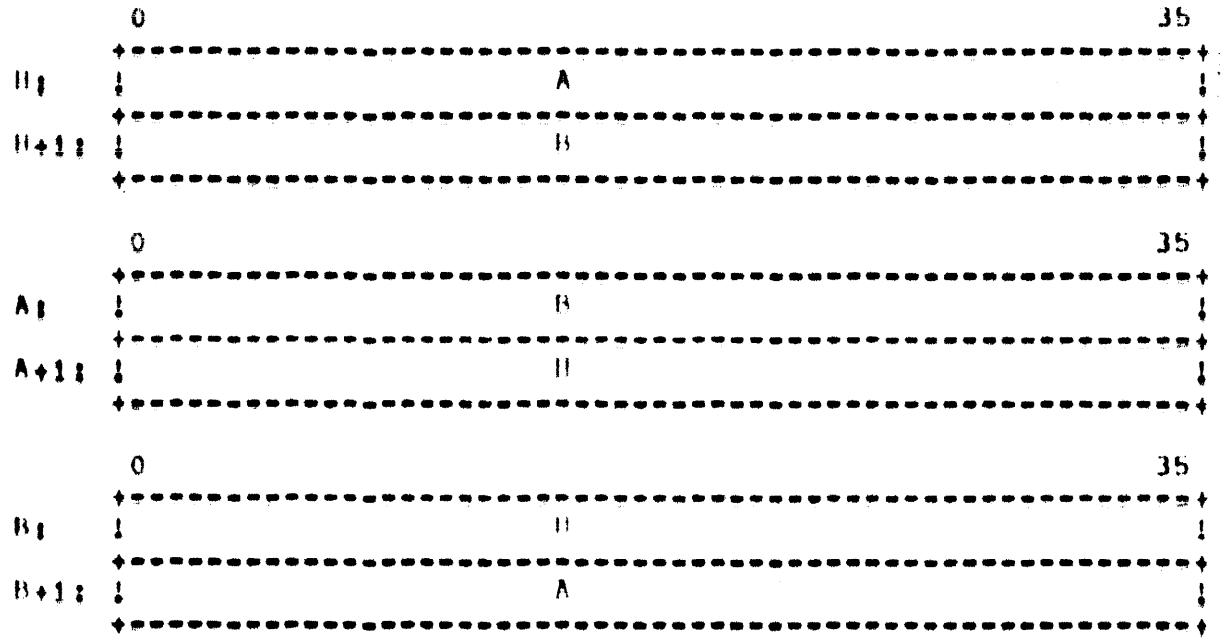
If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown below:



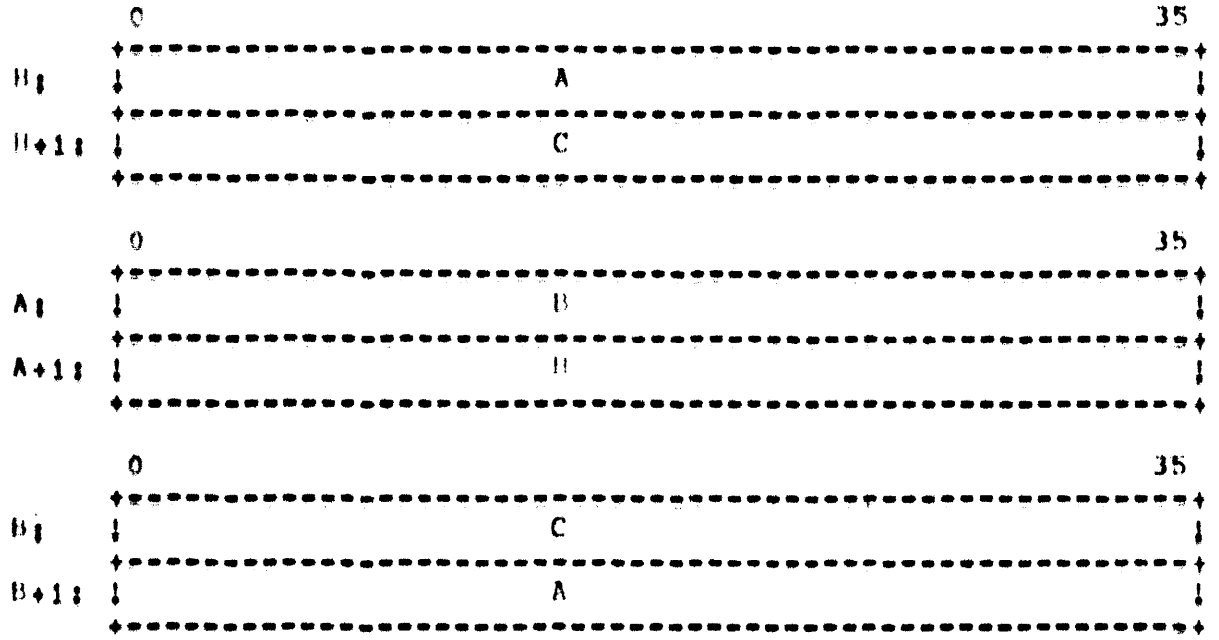


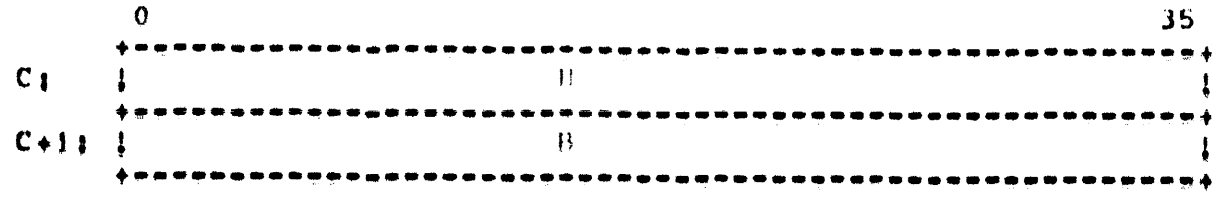
Operations

If an entry at address A is inserted at the head of the queue, the queue is show below:



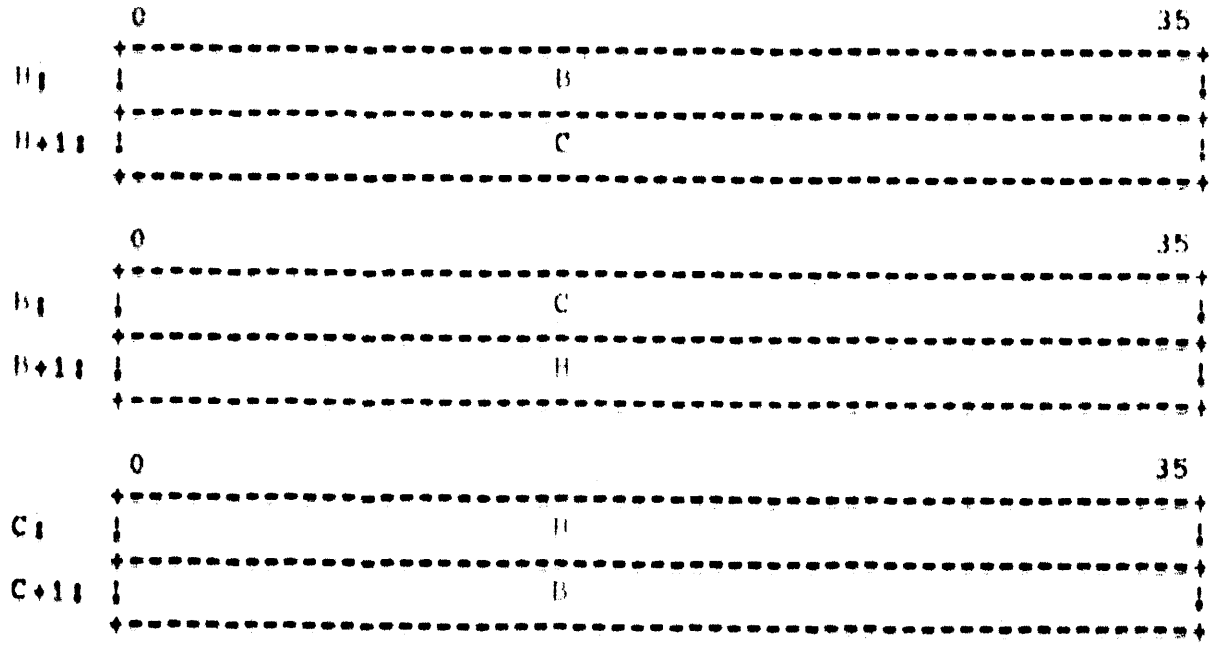
Finally, if an entry at address C is inserted at the tail, the queue appears as follows:





6.3.2 Removal

In the example above with the queue containing entries A, B, and C, the entry at address A can be removed giving:



6.4 Interlocks

Cooperating users of a queue can ensure that no conflicts occur by using only the queue instructions when adding or deleting entries.

When executing a queue instruction, the CPU uses two interlocks. First it uses the MBOX "read-interlock" function to read the queue header. This function sets a hardware interlock that delays any subsequent read-interlock request.

Bit 0 of the queue header provides a secondary interlock. If the bit is off, the queue is available, and the CPU uses the MBOX "write-release" function to set the bit in the header word and release the interlock. At this point, any pending read-interlock finds the bit set in the header.

Having obtained the secondary interlock, the CPU performs the queue manipulation specified by the instruction. It then performs another read-interlock on the header, clears the secondary interlock, and performs a write-release.

Alternatively, if the CPU finds the secondary interlock set, it performs the write-release without changing the header and retries <TBS> times in an attempt to get the secondary interlock. If all retries are unsuccessful, control returns to the user. at +1 ?

The I/O ports manipulate the queues in a similar way. This allows the ports and CPU to cooperate in the use of I/O queues.

If the queue instruction returns an interlock failure, it may be necessary for the CPU to free the interlock. This action would probably consist of reinitializing the port that has the interlock, cleaning up the queue, and then clearing the secondary interlock bit in the queue header so that the queue is accessible again. It is assumed that there is a direct association between the queue that is interlocked and a particular port.

6.5 The Instructions

The queue instructions have common characteristics, as follows:

1. For insertions, the AC contains the physical address of word zero of an entry to be inserted. Bits 0-10 must be 0. For removals, the physical address of word zero of the entry that was removed is returned in the AC. If the AC contains a value in the range 0-17, it is interpreted as a physical address, not an AC.

2. E addresses a physical EA-calc word that is evaluated to produce a 25 bit physical address of the queue header. See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

3. If the secondary interlock is locked, the instruction returns +1. Otherwise, it returns +2.

4. If the instruction skips, it may provide further information. For insertion instructions, if the queue is empty before the insertion, the instruction sets bit 0 of the AC provided. For removal, if the queue is empty, the instruction sets bit 0 of the AC provided. Note that the instructions never clear bit 0; software must clear it in order to test for an empty queue.

5. No CST update is performed.

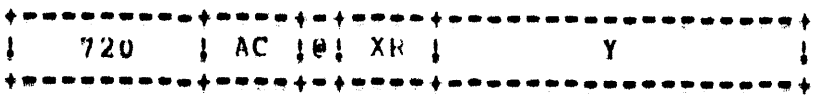
after? or in error?

6.6 Errors

To be supplied

Can AC be
re-called 400?

6.7 INSQHI



Insert Entry into Queue at Head, Interlocked

Perform a physical EA-calc using the word addressed by E, then insert the entry specified by the physical address contained in AC into a queue following the header specified by the result of the physical EA-calc.

If the secondary interlock was unavailable (i.e., bit 0 of the queue header = 1), the instruction returns to PC+1, otherwise it returns to PC+2.

If the entry inserted was the first one in the queue (i.e., E = C(E) before insertion), the instruction sets bit 0 in the AC (bits 1-35 are unchanged). If the entry inserted was not the first one in the queue, AC is unchanged.

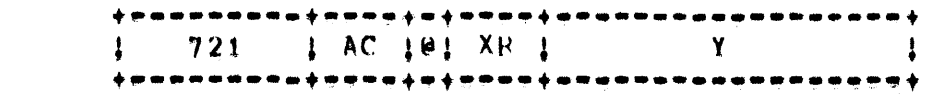
The correct way to insert an item at the head of a queue is as follows:

```
INSQHI AC,E
 CALL INTERR            ;interlock error
JUMPL AC,EMPTYQ         ;entry into empty queue
<entry into non-empty queue>
```

At the completion of the instruction, the IBOX is flushed and restarted.

See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

6.8 INSQTI



Insert Entry into Queue at tail, Interlocked

Perform a physical EA-calc using the word addressed by E, then insert the entry specified by the physical address contained in AC into a queue preceding the header specified by the result of the physical EA-calc.

If the secondary interlock was unavailable (i.e., bit 0 of the queue header = 1), the instruction returns to PC+1, otherwise it returns to PC+2.

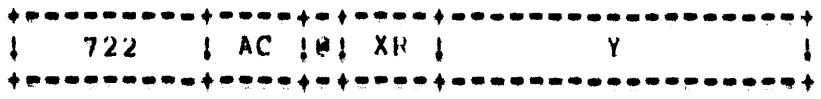
If the entry inserted was the first one in the queue (i.e., E = C(E) before insertion), the instruction sets bit 0 in the AC (bits 1-35 are unchanged). If the entry inserted was not the first one in the queue, AC is unchanged.

The correct way to insert an item at the tail of a queue is as follows:

```
INSQTI AC,E
 CALL INTERR            ;interlock error
JUMPL AC,EMPYQ          ;entry into empty queue
<entry into non-empty queue>
```

At the completion of the instruction, the IBUF is flushed and restarted. See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

6.9 REMQHI



Remove Entry from Queue at Head, Interlocked

Perform a physical EA-calc using the word addressed by E, then remove the queue entry following the header specified by the result of the physical EA-calc.

If the secondary interlock was unavailable (i.e., bit 0 of the queue header = 1) the instruction returns to PC+1, otherwise is returns to PC+2.

If there was no entry in the queue (i.e., E = C(E) before removal), the instruction sets bit 0 in the AC (bits 1-35 are lost). If there was an entry in the queue, the 25-bit physical address of the entry removed is placed in AC.

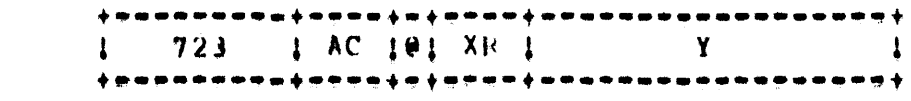
The correct way to remove an item from the head of a queue is as follows:

```
REMQHI AC,E
 CALL INTERR            ;interlock error
JUMPL AC,NOENTR         ;no entry found
<entry returned in AC>
```

At the completion of the instruction, the IBUX is flushed and restarted.

See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

6.10 REMQTI



Remove Entry from Queue at Tail, Interlocked

Perform a physical EA-calc using the word addressed by E, then remove the queue entry preceding the header specified by the result of the physical EA-calc.

If the secondary interlock was unavailable (i.e., bit 0 of the queue header = 1) the instruction returns to PC+1, otherwise is returns to PC+2.

If there was no entry in the queue (i.e., E = C(E) before removal), the instruction sets bit 0 in the AC (bits 1-35 are lost). If there was an entry in the queue, the 25-bit physical address of the entry removed is placed in AC.

The correct way to remove an item from the tail of a queue is as follows:

```
REMQTI AC,E
 CALL INTERR            ;interlock error
JUMPL AC,NOENTR         ;no entry found
<entry returned in AC>
```

At the completion of the instruction, the IBOX is flushed and restarted.

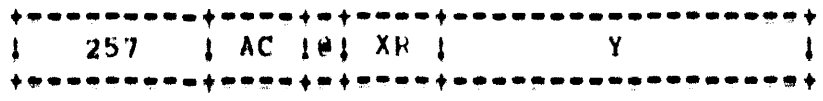
See the chapter on Miscellany for a discussion of the physical EA-calc algorithm.

CHAPTER 7

FUNCTIONAL CHANGES FROM PREVIOUS MACHINES

This chapter describes the functional changes that distinguish the KC10 from previous machines, notably the KL10. In some instances, the change is described in detail in this chapter. In other cases, the change is simply noted and the reader is referred to another chapter where the change is discussed.

7.1 MAP



Map an address

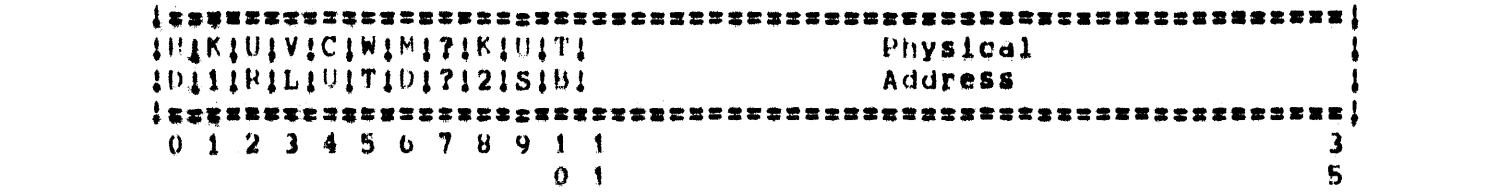
If the pager is on and the processor is in executive or user I/O mode, this instruction reads the hardware translation buffer location corresponding to the effective address. If the translation buffer contains a valid mapping for that page, the mapping is returned in the format described below. If the translation buffer does not contain a valid mapping for the page, the EBUX microcode does a page refill pointer chase to compute the mapping and returns that in the format described below. The result of the mapping is returned in the AC.

This instruction does not change the hardware translation buffer mapping for the page specified by the effective address calculation.

This instruction cannot be performed in a user program unless user I/O is set. Instead of mapping the address, it executes as an MUUO, dispatching through the user undefined I/O opcode dispatch in location 435 of the UPT.

If the pager is off, the effective address and the "valid mapping" bit are returned in AC. See below.

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



The fields are as follows:

0 This bit is a one if the instruction failed to generate a valid mapping because of a hardware error. In this case, bits 1-4 contain a failure code instead of the bits described below. These codes are the same as those returned for a page fail with bit 0 set in the page fail word.

1 See the description of bit 8 below.

2 If bit 10 is a zero, this bit gives the state of the user request bit for the MBOX reference that returned a valid mapping. If bit 10 is a one, the state of this bit is undefined.

3 If this bit is a one, the rest of the information returned, including the physical address, is valid. If this bit is a zero, there is no valid mapping for the virtual address and bits 18-35 contain the reason the microcode couldn't find a valid mapping. This information has the same format as bits 18-35 of the page fail word that would be returned if the specified page was referenced.

4 If this bit is a one, the next virtual reference to the page being mapped will cause the EBOX microcode to perform a CST update operation for the page.

5 If this bit is a one, the page being mapped is writable. If this bit is a zero, the page being mapped is write protected.

6 If this bit is a one, the page being mapped has been modified since being brought into memory, i.e., the page is newer than any backup copy. If this bit is a zero, the page has not been modified since being brought into memory. How is it set?

7 The state of this bit is undefined.

8 If bit 1 or this bit is a one, the hardware virtual-to-physical mapping for the page being mapped will not be invalidated on a conditional pager clear. If both bit 1 and this bit are zeros, the mapping will be invalidated on all pager clears.

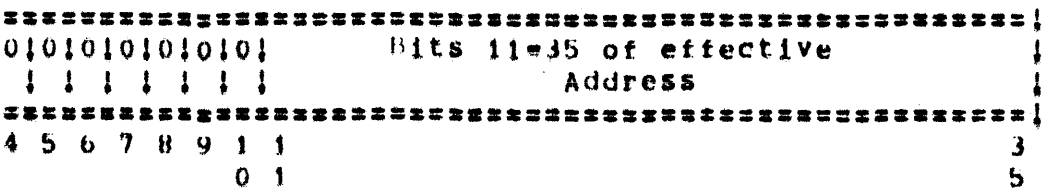
9 If this bit is a one, the mapping for this page is in user space. If this bit is a zero, the mapping for this page is in executive space.

10 If this bit is a zero, the microcode found valid information in the hardware translation buffer for this mapping. If this bit is a one, the microcode performed a pointer trace to compute the mapping.

11-35 The physical address corresponding to the virtual address of the effective address calculation.

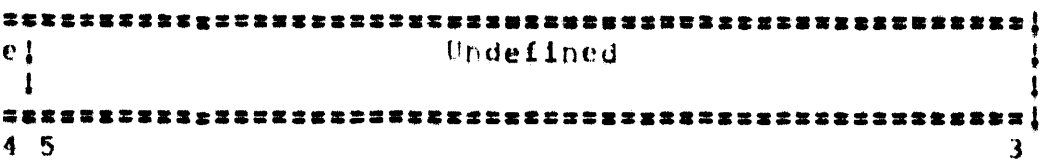
The information returned in the AC by the MAP instruction can be in one of four different formats, as follows:

If the pager is off, bits 11-35 of the effective address of the instruction are returned in bits 11-35 of AC. Bit 3 (the valid bit) is set, and all other bits are zero. This looks as follows:



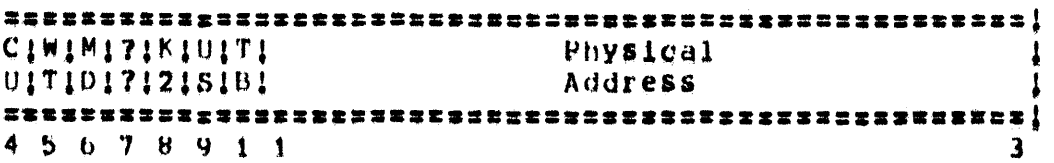
Format of AC if paging is off

If the read of the translation buffer mapping information results in an error, bit 0 is set and bits 1-4 contain a failure code that describes the error. This code is the same as that returned for a page fail with bit 0 set. Bits 5-35 of the AC are undefined. This format looks as follows:



Format of AC if a hard error is detected

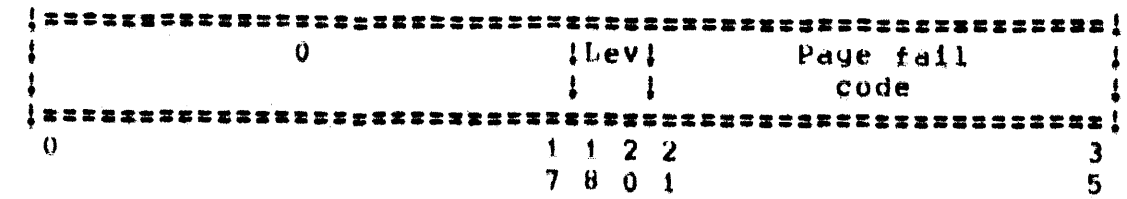
If the pager is on and there is a valid mapping for the specified virtual address (either in the MBOX translation buffer or as the result of a pointer trace), the format of the information returned in AC is as follows:



Format of AC if a valid mapping was found

15

If the pager is on and there was no valid mapping for the effective address found (either in the MBOX translation buffer or by performing a pointer trace), the entire left half of 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 the specified page was referenced. This format looks as follows:



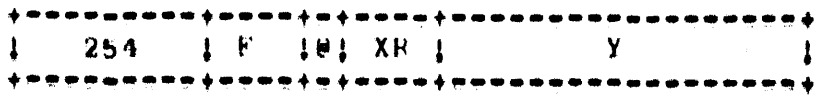
The correct way to test for a valid mapping returned by MAP is as follows:

```
        TLNN    AC,(1B0)        ;Hard error?
        TLNN    AC,(1B3)        ;No, is valid bit on?
        FAILURE                 ;Here if hard error or valid bit off
        SUCCESS                 ;Here if AC contains a valid mapping
```

KL/KS compatibility

The KL and KS returned different bits describing the mapping,

7.2 JRST



Jump and Restore

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

F	Mnemonic	Function
01	JRSTCI	Flush the IBOX, then jump to location E. This instruction (previously defined as PORTAL) should be used anytime the program modifies the instruction stream.
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	JEN	Always execute as an MUUO through the I/O undefined opcode new PC words in the UPT. Since the KC10 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.

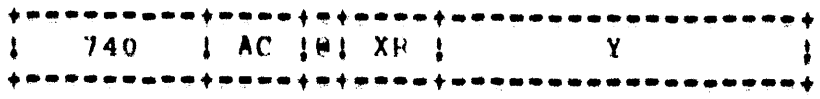
For each of the 16 possible JRST functions, the table given below indicates where each form of the instruction is legal. The meanings of the symbols used to define the legal domains of the functions are as follows:

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

7.3 PUSHM



PUSH multiple ACs

This instruction pushes ACs onto the stack addressed by the stack pointer in AC. The instruction interprets the contents of E 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, on 1st When this is complete, push the full 30-bit effective address of the instruction on the stack.

PC?

Why 2 values?

PUSHM seems to assume instr. goes

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).

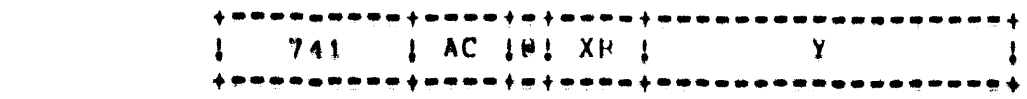
When all designated ACs are pushed onto the stack, the effective address of the instruction (including section number) is also pushed onto the stack if the function code is 2 or 3.

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.

Strange effect. Do have a precedent?

7.4 POPM



POP multiple ACs

This immediate instruction pops ACs from the stack addressed by the stack pointer in AC. The instruction interprets E as follows:

6-17	Ignored
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,

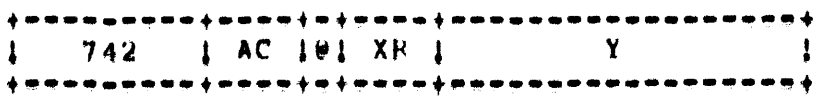
When all designated ACs are popped from the stack, a non-skip or skip return may be performed if the function code is 2 or 3.

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.

so we can't find the POPM?
seems inconsistent with leaving P unchanged

7.5 PUSHI



PUSH immediate

If PC section is non-zero, this instruction pushes 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~~ onto 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
```

7.6 Other functional changes

This section lists other functional changes that distinguish the KC10 from previous machines.

7.6.1 Changes to privileged instructions

Many of the instructions whose opcodes are in the range 700-737 have changed to adapt to the KC10 exec-mode environment. See the chapter on APR0, APR1, and APR2 Instructions for more detail.

7.6.2 Elimination of public and supervisor modes

The KL10 public and supervisor processor modes have been eliminated from the KC10. The processor acts as if it were in kernal mode while in exec mode and as if it were in concealed mode while in user mode.

In addition to removing the modes, the KC10 eliminates other things that are related to the processor modes. Among these are the previous context public PC flag, the "P" bit in TOPS-20 paging pointers, the "P" bit in the page fail word, etc.

7.6.3 Overflow in exec mode

In exec mode on the KL10, PC flag bit 0 was used to indicate previous context public. On the KC10, PC flag bit 0 is used to indicate overflow both in user and exec modes. Therefore, an exec-mode program can cause a trap 1 event that is processed in the same manner as a user-mode trap 1 event.

7.6.4 Cachable page status

The ability to make single pages cachable or not has been removed from the KC10. Therefore, there is no longer a "C" bit in the TOPS-20 paging pointers, nor a "C" bit in the page fail word. On the KC10, all memory references are assumed to be cachable.

7.6.5 XBLT in section zero

The XBLT instruction is legal when PC section is zero on the KC10. See the chapter on Extended Addressing for more information on this subject.

7.6.6 JRST changes

Some of the JRST-class instructions have changed and/or been redefined on the KC10. The key changes are as follows:

- PORTAL has been renamed to JRSTCI and is now used to flush the IBOX if the program overwrites the instruction stream.
- JRST 10, and JEN are illegal and are treated as MUUOs.
- SFM is now legal in all sections.
- XJRST has been added.

See the section on JRST above more more information.

7.6.7 PXCT

PXCT has been changed to make it obey the architecture. Also, in places where the architecture defined conflicting requirements, it was changed also. The PXCT section in the chapter on Extended Addressing contains more information about this topic. The key functional changes are as follows:

- The allowable cases of PXCT [BLT] have been limited to remove some conflicting requirements. LDPAC and STPAC have been added to make up for the loss.
- Previous context stack references for PUSH and POP are no longer allowed.
- The interpretation of the AC bits for PXCT [MOVSLJ] has changed to reduce the complexity.

7.6.8 Paging

The KC10 implements the full 4096-section virtual address space as defined in the architecture. To do this, an additional level of pointer data structure was defined to support the change. Also, additional bits have been defined in the pointers and in the CST.

The KI10 paging mode that was used on the KI10 and early TOPS-10 versions of the KL10 has been decommitted.

See the chapter on Paging for more information.

7,6,9 Flags-PC double word

The format of the flags-PC double word has changed to allow for a 12-bit PCS field and to include the CAB and PAB fields. See the chapter on Process Context Variables for more information.

7,6,10 Process context variables

The action of the processor with regard to process context variables on a context switch has changed. See the table at the end of the chapter on Process Context Variables for more information.

7,6,11 System timers

The format of the timebase has changed to right-justify the double-precision integer in the double-word. In addition, the KL10 accounting meters have been replaced by the user runtime meter. See the chapter on System Timers for more information.

CST write bit
Keep me
CST update needed
page fault data

CHAPTER 8

PAGING

8.1 Introduction

The KL10 implemented both TOPS-10 paging, which supported only one section of virtual address space, and TOPS-20 paging, which supported a maximum of 32 sections of virtual address space. The paging data structures used on the KL10 imposed these limitations.

The KC10 will only implement a TOPS-20 paging KL10 compatible sub-mode that will support a maximum of 32 sections, and a new mode that supports 4096 sections of virtual address space with TOPS-20 paging. This will be done in such a manner as to allow different processes using TOPS-20 paging to be in different paging modes without having to implement a new "mode" bit in the process context variables.

8.2 Paging hardware and microcode

The KC10 translation buffer (or page table as it was known on the KL10) is 2K words long, 1 way associative, and has a 1 word block size. In the KL10, translation buffer refills for only TOPS-20 paging were done by the EBOX microcode. In the KC10, all translation buffer refills are done by the EBOX microcode.

The KC10 translation buffer is indexed by virtual address bits 16-26 with the state of bit 16 inverted in user space to separate user and exec entries for the same page. Each translation buffer slot contains virtual address bits 6-15 for the current entry, state bits (described below), and the corresponding physical address bits 11-26 for the current entry. The translation buffer state bits are as follows:

User — A 1 in this bit indicates that this entry describes a user page. A 0 indicates that this entry describes an exec page.

Valid — A 1 in this bit indicates that this entry contains a valid mapping. A 0 in this bit indicates that no valid mapping exists in this translation buffer entry and that an EBOX translation buffer refill is required when a virtual reference is made.

Modified — A 1 in this bit indicates that the mapping describes a page that has been modified since being brought into memory, i.e., that this page is newer than any backup copy. A 0 in this bit indicates that the mapping describes a page that has not been modified since being brought into memory. A write reference to a page whose translation buffer "modified" bit is 0 will cause an EBOX page fail trap. The EBOX will update the CST entry for that page to set the M bit (bit 35), set this bit in the translation buffer entry, and restart the reference.

Writable — A 1 in this bit indicates that the mapping describes a page that is writable. A 0 in this bit indicates that the mapping describes a page that is write-protected. A write reference to a page whose translation buffer "writable" bit is 0 will cause an EBOX page fail trap and a corresponding page fail trap to the monitor.

Keep — A 1 in this bit indicates that this mapping is not to be invalidated if the translation buffer is cleared with a WRCTX instruction that does not specify all pages (bit 3 in E of WRCTX). It has no effect on translation buffer clears caused by WREBR or CLRPT instructions. A 0 in this bit indicates that there are no restrictions in clearing this entry on a translation buffer clear.

CST update — A 1 in this bit causes an EBOX page fail trap on the next virtual reference to the page described by this entry. The EBOX performs a CST update operation and clears this bit in the entry without clearing the rest of the mapping. A SETCU instruction sets this bit in all entries in the translation buffer and it is cleared by the EBOX for individual entries when the CST update has been performed.

8.3 Caching of paging information

In an attempt to make the virtual-to-physical translation performed by the pager as fast as possible, the KC10 keeps a cache of several levels of information about recent translations. The most obvious example of this caching is the MBOX translation buffer which stores, in hardware, up to 2K translations. In addition to this, the EBOX microcode caches some information about the last few translation buffer refills that it performed in working storage inside the EBOX. The EBOX cache is intended to make translation buffer refills as fast as possible in the case where there is no valid translation in the MBOX.

One aspect of this caching is that the monitor must tell the microcode and hardware when it changes a mapping. In previous machines, this simply meant that the monitor did a CLRPT instruction to clear a translation for a single virtual page or did a CONO PAG, or DATAO PAG, to clear the entire translation buffer. The same concept holds for

the KC10, although the invalidation also effects the EBOX caching.

The CLRPT, WRCTX, and WREBR instructions still clear the MBOX translation buffer entry or entries as appropriate but they also clear all or part of the EBOX information. This process should be transparent to the monitor programmer; if the invalidation would work on a KL10, it will work on a KC10 since the same algorithms apply.

There is one case, however, where the KC10 is different. In order to optimize the processing of KL compatible paging vs. KC paging, the EBOX microcode caches the first super section pointers from EPT and UPT locations 520. These two locations are read and cached anytime the monitor does a WREBR instruction or a WRCTX that changes the UBR, and the information is NOT cleared on a CLRPT. The ONLY way to flush this information is with another WREBR or WRCTX.

8.4 TOPS-20 paging

8.4.1 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.

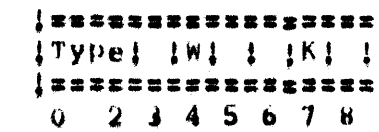
3. VMA<6:8> will be called the "Super Section Number" and will be used to determine which of the 8 Section Tables to look in.

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.

8.4.2 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, 6, 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. These bits, W and K, indicate whether the super section, section, or page is writable, or kept. Throughout the evaluation procedure the microcode effectively ands these bits from one pointer to the next, so the final result requires that the given characteristics be specified at every step.

In other words, if W is 1 in the final pointer for the mapping, the page is writable provided the 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.

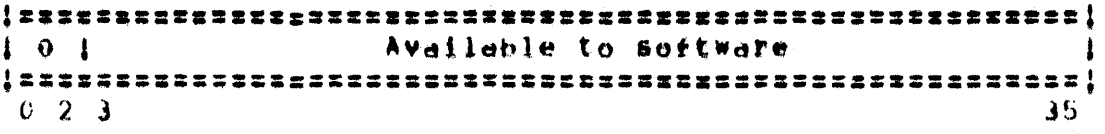
Note that the W bit is also ANDed with the W bit in the CST entry for the final data page to determine the state of the translation buffer W state bit. This final operation is not done if the CST base address is zero.

? So, that different from any other CST thing?

8.4.2.1 Super Section Pointers

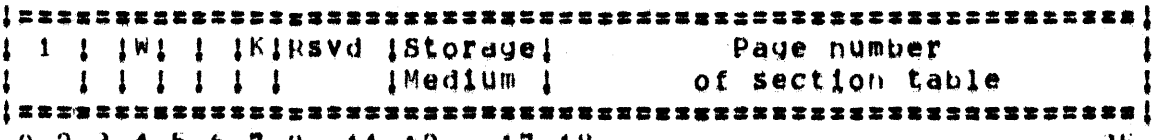
Entries in the Super Section Table in EPT/UPT locations 520-527 are of the following five types. All other types are reserved and will cause a page fail if the microcode encounters them on a refill.

No access



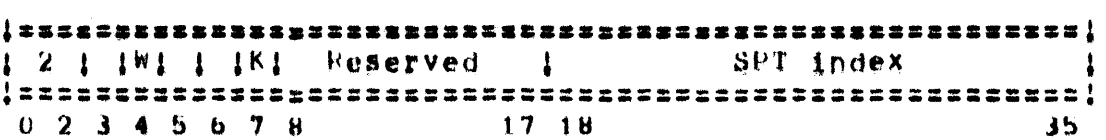
The super section is inaccessible.

Immediate



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

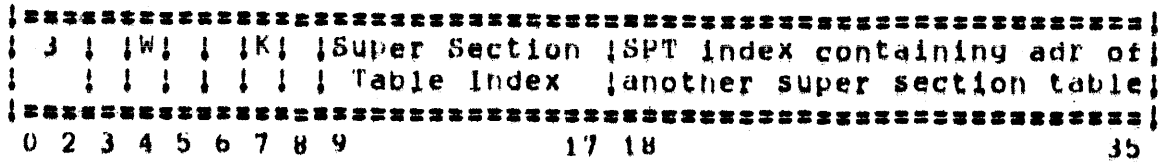
Shared



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

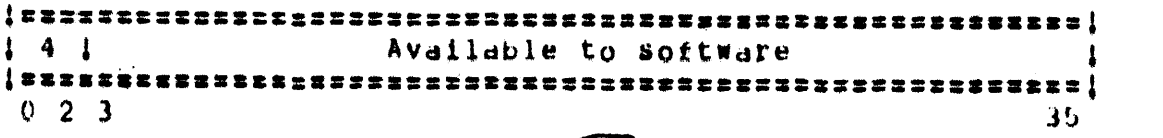
define?

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

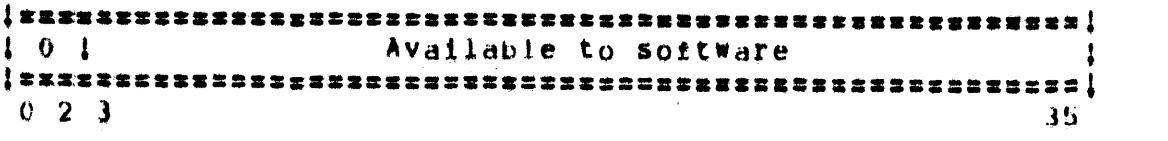


This type of pointer may (ONLY) appear in EPT/UPT offset 520 and indicates that KL compatible paging is to be used. If VMA<6:12> is zero, use VMA<13:17> as an index into the KL compatible section table starting at EPT/UPT offset 540 and perform the pointer evaluation exactly as a KL10 would. If VMA<6:12> 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.

8,4,2,2 Section Pointers

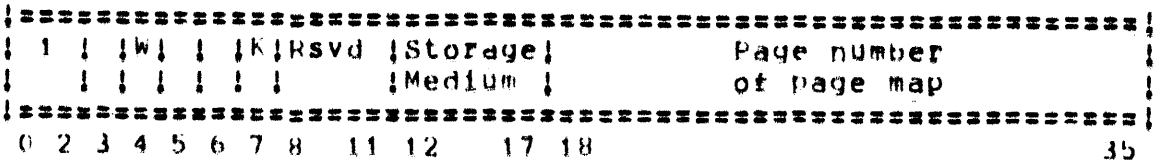
Entries in a section table are of the following four types. All other types are reserved and will cause a page fail if the microcode encounters them on a refill.

No access



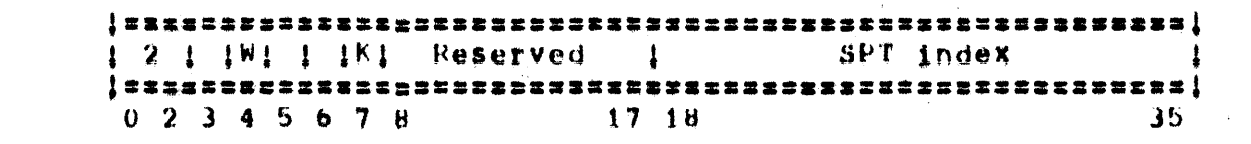
The section is inaccessible.

Immediate



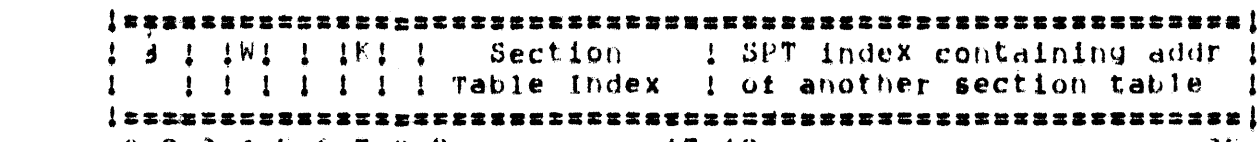
If bits 12-17 are zero, the page map is in the page specified by bits 18-35. Otherwise, the page is not in memory.

Shared



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

Indirect

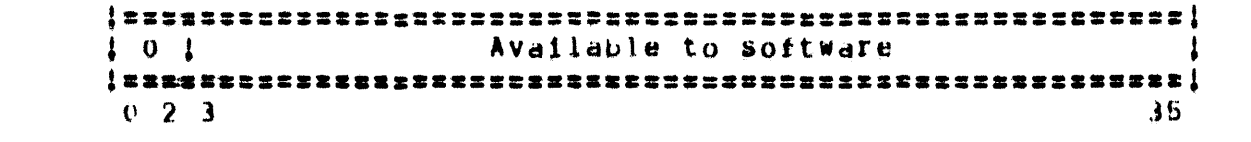


In the SPT offset specified by bits 18-35 is the page address of a secondary section table. The next section pointer to be evaluated is in that table at the offset specified by bits 9-17.

8,4,2,3 Map pointers

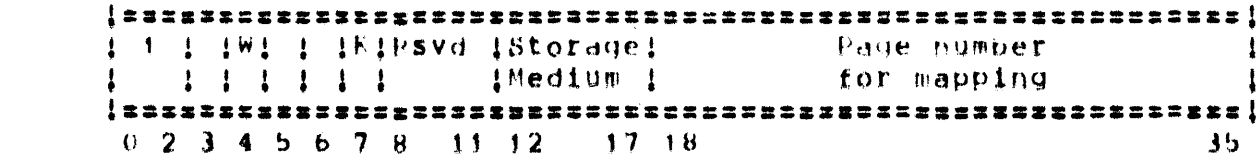
Entries in a page map are of these four types. All other types are reserved and will cause a page fail if the microcode encounters them on a refill.

No access



The page is inaccessible.

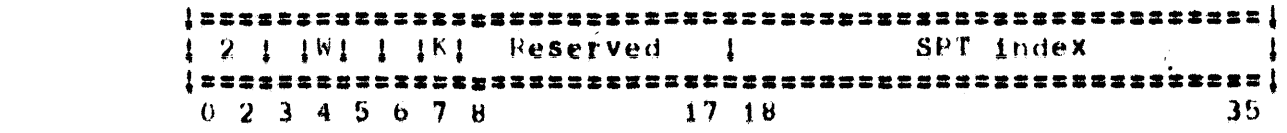
Immediate



If bits 12-17 are zero, the physical page specified by bits 18-35

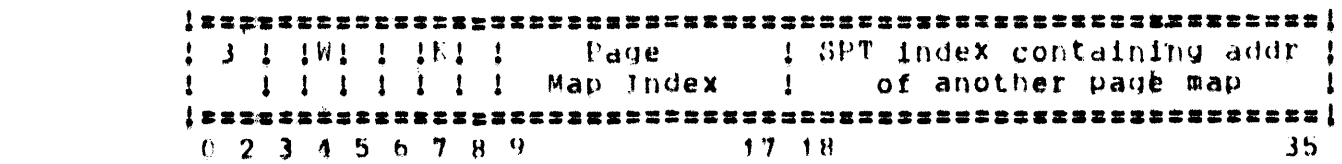
corresponds to the referenced virtual page. Otherwise, the page is not in memory.

Shared



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

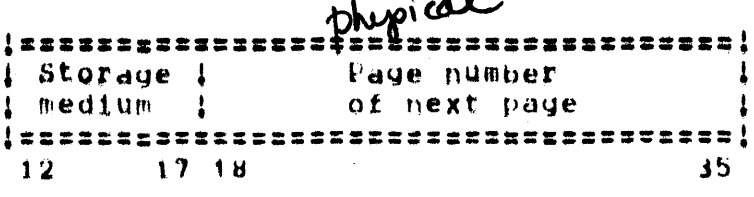
Indirect



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.

8.4.3 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.

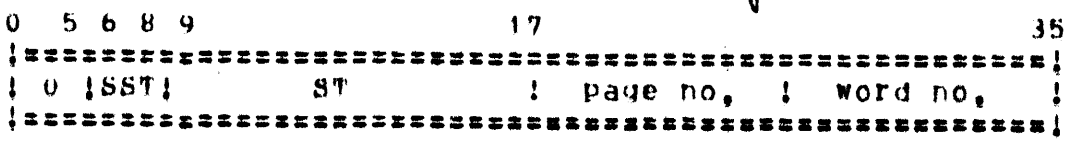
For indirect pointers, the microcode will actually encounter more than one page address word.

8.4.4 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:

what about "KL compatible"?



8.4.5 Page refill

8.4.5.1 CST updates

The EBOX microcode performs an operation called a "CST update" at several points during the processing of a page fault detected by the MBOX 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:

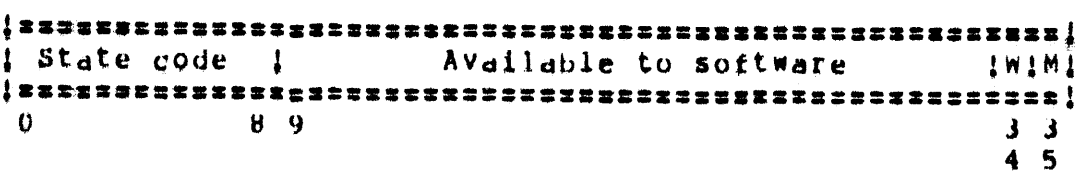
1. A page fault caused by a write reference to a page that is writable but not yet modified. This case sets the modified bit in the entry and writes it back into the CST.

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.

8,4,5,2 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 EBOX 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 EBOX page fail trap.

8.4.5.3 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).

8.4.5.4 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.

8.4.5.5 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.
Modified	1 if the physical page corresponding to this mapping has already been modified according to the CST entry for that page.
Writable	1 if the logical and of the W pointer use bits of all pointers evaluated was a 1.
Keep	1 if the logical and of the K pointer use bits of all pointers evaluated was a 1.
CST update	Always set to a 0

8.4.5.6 Write references

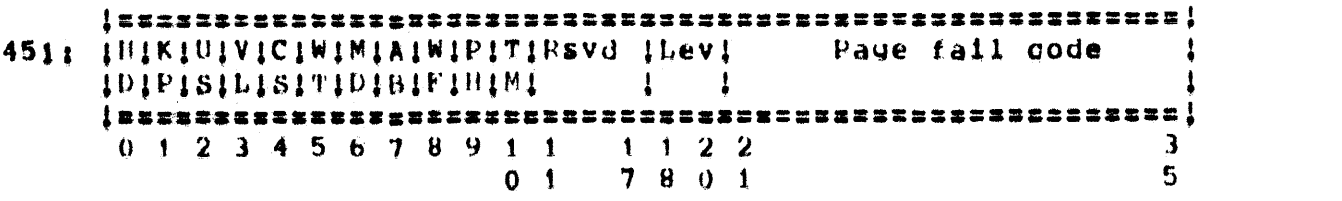
When a virtual write reference is made to the MBOX, the result is a function of the translation buffer entry corresponding to the virtual address specified by the EBOX. Write references are particularly interesting because they can succeed or fail based on the state of the writable and modified bits in the translation buffer. The relationship between write references and the four possible combinations of the writable and modified bits is as follows:

Writable	Modified	Effect
0	0	The page is not writable. A write failure page fail trap will be given to the monitor.
0	1	The page is not writable. A write failure page fail trap will be given to the monitor.
1	0	The page is writable but not yet modified. The EBOX microcode will get a page fail trap, update the CST entry for the page to set the M bit, set the modified bit in the translation buffer, and retry the reference. Note that the EBOX microcode can give an illegal age page fail trap to the monitor if the CST age for the referenced page is illegal.
1	1	The write will succeed.

8.4.6 Page fail conditions and formats

A page failure occurs when the pager is unable to make a desired memory reference, the EBOX detects an illegal condition while executing an instruction (e.g., incorrectly formated indirect word, illegal one-word-global byte pointer, etc.), or the MBOX detects a hardware failure while processing a memory request. When such a condition occurs, the EBOX 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:



The definition of each field is as follows:

0 This page fail was caused by a "hard" error. This does not necessarily mean that a hardware failure occurred. If this bit is set, bits 1-4 contain a code that describes the failure. The EBUX microcode copies the code to bits 27-35 and the valid codes are described below.

1 This bit gives the state of the translation buffer "keep" state bit for a page fail that resulted from a virtual translation failure.

2 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.

3 This bit gives the state of the translation buffer "valid" state bit for a page fail that resulted from a virtual translation failure.

4 This bit gives the state of the translation buffer "CST update needed" state bit for a page fail that resulted from a virtual translation failure.

5 This bit gives the state of the translation buffer "writable" state bit for a page fail that resulted from a virtual translation failure.

6 This bit gives the state of the translation buffer "modified" state bit for a page fail that resulted from a virtual translation failure.

7 If this bit is a 1, the memory reference caused an address break match.

8 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. This bit is valid only for a page fail that resulted from a virtual reference.

9 If this bit is a 1, the memory request was a physical reference. If the bit is a 0, the memory request was a virtual reference.

10 If this bit is a 1, there was no valid translation buffer mapping for the virtual address in the request.

11-17 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.

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 (hard), 2 (user), 9 (physical reference), 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. There are two types of codes that are returned in this field, depending on the state of bit 0. If bit 0 is a zero, the page fail and code are the result of one of the following conditions:

1. There was no valid translation for the reference address.

2. A write reference failed because the page wasn't writable.

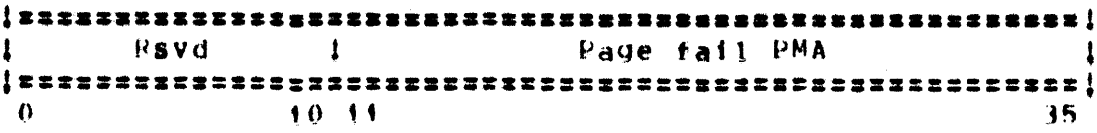
3. An address break occurred.

4. The EBOX detected an illegal condition while executing an instruction.

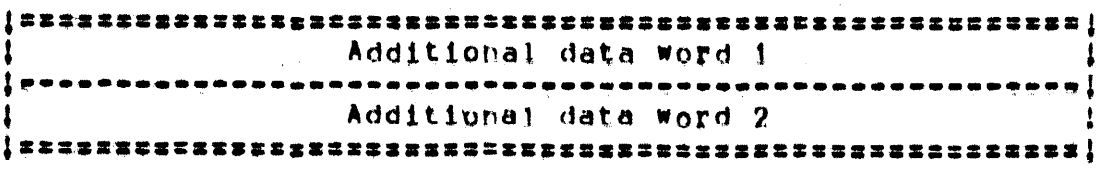
If bit 0 is a one, the page fail and code are the result of a "hard" error. Each case is described separately in the section on page fail codes.

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.

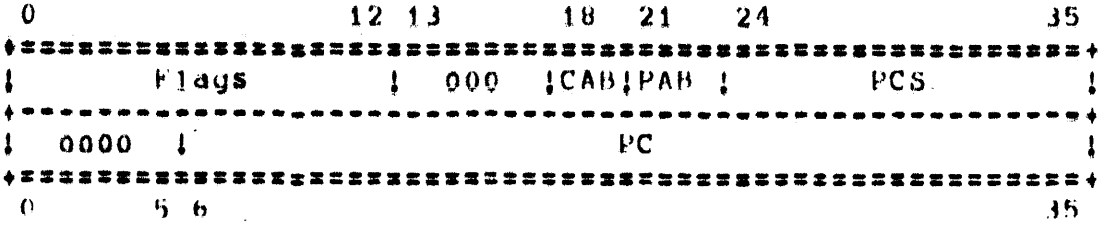
UPT location 453 contains the physical memory address (if any) for the request that page failed. It is only valid for those page fail conditions that have a valid PMA. The table at the end of this section describes under which page fail conditions it is valid.



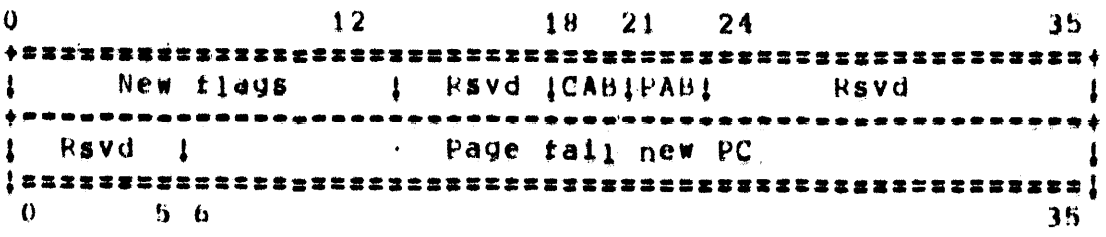
UPT locations 454 and 455 contain additional data that is different for each type of page fail. The contents of these words are given for each page fail at the end of this section. The format of these words is as follows:



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:



8.4.6.1 Tops-20 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 EBOX 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.

If bit 0 is off in the page fail word (indicating that this page fail is not the result of a "hard" error), the codes that may appear in bits 21-35 of the page fail word are as follows:

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.

PAD 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.

PAD 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.

PAD 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.

PAD 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.

PAD Address of word containing the illegal indirect,

PMA Undefined,

AD1 The illegal indirect word,

AD2 Undefined,

14 Illegal PXCT - A PXCTed instruction that stores into the ACs was executed with CAB = PAB,

PAD Undefined,

PMA Undefined,

AD1 Undefined,

AD2 Undefined,

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

PAD Address of illegal physical effective address word,

PMA Undefined,

AD1 The illegal physical effective address word,

AD2 Undefined,

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

PAD Address of the illegal one-word-global byte pointer,

PMA Undefined,

AD1 Undefined,

AD2 The illegal one-word-global byte pointer,

17 Illegal interrupt vector - An illegal interrupt vector (all zeros) was discovered (I/O page fail only),

PAD Address of the illegal interrupt vector,

PMA Undefined,

AD1 Undefined,

AD2 Undefined,

20 Illegal PUSHM function code. A function code of 2 was discovered during the processing of the PUSHM instruction.

RAD Stack address.

PMA Undefined.

AD1 Illegal PUSHM argument.

AD2 Undefined.

21 Illegal POPM function code. A function code of 2 was discovered during the processing of the POPM instruction.

RAD Stack address.

PMA Undefined.

AD1 Undefined.

AD2 Undefined.

If bit 0 is on in the page fail word (indicating that this page fail is the result of a "hard" error), the codes that may appear in bits 21-35 of the page fail word are as follows:

To be supplied

8.4.6.1.1 Additional data words for a pointer trace

When the EBOX 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".

8.5 Address Break

The address break feature of the hardware implements a superset of the KL10 address break capability. It may be used to determine whether a program is reading, writing, or fetching instructions from a range of locations in either user or executive address space and in either virtual or physical memory. The address break feature may also be used to determine if a port is reading or writing a range of locations in physical memory.

The address break enable and break conditions may be set by the WRCTX instruction and read by the RDCTX instruction. A description of the address break related fields in the WRCTX instruction follows:

The first word of the effective address (E) of the WRCTX instruction controls the action of the instruction. The bits in this word that affect address break are:

7 Inhibit all address break conditions for the next instruction executed. The effect of setting this bit is to set the inhibit address break PC flag for the next instruction. The intended use of this bit is to allow the instruction sequence:

```
WRCTX   ADR1        ;Turn on address break
XJRSTF  ADR2        ;Dismiss page fault
```

to be placed at the end of the monitor page fail routine. If the address break conditions are such that the hardware is breaking on all monitor instruction fetches, this bit allows the monitor to execute the XJRSTF to dismiss the address break page fault. It is assumed that the PC flags that are the argument to the XJRSTF will also contain the inhibit address break bit to allow the monitor to execute the instruction that caused the original address break page fault.

8 Load address break conditions from the words at E+2 through E+4. If this bit is on, the address break qualifiers are loaded from word E+2, lower bound break address is loaded from E+3 and the upper bound break address is loaded from E+4. If this bit is off, the address break conditions remain unchanged.

CAUTION

Paging must be enabled (with WRPUR bit 4) to load the address break conditions. If paging is not enabled, the result of loading the address break conditions is undefined.

9 Load address break enable from bit 10. If this bit is on, address break is turned on or off based on the state of bit 10. If this bit is off, the state of address break enable remains unchanged.

10 Enable/disable address break. If both bit 9 and this bit are on, turn on address break. If bit 9 is on and this bit is off, turn off address break. If bit 9 is off, the state of this bit is ignored.

The third word of the effective address (E+2) of the WRCTX instruction defines the conditions that determine when an address break will occur. The condition bits are as follows:

10 If this bit is on, enable address break for a normal fetch of an instruction in the program under control of PC.

11 If this bit is on, enable address break for any reference that reads except the normal fetch of an instruction. This includes retrieval of operands, address words in an effective address calculation, or an instruction to be executed by an XCT.

12 If this bit is on, enable address break for any reference that writes to memory.

13 If this bit is on, enable address break for a reference made in user virtual address space. If this bit is off, enable address break for a reference made in executive space (either virtual or physical depending on the state of bit 16).

14 If this bit is on, enable address break for any reference made from the CPU, i.e., from the IBOX or EBOX.

15 If this bit is on, enable address break for any reference made by a port, i.e., from the IOBOX. No address break page fail is generated for an address break that occurs as the result of a port reference. Instead, the MBOX completes the request normally (i.e., the read or write succeeds) and notifies the port that the request caused an address break. The port sets bit 12 in the port status register to indicate to the monitor that an address break occurred as the result of the transfer.

CAUTION

Due to the implementation of this feature, the monitor cannot be assured that the transfer completed without errors if an address break occurs. Therefore, the monitor must retry the transfer with port address break disabled.

16 If this bit is on, enable address break for a physical memory reference. If this bit is off, enable address break for a virtual memory reference. Note that the break addresses must be physical if this bit is on and virtual if this bit is off.

17 If this bit is on, compare only the low order 18 bits of the reference address with the address range when doing address compares. This allows the program to cause an address break on an address in any section.

There are certain combinations of the above bits that produce unspecified results. These combinations are as follows:

If bit 10 is on, then bit 15 must be off because ports never fetch instructions.

If bit 10 is on, then bit 16 must be off because instruction fetches are always done from virtual memory.

If bit 13 is on, then bit 16 must be off because user references are always done through virtual space.

If bit 15 is on, then bit 16 must be on because ports always make physical references.

The fourth and fifth words of the effective address (E+3 and E+4) of the WRCTX instruction specify the lower and upper bound break addresses. When doing address break compares, the MBOX compares the reference address with the upper and lower bound break addresses. Normally, the full reference address is used in the compares (i.e., 30 bits of virtual address for virtual compares and 25 bits of physical address for physical compares). However, if bit 17 is on in the third word of the WRCTX argument block, only bits 18 through 35 of the reference address are used in the compares. If the reference address is greater than or equal to the lower bound break address and less than or equal to the upper bound break address, the address compare succeeds.

The conditions under which an address break will occur in the MBOX may be described as follows:

Let:

A := Condition bit 10 on and an instruction fetch reference.
B := Condition bit 11 on and a read reference.
C := Condition bit 12 on and a write reference.
D := Condition bit 13 on and a user reference, or condition bit 13 off and a executive reference.
E := Condition bit 14 on and a reference made by the CPU.
F := Condition bit 15 on and a reference made by a port.

G := Condition bit 16 on and a physical reference, or condition bit 16 off and a virtual reference,
H := Condition bit 17 is off and the reference address is within the range described by the lower and upper break addresses or condition bit 17 is on and bits 18 through 35 or the reference address is within the range described by the lower and upper break addresses,

Then a port address break will occur if the following expression is true:

(A OR B OR C) AND (F) AND (H)

and a CPU address break page fail will occur if the following expression is true:

(A OR B OR C) AND (E) AND (D) AND (G) AND (H)

The expressions are given separately for CPU and port references because that is the way they are implemented in the MBOX hardware,

If an address break page fault does occur, the microcode will turn off address break before dispatching to the monitor page fault handler, It is the monitor's responsibility to turn address break back on before dismissing the page fault if the page fault was the result of an address break, The microcode WILL NOT turn off address break for any page fault except an address break page fault, This allows the monitor to trace executive instruction fetches by setting the address break conditions to cause a page fail for each instruction fetched from executive virtual space,

CAUTION

If address break is enabled for a range of memory addresses, an instruction that references multiple words in this range will only cause an address break condition for the first word referenced if the monitor restarts the instruction with the "inhibit address break" PC flag set, This is because the "inhibit address break" PC flag remains set for the completion of execution of the instruction and blocks further address breaks,

CHAPTER 9

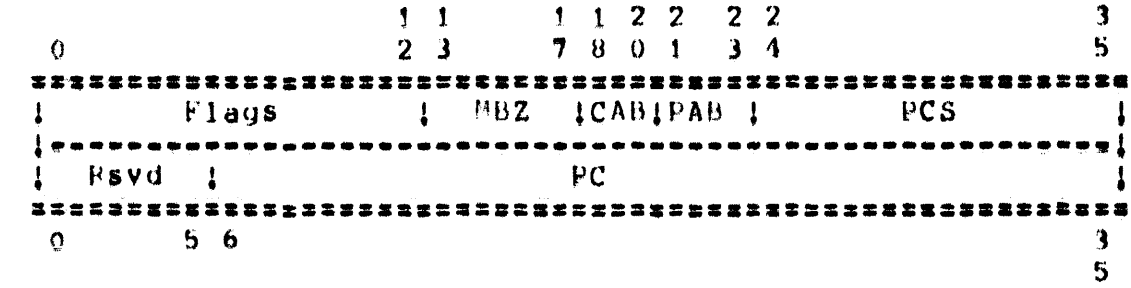
PROCESS CONTEXT VARIABLES

9.1 Introduction

In order to take advantage of the full 4096 section virtual address space implemented by the KC10 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.

9.1.1 New flag+PC double word

The format of the double word is as follows:



Where:

Flags — PC flags. The action of these flags is the same as for the KL10, unless stated otherwise. The flags are as follows:

0 — Overflow. On the KL10 in exec mode, this flag was used as previous context public. On the KC10, it is overflow in both user and exec modes.

1 — Carry 0.

2 — Carry 1.

3 Floating Overflow,

4 First Part Done, This PC flag is used by the microcode as necessary to restart a multi-part instruction, It does not necessarily act the same as any previous machine and the use may change at any time without notice, The monitor should save and restore this flag when changing contexts, The user should never touch it,

5 User,

6 User In-out/Previous Context User,

7 Unused by the KC10 hardware and microcode, On previous machines, this was the Public flag, The KC10 always stores it as zero, and ignores any attempt to set it,

8 Address Failure Inhibit,

9 Trap 2,

10 Trap 1,

11 Floating Underflow,

12 No Divide,

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),

9.1.2 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. See the chapter on Special System Pagers for a description of the changes to the EPT and UPT.

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
	KC	Store	Store	Store	Store	Store
		Load	Load	Load	No	No
Inter-rupt	KL	Store	Store	No	No	Store
		** Load	Load	No	No	Set(PCS)
	KC	Store	Store	Store	Store	Store
		Load	Load	Load	No	No
MUUO	KL	Store	Store	* Store	* Store	Store
		Clear	Load	No	No	Set (PCS)
	KC	Store	Store	Store	Store	Store
		Load	Load	Load	Load	Set
Page Fail	KL	Store	Store	No	No	Store
		Clear	Load	No	No	Set(PCS)
	KC	Store	Store	Store	Store	Store
		Load	Load	Load	Load	Set
LUUO	KL	Store	Store	No	No	No
		No	Load	No	No	No
	KC	Store	Store	No	No	No
		No	Load	No	No	No
XJRSTF	KL	No	No	No	No	No
		Load	Load	No	No	Load
XJEN	KC	No	No	No	No	No
		Load	Load	Load	Load	Load

CHAPTER 10

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.

10.1 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.

10.2 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 0-17 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.

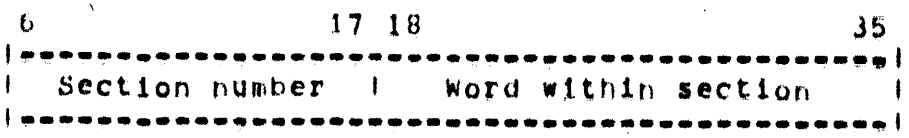
To insure compatibility with programs written for non-extended processors, section zero is treated exactly as it is on non-extended processors. This means that if a program is executing in section zero, nearly all instructions behave exactly as they would if the program were executed on a non-extended 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.

10.3 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

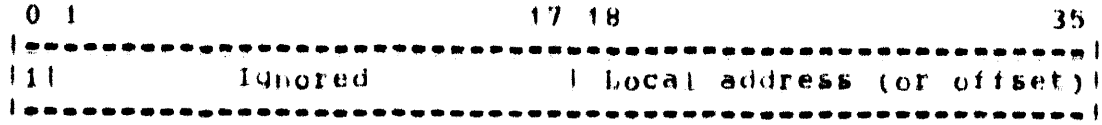
  PC has the same format as a virtual address.

- 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.

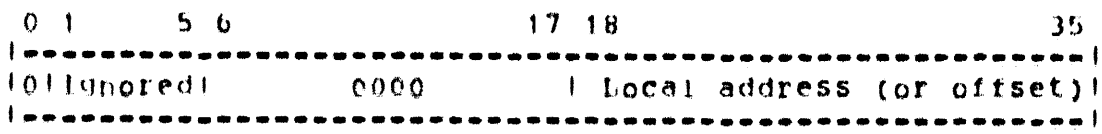
- A **local address** is an 18-bit in-section address that, when combined with a default section number, specifies a full 30-bit address. The section number is supplied by something other than the address word or index register.

- 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:

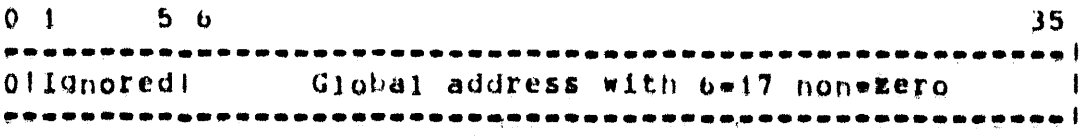


Local index format (bit 0 = 1)



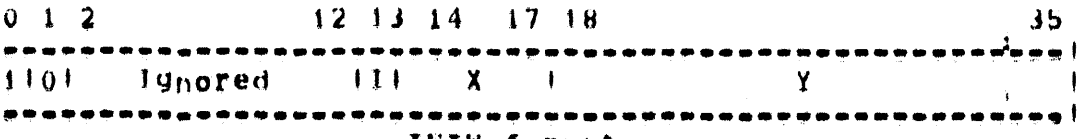
Local index format (bits 6-17 = 0)

- 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:



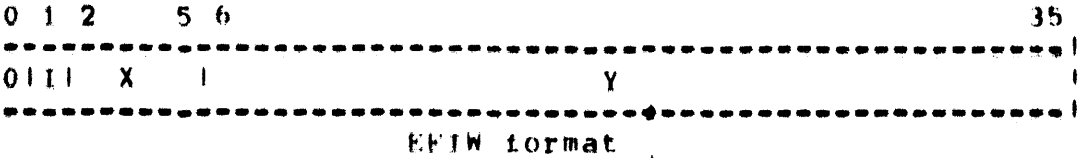
Global index format

- 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:

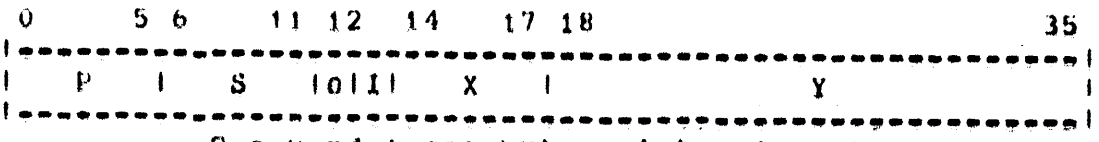


EFIW format

- 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:

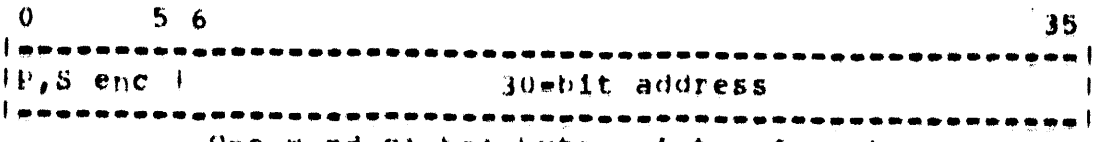
Illegal indirect word format

- A **one-word local byte pointer** is any byte pointer in section zero, or a byte pointer in a non-zero section 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:



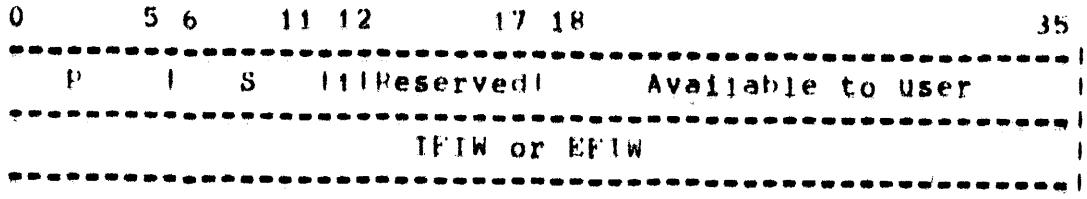
One-word local byte pointer format

- A **one-word global byte pointer** is any byte pointer in a non-zero section 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.

10.4 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).

10.4.1 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.

10.4.1.1 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.

10.4.1.2 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.

10.4.1.3 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.

10.4.1.4 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.

10.4.1.5 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.

10.4.1.6 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	Address Word Type: 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

10.4.2 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.**

10.4.3 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.

10.5 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 0-17. 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.

10.5.1 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 AC's 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.

10.5.2 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.

10.6 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.

10.7 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.

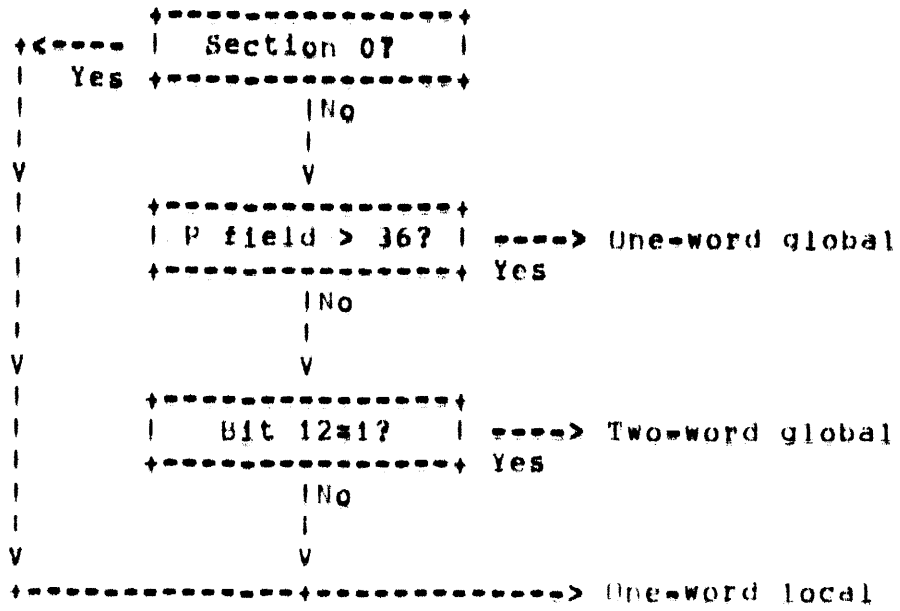
Examples in this section sometimes use the POINT pseudo-op to describe a byte pointer. For those readers who do not know what this pseudo-op generates, a description can be found in the Macro manual.

10.7.1 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.

10.7.1.1 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 byte pointer (more precisely, the first word of the byte pointer if it is a two-word global) 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 global byte pointers is done based on the section from which the byte pointer was fetched. That is, if the section from which the byte pointer was fetched is non-zero, the byte pointer may be global.

10.7.1.2 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.

10.7.2 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.

10.7.2.1 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 global byte pointers is done based on PC section. That is, if PC section is non-zero, the byte pointers may be global.

10.7.2.2 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.

10.7.2.3 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 EPIW 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.**

10.7.2.4 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.

10.7.3 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.

10.7.4 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.

10.7.5 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 some of the non-zero section rules are ignored.

In the case of JSA, the effective address is always treated as a local address in PC section, even if the EA-calc results in a global address in another section. In addition, only the in-section parts of E and PC are stored in the halves of AC. For example,

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

stores the current contents of AC into location 2,,200 (PC section plus the in-section part of EA), stores 200 (in-section part of E) in AC left, 101 (in-section part of PC+1) in AC right, and jumps to 2,,201.

Similarly, the effective address for JRA is always treated as a local address in PC section. In addition, the AC is restored from the local address in PC section contained in AC left. 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).

This behavior is consistent with the operation of the instructions in section zero.

- JSA and JRA always compute an effective address local to PC section even if the EA-calc generates a 30-bit address outside of PC section.

10.7.6 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).

10.7.7 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.

10.7.8 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.

10.7.9 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.

10.7.10 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 EPIW 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.

10.7.11 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.

10.7.11.1 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.**

10.7.11.2 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.

10.7.11.3 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.

10.7.11.4 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,,3001
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.

10.7.11.5 Generalizations for XCT

The examples above cover specitic 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.

10.8 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

10.9 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.

10.10 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.

10.10.1 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.

10,10,2 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.

10.10.3 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.

10.10.4 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.

10.10.5 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.

10.11 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.

10.11.1 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.

10.11.2 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.

10.11.3 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.

10.11.4 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.

10.11.5 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.

10.11.6 Modifications to the EA-calc algorithm

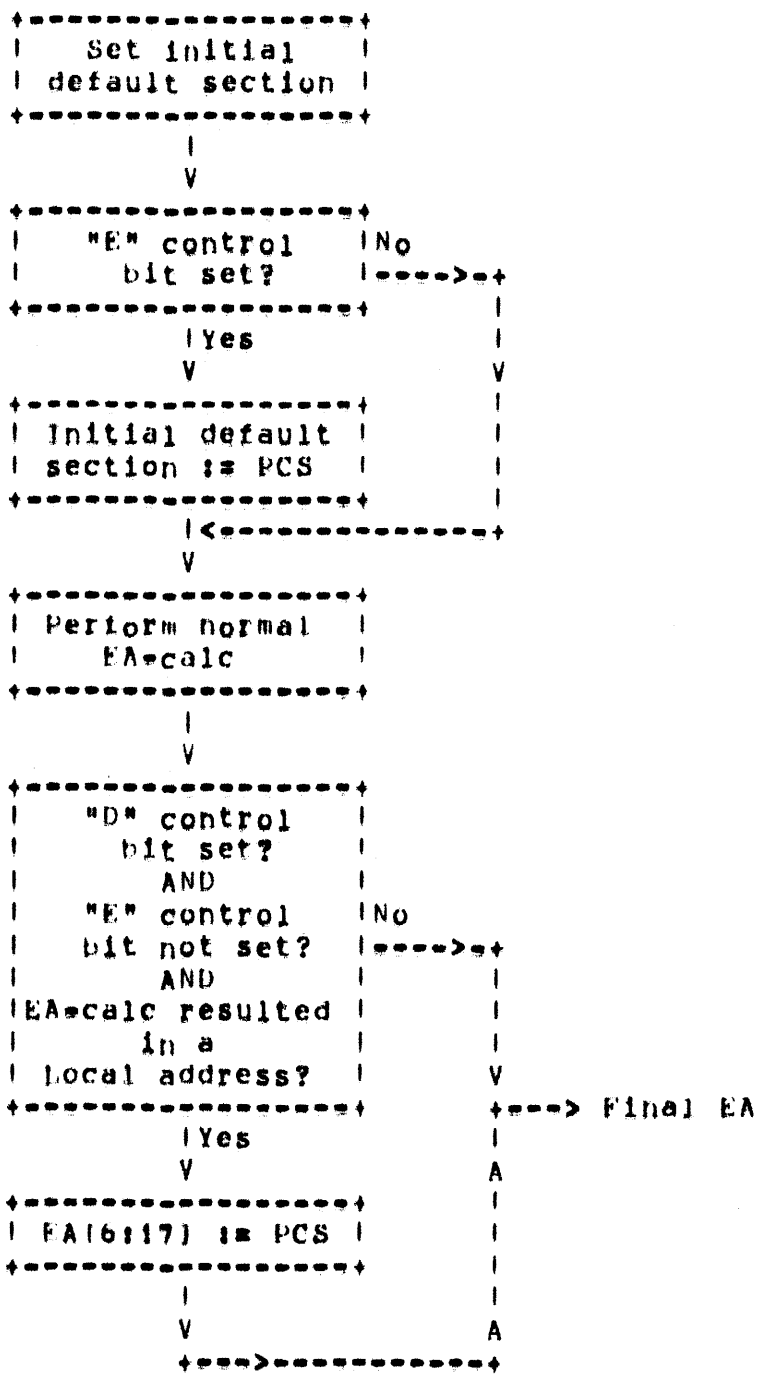
The appropriate "E" and "D" control bits from the AC field of the PXCT instruction are used to modify an EA-calc done on the executed instruction or a subsequent EA-calc done by the instruction (e.g., byte pointer). This modification involves pre- and post-processing the normal effective address calculation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

10,11,7 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.

10,11,7,1 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.

10,11,7,2 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.

10.11.7.3 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 11

SYSTEM TIMERS

11.1 Summary

The KC10 processor implements several kinds of system timers using a combination of hardware and microcode assistance. There are three kinds of timers implemented in the basic CPU, as follows:

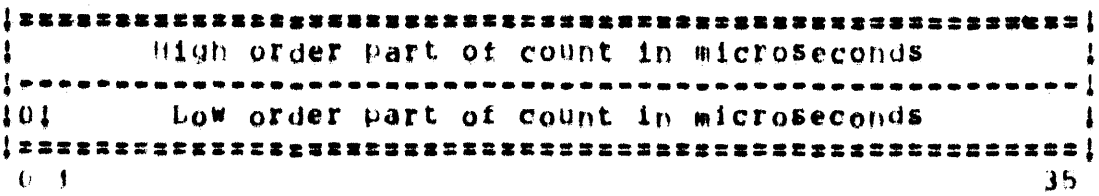
1. Time base clock
2. Interval timer
3. User runtime meter

In addition, the console contains a battery backed-up time-of-year clock that can be used to maintain the correct time through a power failure.

Unlike the KL10, the clocks on this machine will never update locations in memory unless requested to do so by the appropriate instruction in the monitor.

11.2 Time clocks

The time base and the user runtime meter are returned as a double precision integer with units of 1 microsecond. Both have the following format:



11.2.1 Time Base

The time base is implemented as a 72-bit (two word) register in internal EBOX storage, and a 16 bit hardware counter which counts in 1 microsecond units. The hardware counter requests a microcode interrupt approximately every millisecond and the microcode reads and zeros the counter and updates the 72-bit register with the accumulated count. The 72-bit register is also updated from the hardware counter when a RDTIME instruction is done.

The time base is controlled by the WRTMB and RDTMB instructions and may be read with the RDTIME instruction.

The counter will overflow every 7,47 E7 years.

11.2.2 User Runtime Meter

The user runtime meter is implemented in a manner similar to that described for the time base above. In fact, the same 16-bit hardware counter is used for both the user runtime meter and for the time base. As with the time base, the user runtime meter is a 72-bit register kept in internal EBOX storage.

If the user runtime meter is enabled, the count is maintained in the EBOX registers. When a context switch occurs as the result of a WRCTX instruction that changes the UBP, the 72-bit register is updated from the hardware counter, and the result is written into UPT locations 504 and 505 of the previous user context. The new value of the user runtime meter is then loaded from UPT locations 504 and 505 of the new user context. This update process can be inhibited via a bit in the WRCTX argument.

If it is turned on, the user runtime meter counts during the time that the processor is in user mode. It can also be enabled to count during exec PI time and exec non-PI time. That is, it can be made to count when the processor is in exec mode processing an interrupt, or when the processor is in exec mode doing something other than processing an interrupt (e.g., page fail or MUUO processing).

The user runtime meter is controlled by the WRACT and RDACT instructions and can be read with the RDURTM instruction.

11.3 Interval Timer

The interval timer is used to supply a source of interrupts with programmable periods. It is implemented as a 12 bit hardware counter (different from the time base) that counts in 10 microsecond units. It can therefore count and cause interrupts of any interval from 10us to 40,95ms.

When it is enabled, the interval timer counts up until the requested period is reached, sets interval done, and requests an interrupt on the PI channel assigned to it. The count is then automatically reset to zero. If no interval period has been set, the interval timer can overflow. This event sets interval done, and requests an interrupt. As before, the count is automatically reset to zero.

The interval timer is controlled by the WRTMB, RDTMB, WRINT, and RDINT instructions.

CHAPTER 12

TRAP, UUO AND INTERRUPT HANDLING

12.1 Introduction

This chapter discusses the KC10 implementation of trap, MUUO, LUUO, interrupt, and I/O page failure handling.

Trap handling has been changed considerably from the KL10 in that traps on the KC10 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.

MUUO handling is similar to the KL10, but the microcode provides much more information to the monitor as part of the MUUO processing. The major change is in the fact that there are separate new-PC words for different categories of MUUOs.

LUUO processing is done in the same manner as on the KL10 and is included in this specification for completeness.

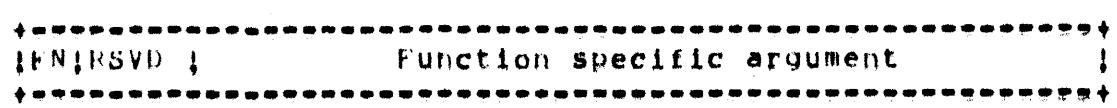
Interrupt processing is done via interrupt vectors in the IO page that specify the address of XPCW-like blocks through which the interrupt is to be processed.

I/O page failure processing is done through a block in the I/O page. The KC10 provides significantly more information to the monitor than the KL10 did.

12.2 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.

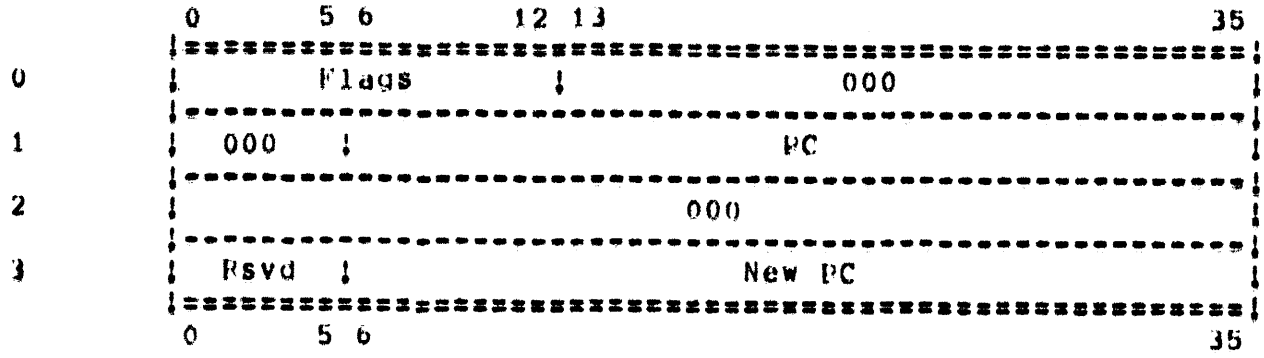
  This function stores only the program flags, CAB, PAB, PCS and the PC in UPT locations 424-425. The opcode, AC, and effective address of the instruction are NOT stored in UPT locations 426-427. The new program flags, CAB, and PAB are loaded from UPT location 430 as in a normal MUUO.
- 2 Virtual address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

12.3 Virtual Machine Simulation Mode

The virtual machine simulation mode (VM mode) implemented by the KC10 allows an operating system to run a program in user mode in such a way that the program cannot distinguish its environment from a stand-alone exec mode machine. The primary use for this mode is to allow a monitor to be tested and/or debugged on a timesharing machine in user mode by concealing the fact that it is indeed running in user mode.

This mode is enabled for a user process with bit 9 of word E of WPCTX. In order to do this, the EBOX microcode must generate an MUUO trap for any instruction that differs between exec and user mode. It is then up to the (real) monitor to simulate the instruction properly to conceal the fact that the program is really running in user mode.

If VM mode is enabled for a user process, the EBOX microcode will generate an MUUO trap through the VM mode new PC word (location 431) in the UPT. There are four classes of instructions that trap through the VM mode new PC word as follows:

1. Any instruction that would normally trap as an MUUO through one of the other MUUO new PC pairs. This includes all unassigned opcodes, all legal MUUOs, all undefined EXTEND opcodes, JSYS, I/O instructions, MAP, JRST 3, HALT, XJEN, XPCW, JRST 10, JRST 11, JEN, JRST 13, JRST 16, and JRST 17. This class of instructions is included because the new PC word for MUUOs is taken from different UPT locations based on whether the MUUO was executed in user or exec mode.

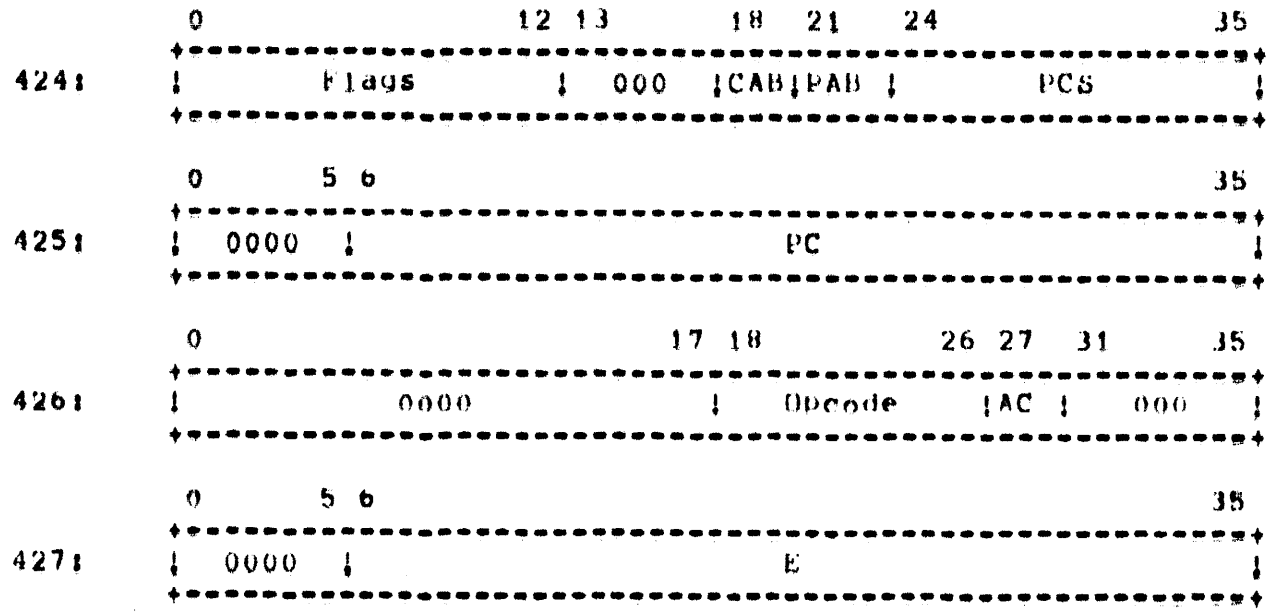
2. XCT with non-zero AC. This class is included because XCT with a non-zero AC in exec mode specifies a PXCT.

3. All LUUOs. This class is included because LUUOs use blocks in either the EPT or UPT based on whether the LUUO was executed in user or exec mode.

4. PUSHJ, JSR, and JSP in section 0. This class is included because the specified instructions store the flags (with the user-mode bit) if they are executed in section 0.

12.4 MUUO handling

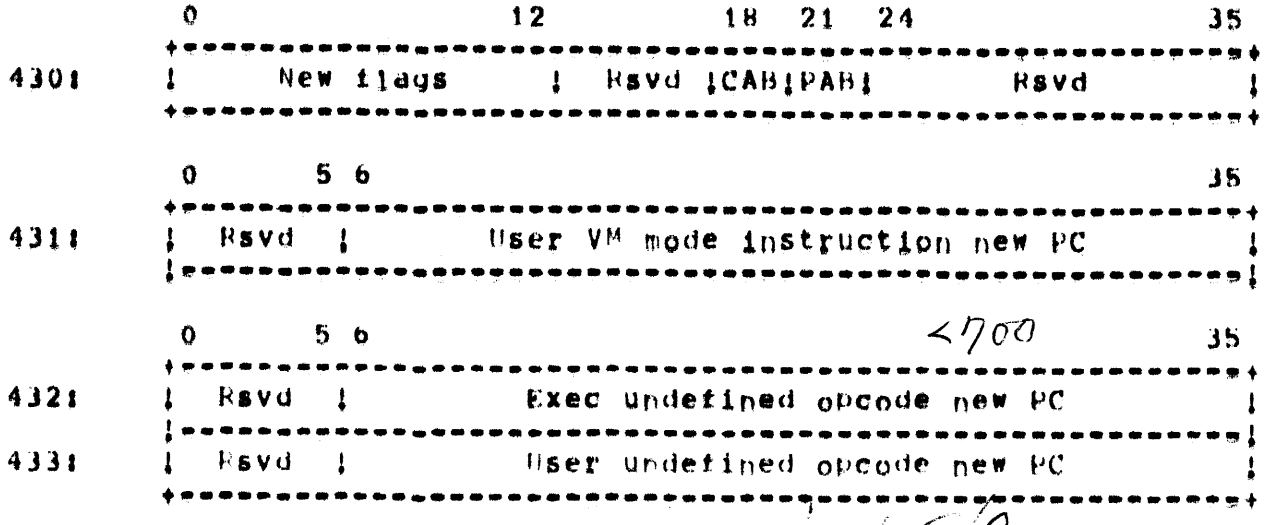
MUUO handling on the KC10 is significantly different from that of any previous processor. Instead of the previous format of UPT locations 424-427, the following format is used to store the program flags, CAB, PAB, PCS, PC, Opcode, AC, and effective address of the MUUO:



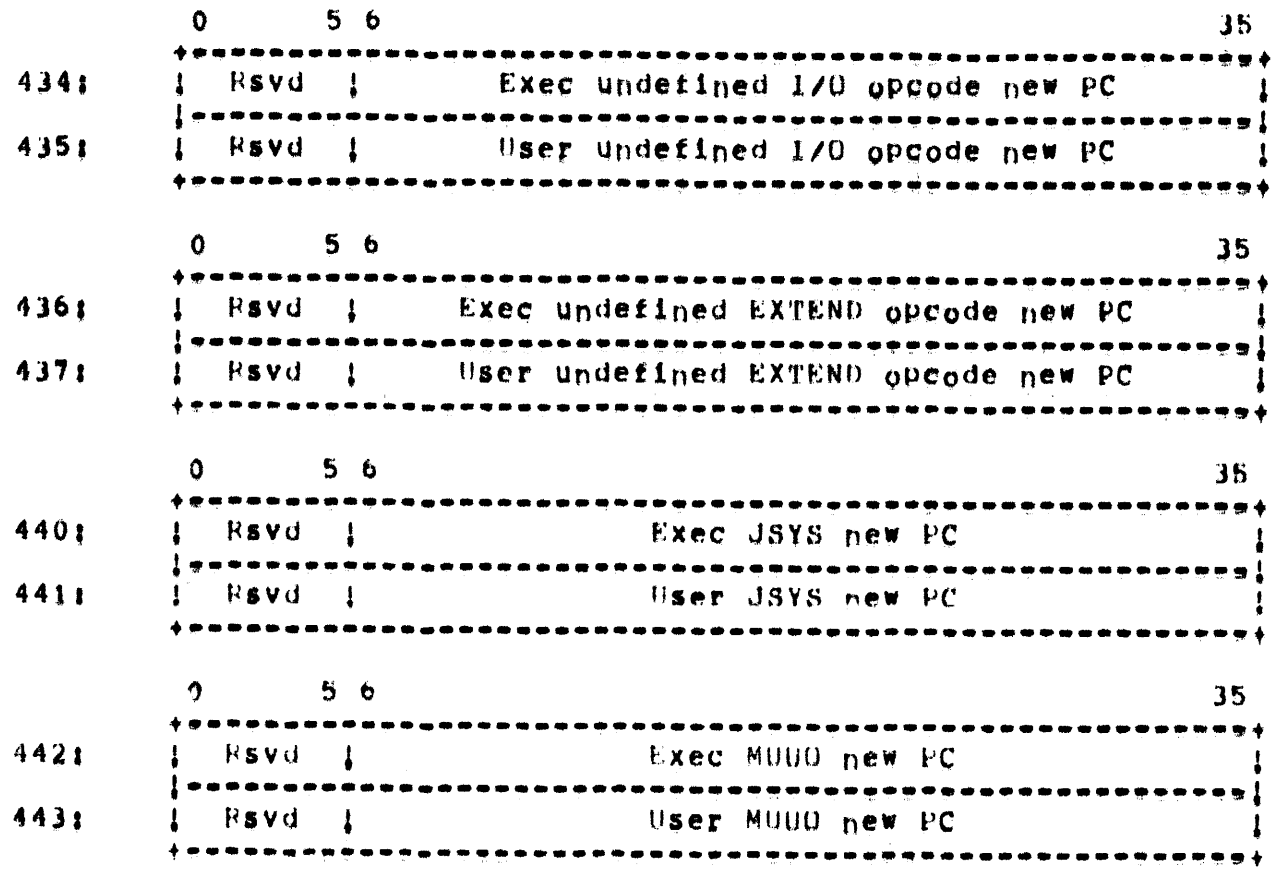
The new current and previous AC blocks, and the new program flags are loaded from the word at UPT location 430. The new PC is taken from one of the words of the dispatch vector beginning at UPT location 431, based on the MUUO opcode and whether the MUUO was executed in user or executive mode. The dispatch vector consists of pairs of words, one for user and one for exec, (location 431 is the exception to this rule) and contains 5 separate MUUO dispatches plus words reserved for future expansion. The dispatches are as follows:

Offset	Use
431	Instructions trapped in user mode as the result of virtual machine simulation mode enabled. See the discussion above.
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 what's left?

The format of these words is as follows:



not ~~-10~~ not extend JSYS

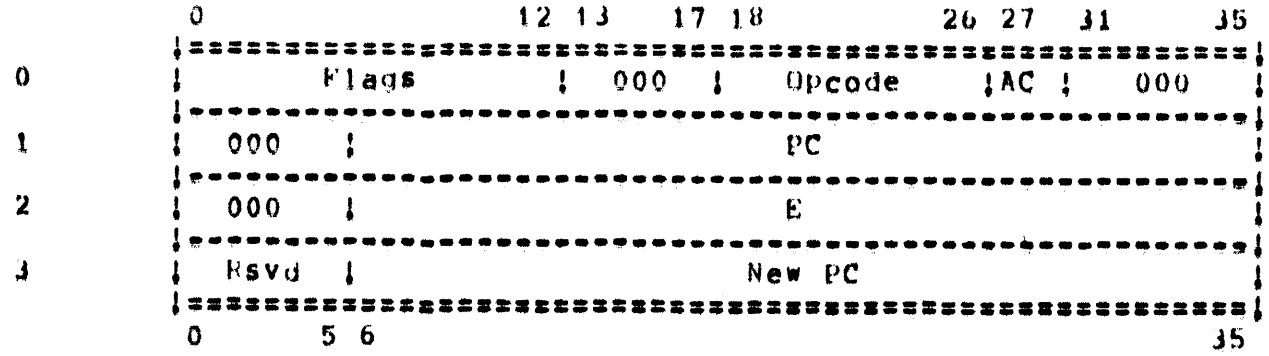

0 5 6 35
434: Rsvd Exec undefined I/O opcode new PC
435: Rsvd User undefined I/O opcode new PC
0 5 6 35
436: Rsvd Exec undefined EXTEND opcode new PC
437: Rsvd User undefined EXTEND opcode new PC
0 5 6 35
440: Rsvd Exec JSYS new PC
441: Rsvd User JSYS new PC
0 5 6 35
442: Rsvd Exec MUUO new PC
443: Rsvd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

12.5 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:



12.6 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.

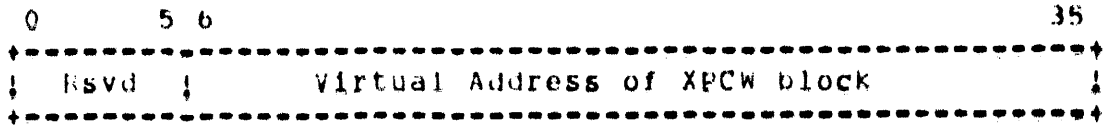
4. Page fails. Page fails that must be processed by the monitor will halt the machine. Page fails that can be resolved entirely by the EBOX microcode will continue to be processed normally.

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.

12.7 Interrupt vectors

All interrupts happen through interrupt vectors located in the I/O page. A 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 a preceding section. 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:

0-5 Reserved

6-35 Vector address of control block.

12.8 I/O page failure

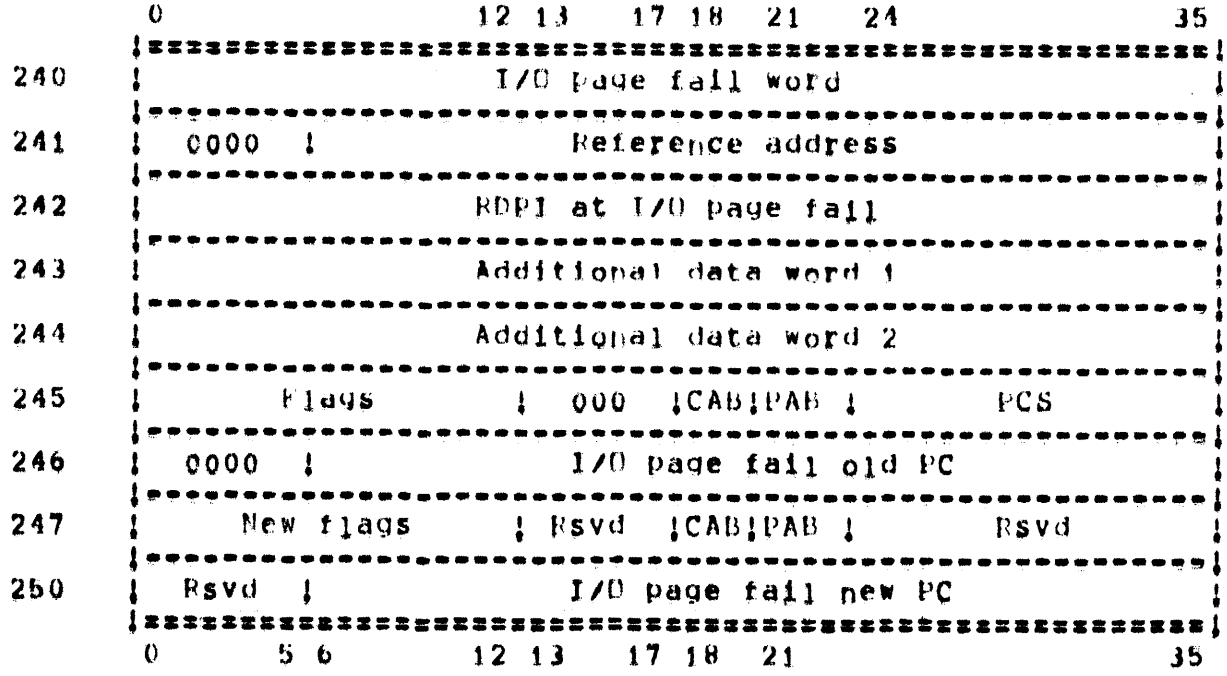
An I/O page fail can occur if the EBOX microcode is unable to fetch a word necessary to process an interrupt request. This condition can occur if a hardware error or address break page fault occurs while trying to read a port interrupt vector word (I/O page locations 210-217), a port interrupt PI status word (I/O page locations 220-227), or a software interrupt vector word (I/O page locations 231-237). It can also occur if a request to access one of the four words pointed to by an interrupt vector page fails. In this case, the EBOX microcode generates an I/O page failure.

This page fail will be similar to a normal page fail trap, but the page fail information is contained in I/O page locations 240-250 instead of in the UPT. The EBOX stores a page fail word, reference address, PI status at the time of the failure, and the additional data words (identical in format to those stored by a normal page fail) in locations 240-244. The old PC double word is stored in locations 245-246 and the new program flags, CAB, PAB, and PC will then be taken from I/O page locations 247-250 and the processor will resume execution at the PI level on which the failure occurred.

The format and contents of words 240-241 and 243-244 are identical in format to the words stored in UPT locations 450-451 and 453-454 for a normal page fail. The format of these words is described in the chapter on paging.

The PI status stored in word 242 is identical in format to that returned by a RDPI instruction.

The format of the I/O page fail locations in the I/O page is as follows:



12.9 Interrupt request protocol

When a port wants service from the CPU, it makes an interrupt request on the PI level assigned to it by the -10 program. This request is made using a protocol that is described here.

When a port wants interrupt service from the CPU, it does the following:

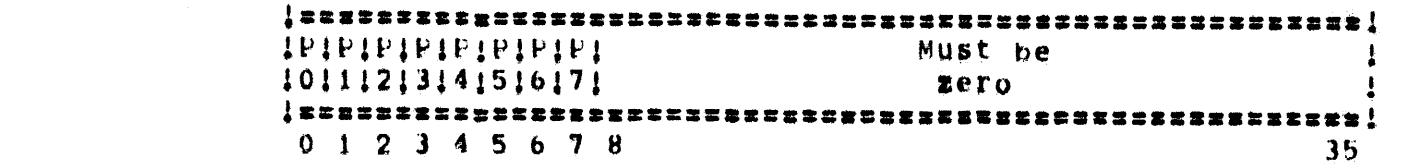
1. It writes the current contents of the Port Status Register into word 4 of the port's Register Access Block in the IO page.

2. It performs a read-interlock request on the Port PI Status Word in the IO page (words 220-227) corresponding to the PI level on which the interrupt request is being made. The Port PI Status Words are indexed by PI level, so a port making an interrupt request on PI level 1 would read-interlock word 221 of the IO page, a port making an interrupt request on PI

level 2 would read-interlock word 222, etc.

3. It sets the bit corresponding to the number of the port making the request and then write-releases the word back into the IO page.

The Port PI Status word has the following format:



where bit 0 corresponds to port 0, bit 1 corresponds to port 1, etc. It is important that the port set no other bits, including bits 8-35 which must remain zero.

4. The port then asserts the hardware interrupt request line for the PI level on which the interrupt request is being made. This line remains asserted until the program running in the -10 clears the condition causing the interrupt.

When the KC10 microcode decides to service the interrupt request, it does the following:

1. It reads from the IO page the Port PI Status Word corresponding to the PI level which is being serviced.

2. By looking at the bits set in the Port PI Status Word, the microcode can determine which ports are requesting interrupt service at this PI level. The microcode selects the port to be serviced and starts the interrupt sequence using the interrupt vector word corresponding to the port number from IO page locations 210-217. The Port Interrupt Vector words are indexed by port number, so an interrupt sequence for port 0 would use word 210, an interrupt sequence for port 1 would use word 211, etc.

3. The interrupt service routine in the -10 processes the interrupt request. When done, the interrupt service routine requests that the port clear the interrupt condition by issuing one or more register write commands through the Register Access Block for the port being serviced.

When the -10 requests that the port clear the interrupt condition, the port does the following:

1. It performs a read-interlock request on the Port PI Status Word in the IO page corresponding to the PI level on which the original interrupt request was made.

2. It deasserts the hardware interrupt request line for the PI level on which the original interrupt was made.

3. It clears the bit in the Port PI Status Word corresponding to the number of the port and then write-releases the word back into the IO page.

Note that the Port PI Status Words are written only by the ports and never by the KC10 microcode. The microcode uses the words only to determine which ports are requesting service at a particular PI level. The microcode does not clear the bit of the port when it grants the interrupt request.

CHAPTER 13

MISCELLANY

This chapter contains miscellaneous information about the KC10 that doesn't fit anywhere else.

13.1 Halt status codes

When the EBOX microcode halts the EBOX for some reason, it stores a halt status code that describes the reason for the halt. This code can be retrieved by the console and printed on the CTY when a halt occurs. Note that such a code is not stored on an EBOX halt that wasn't caused by the EBOX microcode. The halt status codes are as follows:

0 The processor executed a HALT (JRST 4,) instruction.

1 A non-zero section LUUO was executed and trap enable was off in the WREBR argument word.

2 An MUUO was executed and trap enable was off in the WREBR argument word.

3 A page fail that must be resolved by the monitor occurred and trap enable was off in the WREBR argument word.

4 An illegal destination address was generated from EBOX dispatch 67. Early decode bits 4-6 are probably incorrect for the instruction being executed.

5 A MOVSxx memory read that previously caused a page fault did not do so when the reference was retried.

6 A MOVSxx memory write page failed.

7 A page fault generated as the result of a physical memory reference didn't result in a monitor page fail trap. All page fails that can result from a physical reference should require monitor intervention, so the page fault or the page fail word was illegal.

10 An interrupt was requested on PI level 0.

11 The EBOX microcode was trapped to an unimplemented (777x) microtrap vector.

12 The determination of the reason for an IBOX trap to EBOX with EBOX dispatch 25 resulted in an illegal trap reason (no reason bits were indicated in the dispatch).

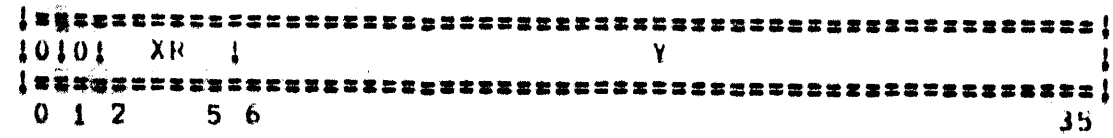
13 The EBOX microcode page fail handler attempted to decode the reason for the page fail from the page fail word supplied to it and couldn't find a reason for the fault. The page fail word was probably illegal.

14 An instruction that is currently unimplemented in the EBOX microcode was executed.

13.2 Physical EA-calc

Certain classes of instructions (PMOVE, PMOVE, and the queue instructions) perform a physical EA-calc on the word addressed by E to produce a 25 bit physical address. This physical address is then used to reference data in physical memory.

The physical EA-calc evaluates a physical EA-calc word. A physical EA-calc word is very 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 index register are added to Y to produce a 25 bit physical effective address. A physical effective address in the range 0-17, inclusive, addresses physical memory locations 0-17, not the ACs.

Bits 0 and 1 of the EA-calc word must be zero and the execution of the instruction will generate a page fail if they are not.

CHAPTER 14

SPECIAL SYSTEM PAGES (EPT / UPT / IOP)

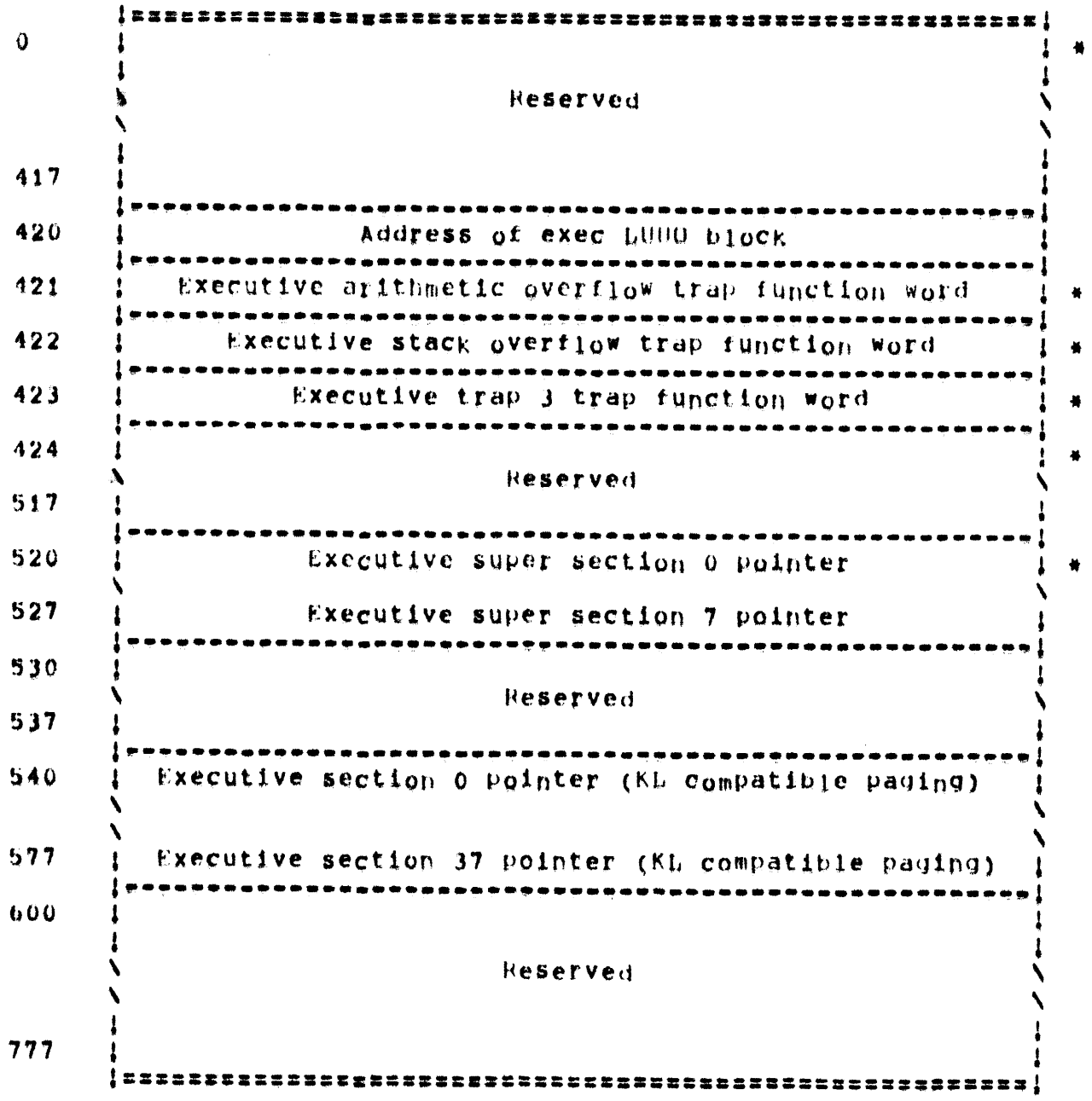
The following EPT/UPT layouts are proposed for the KC10. In addition, there is a new page called the I/O page (IOP) that is used by the KC10 ports and the console for communication with the CPU.

Upon processor reset, the base address of the EPT and UPT will be reset to page 0 and the I/O page will be reset to page 1.

NOTE

All areas that differ from the KL10 are marked with an asterisk (*).

TOPS-20 paging executive process table configuration



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

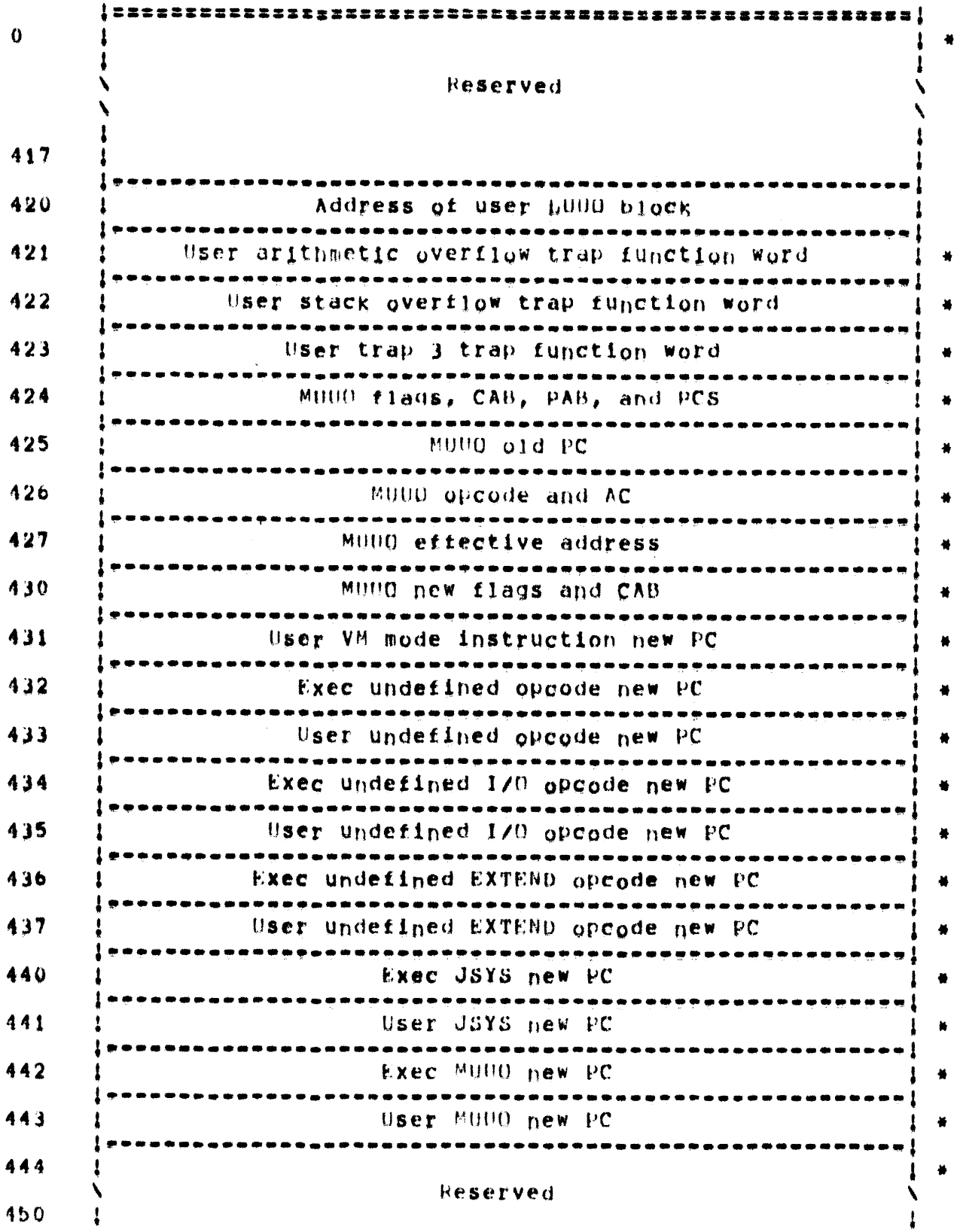
420 Address of exec LUUO block. Exec LUUOs executed with PC section non-zero are processed through the four-word LUUO block whose 30-bit virtual address is contained in this word. For more information on the format of the four-word block, see the chapter on Trap, UUO, and Interrupt Handling.

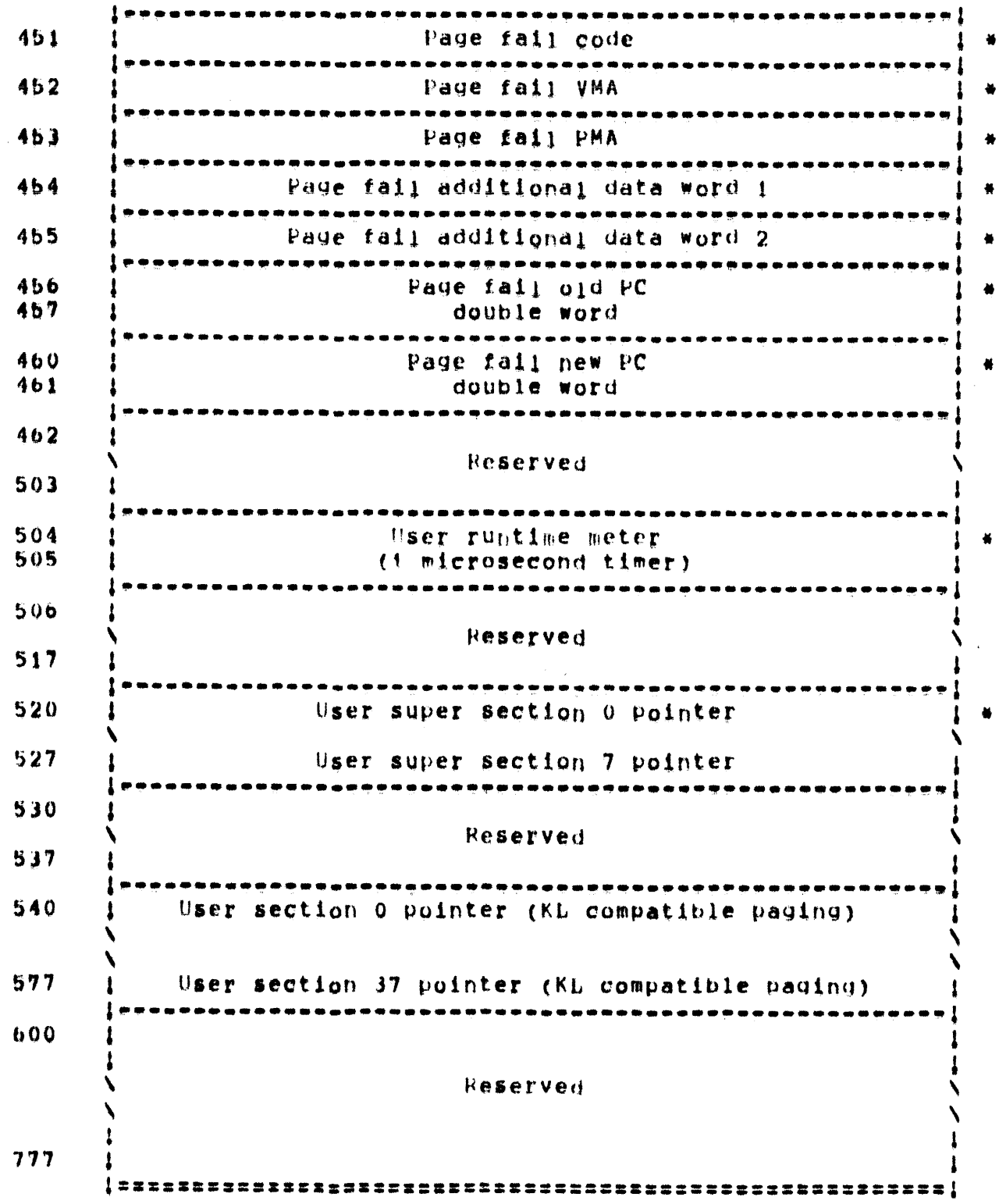
421-423 Exec trap function words for trap 1, 2, and 3. These function words are interpreted to process exec trap 1 (arithmetic overflow), trap 2 (pushdown list overflow), and trap 3 exceptions. For more information on the format of a trap function word, see the chapter on Trap, UUO, and Interrupt Handling.

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 chapter 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 chapter on paging.

TOPS-20 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 chapter on Trap, UUO, and Interrupt 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 chapter on Trap, UUO, and Interrupt Handling.

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

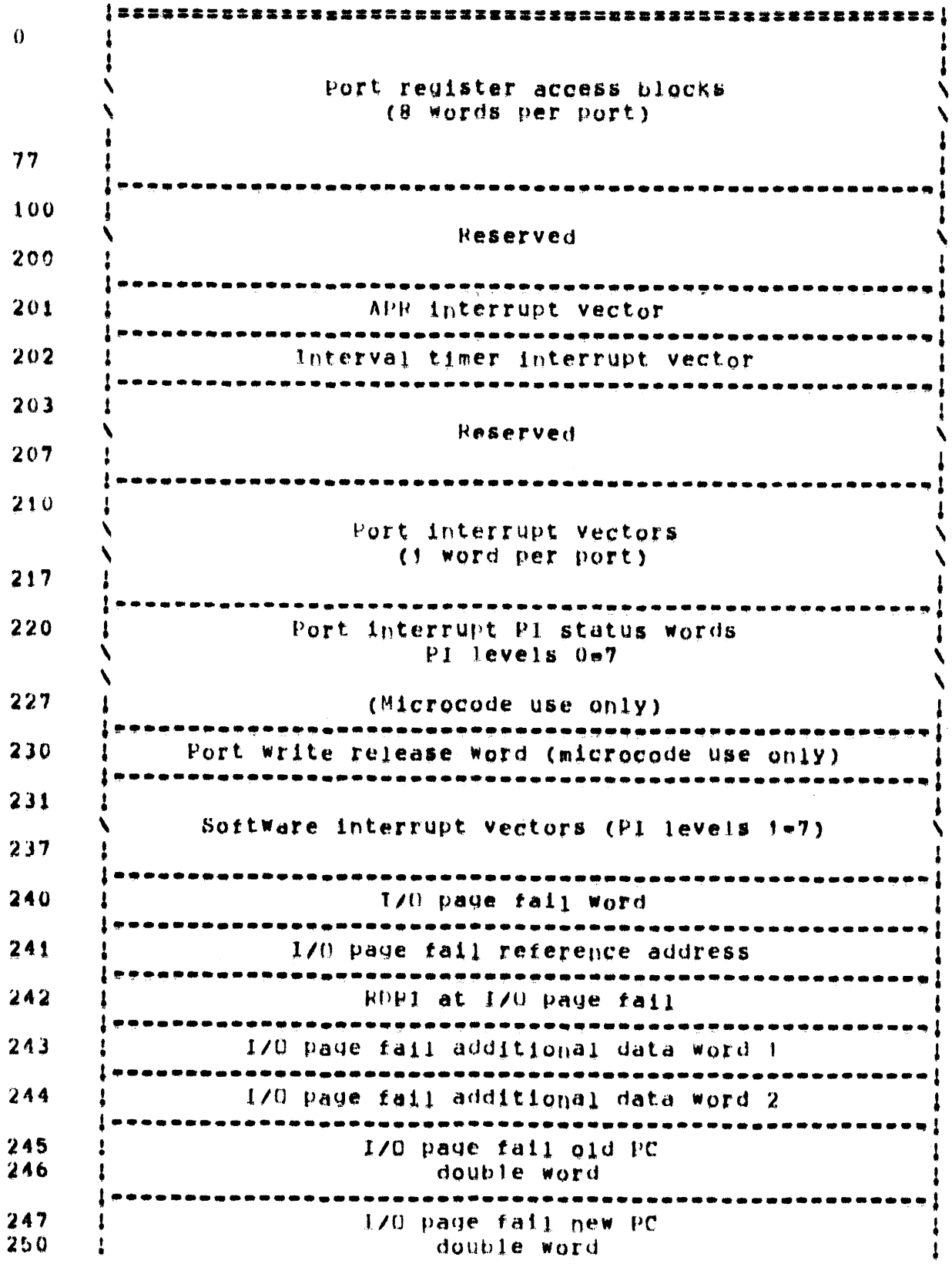
451-461 Page fail processing locations. These locations are used to process user and exec page fails. For the format of each word, see the chapter on Paging.

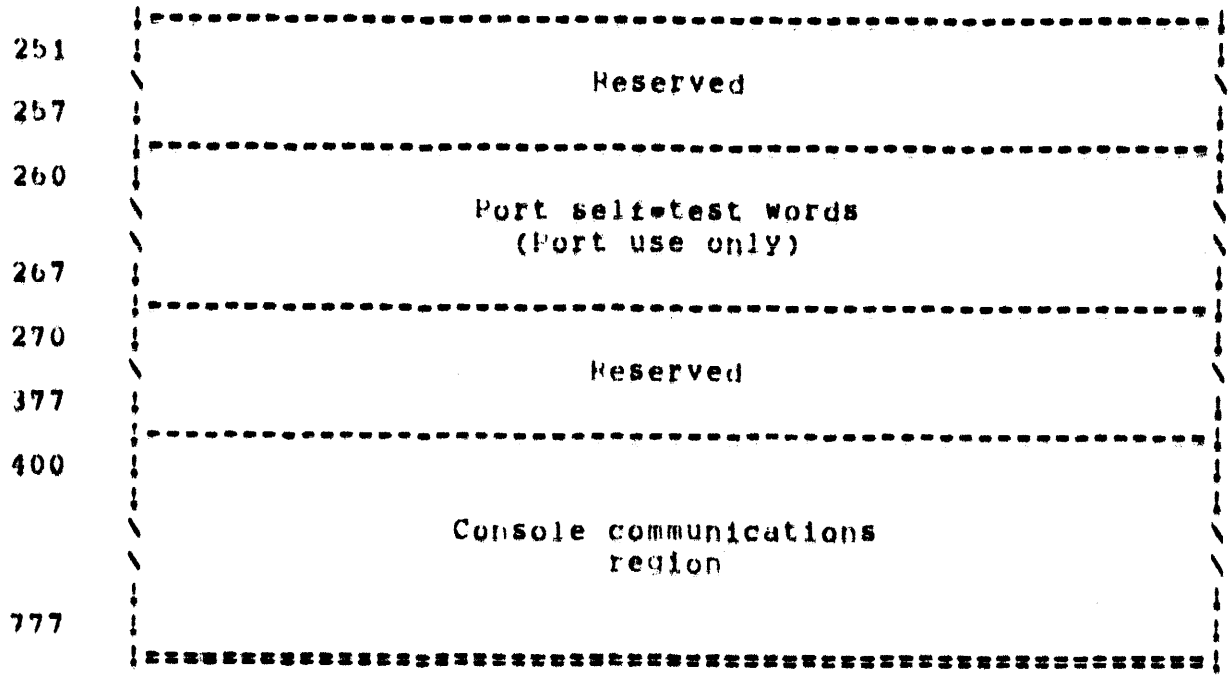
504-505 User runtime meter. These locations contain the current value of the user runtime meter counter for this process. The user runtime meter is a one microsecond counter maintained by the hardware and microcode. For more information on the format of these words, see the chapter on System Timers.

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 chapter 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 chapter on Paging.

I/O Page


0
Port register access blocks
(8 words per port)
77
100
Reserved
200
201
APR interrupt vector
202
Interval timer interrupt vector
203
Reserved
207
210
Port interrupt vectors
(1 word per port)
217
220
Port interrupt PI status words
PI levels 0-7
227
(Microcode use only)
230
Port write release word (microcode use only)
231
Software interrupt vectors (PI levels 1-7)
237
240
I/O page fail word
241
I/O page fail reference address
242
RDPI at I/O page fail
243
I/O page fail additional data word 1
244
I/O page fail additional data word 2
245
246
I/O page fail old PC
double word
247
250
I/O page fail new PC
double word



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

0-77 Port register access blocks. These locations are logically divided into 8 blocks of 8 word each, with one block for each port. The blocks are indexed by port number with port 0 using block 0 (words 0-7), port 1 using block 1 (words 10-17), etc.

The register access block (or RAB) is used as part of the communications protocol between a program running in the -10 and the ports. When a program running in the -10 wants to issue a command to a port, it first stores the command in one of the words of the RAB for the port and then does a RNGB instruction. Depending on the command, the port may return information in another word in the RAB.

The register access block definition may differ for each adapter (although there are common definitions for existing adapters). For the exact usage of each word in the RAB, see the spec for the specific adapter in question.

201 APR interrupt vector. This location contains the interrupt vector word for interrupts requested by the APR as the result of an enabled APR flag bit setting. The flag bits are given in the description for the WRAPR and RDAPR instructions. For more information on the format of a interrupt vector word, see the chapter on Trap, UUO, and Interrupt Handling.

202 Interval timer interrupt vector. This location contains the interrupt vector word for interrupts requested by the interval timer when the current count equals the interval period or when it overflows. For more information on the format of a interrupt vector word, see the chapter on Trap, UUO, and Interrupt Handling.

210-217 Port interrupt vectors. These locations contain the interrupt vector words for each port. The locations are indexed by port number such that port 0 uses word 210, port 1 uses word 211, etc. For more information on the format of a interrupt vector word, see the chapter on Trap, UUO, and Interrupt Handling.

220-227 Port interrupt PI status words. These locations are used as part of the interrupt request protocol between the ports and the KC10 microcode. The locations are indexed by PI level such that PI level 0 (not currently used) uses word 220, PI level 1 uses word 221, etc.

These locations are written only by the ports and read by the KC10 microcode. They should never be written by the program running in the -10 or by the microcode. For a description of the use of these words in the interrupt request protocol, see the chapter on Traps, UUO, and Interrupt Handling.

230 Port write-release word. This location is used by a port to perform a write-release to free up the MBOX primary interlock. It is used only when a port cannot do a normal write-release to the queue header word after obtaining the primary interlock with a read-interlock request. This location may be used by any port and the data stored is undefined.

231-237 Software interrupt vectors. These locations contain the interrupt vector words for software interrupts requested as the result of the program running in the -10 issuing a WRPI with bit 24 set. The locations are indexed by PI level number, with a software request on PI level 1 using word 231, a software request on PI level 2 using word 232, etc. For more information on the format of a interrupt vector word, see the chapter on Trap, UUO, and Interrupt Handling.

240-250 I/O page fail processing locations. These locations are used to process I/O page fails. For the format of each word, see the chapter on Trap, UUO, and Interrupt Handling.

260-267 Port self-test locations. These locations are used during port self-testing to insure that the port can read and write memory. The locations are indexed by port number such that port 0 uses location 260, port 1 uses location 261, etc.

400-777 Console communications region. These locations are reserved for the console and are used by the console to implement the console protocols. For more information on the use of these locations, refer to the console spec.

INDEX

AC references 10-36
  global 10-36
  local 10-36
Address break
  algorithms 8-24
  conditions 2-19, 8-23
  discussion 8-22
  loading 2-18
  reading 2-14
Address word 10-4
APRID 2-2

BLINK 6-2
BLT 10-26
  AC references 10-27
  source and destination addresses 10-26
Byte instructions 10-17
Byte pointer decode 10-17
Byte pointer EA-calc
  byte instructions 10-18
  EXTEND instructions 10-19
Byte pointer type
  byte instructions 10-17
  EXTEND instructions 10-19

CAB
  loading 2-18
  reading 2-14, 2-29
Cache enable
  loading 2-23
  reading 2-26
Cache sweep
  invalidate 2-31
  unload 2-32
CLRPT 2-17
CST base register
  loading 2-42
  reading 2-34
CST format 8-10
CST mask register
  loading 2-44
  reading 2-36
  use 8-11
CST update
  forcing 2-13
CST updates 8-9

Doorbell . . . . . 3-2 to 3-3
DUMPTB 5-8

Context

EA-calc 10-8
  algorithm 10-8
    EFIW with global index 10-10
    IFIW with global index 10-9
    IFIW with local index 10-9
    no indexing 10-8
    section 0 10-10
    summary 10-10
  byte instructions 10-17
  byte pointers 10-18
  default section 10-11
  default sections 10-33
  EXTEND instructions 10-19
  local or global result 10-11
  local/global flag 10-13
  multi-section 10-15
  results 10-11
  section zero 10-15
EBR
  loading 2-23
  reading 2-26
Effective address calculation . . 10-8
EFIW 10-5
EPT
  TOPS-20 14-2
EXTEND instructions 10-19
  byte pointer EA-calc 10-19
  byte pointer type 10-19
  extended opcode EA-calc 10-21
EXTEND opcode map 1-4
Extended addressing
  EA-calc 10-8
  historical summary 10-3
  reference materials 10-2
  terms 10-4
    address word 10-4
    EFIW 10-5
    global address 10-4
    global index 10-5
    global stack pointer 10-7
    IFIW 10-5
    illegal indirect word 10-6
    local address 10-4
    local index 10-4
    local stack pointer 10-7
    one-word global byte pointer . 10-6
    one-word local byte pointer . 10-6
    two-word global byte pointer . 10-7
    virtual address 10-4
Extended format indirect word . . 10-5
Extended opcode EA-calc 10-21

Flags/PC double words 9-1
FLINK 6-2

Global AC address 10-13
Global address 10-4
Global index 10-5
Global stack pointer 10-7
Global stack pointers 10-23

Halt status codes 13-1

I/O page
  diagram 14-7
  loading 2-27
  reading 2-28
I/O page . . . . . 4-6 to 4-7
I/O page failure 12-10
I/O page relative move 4-6
I/O page relative movem 4-7
I/O reset 2-3
IBUX flush . . . . . 2-13, 2-17, 2-21, 2-23, 2-31, 4-5, 4-7, 5-6, 5-8, 6-8 to 6-11, 7-6
IFIW 10-5
Illegal indirect word 10-6
Incrementing EA 10-14
INBOMI 6-8
INBOTI 6-9
Instruction fetches 10-37
Instruction format indirect word . 10-5
Interrupt vectors 12-10
Interval timer
  controlling 2-47
  loading PI assignment 2-45
  reading PI assignment 2-37
  reading status 2-38
IOPMOV 4-6
IOPMVM 4-7

JRA 10-25
  EA-calc 10-25
JRST 7-6
JRSTF 10-29
JSA 10-25
  EA-calc 10-25
JSP 10-21
  storing PC 10-22
JSR 10-21
  storing PC 10-22

LDPAC 4-8
Local AC references 10-13
Local address 10-4
Local index 10-4
Local stack pointer 10-7
Local stack pointers 10-23
Local/global flag 10-13
LUUO 10-26, 12-8

MAP 7-2
Map pointers 8-7
Microcode version number 2-2
Multi-section EA-calc 10-15
MUUO 12-5

Non-zero section rules 10-34

One-word global byte pointer . . . 10-6
One-word local byte pointer . . . 10-6
OPCODE assignment map 1-3

PAB . . . . . 10-40 to 10-41
  loading 2-18
  reading 2-14, 2-29
Page address words 8-8
Page fail word 8-12
Page refill 8-9
Pager enable
  loading 2-23
  reading 2-26
Paging information cache 8-2
Paging pointers 8-4
PC flags 9-1
PC store 10-38
PC trace 5-2
Pc trace 5-3
PCS . . . . . 10-40 to 10-41
  loading 2-18
  reading 2-14, 2-29
PCU . . . . . 10-40 to 10-41
Physical ea-calc . . . . . 4-4 to 4-5
  definition 13-2
Physical memory . . . . . 4-4 to 4-5
PI system
  control 2-8
  status . . . . . 2-10 to 2-12
PMOVE 4-4
PMOVEM 4-5
Pointers 8-4
POPM 7-10
Previous context . . . . . 4-2 to 4-3, 4-8 to 4-9
  applicable instructions 10-42
  references 10-41
  state registers 10-40
    use 10-41
Previous context execute . . . . . 4-2 to 4-3
Process context variables 9-1
Process use register
  loading 2-43
  reading 2-35
  use 8-11
Processor serial number 2-2
PUSHJ 7-12

PUSHM 7-8
PXCT . . . . . 4-2 to 4-3, 10-40
  AC field bits 10-42
  EA-calc algorithm 10-44
    flow chart 10-46
  local/global flag
    byte instructions 10-49
    MOVSLJ 10-50
    stack instructions 10-48

Queue formats 6-2
Queue headers 6-2
Queue insertion 6-3
Queue interlocks 6-5
Queue removal 6-5

RDACT 2-50
RDAPR 2-5
RDCSB 2-34
RDCSTM 2-36
RDCTX 2-14
RDEBR 2-26
RDINT 2-38
RDIOP 2-28
RDPI 2-10
RDPUR 2-35
RDSPB 2-33
RDTIMF 2-39
RDTMB 2-37
RDTRAX 5-2
RDUBR 2-29
RDURTM 2-40
READTB 5-4
REMQHI 6-10
REMQTI 6-11
RNGB 3-2
RNGBW 3-3

Secondary queue interlock 6-2
Section pointers 8-6
Section zero rules 10-34
SETCU 2-13
SNAPR 2-7
SNBSY 3-4
SNPI 2-12
SPT base register
  loading 2-41
  reading 2-33
Stack instructions 10-23
  storing PC 10-24
Stack pointers 10-23
  default section 10-23
  incrementing 10-23
State bits 8-1

Storing EA 10-38
STPAC 4-9
Super section pointers 8-5
SWPIA 2-31
SWPUA 2-32
SZAPR 2-6
SZPI 2-11

Time base
  controlling 2-45
  reading status 2-37
  reading value 2-39
TOPS-20 page fail 8-12
TOPS-20 page fail codes 8-16
TOPS-20 paging 8-4
Tracks . . . . . 5-2 to 5-3
Translation buffer
  clearing 2-17
  conditional clear 2-18
  dumping 5-8
  hardware 8-1
  mapping 7-2
  reading 5-4
  state bits 8-1, 8-11
  writing 5-6
Trap enable
  definition 12-9
  loading 2-23
  reading 2-26
Trap function word 12-2
Two-word global byte pointer . . . 10-7

UBR
  loading 2-18
  reading 2-14, 2-29
UMOVE 4-2
UMOVEM 4-3
UPT
  TOPS-20 14-4
User runtime meter 2-18
  controlling 2-48
  reading status 2-50
  reading value 2-40

Virtual address 10-4
VM mode
  definition 12-4
  invoking 2-18
  reading 2-14, 2-29

WRACT 2-48
WRAPR 2-3
WRCSB 2-42
WRCSTM 2-44

that's all ?

WRCTX 2-18
WREBR 2-23
WRINT 2-47
WRIOP 2-27
WRITTB 5-6
WRPI 2-8
WRPUR 2-43
WRSPB 2-41
WRTMB 2-45
WRTRAX 5-3

XBLT 10-28
  non-zero section references . . 10-28
XCT 10-30
  default section for EA-calc . . 10-30
  local stack references 10-31
  PC storing instructions 10-31
  skip and jump instructions . . . 10-30
  stack instructions 10-31
XHLLI 10-29
  AC references 10-29
XMOVEI 10-29
  AC references 10-29

* * * L P T S P L R u n L o g * * *

```
7:34:03 LPDAT  LPTSPL version 104(3103)          KL2102, TOPS-20 Development
7:34:03 LPDAT  Job KCIO sequence #2814 on Printer 0 at 28-Apr-83  7:34:03
7:40:28 LPMSG  Starting File SNARK:<JUPITER,FUNCTIONAL-SPECS>KCIO.VER-8,1
7:50:52 LPMSG  Finished File SNARK:<JUPITER,FUNCTIONAL-SPECS>KCIO.VER-8,1
7:50:52 LPEND  Summary:   231 Pages of Output
7:50:52 LPEND               161 Disk Pages Read
7:50:52 LPEND             5.381 Seconds CPU Time Used
```