

FILEID**VAXASHP

VV	VV	AAAAAA	XX	XX	AAAAAA	SSSSSSS	HH	HH	PPPPPPP
VV	VV	AAAAAA	XX	XX	AAAAAA	SSSSSSS	HH	HH	PPPPPPP
VV	VV	AA	AA	XX	XX	AA	SS	HH	PP PP
VV	VV	AA	AA	XX	XX	AA	SS	HH	PP PP
VV	VV	AA	AA	XX	XX	AA	SS	HH	PP PP
VV	VV	AA	AA	XX	XX	AA	SS	HH	PP PP
VV	VV	AA	AA	XX	XX	AA	AA	SSSSSS	HHHHHHHHHHH PPPPPP
VV	VV	AA	AA	XX	XX	AA	AA	SSSSSS	HHHHHHHHHHH PPPPPP
VV	VV	AAAAAAA	XX	XX	AAAAAAA	SS	HH	HH	PP
VV	VV	AAAAAAA	XX	XX	AAAAAAA	SS	HH	HH	PP
VV	VV	AA	AA	XX	XX	AA	AA	SS	HH HH PP
VV	VV	AA	AA	XX	XX	AA	AA	SS	HH HH PP
VV	VV	AA	AA	XX	XX	AA	AA	SSSSSS	HH HH PP
VV	VV	AA	AA	XX	XX	AA	AA	SSSSSS	HH HH PP
..
VV	VV	AA	AA	XX	XX	AA	AA	SSSSSS	HH HH PP
VV	VV	AA	AA	XX	XX	AA	AA	SSSSSS	HH HH PP
..

LL	IIIIII	SSSSSSS
LL	IIIIII	SSSSSSS
LL	II	SS
LLLLLLLL	IIIIII	SSSSSSS
LLLLLLLL	IIIIII	SSSSSSS

(2)	69	Declarations
(3)	121	VAX\$ASHP - Arithmetic Shift and Round Packed
(4)	302	ASHP SHIFT - Perform Shift
(5)	565	VAX\$DECIMAL_EXIT - Exit Path for Decimal Instructions
(7)	645	ASHP COPY SOURCE - Copy Source String to Work Area
(8)	696	DECIMAL_R0PRAND
(9)	731	ASHP_ACCEVIO - Reflect an Access Violation
(10)	901	DECIMAL\$BOUNDS_CHECK - Bounds Check on Exception PC
(11)	967	Context-Specific Access Violation Handling for VAX\$ASHP
(12)	1074	VAX\$DECIMAL_ACCVIO - Common Access Violation Handling

0000 1 .TITLE VAX\$ASHP - VAX-11 Instruction Emulator for ASHP
0000 2 .IDENT /V04-000/
0000 3
0000 4 :
0000 5 *****
0000 6 *
0000 7 * COPYRIGHT (c) 1978, 1980, 1982, 1984 BY
0000 8 * DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.
0000 9 * ALL RIGHTS RESERVED.
0000 10 *
0000 11 * THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
0000 12 * ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
0000 13 * INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
0000 14 * COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
0000 15 * OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
0000 16 * TRANSFERRED.
0000 17 *
0000 18 * THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
0000 19 * AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
0000 20 * CORPORATION.
0000 21 *
0000 22 * DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
0000 23 * SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
0000 24 *
0000 25 *
0000 26 *****
0000 27 :
0000 28 :
0000 29 :++
0000 30 : Facility:
0000 31 : VAX-11 Instruction Emulator
0000 32 :
0000 33 : Abstract:
0000 34 :
0000 35 : The routine in this module emulates the VAX-11 packed decimal
0000 36 : ASHP instruction. This procedure can be a part of an emulator
0000 37 : package or can be called directly after the input parameters
0000 38 : have been loaded into the architectural registers.
0000 39 :
0000 40 :
0000 41 : The input parameters to this routine are the registers that
0000 42 : contain the intermediate instruction state.
0000 43 :
0000 44 : Environment:
0000 45 :
0000 46 : This routine runs at any access mode, at any IPL, and is AST
0000 47 : reentrant.
0000 48 :
0000 49 : Author:
0000 50 :
0000 51 : Lawrence J. Kenah
0000 52 :
0000 53 : Creation Date
0000 54 :
0000 55 : 18 October 1983
0000 56 :
0000 57 : Modified by:

0000	58		
0000	59		
0000	60	V01-003 LJK0042	Lawrence J. Kenah
0000	61		Final cleanup pass for source kit.
0000	62	V01-002 LJK0024	Lawrence J. Kenah
0000	63		Add code to handle access violations. Perform minor cleanup.
0000	64		
0000	65	V01-001 LJK0008	Lawrence J. Kenah
0000	66		18-Oct-1983
0000	67		The emulation code for ASHP was moved into a separate module.
	--		

```
0000 69      .SUBTITLE Declarations
0000 70
0000 71 ; Include files:
0000 72
0000 73      .NOCROSS
0000 74      .ENABLE      SUPPRESSION ; No cross reference for these
0000 75          ; No symbol table entries either
0000 76      ASHP_DEF
0000 77
0000 78      ADDP4_DEF
0000 79      ADDP6_DEF
0000 80      DIVP_DEF
0000 81      MULP_DEF
0000 82      SUBP4_DEF
0000 83      SUBP6_DEF
0000 84
0000 85      CVTPS_DEF
0000 86      CVTPT_DEF
0000 87      CVTSP_DEF
0000 88      CVTTP_DEF
0000 89
0000 90      PACK_DEF
0000 91      STACR_DEF
0000 92
0000 93      $PSLDEF      ; Stack usage for reflecting exceptions
0000 94      $$SRMDEF    ; Stack usage for original exception
0000 95
0000 96      .DISABLE     SUPPRESSION ; Define bit fields in PSL
0000 97      .CROSS       ; Define arithmetic trap codes
0000 98
0000 99 ; External declarations
0000 100
0000 101      .DISABLE     GLOBAL
0000 102
0000 103      .EXTERNAL -   DECIMAL$STRIP_ZEROS_R0_R1
0000 104
0000 105
0000 106      .EXTERNAL -   VAX$EXIT_EMULATOR,-
0000 107          VAX$ADD_PACKED_BYTE_R6_R7,-
0000 108          VAX$REFLECT_FAULT,-
0000 109          VAX$REFLECT_TRAP,-
0000 110          VAX$ROPRAND
0000 111
0000 112
0000 113 ; PSECT Declarations:
0000 114
0000 115      .DEFAULT     DISPLACEMENT , WORD
0000 116
0000 117      .PSECT _VAX$CODE PIC, USR, CON, REL, LCL, SHR, EXE, RD, NOWRT, LONG
0000 118
0000 119      BEGIN_MARK_POINT
```

0000 121
0000 122
0000 123
0000 124
0000 125
0000 126
0000 127
0000 128
0000 129
0000 130
0000 131
0000 132
0000 133
0000 134
0000 135
0000 136
0000 137
0000 138
0000 139
0000 140
0000 141
0000 142
0000 143
0000 144
0000 145
0000 146
0000 147
0000 148
0000 149
0000 150
0000 151
0000 152
0000 153
0000 154
0000 155
0000 156
0000 157
0000 158
0000 159
0000 160
0000 161
0000 162
0000 163
0000 164
0000 165
0000 166
0000 167
0000 168
0000 169
0000 170
0000 171
0000 172
0000 173
0000 174
0000 175
0000 176
0000 177

.SUBTITLE VAXSASHP - Arithmetic Shift and Round Packed

Functional Description:

The source string specified by the source length and source address operands is scaled by a power of 10 specified by the count operand. The destination string specified by the destination length and destination address operands is replaced by the result.

A positive count operand effectively multiplies; a negative count effectively divides; and a zero count just moves and affects condition codes. When a negative count is specified, the result is rounded using the Round Operand.

Input Parameters:

R0<15:0> = srclen.rw Number of digits in source character string
R0<23:16> = cnt.rb Shift count
R1 = srcaddr.ab Address of input character string
R2<15:0> = dstlen.rw Length in digits of output decimal string
R2<23:16> = round.rb Round operand used with negative shift count
R3 = dstaddr.ab Address of destination packed decimal string

Output Parameters:

R0 = 0
R1 = Address of byte containing most significant digit of
the source string
R2 = 0
R3 = Address of byte containing most significant digit of
the destination string

Condition Codes:

N <- destination string LSS 0
Z <- destination string EQL 0
V <- decimal overflow
C <- 0

Algorithm:

The routine tries as much as possible to work with entire bytes. This makes the case of an odd shift count more difficult than of an even shift count. The first part of the routine reduces the case of an odd shift count to an equivalent operation with an even shift count.

The instruction proceeds in several stages. In the first stage, after the input parameters have been verified and stored, the operation is broken up into four cases, based on the sign and parity (odd or even) of the shift count. These four cases are treated as follows, in order of increasing complexity.

Case 1. Shift count is negative and even

The actual shift operation can work with the source string in place. There is no need to move the source string to an intermediate work area.

				0000 178	
				0000 179	Case 2. Shift count is positive and even
				0000 180	
				0000 181	The source string is moved to an intermediate work area and the
				0000 182	sign "digit" is cleared before the actual shift operation takes
				0000 183	place. If the source is worked on in place, then a spurious sign
				0000 184	digit would be moved to the middle of the output string instead
				0000 185	of a zero. The alternative is to keep track of where, in the
				0000 186	several special cases of shifting, the sign digit is looked at.
				0000 187	We chose to use the work area to simplify the later stages of
				0000 188	this routine.
				0000 189	
				0000 190	
				0000 191	
				0000 192	Cases 3 and 4. Shift count is odd
				0000 193	
				0000 194	The case of an odd shift count is considerably more difficult
				0000 195	than an even shift count, which is only slightly more complicated
				0000 196	than MOVP. In the case of an even shift count, various digits
				0000 197	remain in the same place (high nibble or low nibble) in a byte.
				0000 198	For odd shift counts, high nibbles become low nibbles and vice
				0000 199	versa. In addition, digits that were adjacent when viewing the
				0000 200	decimal string as a string of bits proceeding from low address to
				0000 201	high are now separated by a full byte.
				0000 202	
				0000 203	
				0000 204	
				0000 205	
				0000 206	
				0000 207	
				0000 208	
				0000 209	
				0000 210	
				0000 211	
				0000 212	ASHP_SHIFT_MASK = ^XFOFOFOFO ; Mask used to shift string by one
				0000 213	VAX\$ASHP::
			OFFF 8F BB	0000 213	PUSHR #^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; Save the lot
			5B DC	0004 214	MOVPSL R11 ; Get initial PSL
5B	04	00 04	F0	0006 215	INSV #PSLSM Z,#0,#4,R11 ; Set Z-bit, clear the rest
				000B 216	ESTABLISH_HANDLER - ; Store address of access
				000B 217	ASHP ACCVIO ; violation handler
				0010 218	ROPRAND_CHECK R2 ; Insure that R2 LEQU 31
				001B 219	ROPRAND_CHECK R0 ; Insure that R0 LEQU 31
			58 5E D0	0023 220	MOVL SP R8 ; Remember current top of stack
			5E 14 C2	0026 221	SUBL #20,SP ; Allocate work area on stack
				0029 222	MARK_POINT ASHP BSBW 20
			FFD4' 30	0029 223	BSBW DECIMALSSTRIP_ZEROS_R0_R1 ; Eliminate any high order zeros
			01 EF	002C 224	EXTZV #1,#4,R2,R2 ; Convert output digit count to bytes
52	52	04 04	52 D6	0031 225	INCL R2 ; Make room for sign as well
			01 EF	0033 226	EXTZV #1,#4,R0,R0 ; Same for input string
50	50	04 04	51 50	0038 227	ADDL3 R0,R1,R6 ; Get address of sign digit
			50 D6	003C 228	INCL R0 ; Include byte containing sign
				003E 229	MARK_POINT ASHP 20
				003E 230	BICB3 #^B11110000,T(R6),R6 ; Extract sign digit
				0043 231	
				0043 232	: Form sign of output string in R9 in preferred form (12 for "+" and 13 for "-")
				0043 233	
				0043 234	MOVZBL #12,R9 ; Assume that input sign is plus

				0046	235	CASE	R6 LIMIT=#10,TYPE=B,<-	; Dispatch on sign
				0046	236		40\$,-	; 10 => +
				0046	237		30\$,-	; 11 => -
				0046	238		40\$,-	; 12 => +
				0046	239		30\$,-	; 13 => -
				0046	240		40\$,-	; 14 => +
				0046	241		40\$,-	; 15 => +
				0046	242		>	
				0056	243			
5B	59	D6	0056	244	30\$: INCL R9			; Change preferred plus to minus
	08	88	0058	245	BISB #PSL\$M_N,R11			; Set N-bit in saved PSW
			0058	246				
			0058	247				: We now retrieve the shift count from the saved R0 and perform the next set
			0058	248				: of steps based on the parity and sign of the shift count. Note that the
			0058	249				: round operand is ignored unless the shift count is strictly less than zero.
54	02	A8	98	005B	250			
	0D		19	005F	251	40\$: CVTBL ASHP_B_CNT(R8),R4		; Extract sign-extended shift count
	55	D4	0061	252	BLSS 50\$; Branch if shift count negative
	0151	30	0063	253	CLRL R5			; Ignore "round" for positive shift
78	11	54	E8	0066	254	BSBW ASHP_COPY_SOURCE		; Move source string to work area
	OF	8A	0069	255	BLBS R4,60\$; Do shift by one for odd shift count
	3C	11	006C	256	BICB2 #^B00001111,-(R8)			; Drop sign in saved source string
			006E	257	BRB ASHP_SHIFT_POSITIVE			; Go do the actual shift
			006E	258				
			006E	259				: The "round" operand is important for negative shifts. If the shift count
			006E	260				: is even, the source can be shifted directly into the destination. For odd
			006E	261				: shift counts, the source must be moved into the work area on the stack and
			006E	262				: shifted by one before the rest of the shift operation takes place.
55	00	EF	006E	263				
	04		0070	264	50\$: EXTZV #ASHP_V_ROUND,-			
	0A	A8	0071	265	#ASHP_S_ROUND,-			
	54	54	E9	266	ASHP_B_ROUND(R8),R5			
	013D	30	0074	267	BLBC R4,ASHP SHIFT NEGATIVE			; Store "round" in a safe place
			0077	268	BSBW ASHP_COPY_SOURCE			; Get right to it for even shift count
			007A	269				; Move source string to work area
			007A	270				
			007A	271				: For odd shift counts, the saved source string is shifted by one in place.
			007A	272				: This is equivalent to a shift of -1 so the shift count (R4) is adjusted
			007A	273				: accordingly. The least significant digit is moved to the place occupied by
			007A	274				: the sign, the tens digit becomes the units digit, and so on. Because the
			007A	275				: work area was padded with zeros, this shift moves a zero into the high
			007A	276				: order digit of a source string of even length.
50	50	50	DD	007A	277	60\$: PUSHL R0		
			D7	007C	278	DECL R0		; We need a scratch register to count
			78	007E	279	ASHL #2,R0,R0		; Want to map {1..16} onto {0..3}
				0083	280			; Convert a byte count to longwords
				0083	281			
				0083	282			: The following loop executes from one to four times such that the entire
				0083	283			: source, taken as a collection of longwords, is shifted by one. Note that
				0083	284			: the two pieces of the source are shifted (rotated) in opposite directions.
				0083	285			: Note also that the shift mask is applied to one string before the shift and
				0083	286			: to the other string after the shift. (This points up the arbitrary choice
				0083	287			: of shift mask. We just as well could have chosen the one's complement of
				0083	288			: the shift mask and reversed the order of the shift and mask operations for
				0083	289			: the two pieces of the source string.)
56	78	FC	8F	9C	0083	290	70\$: ROTL #4,-(R8),R6	; Shift left one digit
56	FOFOFOFO	BF	CA	0088	291	BICL2 #ASHP_SHIFT_MASK,R6		; Clear out old low order digits

57 FF A8 FOF0FOFO 8F	CB 008F 292	BICL3 #ASHP_SHIFT_MASK,-1(R8),R7 ; Clear out high order digits
57 57 04 9C 0098 293	ROTL #4,R7,R7 ; Shift these digits right one digit	
68 57 56 C9 009C 294	BISL3 R6,R7,(R8) ; Combine the two sets of digits	
E0 50 F4 00A0 295	SOBGEQ R0,70\$; Keep going if more	
	00A3 296	
50 8E D0 00A3 297	MOVL (SP)+,R0 ; Restore source string byte count	
54 D6 00A6 298	INCL R4 ; Count the shift we did	
21 19 00A8 299	BLSS ASHP_SHIFT_NEGATIVE ; Join common code at the right place	
00AA 300		; Drop through to ASHP_SHIFT_POSITIVE

00AA 302 .SUBTITLE ASHP_SHIFT - Perform Shift
 00AA 303 ;+
 00AA 304 Functional Description:
 00AA 305
 00AA 306 This routine completes the work of the ASHP instruction in the case of
 00AA 307 an even shift count. (If the original shift count was odd, the source
 00AA 308 string has already been shifted by one and the shift count adjusted by
 00AA 309 one.) A portion (from none to all) of the source string is moved to
 00AA 310 the destination string. Pieces of the destination string at either end
 00AA 311 may be filled with zeros. If excess digits of the source are not
 00AA 312 moved, they must be tested for nonzero to determine the correct
 00AA 313 setting of the V-bit.
 00AA 314
 00AA 315 Input Parameters:
 00AA 316
 00AA 317 R0<3:0> - Number of bytes in source string
 00AA 318 R1 - Address of source string
 00AA 319 R2<3:0> - Number of bytes in destination string
 00AA 320 R3 - Address of destination string
 00AA 321 R4<7:0> - Count operand (signed longword of digit count)
 00AA 322 R5<3:0> - Round operand in case of negative shift
 00AA 323 R9<3:0> - Sign of source string in preferred form
 00AA 324
 00AA 325 Implicit Input:
 00AA 326
 00AA 327 R4 is presumed (guaranteed) even on input to this routine
 00AA 328
 00AA 329 The top of the stack is assumed to contain a 20-byte work area (that
 00AA 330 may or may not have been used). The space must be allocated for this
 00AA 331 work area in all cases so that the exit code works correctly for all
 00AA 332 cases without the need for lots of extra conditional code.
 00AA 333
 00AA 334 Output Parameters:
 00AA 335
 00AA 336 This routine completes the operation of VAX\$ASHP. See the routine
 00AA 337 header for VAX\$ASHP for details on output registers and condition codes.
 00AA 338
 00AA 339 Details:
 00AA 340
 00AA 341 PUT SOME OF THE STUFF FROM ASHP.TXT HERE.
 00AA 342 ;-
 00AA 343
 00AA 344 .ENABLE LOCAL_BLOCK
 00AA 345
 00AA 346 ASHP_SHIFT POSITIVE:
 57 54 02 C6 00AA 347 DIVL #2,R4 ; Convert digit count to byte count
 52 54 C3 00AD 348 SUBL3 R4,R2,R7 ; Modify the destination count
 0E 19 00B1 349 BLSS 30\$; Branch if simply moving zeros
 58 56 54 D0 00B3 350
 57 50 C3 00B6 351 MOVL R4,R6 ; Number of zeros at low order end
 26 19 00BA 352 10\$: SUBL3 R0,R7,R8 ; Are there any excess high order digits?
 00BC 353 BLSS 60\$; No, excess is in source.
 00BC 354
 00BC 355 ; We only move "srcien" source bytes. The rest of the destination string is
 00BC 356 ; filled with zeros.
 00BC 357
 57 50 D0 00BC 358 MOVL R0,R7 ; Get number of bytes to actually move

3E 11 00BF 359 BRB 100\$; ... and go move them

00C1 360

00C1 361 : The count argument is larger than the destination length. All of the source

00C1 362 : is checked for nonzero (overflow check). All of the destination is filled

00C1 363 : with zeros.

00C1 364

56 52 D0 00C1 365 30\$: MOVL R2,R6 ; Number of low order zeros

57 57 D4 00C4 366 CLRL R7 ; The source string is untouched

58 50 D0 00C6 367 MOVL R0,R8 ; Number of source bytes to check

27 11 00C9 368 BRB 80\$; Go do the actual work

00CB 369

00CB 370 : If the count is negative, then there is no need to fill in low order zeros

00CB 371 : (R6 is zero). The following code is similar to the above cases, differing

00CB 372 : in the roles played by source length (R0) and destination length (R2) and

00CB 373 : also in the first loop (zero fill or overflow check) that executes.

00CB 374

00CB 375 ASHP_SHIFT NEGATIVE:

54 56 D4 00CB 376 CLRL R6 ; No zero fill at low end of destination

54 54 CE 00CD 377 MNEGL R4,R4 ; Get absolute value of count

54 02 C6 00D0 378 DIVL #2,R4 ; Convert digit count to byte count

57 50 54 C3 00D3 379 SUBL3 R4,R0,R7 ; Get modified source length

0E 19 00D7 380 BLSS 70\$; Branch if count is larger

58 52 57 C3 00D9 381 SUBL3 R7,R2,R8 ; Are there zeros at high end?

20 18 00DD 382 BGEQ 100\$; Exit to zero fill loop if yes

00DF 383

00DF 384

00DF 385 : The modified source length is larger than the destination length. Part

00DF 386 : of the source is moved. The rest is checked for nonzero.

00DF 387

57 52 D0 00DF 388 MOVL R2,R7 ; Only move "dstlen" bytes

00E2 389

00E2 390 : In these cases, some digits in the source string will not be moved. If any

00E2 391 : of these digits is nonzero, then the V-bit must be set.

00E2 392

58 58 CE 00E2 393 60\$: MNEGL R8,R8 ; Number of bytes in source to check

0B 11 00E5 394 BRB 80\$; Exit to overflow check loop

00E7 395

00E7 396 : The count argument is larger than the source length. All of the destination

00E7 397 : is filled with zeros. The source is ignored.

00E7 398

58 57 D4 00E7 399 70\$: CLRL R7 ; No source bytes get moved

58 52 D0 00E9 400 MOVL R2,R8 ; All of the destination is filled

11 11 00EC 401 BRB 100\$; Join the zero fill loop

00EE 402

00EE 403 :+ At this point, the three separate counts have all been calculated. Each

00EE 404 : loop is executed in turn, stepping through the source and destination

00EE 405 : strings, either alone or in step as appropriate.

00EE 406

00EE 407

00EE 408 : R6 - Number of low order digits to fill with zero

00EE 409 : R7 - Number of bytes to move intact from source to destination

00EE 410 : R8 - Number of excess digits in one or the other string.

00EE 411

00EE 412 : If excess source digits, they must be tested for nonzero to

00EE 413 : correctly set the V-bit.

00EE 414

00EE 415 : If excess destination bytes, they must be filled with zero.

```

        00EE 416 ;-
        00EE 417
        00EE 418 ; Test excess source digits for nonzero
        00EE 419
        00EE 420      MARK_POINT      ASHP_20
  81 95 00EE 421 75$    TSTB   (R1)+          ; Is next byte nonzero
  05 12 00FO 422  BNEQ   90$          ; Handle overflow out of line
 F9 58 00F2 423 80$:   SOBGEQ  R8,75$      ; Otherwise, keep on looking
        00F5 424
        11 11 00F5 425      BRB    120$          ; Join top of second loop
        00F7 426
        5B 02 88 00F7 427 90$:   BISB   #PSL$M_V,R11   ; Set saved V-bit
  51 58 C0 00FA 428  ADDL   R8,R1          ; Skip past rest of excess
  09 11 00FD 429      BRB    120$          ; Join top of second loop
        00FF 430
        00FF 431 ; In this case, the excess digits are found in the destination string. They
        00FF 432 ; must be filled with zero.
        00FF 433
  58 D5 00FF 434 100$:  TSTL   R8          ; Is there really something to do?
  05 13 0101 435  BEQL   120$          ; Skip first loop if nothing
        0103 436
        83 94 0103 437      MARK_POINT      ASHP_20
 FB 58 F5 0105 438 110$:  CLRB   (R3)+          ; Store another zero
        0108 439      SOBGTR  R8,110$      ; ... and keep on looping
        0108 440
        0108 441 ; The next loop is where something interesting happens, namely that parts of
        0108 442 ; the source string are moved to the destination string. Note that the use of
        0108 443 ; bytes rather than digits in this operation makes the detection of nonzero
        0108 444 ; digits difficult because the presence of a nonzero digit in the place
        0108 445 ; occupied by the sign or in the high order nibble of an even length output
        0108 446 ; string and nowhere else would cause the Z-bit to be incorrectly cleared.
        0108 447 ; For this reason, we ignore the Z-bit here and make a special pass over the
        0108 448 ; output string after all of the special cases have been dealt with. The
        0108 449 ; extra overhead of a second trip to memory is offset by the simplicity in
        0108 450 ; other places in this routine.
        0108 451
  57 D5 0108 452 120$:  TSTL   R7          ; Something to do here?
  06 13 010A 453  BEQL   140$          ; Skip this loop if nothing
        010C 454
        83 81 90 010C 455      MARK_POINT      ASHP_20
 FA 57 F5 010F 456 130$:  MOVB   (R1)+,(R3)+  ; Move the next byte
        0112 457      SOBGTR  R7,130$      ; ... and keep on looping
        0112 458
        0112 459 ; The final loop occurs in some cases of positive shift count where the low
        0112 460 ; order digits of the destination must be filled with zeros.
        0112 461
  56 D5 0112 462 140$:  TSTL   R6          ; Something to do here?
  05 13 0114 463  BEQL   160$          ; Skip if loop count is zero
        0116 464
        83 94 0116 465      MARK_POINT      ASHP_20
 FB 56 F5 0118 466 150$:  CLRB   (R3)+          ; Store another zero
        011B 467      SOBGTR  R6,150$      ; ... until we're done
        011B 468
        011B 469 ;+
        011B 470 ; At this point, the destination string is complete except for the sign.
        011B 471 ; If there is a round operand, that must be added to the destination string.
        011B 472 ;

```

011B 473 ; R3 - Address one byte beyond destination string
 011B 474 ; R5 - Round operand
 011B 475 ;
 011B 476 ;

52 08 AE 5E 14 C0 011B 477 160\$: ADDL #20,SP ; Deallocate work area
 04 01 EF 011E 478 EXTZV #1, #4, ASHP_W_DSTLEN(SP),R2 ; Get original destination byte count
 7E 52 7D 0124 479 MOVQ R2,-(SP) ; Save address and count for Z-bit loop
 58 55 9A 0127 480 MOVZBL R5,R8 ; Load round into carry register
 15 13 012A 481 BEQL 180\$; Skip next mess unless "round" exists
 55 53 D0 012C 482 MOVL R3,R5 ; R5 tracks the addition output
 56 D4 012F 483 CLRL R6 ; We only need one term and carry in sum
 0131 484 ;
 0131 485 ;
 57 73 9A 0131 486 MARK_POINT ASHP_8
 0134 487 170\$: MOVZBL -(R3),R7 ; Get next digit
 FEC9' 30 0134 488 MARK_POINT ASHP_BSBW_8
 58 D5 0137 489 BSBW- VAX\$ADD_PACKED_BYT_E_R6_R7 ; Perform the addition
 06 13 0139 490 TSTL R8 ; See if this add produced a carry
 F3 52 F4 013B 491 BEQL 180\$; All done if no more carry
 013E 492 SOBGEQ R2,170\$; Back for the next byte
 013E 493 ;
 013E 494 : If we drop through the end of the loop, then the final add produced a carry.
 013E 495 : This must be reflected by setting the V-bit in the saved PSW.
 013E 496 ;
 5B 02 88 013E 497 BISB #PSL\$M_V,R11 ; Set the saved V-bit
 0141 498 ;
 0141 499 : All of the digits are now loaded into the destination string. The condition
 0141 500 : codes, except for the Z-bit, have their correct settings. The sign must be
 0141 501 : set, a check must be made for even digit count in the output string, and
 0141 502 : the various special cases (negative zero, decimal overflow trap, ans so on)
 0141 503 : must be checked before completing the routine.
 0141 504 ;
 0141 505 : This entire routine worked with entire bytes, ignoring whether digit counts
 0141 506 : were odd or even. An illegal digit in the upper nibble of an even input string
 0141 507 : is ignored. A nonzero digit in the upper nibble of an even output string is
 0141 508 : not allowed but must be checked for. If one exists, it indicates overflow.
 0141 509 ;
 10 AE E8 0141 510 180\$: BLBS <8+ASHP_W_DSTLEN>(SP),-
 OF 0144 511 185\$; Skip next if output digit count is odd
 F0 8F 93 0145 512 MARK_POINT ASHP_8
 14 BE 0145 513 BITB #^B11110000,= ;
 08 13 0148 514 @<8+ASHP_A_DSTADDR>(SP) ; Is most significant digit nonzero?
 F0 8F 8A 014C 515 BEQL 185\$; Nothing to worry about if zero
 14 BE 014F 516 MARK_POINT ASHP_8
 5B 02 88 0151 517 BICB #^B11110000,= ;
 0154 518 @<8+ASHP_A_DSTADDR>(SP) ; Make the digit zero
 0154 519 BISB #PSL\$M_V,RT1 ; ... and set the overflow bit
 0154 520 ;
 0154 521 : We have not tested for nonzero digits in the output string. This test is
 0154 522 : made by making another pass over the ouptut string. Note that the low
 0154 523 : order digit is unconditionally checked.
 0154 524 ;
 52 6E 7D 0154 525 185\$: MOVQ (SP),R2 ; Get address and count
 73 F0 8F 93 0157 526 MARK_POINT ASHP_8
 19 12 015B 527 BITB #^B11110000,=(R3) ; Do not test sign in low order byte
 04 11 015D 528 BNEQ 187\$; Skip loop if nonzero
 BRB 186\$; Start at bottom of loop

```

      015F 530
      015F 531      MARK_POINT     ASHP_8
73 95 015F 532 183$: TSTB -(R3)      ; Is next higher byte nonzero?
13 12 0161 533 BNEQ 187$      ; Exit loop if yes
F9 52 F4 0163 534 186$: SOBGEQ R2,183$    ; Keep looking for nonzero if more bytes
      0166 535
      0166 536 : The entire output string has been scanned and contains no nonzero
      0166 537 : digits. The Z-bit retains its original setting, which is set. If the
      0166 538 : N-bit is also set, then the negative zero must be changed to positive
      0166 539 : zero (unless the V-bit is also set). Note that in the case of overflow,
      0166 540 : the N-bit is cleared but the output string retains the minus sign.
      0166 541
      0166 542 BBC #PSL$V_N,R11,190$      ; N-bit is off already
      016A 543 BICB #PSL$M_N,R11      ; Turn off saved N-bit unconditionally
      016D 544 BBS #PSL$V_V,R11,190$      ; No fixup if V-bit is also set
      0171 545 MOVB #12,R9      ; Use preferred plus as sign of output
      0174 546 BRB 190$      ; ... and rejoin the exit code
      0176 547
      0176 548 : The following instruction is the exit point for all of the nonzero byte
      0176 549 : checks. Its direct effect is to clear the saved Z-bit. It also bypasses
      0176 550 : whatever other zero checks have not yet been performed.
      0176 551
      0176 552 187$: BICB #PSL$M_Z,R11      ; Clear saved Z-bit
      0179 553
      0179 554 : The following code executes in all cases. It is the common exit path for
      0179 555 : all of the ASHP routines when the count is even.
      0179 556
      0179 557 190$: MOVQ (SP)+,R2      ; Get address of end of output string
      017C 558 MARK_POINT ASHP_0
      017C 559 INSV R9,#0,#4,-1(R3)      ; Store sign that we have been saving
      0182 560
      0182 561 .DISABLE LOCAL_BLOCK
      0182 562
      0182 563 ; Drop into VAX$DECIMAL_EXIT for final processing

```

0182 565 .SUBTITLE VAX\$DECIMAL_EXIT - Exit Path for Decimal Instructions
 0182 566 :+
 0182 567 This is the common exit path for many of the routines in this module. This
 0182 568 exit path can only be used for instructions that conform to the following
 0182 569 restrictions.
 0182 570
 0182 571 1. Registers R0 through R11 were saved on entry.
 0182 572
 0182 573 2. The architecture requires that R0 and R2 are zero on exit.
 0182 574
 0182 575 3. All other registers that have instruction-specific values on exit are
 0182 576 correctly stored in the appropriate locations on the stack.
 0182 577
 0182 578 4. The saved PSW is contained in R11
 0182 579
 0182 580 5. This instruction/routine should generate a decimal overflow trap if
 0182 581 both the V-bit and the DV-bit are set on exit.
 0182 582 :-
 0182 583
 0182 584 .ENABLE LOCAL_BLOCK
 0182 585
 0182 586 VAX\$DECIMAL_EXIT:::
 09 5B 08 AE 6E D4 0182 587 CLRE (SP) ; R0 must be zero on exit
 01 01 E0 D4 0184 588 CLRL 8(SP) ; R2 must also be zero
 OF B9 018B 589 BBS #PSL\$V_V,R11,20\$; See if exceptions are enabled
 5B B8 018D 590 10\$: BICPSW #<PSL\$M_N!PSLSM_Z!PSLSM_V!PSL\$M_C> ; Clear condition codes
 OFFF 8F BA 018F 591 BISPSW R11 ; Set appropriate condition codes
 05 0193 592 POPR #^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; Restore
 RSB ; ... and return
 0194 593
 0194 594 : If the V-bit is set and decimal traps are enabled (DV-bit is set), then
 0194 595 : a decimal overflow trap is generated. Note that the DV-bit can be set in
 0194 596 : the current PSL or, if this routine was entered as the result of an emulated
 0194 597 : instruction exception, in the saved PSL on the stack. Note that the final
 0194 598 : condition codes in the PSW have not yet been set. This means that all exit
 0194 599 : paths out of this code must set the condition codes to their correct values.
 0194 600
 0194 601
 0194 602 VAX\$EDITPC_OVERFLOW:::
 50 10 5B 07 E0 0194 603 20\$: BBS #PSL\$V_DV,R11,30\$; Report exception if current DV-bit set
 0000'CF 9E 0198 604 MOVAB VAX\$EXIT_EMULATOR,R0 ; Set up R0 for PIC address comparison
 30 AE 50 D1 019D 605 CMPL R0,<4*125(SP) ; Is return PC EQLU VAX\$EXIT_EMULATOR ?
 E3 60 AE 07 E1 01A1 606 BNEQU 10\$; No. Simply return V-bit set
 01A8 607 BBC #PSL\$V_DV,<<4*<12+1>>+EXCEPTION_PSL>(SP),10\$; Only return V-bit if DV-bit is clear
 01A8 608
 01A8 609
 01A8 610 ; Restore all of the saved registers, reset the condition codes, and drop
 01A8 611 ; into DECIMAL_OVERFLOW.
 01A8 612
 01A8 613 30\$: BICPSW #<PSL\$M_N!PSLSM_Z!PSLSM_V!PSL\$M_C> ; Clear condition codes
 5B B8 01A8 614 BISPSW R11 ; Set appropriate condition codes
 OFFF 8F BA 01AC 615 POPR #^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>

01B0 617 :+
01B0 618 This code path is entered if the decimal string result is too large to
01B0 619 fit into the output string and decimal overflow exceptions are enabled.
01B0 620 The final state of the instruction, including the condition codes, is
01B0 621 entirely in place.
01B0 622
01B0 623 Input Parameter:
01B0 624
01B0 625 (SP) - Return PC
01B0 626
01B0 627 Output Parameters:
01B0 628
01B0 629 0(SP) - SRM\$K_DEC_OVF_T (Arithmetic trap code)
01B0 630 4(SP) - Final state PSL
01B0 631 8(SP) - Return PC
01B0 632
01B0 633 Implicit Output:
01B0 634
01B0 635 Control passes through this code to VAX\$REFLECT_TRAP.
01B0 636 :-
01B0 637
01B0 638 VAX\$DECIMAL_OVERFLOW::
7E DC 01B0 639 MOVPSL -(SP) ; Save final PSL on stack
06 DD 01B2 640 PUSHL #SRM\$K_DEC_OVF_T ; Store arithmetic trap code
FE49' 31 01B4 641 BRW VAX\$REFLECT_TRAP ; Report exception
01B7 642
01B7 643 .DISABLE LOCAL_BLOCK

01B7 645 .SUBTITLE ASHP_COPY_SOURCE - Copy Source String to Work Area

01B7 646 + Functional Description:

01B7 647
01B7 648
01B7 649
01B7 650
01B7 651
01B7 652
01B7 653
01B7 654
01B7 655
01B7 656
01B7 657
01B7 658
01B7 659
01B7 660
01B7 661
01B7 662
01B7 663
01B7 664
01B7 665
01B7 666
01B7 667
01B7 668
01B7 669
01B7 670
01B7 671
01B7 672
01B7 673
01B7 674
01B7 675
01B7 676
01B7 677
01B7 678
01B7 679
01B7 680 :-
01B7 681
01B7 682 ASHP_COPY_SOURCE:

F8 A8 7C 01B7 683 F0 A8 7C 01BA 684 EC A8 D4 01BD 685 51 50 C1 01C0 686 51 58 D0 01C4 687 56 50 D0 01C7 688 01CA 689 01CA 690 71 77 90 01CA 691 FA 56 F5 01CD 692 01D0 693 05 01D0 694	C[RQ -8(R8) : Insure that the work area CLRQ -16(R8) ... is entirely filled CLRL -20(R8) ... with zeros ADDL3 R0,R1,R7 : R7 points one byte beyond source MOVL R8,R1 : R1 will step through work area MOVL R0,R6 : Use R6 as the loop counter MARK_POINT ASHP_BSBW_20 MOVB -(R7),-(R1) : Move the next source byte SOBGTR R6,10\$: Check for end of loop RSB : Return with R1 properly loaded
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

01D1 696 .SUBTITLE DECIMAL_ROPRAND
01D1 697 :-
01D1 698 Functional Description:
01D1 699 This routine receives control when a digit count larger than 31 is
01D1 700 detected. The exception is architecturally defined as an abort so
01D1 701 there is no need to store intermediate state. The VAX\$ASHP routine
01D1 702 saves all registers R0 through R11 before performing the digit check.
01D1 703 These registers must be restored before control is passed to
01D1 704 VAX\$ROPRAND.
01D1 705
01D1 706
01D1 707 Input Parameters:
01D1 708
01D1 709 00(SP) - Saved R0
01D1 710 .
01D1 711
01D1 712 44(SP) - Saved R11
01D1 713 48(SP) - Return PC from VAX\$xxxxxx routine
01D1 714
01D1 715 Output Parameters:
01D1 716
01D1 717 00(SP) - Offset in packed register array to delta PC byte
01D1 718 04(SP) - Return PC from VAX\$xxxxxx routine
01D1 719
01D1 720 Implicit Output:
01D1 721
01D1 722 This routine passes control to VAX\$ROPRAND where further
01D1 723 exception processing takes place.
01D1 724 :-
01D1 725
01D1 726 DECIMAL_ROPRAND:
0FFF 8F BA 01D1 727 POPR #^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
03 DD 01D5 728 PUSHL #ASHP B DELTA_PC : Store offset to delta PC byte
FE26' 31 01D7 729 BRW VAX\$ROPRAND : Pass control along

01DA 731
01DA 732 :+
01DA 733
01DA 734
01DA 735
01DA 736
01DA 737
01DA 738
01DA 739
01DA 740
01DA 741
01DA 742
01DA 743
01DA 744
01DA 745
01DA 746
01DA 747
01DA 748
01DA 749
01DA 750
01DA 751
01DA 752
01DA 753
01DA 754
01DA 755
01DA 756
01DA 757
01DA 758
01DA 759
01DA 760
01DA 761
01DA 762
01DA 763
01DA 764
01DA 765
01DA 766
01DA 767
01DA 768
01DA 769
01DA 770
01DA 771
01DA 772
01DA 773
01DA 774
01DA 775
01DA 776
01DA 777
01DA 778
01DA 779
01DA 780
01DA 781
01DA 782
01DA 783
01DA 784
01DA 785
01DA 786
01DA 787 :

.SUBTITLE ASHP_ACCVIO - Reflect an Access Violation

The general approach to handling access violations that occur while emulating the packed decimal instructions is described here. We take advantage of the fact that there are no architectural constraints on the way that access violations must be handled. In general, we back the instruction up to the beginning, to the point where initial state is stored in the set of general registers modified by each instruction. Thus, the only step that is avoided when an instruction is restarted is operand evaluation.

Functional Description:

This routine (or its counterpart in other decimal emulation modules) receives control when an access violation occurs while executing within one of the VAX\$xxxxxx routines that emulated a decimal string instruction. This routine determines whether the exception occurred while accessing a source or destination string or whether the access violation is peculiar to emulation, such as stack overflow. (This check is made based on the PC of the exception.)

If the PC is one that is recognized by this routine, then the state of the instruction (digit counts, string addresses, and the like) are restored to a state where the instruction/routine can be restarted after (if) the cause for the exception is eliminated. Control is then passed to a common routine that sets up the stack and the exception parameters in such a way that the instruction or routine can restart transparently.

If the exception occurs at some unrecognized PC, then the exception is reflected to the user as an exception that occurred within the emulator.

There are two exceptions that can occur that are not backed up to appear as if they occurred at the site of the original emulated instruction. These exceptions will appear to the user as if they occurred inside the emulator itself.

1. If stack overflow occurs due to use of the stack by one of the routines, it is unlikely that this routine will even execute because the code that transfers control here must first copy the parameters to the exception stack and that operation would fail. (The failure causes control to be transferred to VMS, where the stack expansion logic is invoked and the routine resumed transparently.)
2. If assumptions about the address space change out from under these routines (because an AST deleted a portion of the address space or a similar silly thing), the handling of the exception is UNPREDICTABLE.

Input Parameters:

R0 - Value of SP when exception occurred
R1 - PC at which exception occurred
R2 - scratch
R3 - scratch

01DA 788 : R10 - Address of this routine (no longer needed)
01DA 789 :
01DA 790 : 00(SP) - Value of R0 when exception occurred
01DA 791 : 04(SP) - Value of R1 when exception occurred
01DA 792 : 08(SP) - Value of R2 when exception occurred
01DA 793 : 12(SP) - Value of R3 when exception occurred
01DA 794 : 16(SP) - Return PC in exception dispatcher in operating system
01DA 795 :
01DA 796 : 20(SP) - First longword of system-specific exception data
01DA 797 :
01DA 798 :
01DA 799 : xx(SP) - Last longword of system-specific exception data
01DA 800 :
01DA 801 : The address of the next longword is the position of the stack when
01DA 802 : the exception occurred. R0 locates this address.
01DA 803 :
01DA 804 : R0 -> xx+4(SP) - Instruction-specific data
01DA 805 : . - Optional instruction-specific data
01DA 806 : . - Optional instruction-specific data
01DA 807 : xx+<4*M>(SP) - Return PC from VAX\$xxxxxx routine (M is the number
01DA 808 : of instruction-specific longwords)
01DA 809 :
01DA 810 : Implicit Input:
01DA 811 :
01DA 812 : It is assumed that the contents of all registers coming into this
01DA 813 : routine are unchanged from their contents when the exception occurred.
01DA 814 : (For R0 through R3, this assumption applies to the saved register
01DA 815 : contents on the top of the stack. Any modification to these four
01DA 816 : registers must be made to their saved copies and not to the registers
01DA 817 : themselves.)
01DA 818 :
01DA 819 : Finally, the macro BEGIN_MARK_POINT should have been invoked at the
01DA 820 : beginning of this module to define the symbols
01DA 821 :
01DA 822 : MODULE_BASE
01DA 823 : PC_TABLE_BASE
01DA 824 : HANDLER_TABLE_BASE
01DA 825 : TABLE_SIZE
01DA 826 :
01DA 827 : PC_TABLE_BASE is the base of a word array with one entry for each
01DA 828 : PC (relative to MODULE_BASE) that can cause an access
01DA 829 : violation that is capable of being backed up.
01DA 830 :
01DA 831 : HANDLER_TABLE_BASE is the base of a corresponding word array with an
01DA 832 : entry that locates the context specific code to handle each
01DA 833 : of the recognized access violations.
01DA 834 :
01DA 835 : Output Parameters (Exit via JMP instruction):
01DA 836 :
01DA 837 : If the exception is recognized (that is, if the exception PC is
01DA 838 : associated with one of the mark points), control is passed to the
01DA 839 : context-specific routine that restores the instruction state to
01DA 840 : its initial state.
01DA 841 :
01DA 842 : These are the register values and stack state when the context
01DA 843 : specific code begins execution.
01DA 844 :

01DA 845 : R0 - Value of SP when exception occurred
 01DA 846 : R1 - scratch
 01DA 847 : R2 - scratch
 01DA 848 : R3 - scratch
 01DA 849 : R10 - scratch
 01DA 850
 01DA 851 : R0 -> zz(SP) - Instruction-specific data begins here
 01DA 852
 01DA 853 : Implicit Output:
 01DA 854
 01DA 855 : The context-specific code accomplishes essentially the same thing for
 01DA 856 : all of the emulated instructions.
 01DA 857
 01DA 858 : The register contents are restored to the values that they had on
 01DA 859 : entry to the VAX\$xxxxxx routine. This causes the instruction to be
 01DA 860 : backed up almost (but not quite) to its starting point. (The operand
 01DA 861 : evaluation is not lost. The operands are saved in registers.) Any
 01DA 862 : registers saved on entry are restored.
 01DA 863
 01DA 864 : Output Parameters (Exit via RSB instruction):
 01DA 865
 01DA 866 : If the exception PC occurred somewhere else (such as a stack access),
 01DA 867 : the saved registers are restored and control is passed back to the
 01DA 868 : host system with an RSB instruction.
 01DA 869
 01DA 870 :-
 01DA 871
 01DA 872 ASHP_ACCVIO:
 FE20 52 D4 01DA 873 CLRL R2 : Initialize the counter
 CF 9F 01DC 874 PUSHAB MODULE_BASE : Store base address of this module
 0244'CF 9F 01E0 875 PUSHAB MODULE-END : Store module end address
 0020 30 01E4 876 BSBW DECIMA[\$BOUNDS_CHECK] : Check if PC is inside the module
 5E 04 C0 01E7 877 ADDL #4,SP : Discard end address
 51 8E C2 01EA 878 SUBL2 (SP)+,R1 : Get PC relative to this base
 01ED 879
 0000'CF42 51 B1 01ED 880 10\$: CMPW R1,PC_TABLE_BASE[R2] : Is this the right PC?
 07 13 01F3 881 BEQL 30\$: Exit loop if true
 F4 52 0E F2 01F5 882 AOBLLS #TABLE_SIZE,R2,10\$: Do the entire table
 01F9 883
 01F9 884 : If we drop through the dispatching based on PC, then the exception is not
 01F9 885 : one that we want to back up. We simply reflect the exception to the user.
 01F9 886
 OF BA 01F9 887 20\$: POPR #^M<R0,R1,R2,R3> : Restore saved registers
 05 01FB 888 RSB : Return to exception dispatcher
 01FC 889
 01FC 890 : The exception PC matched one of the entries in our PC table. R2 contains
 01FC 891 : the index into both the PC table and the handler table. R1 has served
 01FC 892 : its purpose and can be used as a scratch register.
 01FC 893
 51 0000'CF42 3C 01FC 894 30\$: MOVZWL HANDLER_TABLE_BASE[R2],R1 ; Get the offset to the handler
 FDF9'CF41 17 0202 895 JMP MODULE_BASE[RT] ; Pass control to the handler
 0207 896
 0207 897 : In all of the instruction-specific routines, the state of the stack
 0207 898 : will be shown as it was when the exception occurred. All offsets will
 0207 899 : be pictured relative to R0.

0207 901 .SUBTITLE DECIMAL\$BOUNDS_CHECK - Bounds Check on Exception PC
 0207 902 :+
 0207 903 Functional Description:
 0207 904
 0207 905 This routine is called by the exception handlers for the various
 0207 906 decimal string instruction emulation routines to perform a bounds
 0207 907 check on the exception PC. The real reason for performing this check
 0207 908 is that certain exceptions can occur in subroutines that are outside a
 0207 909 given module. In this case, it is not the exception PC but rather the
 0207 910 return PC on the top of the stack that determines the context of the
 0207 911 exception (and therefore, the code necessary to back up the
 0207 912 instruction state).
 0207 913
 0207 914 The basic mode of operation is that, if the exception PC is outside
 0207 915 the current module boundarise, then R1 (the exception PC) is replaced
 0207 916 by the return PC (presumed pointed to by R0).
 0207 917
 0207 918 Input Parameters:
 0207 919
 0207 920 R0 - Top of stack when exception occurred
 0207 921 R1 - PC at time of exception
 0207 922
 0207 923 (R0) - Return PC from subroutine in which access violation occurred
 0207 924
 0207 925 00(SP) - Return PC from caller of this routine
 0207 926 04(SP) - End address of module
 0207 927 08(SP) - Start address of module
 0207 928
 0207 929 Output Parameters:
 0207 930
 0207 931 If the exception PC is outside the bounds of the module (as defined by
 0207 932 the two longwords on the stack, then R1 is replaced by the "return
 0207 933 PC", the contents of the longword located by R0.
 0207 934
 0207 935 If the exception PC is inside the module, nothing is changed by the
 0207 936 execution of this module.
 0207 937
 0207 938 Assumptions:
 0207 939
 0207 940 There are two assumptions that must hold for these subroutines
 0207 941 in which access violations can occur.
 0207 942
 0207 943 They must not use the stack. This keeps the return PC on the
 0207 944 top of the stack, located by R0.
 0207 945
 0207 946 They must be called with a BSBW instruction. This causes the
 0207 947 return PC to be exactly three bytes beyond instruction that
 0207 948 transferred control to the subroutine.
 0207 949 :-
 0207 950
 0207 951 DECIMAL\$BOUNDS_CHECK::
 04 AE 51 D1 0207 952 CMPL R1,4(SP) ; Beyond upper end?
 07 07 1E 020B 953 BGEQU 10\$; Branch if out of bounds
 08 AE 51 D1 020D 954 CMPL R1,8(SP) ; Within lower limit?
 01 01 1F 0211 955 BLSSU 10\$; Branch if out of bounds
 05 05 0213 956 RSB ; Return with R1 intact
 0214 957

0214 958 : R1 is out of bounds. Replace it with the return PC from the routine that
0214 959 : was executing when the access violation occurred. Note that the PC is
0214 960 : backed up over the BSBW instruction because the PC offset that appears in
0214 961 : the PC_TABLE will be the PC of the BSBW instruction and not the PC of the
0214 962 ; next instruction.
0214 963

51 60 03 C3 0214 964 10\$: SUBL3 #3,(R0),R1 ; Get new "exception" PC
05 0218 965 RSB

0219 967 .SUBTITLE Context-Specific Access Violation Handling for VAX\$ASHP
0219 968 +
0219 969 Functional Description:
0219 970
0219 971 The only difference among the various entry points is the number of
0219 972 longwords on the stack. R0 is advanced beyond these longwords to point
0219 973 to the list of saved registers. These registers are then restored,
0219 974 effectively backing the routine up to its initial state.
0219 975
0219 976 Input Parameters:
0219 977
0219 978 R0 - Address of top of stack when access violation occurred
0219 979
0219 980 See specific entry points for details
0219 981
0219 982 Output Parameters:
0219 983
0219 984 See input parameter list for VAX\$DECIMAL_ACCVIO
0219 985 :-
0219 986
0219 987 +
0219 988 ASHP_BSBW_20
0219 989
0219 990 An access violation occurred in subroutine STRIP_ZEROS or in subroutine
0219 991 ASHP_COPY_SOURCE while the source string was being copied to the work space
0219 992 on the stack. In addition to the five longwords of work space on the stack,
0219 993 this routine has an additional longword, the return PC, on the stack.
0219 994
0219 995 00(R0) - Return PC in mainline of VAX\$ASHP
0219 996 04(R0) - First longword of scratch space
0219 997 etc.
0219 998 :-
0219 999
50 04 C0 0219 1000 ASHP_BSBW_20:
0219 1001 ADDL #4,R0 ; Skip over return PC and drop into ...
021C 1002
021C 1003 +
021C 1004 ASHP_20
021C 1005
021C 1006 There are five longwords of workspace on the stack for this entry point.
021C 1007
021C 1008 00(R0) - First longword of scratch space
021C 1009
021C 1010
021C 1011 16(R0) - Fifth longword of scratch space
021C 1012 20(SP) - Saved R0
021C 1013 24(SP) - Saved R1
021C 1014 etc.
021C 1015 :-
021C 1016
021C 1017 ASHP_20:
50 14 C0 021C 1018 ADDL #20,R0 ; Discard scratch space on stack
06 11 021F 1019 BRB VAX\$DECIMAL_ACCVIO ; Join common code to restore registers
0221 1020
0221 1021 +
0221 1022 ASHP_BSBW_8
0221 1023

0221 1024 : An access violation occurred in subroutine ADD PACKED BYTE while the round
0221 1025 operand was being propagated. In addition to the saved R2/R3 pair of
0221 1026 longwords on the stack, this routine has an additional longword, the return
0221 1027 PC, on the stack.
0221 1028
0221 1029 00(R0) - Return PC in mainline of VAX\$ASHP
0221 1030 04(R0) - Saved intermediate value of R2
0221 1031 etc.
0221 1032 :-
0221 1033
50 04 CO 0221 1034 ASHP_BSBW 8:
0221 1035 ADDL #4,R0 ; Skip over return PC and drop into ...
0224 1036
0224 1037 :+
0224 1038 ASHP_8
0224 1039
0224 1040 There is a saved register pair (two longwords) on the stack for these entry
0224 1041 points.
0224 1042
0224 1043 00(R0) - Saved intermediate value of R2
0224 1044 04(R0) - Saved intermediate value of R3
0224 1045 08(SP) - Saved R0
0224 1046 12(SP) - Saved R1
0224 1047 16(SP) - Saved R2
0224 1048 20(SP) - Saved R3
0224 1049 etc.
0224 1050 :-
0224 1051
50 08 CO 0224 1052 ASHP_8:
0224 1053 ADDL #8,R0 ; Discard saved register pair
0227 1054
0227 1055 : Drop into VAX\$DECIMAL_ACCVIO to restore saved registers
0227 1056
0227 1057 :+
0227 1058 ASHP_0
0227 1059
0227 1060 The stack is empty. This label is merely a synonym for VAX\$DECIMAL_ACCVIO
0227 1061 because there is no context-specific work to do.
0227 1062
0227 1063 00(SP) - Saved R0
0227 1064 04(SP) - Saved R1
0227 1065 08(SP) - Saved R2
0227 1066 12(SP) - Saved R3
0227 1067 etc.
0227 1068 :-
0227 1069
0227 1070 ASHP_0:
0227 1071
0227 1072 : Drop into VAX\$DECIMAL_ACCVIO to restore saved registers

0227 1074 .SUBTITLE VAX\$DECIMAL_ACCVIO - Common Access Violation Handling

0227 1075 :- Functional Description:

0227 1077 This code is the final access violation processing for those
0227 1078 exceptions that have two things in common.

0227 1081 The instruction/routine is to be backed up to its initial state.

0227 1082 All registers from R0 to R11 were saved on entry to VAX\$xxxxxx.

0227 1084 Input Parameters:

0227 1086 00(R0) - Saved R0 on entry to VAX\$xxxxxx

0227 1088 04(R0) - Saved R1

0227 1089 .

0227 1091 44(R0) - Saved R11 on entry to VAX\$xxxxxx

0227 1092 48(R0) - Return PC from VAX\$xxxxxx routine

0227 1093 00(SP) - Saved R0 (restored by VAX\$HANDLER)

0227 1094 04(SP) - Saved R1

0227 1095 08(SP) - Saved R2

0227 1096 12(SP) - Saved R3

0227 1098 Output Parameters:

0227 1100 R0 is advanced over saved register array as the registers are restored.
0227 1101 R0 ends up pointing at the return PC.

0227 1102 R1 contains the value of delta PC for all of the routines that
0227 1103 use this common code path. The FPD and ACCVIO bits are both set
in R1.

0227 1104 00(R0) - Return PC from VAX\$xxxxxx routine

0227 1105 00(SP) - Value of R0 on entry to VAX\$xxxxxx

0227 1106 04(SP) - Value of R1 on entry to VAX\$xxxxxx

0227 1107 08(SP) - Value of R2 on entry to VAX\$xxxxxx

0227 1108 12(SP) - Value of R3 on entry to VAX\$xxxxxx

0227 1109 R4 through R11 are restored to their values on entry to VAX\$xxxxxx.

0227 1110 :-

0227 1111

0227 1112

0227 1113

0227 1114

0227 1115

0227 1116 :-

0227 1117

0227 1118 VAX\$DECIMAL_ACCVIO::

08 6E 80 7D 0227 1119	MOVQ (R0)+,PACK_L_SAVED_R0(SP)	: "Restore" R0 and R1
AE 80 7D 022A 1120	MOVQ (R0)+,PACK_L_SAVED_R2(SP)	: "Restore" R2 and R3
54 80 7D 022E 1121	MOVQ (R0)+,R4	: Really restore R4 and R5
56 80 7D 0231 1122	MOVQ (R0)+,R6	: ... and R6 and R7
58 80 7D 0234 1123	MOVQ (R0)+,R8	: ... and R8 and R8
5A 80 7D 0237 1124	MOVQ (R0)+,R10	: ... and R10 and R11

023A 1125

023A 1126

023A 1127

023A 1128

023A 1129

023A 1130

ASSUME ADDP4_B_DELTA_PC EQ ASHP_B_DELTA_PC

ASSUME ADDP6_B_DELTA_PC EQ ASHP_B_DELTA_PC

ASSUME SUBP4_B_DELTA_PC EQ ASHP_B_DELTA_PC

ASSUME SUBP6_B_DELTA_PC EQ ASHP_B_DELTA_PC

ASSUME MULP_B_DELTA_PC EQ ASHP_B_DELTA_PC

023A 1131 ASSUME DIVP_B_DELTA_PC EQ ASHP_B_DELTA_PC
023A 1132 ASSUME CVTPS_B_DELTA_PC EQ ASHP_B_DELTA_PC
023A 1133 ASSUME CVTPT_B_DELTA_PC EQ ASHP_B_DELTA_PC
023A 1134 ASSUME CVTSP_B_DELTA_PC EQ ASHP_B_DELTA_PC
023A 1135 ASSUME CVTTP_B_DELTA_PC EQ ASHP_B_DELTA_PC
023A 1136 ASSUME CVTTP_B_DELTA_PC EQ ASHP_B_DELTA_PC
023A 1137
51 00000303 8F D0 023A 1138 MOVL #<ASHP_B_DELTA_PC!~ : Indicate offset for delta PC
0241 1139 PACK M_FPD!- : FPD bit should be set
FDBC' 31 0241 1140 PACK M_ACCVIO>,R1 : This is an access violation
0241 1141 BRW VAX\$REFLECT_FAULT : Continue exception handling
0244 1142 END_MARK_POINT
0244 1143
0244 1144
0244 1145 .END

G 15

- VAX-11 Instruction Emulator for ASHP

16-SEP-1984 01:31:43 VAX/VMS Macro V04-00
5-SEP-1984 00:44:46 [EMULAT.SRC]VAXASHP.MAR;1

Page 26
(12)

...PC...	= 000001CA		VAX\$REFLECT_FAULT	***** X 00
...ROPRAND...	= 00000015	R 02	VAX\$REFLECT_TRAP	***** X 00
ADDP4_B_DELTA_PC	= 00000003		VAX\$ROPRAND	***** X 00
ADDP6_B_DELTA_PC	= 00000003			
ASHP_0	00000227	R 02		
ASHP_20	0000021C	R 02		
ASHP_8	00000224	R 02		
ASHP_ACCVIO	000001DA	R 02		
ASHP_A_DSTADDR	= 0000000C			
ASHP_BSBW_20	00000219	R 02		
ASHP_BSBW_8	00000221	R 02		
ASHP_B_CNT	= 00000002			
ASHP_B_DELTA_PC	= 00000003			
ASHP_B_ROUND	= 0000000A			
ASHP_COPY_SOURCE	000001B7	R 02		
ASHP_SHIFT_MASK	= F0F0F0F0			
ASHP_SHIFT_NEGATIVE	000000CB	R 02		
ASHP_SHIFT_POSITIVE	000000AA	R 02		
ASHP_S_ROUND	= 00000004			
ASHP_V_ROUND	= 00000000			
ASHP_W_DSTLEN	= 00000008			
CVTP5_B_DELTA_PC	= 00000003			
CVTPT_B_DELTA_PC	= 00000003			
CVTSP_B_DELTA_PC	= 00000003			
CVTTP_B_DELTA_PC	= 00000003			
DECIMAL\$BOUNDS_CHECK	00000207	RG 02		
DECIMAL\$STRIP_ZEROS_R0_R1	***** X 00			
DECIMAL_ROPRAND	000001D1	R 02		
DIVP_B_DELTA_PC	= 00000003			
EXCEPTION_PSC	= 0000002C			
HANDLER_TABLE_BASE	00000000	R 04		
MODULE_BASE	= 00000000	R 02		
MODULE_END	= 0000244	R 02		
MULP_B_DELTA_PC	= 00000003			
PACK_L_SAVED_R0	= 00000000			
PACK_L_SAVED_R2	= 00000008			
PACK_M_ACCVIO	= 00000200			
PACK_M_FPD	= 00000100			
PC_TABLE_BASE	00000000	R 03		
PSLSM_C	= 00000001			
PSLSM_N	= 00000008			
PSLSM_V	= 00000002			
PSLSM_Z	= 00000004			
PSL\$V_DV	= 00000007			
PSL\$V_N	= 00000003			
PSL\$V_V	= 00000001			
SRMSK_DEC_OVF_T	= 00000006			
SUBP4_B_DELTA_PC	= 00000003			
SUBP6_B_DELTA_PC	= 00000003			
TABLE_SIZE	= 0000000E			
VAX\$ADD_PACKED_BYTE_R6_R7	***** X 00			
VAX\$ASHP	00000000	RG 02		
VAX\$DECIMAL_ACCVIO	00000227	RG 02		
VAX\$DECIMAL_EXIT	00000182	RG 02		
VAX\$DECIMAL_OVERFLOW	000001B0	RG 02		
VAX\$EDITPC_OVERFLOW	00000194	RG 02		
VAX\$EXIT_EMULATOR	***** X 00			

+-----+
! Psect synopsis !
+-----+

PSECT name	Allocation	PSECT No.	Attributes
ABS .	00000000 (0.)	00 (0.)	NOPIC USR CON ABS LCL NOSHR NOEXE NORD NOWRT NOVEC BYTE
\$ABSS	00000000 (0.)	01 (1.)	NOPIC USR CON ABS LCL NOSHR EXE RD WRT NOVEC BYTE
VAX\$CODE	00000244 (580.)	02 (2.)	PIC USR CON REL LCL SHR EXE RD NOWRT NOVEC LONG
PC_TABLE	0000001C (28.)	03 (3.)	PIC USR CON REL LCL SHR NOEXE RD NOWRT NOVEC BYTE
HANDLER_TABLE	0000001C (28.)	04 (4.)	PIC USR CON REL LCL SHR NOEXE RD NOWRT NOVEC BYTE

+-----+
! Performance indicators !
+-----+

Phase	Page faults	CPU Time	Elapsed Time
Initialization	16	00:00:00.03	00:00:02.51
Command processing	74	00:00:00.47	00:00:05.74
Pass 1	155	00:00:04.64	00:00:24.66
Symbol table sort	0	00:00:00.33	00:00:01.46
Pass 2	204	00:00:02.24	00:00:11.10
Symbol table output	7	00:00:00.07	00:00:00.30
Psect synopsis output	3	00:00:00.03	00:00:00.03
Cross-reference output	0	00:00:00.00	00:00:00.00
Assembler run totals	459	00:00:07.81	00:00:45.80

The working set limit was 1200 pages.
 26813 bytes (53 pages) of virtual memory were used to buffer the intermediate code.
 There were 20 pages of symbol table space allocated to hold 258 non-local and 37 local symbols.
 1145 source lines were read in Pass 1, producing 19 object records in Pass 2.
 30 pages of virtual memory were used to define 28 macros.

+-----+
! Macro library statistics !
+-----+

Macro library name	Macros defined
\$255\$DUA28:[EMULAT.OBJ]VAXMACROS.MLB;1	19
\$255\$DUA28:[SYSLIB]STARLET.MLB;2	6
TOTALS (all libraries)	25

429 GETS were required to define 25 macros.

There were no errors, warnings or information messages.

MACRO/LIS=LI\$:\$:VAXASHP/OBJ=OBJ\$:\$:VAXASHP MSRC\$:\$:VAXASHP/UPDATE=(ENH\$:\$:VAXASHP)+LIB\$:\$:VAXMACROS/LIB

0143 AH-BT13A-SE
VAX/VMS V4.0

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY

UPLOAD
LIS

VAXASHP
LIS

VAXCONVT
LIS

VAXARITH
LIS