


```

LL          IIIIII  BBBB8888  SSSSSSSS  IIIIII  MM      MM  TTTTTTTTTT  RRRRRRRR  AAAAAA
LL          IIIIII  BBBB8888  SSSSSSSS  IIIIII  MM      MM  TTTTTTTTTT  RRRRRRRR  AAAAAA
LL          II      88      88  SS      II      MMMM  MMMM  TT      RR      RR  AA      AA
LL          II      88      88  SS      II      MMMM  MMMM  TT      RR      RR  AA      AA
LL          II      88      88  SS      II      MM  MM  MM  TT      RR      RR  AA      AA
LL          II      88      88  SS      II      MM  MM  MM  TT      RRRRRRRR  AA      AA
LL          II      88      88  SS      II      MM  MM  MM  TT      RRRRRRRR  AA      AA
LL          II      88      88  SS      II      MM  MM  MM  TT      RR  RR  AAAAAAAAAA
LL          II      88      88  SS      II      MM  MM  MM  TT      RR  RR  AAAAAAAAAA
LL          II      88      88  SS      II      MM  MM  MM  TT      RR      RR  AA      AA
LL          II      88      88  SS      II      MM  MM  MM  TT      RR      RR  AA      AA
LLLLLLLLLL IIIIII  BBBB8888  SSSSSSSS  IIIIII  MM      MM  TT      RR      RR  AA      AA
LLLLLLLLLL IIIIII  BBBB8888  SSSSSSSS  IIIIII  MM      MM  TT      RR      RR  AA      AA

```

```

LL          IIIIII  SSSSSSSS
LL          IIIIII  SSSSSSSS
LL          II      SS
LL          II      SS
LL          II      SS
LL          II      SS
LL          II      SSSSSS
LL          II      SSSSSS
LL          II      SS
LL          II      SS
LL          II      SS
LL          II      SS
LLLLLLLLLL IIIIII  SSSSSSSS
LLLLLLLLLL IIIIII  SSSSSSSS

```

(2)	44	Edit History
(3)	52	DECLARATIONS
(4)	86	LIBSSIM_TRAP - Convert Floating Faults to Traps

```
0000 1 .TITLE LIBSSIM_TRAP - Simulate floating trap
0000 2 .IDENT 71-003/ ; File: LIBSIMTRA.MAR Edit: SBL1003
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: General Utility Library
0000 31 :
0000 32 : ABSTRACT:
0000 33 :
0000 34 : LIBSSIM_TRAP is a routine which converts floating faults
0000 35 : to floating traps. It is designed to be enabled as a
0000 36 : condition handler or to be called by one.
0000 37 :
0000 38 : ENVIRONMENT: User Mode, AST Reentrant
0000 39 :
0000 40 :--
0000 41 : AUTHOR: Derek Zave, CREATION DATE: 6-Dec-1979
0000 42 :
```

LIBSSIM_TRAP
1-003

- Simulate floating trap
Edit History

M 16

16-SEP-1984 00:18:50 VAX/VMS Macro V04-00
6-SEP-1984 11:10:54 [LIBRTL.SRC]LIBSIMTRA.MAR;1

Page 2
(2)

```
0000 44      .SBTTL Edit History
0000 45
0000 46 : 1-001 - Original. 6-Dec-1979
0000 47 : 1-002 - Remove restriction that stack frames must be in P1 space.
0000 48 :      SBL 22-May-1980
0000 49 : 1-003 - Change to using SYSSRCHANDLER mechanism to look for handlers.
0000 50 :
```

```
0000 52      .SBTTL  DECLARATIONS
0000 53      :
0000 54      : INCLUDE FILES:
0000 55      :
0000 56      :
0000 57      :
0000 58      : EXTERNAL DECLARATIONS:
0000 59      :
0000 60      .DSABL  GBL                      ; Prevent undeclared
0000 61      :                                           ; symbols from being
0000 62      :                                           ; automatically global.
0000 63      .EXTRN SYSSCALL HANDL          ; System routine that calls handlers
0000 64      .EXTRN SYSSRCHANDLER          ; System routine that looks for handlers
0000 65      .EXTRN SYSSUNWIND              ; $UNWIND system service
0000 66      :
0000 67      :
0000 68      : MACROS:
0000 69      :
0000 70      :
0000 71      :
0000 72      : EQUATED SYMBOLS:
0000 73      :
0000 74      :
0000 75      :
0000 76      : OWN STORAGE:
0000 77      :
0000 78      :
0000 79      :
0000 80      : PSECT DECLARATIONS:
0000 81      :
0000 82      .PSECT _LIB$CODE PIC,USR,CON,REL,LCL,SHR,-
0000 83      EXE, RD, NOWRT, LONG
0000 84
```

```
0000 86 .SBTTL LIBSSIM_TRAP - Convert Floating Faults to Traps
0000 87 :++
0000 88 : FUNCTIONAL DESCRIPTION:
0000 89 :
0000 90 : This routine functions as a condition handler which intercepts
0000 91 : floating overflow, floating underflow, and floating divide by zero
0000 92 : faults. When these conditions are detected, the routine simulates the
0000 93 : instruction causing the condition up to the point where a trap should
0000 94 : be signaled and signals the corresponding floating trap.
0000 95 :
0000 96 :
0000 97 : Introduction
0000 98 : -----
0000 99 :
0000 100 : With Revision 5 of the VAX System Reference Manual, incompatible
0000 101 : changes were made to the VAX architecture in regard to the exceptions
0000 102 : that may occur in connection with floating arithmetic operations.
0000 103 : The exceptions generated by floating underflow, floating overflow,
0000 104 : and floating divide by zero were changed from traps to faults and new
0000 105 : hardware vectors were defined for the new faults. In VMS new condition
0000 106 : codes were defined which correspond to the new hardware-defined
0000 107 : conditions. Since fault exceptions always leave the environment in such
0000 108 : a state that the faulted instruction can be restarted, it is possible,
0000 109 : in principle, to "convert" one of the new exceptions to the exception
0000 110 : it corresponds to under the earlier VAX architecture.
0000 111 :
0000 112 : The routine LIBSSIM_TRAP has been developed to provide a generally
0000 113 : usable facility for converting the new floating faults to the earlier
0000 114 : floating traps so that existing code for handling floating exceptions
0000 115 : will still work. The routine may be used as a condition handler or it
0000 116 : may be called by a condition handler (perhaps with intermediate
0000 117 : procedure calls) to examine a signaled condition. When one of the new
0000 118 : floating faults occurs, the routine "shuts down" the current condition
0000 119 : handling operation, performs any necessary changes to the environment,
0000 120 : and signals the trap condition.
0000 121 :
0000 122 : No effort has been spared to make the traps look exactly like
0000 123 : those generated by the hardware and intercepted by the operating
0000 124 : system. Particular attention has been paid to the problems of
0000 125 : dealing with floating underflow exceptions for the POLYx instructions
0000 126 : which require resuming the instruction.
0000 127 :
0000 128 :
0000 129 : Operation of the Fault to Trap Conversion
0000 130 : -----
0000 131 :
0000 132 : LIBSSIM_TRAP is concerned with instructions which can cause any
0000 133 : of the following three exceptions under the revised VAX architecture:
0000 134 :
0000 135 : SSS_FLTUND_F floating underflow fault
0000 136 : SSS_FLTDIV_F floating divide by zero fault
0000 137 : SSS_FLTOVF_F floating overflow fault
0000 138 :
0000 139 : When detected these faults will be converted to the following
0000 140 : corresponding traps:
0000 141 :
0000 142 : SSS_FLTUND floating underflow trap
```

```

0000 143 :          SSS_FLTDIV      floating/decimal divide by zero trap
0000 144 :          SSS_FLTOVF      floating overflow trap
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 :
0000 178 :
0000 179 :
0000 180 :
0000 181 :
0000 182 :
0000 183 :
0000 184 :
0000 185 :
0000 186 :
0000 187 :
0000 188 :
0000 189 :
0000 190 :
0000 191 :
0000 192 :
0000 193 :
0000 194 :
0000 195 :
0000 196 :
0000 197 :
0000 198 :
0000 199 :

```

In the course of performing the conversion other exceptions may be detected and signaled instead. The possible exceptions are

```

          SSS_ACCVIO      access violation fault
          SSS_ROPRAND     reserved operand fault

```

in addition to the above three traps.

The instructions which can cause the three faults which are recognize are listed below.

```

ACBD      add compare and branch D_floating
ACBF      add compare and branch F_floating
ACBG      add compare and branch G_floating
ACBH      add compare and branch H_floating
ADDD2     add D_floating (two operands)
ADDD3     add D_floating (three operands)
ADDF2     add F_floating (two operands)
ADDF3     add D_floating (three operands)
ADDG2     add G_floating (two operands)
ADDG3     add G_floating (three operands)
ADDH2     add H_floating (two operands)
ADDH3     add H_floating (three operands)
CVTDF     convert D_floating to F_floating
CVTGF     convert G_floating to F_floating
CVTHD     convert H_floating to D_floating
CVTHF     convert H_floating to F_floating
CVTHG     convert H_floating to G_floating
DIVD2     divide D_floating (two operands)
DIVD3     divide D_floating (three operands)
DIVF2     divide F_floating (two operands)
DIVF3     divide F_floating (three operands)
DIVG2     divide G_floating (two operands)
DIVG3     divide G_floating (three operands)
DIVH2     divide H_floating (two operands)
DIVH3     divide H_floating (three operands)
EMODD     extended modulus D_floating
EMODF     extended modulus F_floating
EMODG     extended modulus G_floating
EMODH     extended modulus H_floating
MULD2     multiply D_floating (two operands)
MULD3     multiply D_floating (three operands)
MULF2     multiply F_floating (two operands)
MULF3     multiply F_floating (three operands)
MULG2     multiply G_floating (two operands)
MULG3     multiply G_floating (three operands)
MULH2     multiply H_floating (two operands)
MULH3     multiply H_floating (three operands)
POLYD     evaluate polynomial D_floating
POLYF     evaluate polynomial F_floating
POLYG     evaluate polynomial G_floating
POLYH     evaluate polynomial H_floating
SUBD2     subtract D_floating (two operands)

```

```

0000 200 : SUBD3 subtract D_floating (three operands)
0000 201 : SUBF2 subtract F_floating (two operands)
0000 202 : SUBF3 subtract F_floating (three operands)
0000 203 : SUBG2 subtract G_floating (two operands)
0000 204 : SUBG3 subtract G_floating (three operands)
0000 205 : SUBH2 subtract H_floating (two operands)
0000 206 : SUBH3 subtract H_floating (three operands)
0000 207 :
0000 208 :
0000 209 :
0000 210 :
0000 211 :
0000 212 :
0000 213 :
0000 214 :
0000 215 :
0000 216 :
0000 217 :
0000 218 :
0000 219 :
0000 220 :
0000 221 :
0000 222 :
0000 223 :
0000 224 :
0000 225 :
0000 226 :
0000 227 :
0000 228 :
0000 229 :
0000 230 :
0000 231 :
0000 232 :
0000 233 :
0000 234 :
0000 235 :
0000 236 :
0000 237 :
0000 238 :
0000 239 :
0000 240 :
0000 241 :
0000 242 :
0000 243 :
0000 244 :
0000 245 :
0000 246 :
0000 247 :
0000 248 :
0000 249 :
0000 250 :
0000 251 :
0000 252 :
0000 253 :
0000 254 :
0000 255 :
0000 256 :

```

The conversion of faults to traps is performed as follows: First the instruction operands are scanned and any increments or decrements of the registers required by the operand specifiers is performed. Also, the operands or operand locations are determined during this scan. For all of the faults, a value of zero is stored for written integer operands. For floating underflow, a zero value is also stored for each written floating operand. For floating overflow and floating divide by zero, a reserved floating value which is zero except for a sign bit of one is stored for each floating written operand. The condition codes are set to describe the last result stored. Finally, the trap condition corresponding to the fault is signaled.

There are some minor exceptions to the above outline. For the ACBx instructions, the carry bit in the condition code is preserved and the routine determines whether or not the branch should be taken. For the POLYx instructions, things are a bit more complicated. For a floating overflow a reserved operand is stored in the register or registers intended to hold the result, the condition codes are set to describe it, and the trap is signaled. For floating underflow, however, the result so far is cleared and the instruction is resumed and run to completion if possible. If the instruction is completed, then the underflow trap is signaled. However, an overflow, access violation, or a reserved operand fault, may occur (the last two originating from problems with the table of coefficients). If an overflow occurs, the condition will be treated as if the underflow had never occurred. If an access violation or a reserved operand fault occurs, then the fault will be signaled exactly as described by the architecture with the FPD bit set in the PSL so the instruction can be resumed.

Since the routine LIBSSIM_TRAP "dissolves" the condition handling for the original fault condition, the final condition signaled by the routine will be from the context of the instruction itself rather than from that of the condition handler so the signaling path will be identical to that for a hardware generated trap. The signal array will be correctly placed so that the end of this table will be the user's stack pointer at the completion of the instruction (for traps) or at the beginning of the instruction (for faults).

Differences from the Hardware

Below is what is believed to be a complete list of differences between LIBSSIM_TRAP and the hardware that are in any way detectable by the user. We have not included differences which are related to whether or not the converter is being used rather than to the operation of the converter itself.

0000 257 :
0000 258 :
0000 259 :
0000 260 :
0000 261 :
0000 262 :
0000 263 :
0000 264 :
0000 265 :
0000 266 :
0000 267 :
0000 268 :
0000 269 :
0000 270 :
0000 271 :
0000 272 :
0000 273 :
0000 274 :
0000 275 :
0000 276 :
0000 277 :
0000 278 :
0000 279 :
0000 280 :
0000 281 :
0000 282 :
0000 283 :
0000 284 :
0000 285 :
0000 286 :
0000 287 :
0000 288 :
0000 289 :
0000 290 :
0000 291 :
0000 292 :
0000 293 :
0000 294 :
0000 295 :
0000 296 :
0000 297 :
0000 298 :
0000 299 :
0000 300 :
0000 301 :
0000 302 :
0000 303 :
0000 304 :
0000 305 :
0000 306 :
0000 307 :
0000 308 :
0000 309 :
0000 310 :
0000 311 :
0000 312 :
0000 313 :

1. The simulation of the instruction from the fault up to the point where the trap is signaled takes much longer than the hardware.
2. The simulation of the instruction from the fault up to the point where the trap is signaled is performed by software so it is interruptable. Because of this an AST which becomes active while the simulation is taking place may find the actions of the instruction only partially completed.
3. When fault to trap conversion takes place, the converter garbages the area below the user's stack pointer. This area is used to simulate a portion of the user's stack as well as for local storage for the converter itself. The converter protects itself from stores into its working storage in such a way that it appears that the results of the stores were garbaged after the store.
4. When the converter is active, there is an extra procedure frame on the stack. This might come as an unexpected surprise to an AST that becomes active while the converter is running.
5. When the converter signals faults and traps there may be some differences in the undocumented aspects of the blocks that are pushed onto the stack for signaling.

Notes on this Version of LIBSSIM_TRAP

The following notes apply to this version of LIBSSIM_TRAP and may change in subsequent versions.

1. This version was completed in a hurry and is not fully tested. All of the mechanisms and paths have been tested but not enough examples have been tried to expose interaction bugs.
2. In order to implement the converter with only a modest amount of effort the following assumptions were made about the architecture. These assumptions are not clearly justified by Revision 5 of the SRM but Dileep Bhandarkar tells me that these points will be clarified in future version of the SRM. Curiously both of these points involve the POLYx instructions.
 - A. It is assumed that when a POLYx instruction causes and underflow or overflow fault, that the instruction will be suspended with the FPD bit set in the PSL. The SRM accurately describes the suspended state but makes no guarantees as to when the state is ever reached.
 - B. It is assumed that when a POLYx instruction has been suspended that the user has unlimited liscence to change the result so far, the argument, the number of remaining coefficients, and the location of the next coefficient, but not the implementation specific

```

0000 314 :
0000 315 :
0000 316 :
0000 317 :
0000 318 :
0000 319 :
0000 320 :
0000 321 :
0000 322 :
0000 323 :
0000 324 :
0000 325 :
0000 326 :
0000 327 :
0000 328 :
0000 329 :
0000 330 :
0000 331 :
0000 332 :
0000 333 :
0000 334 :
0000 335 :
0000 336 :
0000 337 :
0000 338 :
0000 339 :
0000 340 :
0000 341 :
0000 342 :
0000 343 :
0000 344 :
0000 345 :
0000 346 :
0000 347 :
0000 348 :
0000 349 :
0000 350 :
0000 351 :
0000 352 :
0000 353 :
0000 354 :
0000 355 :
0000 356 :
0000 357 :
0000 358 :
0000 359 :
0000 360 :
0000 361 :
0000 362 :
0000 363 :
0000 364 :
0000 365 :
0000 366 :
0000 367 :
0000 368 :
0000 369 :
0000 370 :

```

information. The number of remaining coefficients can not be changed to a reserved value or zero. These assumptions imply that the implementation specific information does not depend on these quantities.

3. There is no version of the architecture which specifies that traps occur for floating arithmetic exceptions generated by the G-format and H-format instructions. Nevertheless, the converter performs such a conversion since it that the primary reason for this converter is to allow user's to continue to use their old exception handling routines.
4. The converter does not check for a stack overflow while it is running. Because of the automatic stack expansion feature in Release 2 VMS this should not be a problem. If the user nails his own stack pointer during the simulation (the only values that can be stored into it are zero and 2^{15}), then a reserved operand exception will occur when the converter tries to return.
5. The converter interacts smoothly with the Debugger.
6. The converter is read-only and PIC so there should be no difficulty in incorporating it into any program.

How to Install Floating Trap Simulation

In order to insure that the new floating faults are converted to the old traps, the routine LIBSSIM_TRAP must be used as a condition handler or must be called from a condition handler. If it is used as a condition handler it must be positioned so that it will be notified about conditions generated by the code in question. This will be the case if it is the primary, secondary, or last chance condition handler. If it is installed as the condition handler for some routine, then that routine must be one which executes the instructions which will cause the condition either directly or through a chain of intermediate procedure calls. If the routine LIBSSIM_TRAP is to be called by a condition handler, then that condition handler must have the properties described above. Condition handlers which "cover" the entire program may be set up using the LIB\$INITIALIZE facility described in Appendix E of the Common RTL Reference Manual.

When LIBSSIM_TRAP is called from a condition handler the call should have the following form:

```
CALL LIBSSIM_TRAP (signal_args_adr, mech_args_adr)
```

However it is called, the routine examines the condition to determine if it is one of the faults that are converted to traps. If it is not then the routine returns with R0 containing the condition code `SS$RESIGNAL`. A condition handler which calls LIBSSIM_TRAP should be coded so that it will not leave any loose ends dangling if the call to LIBSSIM_TRAP does not return.

If the routine chooses to handle the condition, all of the

LIBSSIM_TRAP
1-003

H 1
- Simulate floating trap
LIBSSIM_TRAP - Convert Floating Faults t 16-SEP-1984 00:18:50 VAX/VMS Macro V04-00 Page 9
6-SEP-1984 11:10:54 [LIBRTL.SRC]LIBSIMTRA.MAR;1 (4)

```
0000 371 : procedure frames up to and including the handler frame are removed
0000 372 : from the stack and the registers are restored in order to recreate the
0000 373 : stack and register values at the time of the exception. The conversion
0000 374 : is then performed and the resulting condition is signaled. For this
0000 375 : reason the resulting condition will be signaled as a new condition and
0000 376 : not as a continuation of the original fault condition.
0000 377 :
```

LIB
1-0

```
0000 379 ..... *****
0000 380 ..... *
0000 381 ..... *
0000 382 ..... *
0000 383 ..... *
0000 384 ..... *
0000 385 ..... *****
0000 386 .....
0000 387 .....
0000 388 ..... Definition Macro Invocations
0000 389 .....
0000 390 $SSDEF ; System Status Codes
0000 391 $JPIDEF ; Job/Process Information Codes
0000 392 .....
0000 393 ..... Parameters
0000 394 .....
00000028 0000 395 CALL_ARGS = 40 ; flexible stack space (longwords)
0000 396 .....
0000 397 ..... Bits in the Processor Status Longword (PSL)
00000000 0000 398 .....
00000001 0000 399 PSL_C = 0 ; carry indicator
00000002 0000 400 PSL_V = 1 ; overflow indicator
00000003 0000 401 PSL_Z = 2 ; zero indicator
00000004 0000 402 PSL_N = 3 ; negative indicator
00000006 0000 403 PSL_T = 4 ; trace enable indicator
00000018 0000 404 PSL_FU = 6 ; floating underflow fault enable
0000001B 0000 405 PSL_CAM = 24 ; current access mode
0000001E 0000 406 PSL_FPD = 27 ; instruction first part done
0000001E 0000 407 PSL_TP = 30 ; trace pending indicator
0000 408 .....
0000 409 ..... Masks for the Processor Status Longword
00000001 0000 410 .....
00000001 0000 411 PSLM_C = 1@PSL_C ; carry indicator
00000002 0000 412 PSLM_V = 1@PSL_V ; overflow indicator
00000004 0000 413 PSLM_Z = 1@PSL_Z ; zero indicator
00000008 0000 414 PSLM_N = 1@PSL_N ; negative indicator
0000000C 0000 415 PSLM_NZ = PSLM_N+PSLM_Z ; reserved floating value condition
00000007 0000 416 PSLM_ZVC = PSLM_Z+PSLM_V+PSLM_C ; condition bits except for sign
0000000F 0000 417 PSLM_NZVC = PSLM_N+PSLM_ZVC ; condition bits
0000 418 .....
0000 419 ..... Call Frame Layout
0000 420 .....
00000000 0000 421 HANDLER = 0 ; condition handler location
00000004 0000 422 SAVE_PSW = 4 ; save processor status word
00000006 0000 423 SAVE_MASK = 6 ; register save mask
0000000E 0000 424 MASK_ALIGN = 14 ; bit position of alignment bits
00000008 0000 425 SAVE_AP = 8 ; user's argument pointer
0000000C 0000 426 SAVE_FP = 12 ; user's frame pointer
00000010 0000 427 SAVE_PC = 16 ; return point
00000014 0000 428 REG_R0 = 20 ; user's R0
00000018 0000 429 REG_R1 = 24 ; user's R1
0000001C 0000 430 REG_R2 = 28 ; user's R2
00000020 0000 431 REG_R3 = 32 ; user's R3
00000024 0000 432 REG_R4 = 36 ; user's R4
00000028 0000 433 REG_R5 = 40 ; user's R5
0000002C 0000 434 REG_R6 = 44 ; user's R6
00000030 0000 435 REG_R7 = 48 ; user's R7
```

```
00000034 0000 436 REG_R8 = 52 ; user's R8
00000038 0000 437 REG_R9 = 56 ; user's R9
0000003C 0000 438 REG_R10 = 60 ; user's R10
00000040 0000 439 REG_R11 = 64 ; user's R11
00000044 0000 440 FRAME_END = 68 ; end of call frame
0000 441 :
0000 442 : Call Frame Extension Layout
0000 443 :
00000044 0000 444 REG_AP = 68 ; user's AP
00000048 0000 445 REG_FP = 72 ; user's FP
0000004C 0000 446 REG_SP = 76 ; user's SP
00000050 0000 447 REG_PC = 80 ; user's PC
00000054 0000 448 PSL = 84 ; user's PSL
00000058 0000 449 LOCAL_END = 88 ; end of local storage
0000 450 :
0000 451 : Local Storage Layout
0000 452 :
FFFFFFF 0000 453 SAVE_ALIGN = HANDLER-1 ; safe copy of alignment bits
FFFFFFFE 0000 454 CARRY_BIT = SAVE_ALIGN-1 ; original setting of carry bit
FFFFFFFD 0000 455 FAULT_TYPE = CARRY_BIT-1 ; internal code for type of fault
FFFFFFFC 0000 456 ACTION_COUNT = FAULT_TYPE-1 ; number of action bytes remaining
FFFFFFFB 0000 457 ACCVIO_REASON = ACTION_COUNT-1 ; reason mask for access violation
FFFFFFFA 0000 458 UNUSED_BYTE_1 = ACCVIO_REASON-1 ; unused byte
FFFFFFF9 0000 459 UNUSED_BYTE_2 = UNUSED_BYTE_1-1 ; unused byte
FFFFFFF8 0000 460 UNUSED_BYTE_3 = UNUSED_BYTE_2-1 ; unused byte
FFFFFFF4 0000 461 ACTION_PTR = UNUSED_BYTE_3-4 ; pointer to current action byte
FFFFFFF0 0000 462 INST_ADDRESS = ACTION_PTR-4 ; instruction location
FFFFFFEC 0000 463 BRANCH_ADDRESS = INST_ADDRESS-4 ; branch destination address
FFFFFFE8 0000 464 ACCVIO_ADDRESS = BRANCH_ADDRESS-4 ; access violation referenced address
FFFFFFC4 0000 465 READ_AREA = ACCVIO_ADDRESS-36 ; area for input operand values
FFFFFFA0 0000 466 WRITE_AREA = READ_AREA-36 ; area for output operand values
FFFFFFA0 0000 467 LOCAL_START = WRITE_AREA ; start of local storage
0000 468 :
0000 469 : Interpretive Action Codes
0000 470 :
00000001 0000 471 READ_ACCESS = 1 ; read only access operand
00000002 0000 472 WRITE_ACCESS = 2 ; write only access operand
00000003 0000 473 MODIFY_ACCESS = 3 ; modify access operand
00000004 0000 474 ADDRESS_ACCESS = 4 ; address access operand
00000005 0000 475 BRANCH_WORD = 5 ; branch destination operand
00000006 0000 476 POLY_CHECK = 6 ; POLYx post instruction checking
00000007 0000 477 ACB_CHECK = 7 ; ACBx post instruction checking
0000 478 :
0000 479 : Data Type Codes
0000 480 :
00000001 0000 481 FLOAT_TYPE = 1 ; single floating
00000002 0000 482 DOUBLE_TYPE = 2 ; double floating
00000003 0000 483 GRAND_TYPE = 3 ; grand floating
00000004 0000 484 HUGE_TYPE = 4 ; huge floating
00000005 0000 485 BYTE_TYPE = 5 ; byte
00000006 0000 486 WORD_TYPE = 6 ; word
00000007 0000 487 LONG_TYPE = 7 ; longword
0000 488 :
0000 489 : Derived Codes
0000 490 :
00000011 0000 491 READ_FLOAT = <FLOAT_TYPE@4>+READ_ACCESS
00000021 0000 492 READ_DOUBLE = <DOUBLE_TYPE@4>+READ_ACCESS
```

```
00000031 0000 493 READ_GRAND = <GRAND_TYPE@4>+READ_ACCESS
00000041 0000 494 READ_HUGE = <HUGE_TYPE@4>+READ_ACCESS
00000051 0000 495 READ_BYTE = <BYTE_TYPE@4>+READ_ACCESS
00000061 0000 496 READ_WORD = <WORD_TYPE@4>+READ_ACCESS
00000012 0000 497 WRITE_FLOAT = <FLOAT_TYPE@4>+WRITE_ACCESS
00000022 0000 498 WRITE_DOUBLE = <DOUBLE_TYPE@4>+WRITE_ACCESS
00000032 0000 499 WRITE_GRAND = <GRAND_TYPE@4>+WRITE_ACCESS
00000042 0000 500 WRITE_HUGE = <HUGE_TYPE@4>+WRITE_ACCESS
00000072 0000 501 WRITE_LONG = <LONG_TYPE@4>+WRITE_ACCESS
00000013 0000 502 MODIFY_FLOAT = <FLOAT_TYPE@4>+MODIFY_ACCESS
00000023 0000 503 MODIFY_DOUBLE = <DOUBLE_TYPE@4>+MODIFY_ACCESS
00000033 0000 504 MODIFY_GRAND = <GRAND_TYPE@4>+MODIFY_ACCESS
00000043 0000 505 MODIFY_HUGE = <HUGE_TYPE@4>+MODIFY_ACCESS
00000054 0000 506 ADDRESS_BYTE = <BYTE_TYPE@4>+ADDRESS_ACCESS
00000016 0000 507 POLY_FLOAT = <FLOAT_TYPE@4>+POLY_CHECK
00000026 0000 508 POLY_DOUBLE = <DOUBLE_TYPE@4>+POLY_CHECK
00000036 0000 509 POLY_GRAND = <GRAND_TYPE@4>+POLY_CHECK
00000046 0000 510 POLY_HUGE = <HUGE_TYPE@4>+POLY_CHECK
00000017 0000 511 ACB_FLOAT = <FLOAT_TYPE@4>+ACB_CHECK
00000027 0000 512 ACB_DOUBLE = <DOUBLE_TYPE@4>+ACB_CHECK
00000037 0000 513 ACB_GRAND = <GRAND_TYPE@4>+ACB_CHECK
00000047 0000 514 ACB_HUGE = <HUGE_TYPE@4>+ACB_CHECK
00000047 0000 515 ;
```

```

0000 517
0000 518
0000 519
0000 520
0000 521
0000 522
0000 523
0000 524
0000 525
0000 526
0000 527
0000 528
0000 529
0000 530
0000 531
0000 532
0000 533
0000 534
0000 535
0000 536
0000 537
0000 538
0000 539
0000 540
0000 541
0000 542
0000 543
0000 544
0000 545
0000 546
0000 547
0000 548
0000 549
0000 550
0000 551
0000 552
0000 553
0000 554
0000 555
0000 556
0000 557
0000 558
0000 559
51 50 04 AC 000C 0002 560
51 51 04 A0 D0 0006 561
51 000004C4 8F D1 000A 562
18 13 0011 563
51 000004BC 8F D1 0013 564
0F 13 001A 565
51 000004B4 8F D1 001C 566
06 13 0023 567
50 0918 8F 3C 0025 568
04 002A 569
5E F4 AD 9E 002B 570 1$:
OFF0 8F BB 002F 571
5E 10 C2 0033 572
50 5D D0 0036 573

```

LIBSSIM_TRAP - Conversion Condition Handler

parameters: P1 = Signal Array Location
P2 = Mechanism Array Location

returns with R0 = Condition Response

Discussion

This routine is designed to function as a condition handler or it may be called from a condition handler (possibly with a chain of intermediate procedure calls) with the locations of the signal and mechanism array for the condition supplied as parameters.

When this routine is called it examines the signal array to see if the condition was a floating overflow fault, floating underflow fault, or a floating divide by zero fault. If the condition is none of these faults, then the routine returns with the value SSS_RESIGNAL to indicate that it did not handle the condition.

If the condition is one of those checked for above, then the routine examines the procedure frames on the stack up to the condition handler frame and determines the values of the registers at the time of the exception. The registers are then restored and the stack pointer is positioned to the condition code in the signal array (which is immediately before the PC, PSL pair for the detected conditions). The routine then branches to FAULT_TO_TRAP to initiate the conversion.

Note: 1. The method of removing the procedure frames above the condition handler frame is similar to an unwind but differs in that the condition handlers in the frames are not activated with the SSS_UNWIND status. This was done since it appears to be impossible to make the handler call look like it was made by SYSSUNWIND so that overlapping unwinds can be detected.

```

.ENTRY LIBSSIM_TRAP,- ; entrance
^M<R2,R3> ; entry mask
MOVL 4(AP),R0 ; R0 = signal array location
MOVL 4(R0),R1 ; R1 = condition code
CML #SSS_FLTUND_F,R1 ; floating underflow fault ?
BEQL 1$ ; yes - bypass
CML #SSS_FLTDIV_F,R1 ; floating divide by zero fault ?
BEQL 1$ ; yes - bypass
CML #SSS_FLTOVF_F,R1 ; floating overflow fault ?
BEQL 1$ ; yes - bypass
MOVZWL #SSS_RESIGNAL,R0 ; R0 = resignal condition code
RET ; return
MOVAB -12(FP),SP ; allocate space for AP, FP, SP
PUSHR #^M<R4,R5,R6,R7,R8,R9,R10,R11> ; save registers R4-R11
SUBL2 #16,SP ; allocate space for R0, R1, R2, R3
MOVL FP,R0 ; R0 = current frame pointer

```

```

51 06 A0 0C 00 EF 0039 574 2$: EXTZV #0,#12,SAVE_MASK(R0),R1 ; R1 = register save mask
      52 14 A0 9E 003F 575 MOVAB REG_R0(R0),R2 ; R2 = start of registers in R0 frame
      53 51 0C 53 D4 0043 576 CLRL R3 ; clear the register index
      6E43 08 13 004A 578 BEQL 4$ ; find the next saved register
      53 53 D0 004C 579 MOVL (R2)+,(SP)[R3] ; no more saved registers - bypass
      F1 11 0052 581 BRB 3$ ; get the register value
      30 AE 08 A0 7D 0054 582 4$: MOVQ SAVE_AP(R0),48(SP) ; increment the register number
10 A0 00000004'8F D1 0059 583 Cmpl #SYS$CALL_HANDL+4,SAVE_PC(R0) ; look some more
      50 34 AE D0 0063 584 BEQL 5$ ; get the values of AP and SP
      38 AE 04 AC 04 C1 0069 587 5$: ADDL3 #4,4(AP),56(SP) ; is this the handler frame ?
      50 08 AC D0 006F 588 MOVL 52(SP),R0 ; yes - bypass
      6E 0C A0 7D 0073 589 MOVQ 12(R0),(SP) ; R0 = location of next call frame
      7FFF 8F BA 0077 590 POPR #*M<R0,R1,R2,R3,R4,R5,- ; unwind the frame
      007B 591 ; R6,R7,R8,R9,R10,R11,AP,FP,SP> ; point saved SP to condition name
      007B 592 ; FAULT_TO_TRAP ; R0 = mechanism array location
      007B 593 ; ; get R0 and R1
      ; ; restore registers R0-SP
      ; ; with SP pointing to condition
      ; ; enter the conversion routine

```

```

007B 595
007B 596
007B 597
007B 598
007B 599
007B 600
007B 601
007B 602
007B 603
007B 604
007B 605
007B 606
007B 607
007B 608
007B 609
007B 610
007B 611
007B 612
007B 613
007B 614
007B 615
007B 616
007B 617
007B 618
007B 619
007B 620
007B 621
007B 622
007B 623
007B 624
007B 625
007B 626
007B 627
007B 628
007B 629
007B 630
007B 631
007B 632
007B 633
007B 634
007B 635
007B 636
007B 637
007B 638
007B 639
007B 640
007B 641
007B 642
007B 643
007B 644
007B 645
007B 646
007B 647
007B 648
007B 649
007B 650
007B 651 FAULT_TO_TRAP:

```

FAULT_TO_TRAP - Perform Fault to Trap Conversion

entered by branching

parameters: (SP) = Fault Condition Code
4(SP) = Instruction Location
8(SP) = PSL Value

Discussion

This routine sets up the frame for the fault to trap conversion routine and initializes everything. The specified parameters describe the condition that occurred and 12(SP) is assumed to be the user's stack pointer at the time of the fault. All of the other registers are assumed to be the user's registers.

First the stack is extended to CALL_ARGS-3 longwords to provide the area for simulating the top of the user's stack. A CALLS instruction is then executed which specifies all of the CALL_ARGS longwords below the user's stack pointer as the parameter list. The local storage for the frame is then allocated and the alignment bits for the frame are saved in a local storage cell. The condition handler for the routine is then set up. The CALLS instruction also saves the user's registers R0, ..., R11 in order and saves AP and FP elsewhere in the frame. The routine then saves the registers by saving the user's AP, FP, SP, PC, and PSL after the saved registers. AP and FP are taken from the frame, SP is computed, and PC and PSL are taken from the parameter list.

The condition code for the fault is converted to a short internal code for use in branching and the carry bit in the user's PSL is saved. The instruction location is stored in the return PC for the frame so it will be discovered by the traceback handler if the routine blows up. If the T bit is set in the user's PSL then the TP bit is also set to insure that instructions are not lost if the debugger is running.

The opcode of the instruction is analyzed and the location of the action bytes for the instruction is computed. (The details of this computation are given in the table descriptions below.) The action pointer and action count are set up for processing the instruction, the pointers to the read area and the write area are initialized, and control enters the action loop to process the instruction.

Note: 1. From the description of the way that the simulated register area is constructed, it is clear that the length longword of the parameter list is overwritten. All of the methods of leaving the fault to trap converter put this longword back together. The internal condition handler does this if it detects an unwind.

; entrance

50	5E	FF6C	CE	9E	007B	652	MOVAB	-4*<CALL_ARGS-3>(SP),SP ; allocate the flexible stack space		
	00000088	'EF	28	FB	0080	653	CALLS	#CALL_ARGS,1\$; build the procedure frame		
				00	0087	654	HALT	; the call will never return here		
				OFFF	0088	655	1\$:	.WORD ^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; entry mask		
50	06	AD	5E	A0	AD	9E	008A	656	MOVAB LOCAL_START(FP),SP ; allocate the local storage	
			FF	AD	50	90	0094	658	EXTZV #MASK_ALIGN,#2,SAVE_MASK(FP),R0 ; R0 = alignment bits	
			6D	0178	'CF	9E	0098	659	MOVAB W^CONVERT_HANDLER,HANDLER(FP) ; set up the condition handler	
	44	AD		08	AD	7D	009D	660	MOVQ SAVE_AP(FP),REG_AP(FP) ; move user's AP and FP into place	
	4C	AD		00A4	CC	9E	00A2	661	MOVAB 4*<CALL_ARGS+1>(AP),REG_SP(FP) ; move user's SP into place	
	50	AD		009C	CC	7D	00A8	662	MOVQ 4*<CALL_ARGS-1>(AP),REG_PC(FP) ; move PC and PSL into place	
				0098	CC	D0	00AE	663	MOVL 4*<CALL_ARGS-2>(AP),R0 ; R0 = condition name	
				04C4	8F	B1	00B3	664	CMPW #SS\$_FLTUND_F,R0 ; floating underflow fault ?	
					05	12	00B8	665	BNEQ 2\$; no - bypass	
					50	01	00BA	666	MOVL #1,R0 ; R0 = internal code	
					17	11	00BD	667	BRB 5\$; bypass	
	50			04B4	8F	B1	00BF	668	2\$: CMPW #SS\$_FLTOVF_F,R0 ; floating overflow fault ?	
					05	12	00C4	669	BNEQ 3\$; no - bypass	
					50	02	00C6	670	MOVL #2,R0 ; R0 = internal code	
					0B	11	00C9	671	BRB 5\$; bypass	
	50			04BC	8F	B1	00CB	672	3\$: CMPW #SS\$_FLTDIV_F,R0 ; floating divide by zero fault ?	
					01	13	00D0	673	BEQL 4\$; yes - skip	
						00	00D2	674	HALT ; something went wrong !	
					50	03	00D3	675	4\$: MOVL #3,R0 ; R0 = internal code	
				FD	AD	50	00D6	676	5\$: MOVB R0,FAULT_TYPE(FP) ; save the fault code	
					FE	AD	94	00DA	677	CLRB CARRY_BIT(FP) ; clear the carry bit flag
	03	54	AD		00	E1	00DD	678	BBC #PSL_C,PSL(FP),6\$; is the carry bit set ?	
					FE	AD	96	00E2	679	INCB CARRY_BIT(FP) ; yes - make a note
	05	54	AD		04	E1	00E5	680	6\$: BBC #PSL_T,PSL(FP),7\$; the trace enable bit is clear - skip	
	00	54	AD		1E	E2	00EA	681	BBSS #PSL_TP,PSL(FP),7\$; set the trace pending bit	
	10	AD		50	AD	D0	00EF	682	7\$: MOVL REG_PC(FP),SAVE_PC(FP) ; store traceback instruction location	
	FO	AD		50	AD	D0	00F4	683	MOVL REG_PC(FP),INST_ADDRESS(FP) ; save the instruction address	
				52	50	BD	9A	00F9	684	MOVZBL @REG_PC(FP),R2 ; R2 = first byte of instruction
					50	AD	D6	00FD	685	INCL REG_PC(FP) ; increment the PC
				52	FD	8F	91	0100	686	CMPB #^XFD,R2 ; is the byte FD ?
					0A	13	0104	687	BEQL 8\$; yes - bypass	
53	52			04	00	EF	0106	688	EXTZV #0,#4,R2,R3 ; R3 = low-order nibble	
					52	0F	CA	010B	689	BICL2 #15,R2 ; clear low-order nibble in R2
					11	11	010E	690	BRB 9\$; bypass	
					52	50	BD	9A	0110	8\$: MOVZBL @REG_PC(FP),R2 ; R2 = second byte of instruction
					50	AD	D6	0114	692	INCL REG_PC(FP) ; increment the PC
53	52			04	00	EF	0117	693	EXTZV #0,#4,R2,R3 ; R3 = low-order nibble	
52	04			00	01	FO	011C	694	INSV #1,#0,#4,R2 ; store one in the low-order nibble	
				01A3	'CF43	3E	0121	695	9\$: MOVAW W^CLASS_TAB[R3],R3 ; R3 = location of word for class	
				54	01	A3	98	0127	696	CVTBL 1(R3),R4 ; R4 = size of the class
					53	63	98	012B	697	CVTBL (R3),R3 ; R3 = relative position of class
53				01A3	'CF43	9E	012E	698	MOVAB W^CLASS_TAB[R3],R3 ; R3 = location of class table	
	63			54	52	3A	0134	699	LOCC R2,R4,(R3) ; look for the instruction	
					01	12	0138	700	BNEQ 10\$; we found it - skip	
						00	013A	701	HALT ; we didn't find it - internal error	
					51	6144	9A	013B	702	10\$: MOVZBL (R1)[R4],R1 ; R1 = relative position of pattern
50				0225	'CF41	9E	013F	703	MOVAB W^PATTERNS[R1],R0 ; R0 = location of pattern string	
				FC	AD	60	90	0145	704	MOVB (R0),ACTION_COUNT(FP) ; store the action counter
				F4	AD	50	50	0149	705	MOVL R0,ACTION_PTR(FP) ; store the action pointer
				5A	C4	AD	9E	014D	706	MOVAB READ_AREA(FP),R10 ; R10 = location of read operand area
				5B	A0	AD	9E	0151	707	MOVAB WRITE_AREA(FP),R11 ; R11 = location of write operand area
					0143	31	0155	708	BRW ACTION_LOOP ; enter the action loop	

LIBSSIM_TRAP
1-003

C 2
- Simulate floating trap
LIBSSIM_TRAP - Convert Floating Faults t 16-SEP-1984 00:18:50 VAX/VMS Macro V04-00 Page 17
0158 709 ; 6-SEP-1984 11:10:54 [LIBRTL.SRC]LIBSIMTRA.MAR;1 (7)

LIB
1-0

```

0158 711 :
0158 712 : Normal End of Fault to Trap Conversion
0158 713 :
0158 714 : entered by branching
0158 715 :
0158 716 : no parameters
0158 717 :
0158 718 : Discussion
0158 719 :
0158 720 : This routine is entered at the end of instruction
0158 721 : processing. First it writes all of the output operands.
0158 722 : The values of these operands is in the write area in the
0158 723 : frame. When this operation is complete, the routine branches
0158 724 : to the routine which signals the appropriate arithmetic trap.
0158 725 :
0158 726 : Note: 1. Each entry of the write area consists of a longword
0158 727 : containing the operand location, followed by a
0158 728 : longword containing the number of bytes of the
0158 729 : operand, followed by that many bytes containing
0158 730 : the value to be output. For register operands, the
0158 731 : address of the simulated register is used. The
0158 732 : location of the next byte in the write area is
0158 733 : always contained in R11.
0158 734 :
0158 735 : FINISH:
56 A0 AD 9E 0158 736 MOVAB WRITE_AREA(FP),R6 ; entrance
5B 56 D1 015C 737 1$: CMPL R6,R11 ; R6 = location of output value area
OC 13 015F 738 BEQL 2$ ; have we reached the end ?
57 86 7D 0161 739 MOVQ (R6)+,R7 ; yes - bypass
67 66 58 28 0164 740 MOVQ3 R8,(R6),(R7) ; R7 = address, R8 = length
56 58 CO 0168 741 ADDL2 R8,R6 ; output the value
EF 11 016B 742 BRB 1$ ; position past the value
02 01 FD AD 8F 016D 743 2$: CASEB FAULT_TYPE(FP),#1,#2 ; loop
05CE' 0172 744 3$: .WORD UNDERFLOW_TRAP-3$ ; branch on the fault type
05DC' 0174 745 .WORD OVERFLOW_TRAP-3$ ; 1 - floating underflow
05EA' 0176 746 .WORD DIVIDE_TRAP-3$ ; 2 - floating overflow
0178 747 ; 3 - floating divide by zero

```

```

0178 749
0178 750
0178 751
0178 752
0178 753
0178 754
0178 755
0178 756
0178 757
0178 758
0178 759
0178 760
0178 761
0178 762
0178 763
0178 764
0178 765
0178 766
0178 767
0178 768
0178 769
0178 770
017A 771
017E 772
0186 773
0188 774
018C 775
0190 776
0196 777
0199 778
019D 779
01A2 780
01A3 781

```

CONVERT_HANDLER - Internal Condition Handler

parameters: P1 = Signal Array Location
P2 = Mechanism Array Location

returns with R0 = Condition Response

Discussion

This routine is the internal condition handler for the fault to trap converter. Since the converter does not make constructive use of exceptions except in special procedures, this routine requests of all conditions it intercepts.

If the condition is SSS_UNWIND which indicates that an unwind is about to take place, then it restores the argument count longword in the parameter list for the procedure so the unwind will work properly.

```

0000 0178 770 CONVERT_HANDLER:
04 A0 50 04 AC 7D 017A 771 .WORD 0 ; entrance
00000920 8F D1 017E 772 MOVQ 4(AP),R0 ; entry mask
15 12 0186 773 CMPL #SS$_UNWIND,4(R0) ; R0,R1 = condition array locations
50 04 A1 D0 0188 774 BNEQ 1$ ; is this an unwind?
51 FF A0 90 018C 775 MOVL 4(R1),R0 ; no - bypass
06 A0 02 0E 51 F0 0190 776 MOVB SAVE_ALIGN(R0),R1 ; R0 = frame location
50 51 C0 0196 777 INSV R1,#MASK_ALIGN,#2,SAVE_MASK(R0) ; R1 = safe copy of alignment bits
44 A0 28 D0 0199 778 ADDL2 R1,R0 ; store align bits in frame
50 0918 8F 3C 019D 779 1$: MOVL #CALL_ARGS,FRAME_END(R0) ; store the argument count
04 01A2 780 MOVZWL #SS$_RESIGNAL,R0 ; resignal the condition
01A3 781 RET ; return

```

01A3	783	:
01A3	784	:
01A3	785	:
01A3	786	:
01A3	787	:
01A3	788	:
01A3	789	:
01A3	790	:
01A3	791	:
01A3	792	:
01A3	793	:
01A3	794	:
01A3	795	:
01A3	796	:
01A3	797	:
01A3	798	:
01A3	799	:
01A3	800	:
01A3	801	:
01A3	802	:
01A3	803	:
01A3	804	:
01A3	805	:
01A3	806	:
01A3	807	:
01A3	808	:
01A3	809	:
01A3	810	:
01A3	811	:
01A3	812	:
01A3	813	:
01A3	814	:
01A3	815	:
01A3	816	:
01A3	817	:
01A3	818	:
01A3	819	:
01A3	820	:
01A3	821	:
01A3	822	:
01A3	823	:
01A3	824	:
01A3	825	:
01A3	826	:
01A3	827	:
01A3	828	:
01A3	829	:
01A3	830	:
01A3	831	:
01A3	832	:
01A3	833	:
01A3	834	:
01A3	835	:
01A3	836	:
01A3	837	CLASS_TAB:
04'20'	01A3	.BYTE
04'28'	01A5	.BYTE

Instruction Lookup and Action Tables

Discussion

The following tables are used for identifying the instructions which the fault to trap converter is supposed to process and for determining what actions are necessary to process these instructions. Three sets of tables are involved. First there is the class lookup table which is used to locate the class table for a particular instruction class. Next there is the class table which is used to search for a particular instruction and then to determine its action pattern table. Finally there is the action pattern table which specifies the various actions taken for the instruction.

For a one byte opcode, the class of the instruction is the low order nibble of the opcode and the key of the instruction is the opcode with the low order nibble cleared. For a two byte opcode (where the first byte is FD), the class of the instruction is the low order nibble of the second byte and the key of the instruction is the second byte of the opcode with the low order nibble set to one.

The class lookup table CLASS_TAB is a table of words which is indexed by the class number. The first byte of the indexed entry is the position (relative to CLASS_TAB) of the class table for the class and the second byte of the indexed entry is the number of entries in the class table.

The class table for a particular class consists of two byte strings of equal size. The first byte string (the instruction lookup table) contains the keys for the instructions in the class and the corresponding bytes in the second string (the action table) consist of the positions (relative to PATTERNS) of the action pattern table for the instruction.

An action pattern table is a byte string in which the first byte contains the number of actions and the remaining bytes contain the codes for the individual actions. The code for an action contains the action type in the low order nibble and a qualifying data type in the high-order nibble. All of action codes used have been given identifiers which are defined in the definition section of this module.

A short table which maps data type codes into their lengths in bytes has also been included.

Class Lookup Table

CLASS_0-CLASS_TAB,ACTION_0-CLASS_0
CLASS_1-CLASS_TAB,ACTION_1-CLASS_1

```

04'30' 01A7 840 .BYTE CLASS_2-CLASS_TAB,ACTION_2-CLASS_2
05'38' 01A9 841 .BYTE CLASS_3-CLASS_TAB,ACTION_3-CLASS_3
08'42' 01AB 842 .BYTE CLASS_4-CLASS_TAB,ACTION_4-CLASS_4
08'52' 01AD 843 .BYTE CLASS_5-CLASS_TAB,ACTION_5-CLASS_5
07'62' 01AF 844 .BYTE CLASS_6-CLASS_TAB,ACTION_6-CLASS_6
05'70' 01B1 845 .BYTE CLASS_7-CLASS_TAB,ACTION_7-CLASS_7
00 00 01B3 846 .BYTE 0,0
00 00 01B5 847 .BYTE 0,0
00 00 01B7 848 .BYTE 0,0
00 00 01B9 849 .BYTE 0,0
00 00 01BB 850 .BYTE 0,0
00 00 01BD 851 .BYTE 0,0
00 00 01BF 852 .BYTE 0,0
04'7A' 01C1 853 .BYTE CLASS_F-CLASS_TAB,ACTION_F-CLASS_F
      01C3 854 .....
      01C3 855 .....
      01C3 856 .....
      01C3 857 .....
      01C3 858 .....
      01C3 859 .....
      01C3 860 .....
      01C3 861 CLASS_0:
40 01C3 862 .BYTE ^X 40 : 4C ADDF2
60 01C4 863 .BYTE ^X 60 : 60 ADDD2
41 01C5 864 .BYTE ^X 41 : 40FD ADDG2
61 01C6 865 .BYTE ^X 61 : 60FD ADDH2
      01C7 866 .....
      01C7 867 .....
      01C7 868 .....
      01C7 869 ACTION_0:
00' 01C7 870 .BYTE PATTERN_ADDF2-PATTERNS : 40 ADDF2
03' 01C8 871 .BYTE PATTERN_ADDD2-PATTERNS : 60 ADDD2
06' 01C9 872 .BYTE PATTERN_ADDG2-PATTERNS : 40FD ADDG2
09' 01CA 873 .BYTE PATTERN_ADDH2-PATTERNS : 60FD ADDH2
      01CB 874 .....
      01CB 875 .....
      01CB 876 .....
      01CB 877 CLASS_1:
40 01CB 878 .BYTE ^X 40 : 41 ADDF3
60 01CC 879 .BYTE ^X 60 : 61 ADDD3
41 01CD 880 .BYTE ^X 41 : 41FD ADDG3
61 01CE 881 .BYTE ^X 61 : 61FD ADDH3
      01CF 882 .....
      01CF 883 .....
      01CF 884 .....
      01CF 885 ACTION_1:
0C' 01CF 886 .BYTE PATTERN_ADDF3-PATTERNS : 41 ADDF3
10' 01D0 887 .BYTE PATTERN_ADDD3-PATTERNS : 61 ADDD3
14' 01D1 888 .BYTE PATTERN_ADDG3-PATTERNS : 41FD ADDG3
18' 01D2 889 .BYTE PATTERN_ADDH3-PATTERNS : 61FD ADDH3
      01D3 890 .....
      01D3 891 .....
      01D3 892 .....
      01D3 893 CLASS_2:
40 01D3 894 .BYTE ^X 40 : 42 SUBF2
60 01D4 895 .BYTE ^X 60 : 62 SUBD2
41 01D5 896 .BYTE ^X 41 : 42FD SUBG2

```

Class Tables

Class 0 Instruction Lookup Table

Class 0 Action Table

Class 1 Instruction Lookup Table

Class 1 Action Table

Class 2 Instruction Lookup Table

```

61 01D6 897 .BYTE ^X 61 ; 62FD SUBH2
    01D7 898
    01D7 899
    01D7 900
    01D7 901 ACTION_2:
00' 01D7 902 .BYTE PATTERN_SUBF2-PATTERNS : 42 SUBF2
03' 01D8 903 .BYTE PATTERN_SUBD2-PATTERNS : 62 SUBD2
06' 01D9 904 .BYTE PATTERN_SUBG2-PATTERNS : 42FD SUBG2
09' 01DA 905 .BYTE PATTERN_SUBH2-PATTERNS : 62FD SUBH2
    01DB 906
    01DB 907
    01DB 908
    01DB 909 CLASS_3:
40 01DB 910 .BYTE ^X 40 : 43 SUBF3
60 01DC 911 .BYTE ^X 60 : 63 SUBD3
31 01DD 912 .BYTE ^X 31 : 33FD CVTGF
41 01DE 913 .BYTE ^X 41 : 43FD SUBG3
61 01DF 914 .BYTE ^X 61 : 63FD SUBH3
    01E0 915
    01E0 916
    01E0 917
    01E0 918 ACTION_3:
0C' 01E0 919 .BYTE PATTERN_SUBF3-PATTERNS : 43 SUBF3
10' 01E1 920 .BYTE PATTERN_SUBD3-PATTERNS : 63 SUBD3
1F' 01E2 921 .BYTE PATTERN_CVTGF-PATTERNS : 33FD CVTGF
14' 01E3 922 .BYTE PATTERN_SUBG3-PATTERNS : 43FD SUBG3
18' 01E4 923 .BYTE PATTERN_SUBH3-PATTERNS : 63FD SUBH3
    01E5 924
    01E5 925
    01E5 926
    01E5 927 CLASS_4:
40 01E5 928 .BYTE ^X 40 : 44 MULF2
50 01E6 929 .BYTE ^X 50 : 54 EMOF
60 01E7 930 .BYTE ^X 60 : 64 MULD2
70 01E8 931 .BYTE ^X 70 : 74 EMOF
41 01E9 932 .BYTE ^X 41 : 44FD MULG2
51 01EA 933 .BYTE ^X 51 : 54FD EMOF
61 01EB 934 .BYTE ^X 61 : 64FD MULH2
71 01EC 935 .BYTE ^X 71 : 74FD EMOF
    01ED 936
    01ED 937
    01ED 938
    01ED 939 ACTION_4:
00' 01ED 940 .BYTE PATTERN_MULF2-PATTERNS : 44 MULF2
2B' 01EE 941 .BYTE PATTERN_EMOF-PATTERNS : 54 EMOF
03' 01EF 942 .BYTE PATTERN_MULD2-PATTERNS : 64 MULD2
31' 01F0 943 .BYTE PATTERN_EMODD-PATTERNS : 74 EMODD
06' 01F1 944 .BYTE PATTERN_MULG2-PATTERNS : 44FD MULG2
37' 01F2 945 .BYTE PATTERN_EMODG-PATTERNS : 54FD EMODG
09' 01F3 946 .BYTE PATTERN_MULH2-PATTERNS : 64FD MULH2
3D' 01F4 947 .BYTE PATTERN_EMODH-PATTERNS : 74FD EMODH
    01F5 948
    01F5 949
    01F5 950
    01F5 951 CLASS_5:
40 01F5 952 .BYTE ^X 40 : 45 MULF3
50 01F6 953 .BYTE ^X 50 : 55 POLYF

```

60	01F7	954	.BYTE	^X 60	:	65	MULD3
70	01F8	955	.BYTE	^X 70	:	75	POLYD
41	01F9	956	.BYTE	^X 41	:	45FD	MULG3
51	01FA	957	.BYTE	^X 51	:	55FD	POLYG
61	01FB	958	.BYTE	^X 61	:	65FD	MULH3
71	01FC	959	.BYTE	^X 71	:	75FD	POLYH
	01FD	960	:				
	01FD	961	:				
	01FD	962	:				
	01FD	963	ACTION_5:				
0C'	01FD	964	.BYTE	PATTERN_MULF3-PATTERNS	:	45	MULF3
43'	01FE	965	.BYTE	PATTERN_POLYF-PATTERNS	:	55	POLYF
10'	01FF	966	.BYTE	PATTERN_MULD3-PATTERNS	:	65	MULD3
48'	0200	967	.BYTE	PATTERN_POLYD-PATTERNS	:	75	POLYD
14'	0201	968	.BYTE	PATTERN_MULG3-PATTERNS	:	45FD	MULG3
4D'	0202	969	.BYTE	PATTERN_POLYG-PATTERNS	:	55FD	POLYG
18'	0203	970	.BYTE	PATTERN_MULH3-PATTERNS	:	65FD	MULH3
52'	0204	971	.BYTE	PATTERN_POLYH-PATTERNS	:	75FD	POLYH
	0205	972	:				
	0205	973	:				
	0205	974	:				
	0205	975	CLASS_6:				
40	0205	976	.BYTE	^X 40	:	46	DIVF2
60	0206	977	.BYTE	^X 60	:	66	DIVD2
70	0207	978	.BYTE	^X 70	:	76	CVTDF
41	0208	979	.BYTE	^X 41	:	46FD	DIVG2
61	0209	980	.BYTE	^X 61	:	66FD	DIVH2
71	020A	981	.BYTE	^X 71	:	76FD	CVTHD
F1	020B	982	.BYTE	^X F1	:	F6FD	CVTHF
	020C	983	:				
	020C	984	:				
	020C	985	:				
	020C	986	ACTION_6:				
00'	020C	987	.BYTE	PATTERN_DIVF2-PATTERNS	:	46	DIVF2
03'	020D	988	.BYTE	PATTERN_DIVD2-PATTERNS	:	66	DIVD2
1C'	020E	989	.BYTE	PATTERN_CVTDF-PATTERNS	:	76	CVTDF
06'	020F	990	.BYTE	PATTERN_DIVG2-PATTERNS	:	46FD	DIVG2
09'	0210	991	.BYTE	PATTERN_DIVH2-PATTERNS	:	66FD	DIVH2
28'	0211	992	.BYTE	PATTERN_CVTHG-PATTERNS	:	76FD	CVTHG
22'	0212	993	.BYTE	PATTERN_CVTHF-PATTERNS	:	F6FD	CVTHF
	0213	994	:				
	0213	995	:				
	0213	996	:				
	0213	997	CLASS_7:				
40	0213	998	.BYTE	^X 40	:	47	DIVF3
60	0214	999	.BYTE	^X 60	:	67	DIVD3
41	0215	1000	.BYTE	^X 41	:	47FD	DIVG3
61	0216	1001	.BYTE	^X 61	:	67FD	DIVH3
F1	0217	1002	.BYTE	^X F1	:	F7FD	CVTHD
	0218	1003	:				
	0218	1004	:				
	0218	1005	:				
	0218	1006	ACTION_7:				
0C'	0218	1007	.BYTE	PATTERN_DIVF3-PATTERNS	:	47	DIVF3
10'	0219	1008	.BYTE	PATTERN_DIVD3-PATTERNS	:	67	DIVD3
14'	021A	1009	.BYTE	PATTERN_DIVG3-PATTERNS	:	47FD	DIVG3
18'	021B	1010	.BYTE	PATTERN_DIVH3-PATTERNS	:	67FD	DIVH3

```

25' 021C 1011 .BYTE PATTERN_CVTHD-PATTERNS ; F7 CVTHD
    021D 1012
    021D 1013
    021D 1014
    021D 1015 CLASS_F:
40 021D 1016 .BYTE ^X 40 : 4F ACBF
60 021E 1017 .BYTE ^X 60 : 6F ACBD
41 021F 1018 .BYTE ^X 41 : 4FFD ACBG
61 0220 1019 .BYTE ^X 61 : 6FFD ACBH
    0221 1020
    0221 1021
    0221 1022
    0221 1023 ACTION_F:
57' 0221 1024 .BYTE PATTERN_ACBF-PATTERNS : 4F ACBF
5D' 0222 1025 .BYTE PATTERN_ACBD-PATTERNS : 6F ACBD
63' 0223 1026 .BYTE PATTERN_ACBG-PATTERNS : 4FFD ACBG
69' 0224 1027 .BYTE PATTERN_ACBH-PATTERNS : 6FFD ACBH
    0225 1028
    0225 1029
    0225 1030
    0225 1031
    0225 1032
    0225 1033 PATTERNS: ; origin for pattern action tables
    0225 1034
    0225 1035
    0225 1036
    0225 1037 PATTERN_ADDF2:
    0225 1038 PATTERN_DIVF2:
    0225 1039 PATTERN_MULF2:
    0225 1040 PATTERN_SUBF2:
02 0225 1041 .BYTE 2
11 0226 1042 .BYTE READ_FLOAT
13 0227 1043 .BYTE MODIFY_FLOAT
    0228 1044
    0228 1045
    0228 1046
    0228 1047 PATTERN_ADDD2:
    0228 1048 PATTERN_DIVD2:
    0228 1049 PATTERN_MULD2:
    0228 1050 PATTERN_SUBD2:
02 0228 1051 .BYTE 2
21 0229 1052 .BYTE READ_DOUBLE
23 022A 1053 .BYTE MODIFY_DOUBLE
    022B 1054
    022B 1055
    022B 1056
    022B 1057 PATTERN_ADDG2:
    022B 1058 PATTERN_DIVG2:
    022B 1059 PATTERN_MULG2:
    022B 1060 PATTERN_SUBG2:
02 022B 1061 .BYTE 2
31 022C 1062 .BYTE READ_GRAND
33 022D 1063 .BYTE MODIFY_GRAND
    022E 1064
    022E 1065
    022E 1066
    022E 1067 PATTERN_ADDH2:

```

```
022E 1068 PATTERN_DIVH2:
022E 1069 PATTERN_MULH2:
022E 1070 PATTERN_SUBH2:
02 022E 1071 .BYTE 2
41 022F 1072 .BYTE READ_HUGE
43 0230 1073 .BYTE MODIFY_HUGE
0231 1074 .
0231 1075 . Pattern for ADDF3, DIVF3, MULF3, and SUBF3
0231 1076 .
0231 1077 PATTERN_ADDF3:
0231 1078 PATTERN_DIVF3:
0231 1079 PATTERN_MULF3:
0231 1080 PATTERN_SUBF3:
03 0231 1081 .BYTE 3
11 0232 1082 .BYTE READ_FLOAT
11 0233 1083 .BYTE READ_FLOAT
12 0234 1084 .BYTE WRITE_FLOAT
0235 1085 .
0235 1086 . Pattern for ADDD3, DIVD3, MULD3, and SUBD3
0235 1087 .
0235 1088 PATTERN_ADDD3:
0235 1089 PATTERN_DIVD3:
0235 1090 PATTERN_MULD3:
0235 1091 PATTERN_SUBD3:
03 0235 1092 .BYTE 3
21 0236 1093 .BYTE READ_DOUBLE
21 0237 1094 .BYTE READ_DOUBLE
22 0238 1095 .BYTE WRITE_DOUBLE
0239 1096 .
0239 1097 . Pattern for ADDG3, DIVG3, MULG3 and SUBG3
0239 1098 .
0239 1099 PATTERN_ADDG3:
0239 1100 PATTERN_DIVG3:
0239 1101 PATTERN_MULG3:
0239 1102 PATTERN_SUBG3:
03 0239 1103 .BYTE 3
31 023A 1104 .BYTE READ_GRAND
31 023B 1105 .BYTE READ_GRAND
32 023C 1106 .BYTE WRITE_GRAND
023D 1107 .
023D 1108 . Pattern for ADDH3, DIVH3, MULG3, and SUBH3
023D 1109 .
023D 1110 PATTERN_ADDH3:
023D 1111 PATTERN_DIVH3:
023D 1112 PATTERN_MULH3:
023D 1113 PATTERN_SUBH3:
03 023D 1114 .BYTE 3
41 023E 1115 .BYTE READ_HUGE
41 023F 1116 .BYTE READ_HUGE
42 0240 1117 .BYTE WRITE_HUGE
0241 1118 .
0241 1119 . Pattern for CVTDF
0241 1120 .
0241 1121 PATTERN_CVTDF:
02 0241 1122 .BYTE 2
21 0242 1123 .BYTE READ_DOUBLE
12 0243 1124 .BYTE WRITE_FLOAT
```

```
0244 1125      :
0244 1126      : Pattern for CVTGF
0244 1127      :
0244 1128      :
02 0244 1129      : PATTERN_CVTGF:
31 0244 1129      : .BYTE 2
12 0245 1130      : .BYTE READ_GRAND
0246 1131      : .BYTE WRITE_FLOAT
0247 1132      :
0247 1133      : Pattern for CVTHF
0247 1134      :
02 0247 1135      : PATTERN_CVTHF:
41 0247 1136      : .BYTE 2
12 0248 1137      : .BYTE READ_HUGE
0249 1138      : .BYTE WRITE_FLOAT
024A 1139      :
024A 1140      : Pattern for CVTHD
024A 1141      :
02 024A 1142      : PATTERN_CVTHD:
41 024A 1143      : .BYTE 2
22 024B 1144      : .BYTE READ_HUGE
024C 1145      : .BYTE WRITE_DOUBLE
024D 1146      :
024D 1147      : Pattern for CVTHG
024D 1148      :
02 024D 1149      : PATTERN_CVTHG:
41 024D 1150      : .BYTE 2
32 024E 1151      : .BYTE READ_HUGE
024F 1152      : .BYTE WRITE_GRAND
0250 1153      :
0250 1154      : Pattern for EMODF
0250 1155      :
05 0250 1156      : PATTERN_EMODF:
11 0250 1157      : .BYTE 5
51 0251 1158      : .BYTE READ_FLOAT
11 0252 1159      : .BYTE READ_BYTE
72 0253 1160      : .BYTE READ_FLOAT
12 0254 1161      : .BYTE WRITE_LONG
0255 1162      : .BYTE WRITE_FLOAT
0256 1163      :
0256 1164      : Pattern for EMODD
0256 1165      :
05 0256 1166      : PATTERN_EMODD:
21 0256 1167      : .BYTE 5
51 0257 1168      : .BYTE READ_DOUBLE
21 0258 1169      : .BYTE READ_BYTE
72 0259 1170      : .BYTE READ_DOUBLE
22 025A 1171      : .BYTE WRITE_LONG
025B 1172      : .BYTE WRITE_DOUBLE
025C 1173      :
025C 1174      : Pattern for EMODG
025C 1175      :
05 025C 1176      : PATTERN_EMODG:
31 025C 1177      : .BYTE 5
61 025D 1178      : .BYTE READ_GRAND
31 025E 1179      : .BYTE READ_WORD
72 025F 1180      : .BYTE READ_GRAND
0260 1181      : .BYTE WRITE_LONG
```

```
32 0261 1182 .BYTE WRITE_GRAND
    0262 1183 .:
    0262 1184 .: Pattern for EMODH
    0262 1185 .:
    0262 1186 PATTERN_EMODH:
05 0262 1187 .BYTE 5
41 0263 1188 .BYTE READ_HUGE
61 0264 1189 .BYTE READ_WORD
41 0265 1190 .BYTE READ_HUGE
72 0266 1191 .BYTE WRITE_LONG
42 0267 1192 .BYTE WRITE_HUGE
    0268 1193 .:
    0268 1194 .: Pattern for POLYF
    0268 1195 .:
    0268 1196 PATTERN_POLYF:
04 0268 1197 .BYTE 4
11 0269 1198 .BYTE READ_FLOAT
61 026A 1199 .BYTE READ_WORD
54 026B 1200 .BYTE ADDRESS_BYTE
16 026C 1201 .BYTE POLY_FLOAT
    026D 1202 .:
    026D 1203 .: Pattern for POLYD
    026D 1204 .:
    026D 1205 PATTERN_POLYD:
04 026D 1206 .BYTE 4
21 026E 1207 .BYTE READ_DOUBLE
61 026F 1208 .BYTE READ_WORD
54 0270 1209 .BYTE ADDRESS_BYTE
26 0271 1210 .BYTE POLY_DOUBLE
    0272 1211 .:
    0272 1212 .: Pattern for POLYG
    0272 1213 .:
    0272 1214 PATTERN_POLYG:
04 0272 1215 .BYTE 4
31 0273 1216 .BYTE READ_GRAND
61 0274 1217 .BYTE READ_WORD
54 0275 1218 .BYTE ADDRESS_BYTE
36 0276 1219 .BYTE POLY_GRAND
    0277 1220 .:
    0277 1221 .: Pattern for POLYH
    0277 1222 .:
    0277 1223 PATTERN_POLYH:
04 0277 1224 .BYTE 4
41 0278 1225 .BYTE READ_HUGE
61 0279 1226 .BYTE READ_WORD
54 027A 1227 .BYTE ADDRESS_BYTE
46 027B 1228 .BYTE POLY_HUGE
    027C 1229 .:
    027C 1230 .: Pattern for ACBF
    027C 1231 .:
    027C 1232 PATTERN_ACBF:
05 027C 1233 .BYTE 5
11 027D 1234 .BYTE READ_FLOAT
11 027E 1235 .BYTE READ_FLOAT
13 027F 1236 .BYTE MODIFY_FLOAT
05 0280 1237 .BYTE BRANCH_WORD
17 0281 1238 .BYTE ACB_FLOAT
```

```

0282 1239      :
0282 1240      : Pattern for ACBD
0282 1241      :
0282 1242      : PATTERN_ACBD:
05 0282 1243      .BYTE 5
21 0283 1244      .BYTE READ_DOUBLE
21 0284 1245      .BYTE READ_DOUBLE
23 0285 1246      .BYTE MODIFY_DOUBLE
05 0286 1247      .BYTE BRANCH_WORD
27 0287 1248      .BYTE ACB_DOUBLE
0288 1249      :
0288 1250      : Pattern for ACBG
0288 1251      :
0288 1252      : PATTERN_ACBG:
05 0288 1253      .BYTE 5
31 0289 1254      .BYTE READ_GRAND
31 028A 1255      .BYTE READ_GRAND
33 028B 1256      .BYTE MODIFY_GRAND
05 028C 1257      .BYTE BRANCH_WORD
37 028D 1258      .BYTE ACB_GRAND
028E 1259      :
028E 1260      : Pattern for ACBH
028E 1261      :
028E 1262      : PATTERN_ACBH:
05 028E 1263      .BYTE 5
41 028F 1264      .BYTE READ_HUGE
41 0290 1265      .BYTE READ_HUGE
43 0291 1266      .BYTE MODIFY_HUGE
05 0292 1267      .BYTE BRANCH_WORD
47 0293 1268      .BYTE ACB_HUGE
0294 1269      :
0294 1270      : Table of Data Type Lengths
0294 1271      :
0294 1272      : DATA_LENGTHS:
04 0294 1273      .BYTE 4           : table origin
08 0295 1274      .BYTE 8           : 1 - floating
08 0296 1275      .BYTE 8           : 2 - double floating
10 0297 1276      .BYTE 16          : 3 - grand floating
01 0298 1277      .BYTE 1           : 4 - huge floating
02 0299 1278      .BYTE 2           : 5 - byte
04 029A 1279      .BYTE 4           : 6 - word
029B 1280      :           : 7 - longword
:

```

```

029B 1282
029B 1283
029B 1284
029B 1285
029B 1286
029B 1287
029B 1288
029B 1289
029B 1290
029B 1291
029B 1292
029B 1293
029B 1294
029B 1295
029B 1296
029B 1297
029B 1298
029B 1299
029B 1300
029B 1301
029B 1302
029B 1303
029B 1304
029B 1305
029B 1306
029B 1307 ACTION_LOOP:
FC AD 97 029B 1308 DECB ACTION_COUNT(FP) ; entrance
03 18 029E 1309 BGEQ 1$ ; decrement the counter
FEB5 31 02A0 1310 BRW FINISH ; it's not negative - skip
F4 AD D6 02A3 1311 1$: INCL ACTION_PTR(FP) ; finish up
56 50 04 00 EF 02AA 1313 MOVZBL @ACTION_PTR(FP),R0 ; increment the action pointer
57 50 04 04 EF 02AF 1314 EXTZV #0,#4,R0,R6 ; R0 = next action byte
58 DB AF47 98 02B4 1315 EXTZV #4,#4,R0,R7 ; R6 = type of action
06 01 56 CF 02B9 1316 CVTBL DATA_LENGTHS-1[R7],R8 ; R7 = data type
000E' 02BD 1317 2$: CASEL R6,#T,#6 ; R8 = data length
000E' 02BF 1318 .WORD OPERAND-2$ ; branch on the action type
000E' 02C1 1319 .WORD OPERAND-2$ ; 1 - operand with read access
000E' 02C3 1320 .WORD OPERAND-2$ ; 2 - operand with write access
01EF' 02C5 1321 .WORD WORD_BRANCH-2$ ; 3 - operand with modify access
0254' 02C7 1322 .WORD CHECK_POLY-2$ ; 4 - operand with address access
0200' 02C9 1323 .WORD CHECK_ACB-2$ ; 5 - word branch displacement
02CB 1324 ; 6 - post checking for POLYx
; ; 7 - post checking for ACBx

```

ACTION_LOOP - Action Dispatching Routine

entered by branching

no parameters

Discussion

This routine passes control to the routine for the next action required to process the current instruction. The pointer to the current action code is contained in the local cell ACTION_PTR and the number of actions not yet processed is contained in the local cell ACTION_COUNT. When all of the actions for the instruction have been performed, control passes to FINISH. When an action is complete, the routine exits by branching to ACTION_LOOP.

Note: 1. When an action routine is entered, the following values are available in the registers:

R6 = type of action
R7 = data type for action
R8 = data type length for action

; entrance
; decrement the counter
; it's not negative - skip
; finish up
; increment the action pointer
; R0 = next action byte
; R6 = type of action
; R7 = data type
; R8 = data length
; branch on the action type
; 1 - operand with read access
; 2 - operand with write access
; 3 - operand with modify access
; 4 - operand with address access
; 5 - word branch displacement
; 6 - post checking for POLYx
; 7 - post checking for ACBx

02CB 1326
02CB 1327
02CB 1328
02CB 1329
02CB 1330
02CB 1331
02CB 1332
02CB 1333
02CB 1334
02CB 1335
02CB 1336
02CB 1337
02CB 1338
02CB 1339
02CB 1340
02CB 1341
02CB 1342
02CB 1343
02CB 1344
02CB 1345
02CB 1346
02CB 1347
02CB 1348
02CB 1349
02CB 1350
02CB 1351
02CB 1352
02CB 1353
02CB 1354
02CB 1355
02CB 1356
02CB 1357
02CB 1358
02CB 1359
02CB 1360
02CB 1361
02CB 1362
02CB 1363
02CB 1364
02CB 1365
02CB 1366
02CB 1367
02CB 1368
02CB 1369
02CB 1370
02CB 1371
02CB 1372
02CB 1373
02CB 1374
02CB 1375
02CB 1376
02CB 1377
02CB 1378
02CB 1379
02CB 1380
02CB 1381
02CB 1382

Instruction Operand Processing Routines

General Discussion

The following routines perform those actions which specify instruction operand processing. The action code contains the access type and the data type for the operand and these values along with the data type are available in the registers R6, R7, and R8. The routine operand decodes the operand specifier and branches to the routine for handling the particular operand type. When this routine is entered, the operand specifier byte, the low order nibble, and the high order nibble, are available in R0, R1, and R2.

During operand processing all of the side effects which change register values unless FPD is set in the PSL in which case only side effects for PC are performed. This is because operand scanning is only used to determine the instruction length when FPD is set since the operands are in the user's registers or on his stack.

Except for literal mode operands and index mode operands the operand scanning routines simply determine the location of the operand and enter ACCESS_OPERAND with this location in R9. For literal mode operands, the operand value is computed and stored in the read area. For index mode operands, the index modification is computed into R3 and then the routine process the next byte as an operand specifier and branches to the appropriate operand processing routine.

The routine ACCESS_OPERAND performs one of several actions for the operand based on the operand access and data types and on the type of exception being processed. If the operand is read or address, then the operand value or address is copied into the read area. If the operand is write or modify then a value for it is placed in the write area depending on the data type and exception type. For integer data types and for underflow exceptions, this value is always zero. For overflow and divide exceptions and floating operands the value is a reserved floating value. When ACCESS_OPERAND is done control passes back to ACTION_LOOP.

The read area is used to hold the values of all of the read only and address only operands of the instruction in contiguous bytes. The register R10 contains the location of next available byte in the read area. The read area is used by the postprocessing routine for the ACBx instructions in order to determine if the branch should be taken.

The write area is a list of all of the values to be output by FINISH when instruction processing is complete. The values are not output as the operands are processed since the architecture requires that all operands be processed before any values are output. The format of this area is given in the description of FINISH.

```

02CB 1383
02CB 1384
02CB 1385
02CB 1386
02CB 1387
02CB 1388
02CB 1389
02CB 1390
02CB 1391 OPERAND:
50 50 BD 9A 02CB 1392 MOVZBL @REG PC(FP),R0
50 50 AD D6 02CF 1393 INCL REG PC(FP)
51 50 04 00 EF 02D2 1394 EXTZV #0,#4,R0,R1
52 50 04 04 EF 02D7 1395 EXTZV #4,#4,R0,R2
OF 00 52 D4 02DC 1396 CLRL R3
CF 02DE 1397 CASEL R2,#0,#15
0020' 02E2 1398 1$: .WORD LITERAL_MODE-1$
0020' 02E4 1399 .WORD LITERAL_MODE-1$
0020' 02E6 1400 .WORD LITERAL_MODE-1$
0020' 02E8 1401 .WORD LITERAL_MODE-1$
006D' 02EA 1402 .WORD INDEX_MODE-1$
00A8' 02EC 1403 .WORD REGISTER_MODE-1$
00B0' 02EE 1404 .WORD REG_DEF_MODE-1$
00B9' 02F0 1405 .WORD DECR_MODE-1$
00CF' 02F2 1406 .WORD INCR_MODE-1$
00EA' 02F4 1407 .WORD INCR_DEF_MODE-1$
0108' 02F6 1408 .WORD BYTE_DISP_MODE-1$
0111' 02F8 1409 .WORD BYTE_DEF_MODE-1$
011A' 02FA 1410 .WORD WORD_DISP_MODE-1$
0124' 02FC 1411 .WORD WORD_DEF_MODE-1$
012E' 02FE 1412 .WORD LONG_DISP_MODE-1$
0138' 0300 1413 .WORD LONG_DEF_MODE-1$
0302 1414
0302 1415
0302 1416
0302 1417 LITERAL_MODE:
06 01 57 CF 0302 1418 CASEL R7,#1,#6
000E' 0306 1419 1$: .WORD 2$-1$
0019' 0308 1420 .WORD 3$-1$
001F' 030A 1421 .WORD 4$-1$
002C' 030C 1422 .WORD 6$-1$
0039' 030E 1423 .WORD 7$-1$
003E' 0310 1424 .WORD 8$-1$
0043' 0312 1425 .WORD 9$-1$
50 50 04 78 0314 1426 2$: ASHL #4,R0,R0
8A 4000 C0 9E 0318 1427 MOVAB 1@14(R0),(R10)+
2D 11 031D 1428 BRB 10$
50 50 04 78 031F 1429 3$: ASHL #4,R0,R0
BRB 5$
50 50 01 78 0325 1431 4$: ASHL #1,R0,R0
8A 4000 C0 9E 0329 1432 5$: MOVAB 1@14(R0),(R10)+
8A D4 032E 1433 CLRL (R10)+
1A 11 0330 1434 BRB 10$
50 50 1D 9C 0332 1435 6$: ROTL #29,R0,R0
8A 4000 C0 9E 0336 1436 MOVAB 1@14(R0),(R10)+
8A D4 033B 1437 CLRL (R10)+
8A 7C 033D 1438 CLRQ (R10)+
8A 50 90 033F 1439 7$: MOVAB R0,(R10)+

```

OPERAND - Process the Next Operand Specifier
entered by branching

parameters: (See the discussion of ACTION_LOOP.)

```

: entrance
: R0 = operand specifier byte
: increment the PC
: R1 = low order nibble of specifier
: R2 = high order nibble of specifier
: clear the index modification
: branch on the high order nibble
: 0 - literal mode
: 1 - literal mode
: 2 - literal mode
: 3 - literal mode
: 4 - index mode
: 5 - register mode
: 6 - register deferred mode
: 7 - autodecrement mode
: 8 - autoincrement mode
: 9 - autoincrement deferred mode
: A - byte displacement mode
: B - byte displacement deferred mode
: C - word displacement mode
: D - word displacement deferred mode
: E - long displacement mode
: F - long displacement deferred mode

```

Process a Literal Mode Operand Specifier

```

: entrance
: branch on the data type
: 1 - float
: 2 - double
: 3 - grand
: 4 - huge
: 5 - byte
: 6 - word
: 7 - long
: position the literal bits
: save the value
: bypass
: position the literal bits
: skip
: position the literal bits
: save the first longword
: clear the second longword
: bypass
: position the literal bits
: save the first longword
: clear the second longword
: clear the second quadword
: save the byte value

```

```

      08 11 0342 1440 BRB 10$ : bypass
      BA 50 B0 0344 1441 8$: MOVW R0,(R10)+ : save the word value
      03 11 0347 1442 BRB 10$ : skip
      BA 50 D0 0349 1443 9$: MOVL R0,(R10)+ : save the longword value
      FF4C 31 034C 1444 10$: BRW ACTION_LOOP : end of operand processing
      034F 1445
      034F 1446
      034F 1447
      034F 1448
      INDEX_MODE:
53 14 AD41 58 C5 034F 1449 MULL3 R8,REG_R0(FP)[R1],R3 : entrance
      50 50 BD 9A 0355 1450 MOVZBL @REG_PC(FP),R0 : R3 = index modification
      50 50 AD D6 0359 1451 INCL REG_PC(FP) : R0 = next operand specifier byte
51 50 04 00 EF 035C 1452 EXTZV #0,#4,R0,R1 : increment the PC
52 50 04 04 EF 0361 1453 EXTZV #4,#4,R0,R2 : R1 = low order nibble of specifier
      OF 00 52 CF 0366 1454 CASEL R2,#0,#15 : R1 = high order nibble of specifier
      0000 036A 1455 1$: .WORD 0 : branch on the high order nibble
      0000 036C 1456 .WORD 0 : 0 - literal mode
      0000 036E 1457 .WORD 0 : 1 - literal mode
      0000 0370 1458 .WORD 0 : 2 - literal mode
      0000 0372 1459 .WORD 0 : 3 - literal mode
      0000 0374 1460 .WORD 0 : 4 - index mode
      0028 0376 1461 .WORD REG_DEF_MODE-1$ : 5 - register mode
      0031 0378 1462 .WORD DECR_MODE-1$ : 6 - register deferred mode
      0047 037A 1463 .WORD INCR_MODE-1$ : 7 - autodecrement mode
      0062 037C 1464 .WORD INCR_DEF_MODE-1$ : 8 - autoincrement mode
      0080 037E 1465 .WORD BYTE_DISP_MODE-1$ : 9 - autoincrement deferred mode
      0089 0380 1466 .WORD BYTE_DEF_MODE-1$ : A - byte displacement mode
      0092 0382 1467 .WORD WORD_DISP_MODE-1$ : B - byte displacement deferred mode
      009C 0384 1468 .WORD WORD_DEF_MODE-1$ : C - word displacement mode
      00A6 0386 1469 .WORD LONG_DISP_MODE-1$ : D - word displacement deferred mode
      00B0 0388 1470 .WORD LONG_DEF_MODE-1$ : E - long displacement mode
      038A 1471
      038A 1472
      038A 1473
      038A 1474 REGISTER_MODE: : entrance
59 14 AD41 DE 038A 1475 MOVAL REG_R0(FP)[R1],R9 : R9 = location of the operand
      00B1 31 038F 1476 BRW ACCESS_OPERAND : access the operand
      0392 1477
      0392 1478
      0392 1479
      0392 1480 REG_DEF_MODE: : entrance
59 14 AD41 53 C1 0392 1481 ADDL3 R3,REG_R0(FP)[R1],R9 : R9 = location of the operand
      00A8 31 0398 1482 BRW ACCESS_OPERAND : access the operand
      039B 1483
      039B 1484
      039B 1485
      039B 1486 DECR_MODE: : entrance
      03 54 AD 1B E1 039B 1487 BBC #PSL_FPD,PSL(FP),1$ : side effects are allowed - skip
      FEF8 31 03A0 1488 BRW ACTION_LOOP : end of operand processing
      14 AD41 58 C2 03A3 1489 1$: SUBL2 R8,REG_R0(FP)[R1] : subtract data size from the register
59 14 AD41 53 C1 03A8 1490 ADDL3 R3,REG_R0(FP)[R1],R9 : R9 = location of the operand
      0092 31 03AE 1491 BRW ACCESS_OPERAND : access the operand
      03B1 1492
      03B1 1493
      03B1 1494
      03B1 1495 INCR_MODE: : entrance
      51 OF D1 03B1 1496 CMPL #15,R1 : is the register PC ?

```

```

03 54 AD 08 13 03B4 1497 BEQL 1$ ; yes - side effects are required
          1B E1 03B6 1498 BBC #PSL_FPD,PSL(FP),1$ ; side effects are allowed - skip
          FEDD 31 03BB 1499 BRW ACTION_LOOP ; end of operand processing
59 14 AD41 53 C1 03BE 1500 1$: ADDL3 R3,REG_RO(FP)[R1],R9 ; R9 = operand address
          14 AD41 58 C0 03C4 1501 ADDL2 R8,REG_RO(FP)[R1] ; increment the register
          0077 31 03C9 1502 BRW ACCESS_OPERAND ; access the operand
          03CC 1503 ;
          03CC 1504 ; Process an Autoincrement Deferred Mode Operand Specifier
          03CC 1505 ;
          03CC 1506 INCR_DEF MODE: ; entrance
          51 OF D1 03CC 1507 CMPL #15,R1 ; is the register PC ?
          03 54 AD 08 13 03CF 1508 BEQL 1$ ; yes - side effects are required
          1B E1 03D1 1509 BBC #PSL_FPD,PSL(FP),1$ ; side effects are allowed - skip
          FEC2 31 03D6 1510 BRW ACTION_LOOP ; end of operand processing
          59 14 AD41 D0 03D9 1511 1$: MOVL REG_RO(FP)[R1],R9 ; R9 = location of address longword
          59 69 53 C1 03DE 1512 ADDL3 R3,(R9),R9 ; R9 = operand address
          14 AD41 04 C0 03E2 1513 ADDL2 #4,REG_RO(FP)[R1] ; increment the register
          0059 31 03E7 1514 BRW ACCESS_OPERAND ; access the operand
          03EA 1515 ;
          03EA 1516 ; Process a Byte Displacement Mode Operand Specifier
          03EA 1517 ;
          59 50 BD 98 03EA 1518 BYTE_DISP MODE: ; entrance
          50 AD D6 03EA 1519 CVTBL @REG_PC(FP),R9 ; R9 = the byte displacement
          31 11 03EE 1520 INCL REG_PC(FP) ; increment the PC
          03F1 1521 BRB DISP_MODE ; finish processing the specifier
          03F3 1522 ;
          03F3 1523 ; Process a Byte Displacement Deferred Mode Operand Specifier
          03F3 1524 ;
          59 50 BD 98 03F3 1525 BYTE_DEF MODE: ; entrance
          50 AD D6 03F3 1526 CVTBL @REG_PC(FP),R9 ; R9 = the byte displacement
          33 11 03F7 1527 INCL REG_PC(FP) ; increment the PC
          03FA 1528 BRB DISP_DEF_MODE ; finish processing the specifier
          03FC 1529 ;
          03FC 1530 ; Process a Word Displacement Mode Operand Specifier
          03FC 1531 ;
          59 50 BD 32 03FC 1532 WORD_DISP MODE: ; entrance
          50 AD 02 C0 03FC 1533 CVTWL @REG_PC(FP),R9 ; R9 = the word displacement
          1E 11 0400 1534 ADDL2 #2,REG_PC(FP) ; increment the PC
          0404 1535 BRB DISP_MODE ; finish processing the specifier
          0406 1536 ;
          0406 1537 ; Process a Word Displacement Deferred Mode Operand Specifier
          0406 1538 ;
          59 50 BD 32 0406 1539 WORD_DEF MODE: ; entrance
          50 AD 02 C0 0406 1540 CVTWL @REG_PC(FP),R9 ; R9 = the word displacement
          1F 11 040A 1541 ADDL2 #2,REG_PC(FP) ; increment the PC
          040E 1542 BRB DISP_DEF_MODE ; finish processing the specifier
          0410 1543 ;
          0410 1544 ; Process a Long Displacement Mode Operand Specifier
          0410 1545 ;
          59 50 BD D0 0410 1546 LONG_DISP MODE: ; entrance
          50 AD 04 C0 0410 1547 MOVL @REG_PC(FP),R9 ; R9 = the longword displacement
          0A 11 0414 1548 ADDL2 #4,REG_PC(FP) ; increment the PC
          0418 1549 BRB DISP_MODE ; finish processing the specifier
          041A 1550 ;
          041A 1551 ; Process a Long Displacement Deferred Mode Operand Specifier
          041A 1552 ;
          041A 1553 LONG_DEF_MODE: ; entrance

```

```

59 50 BD D0 041A 1554 MOVL @REG_PC(FP),R9 ; R9 = the word displacement
50 AD 04 C0 041E 1555 ADDL2 #4,REG_PC(FP) ; increment the PC
    OB 11 0422 1556 BRB DISP_DEF_MODE ; finish processing the specifier
    0424 1557 ;
    0424 1558 ;
    0424 1559 ;
    0424 1560 DISP_MODE: ; entrance
59 14 AD41 C0 0424 1561 ADDL2 REG_R0(FP)[R1],R9 ; add the register value
59 59 53 C0 0429 1562 ADDL2 R3,R9 ; add the index modification
    0014 31 042C 1563 BRW ACCESS_OPERAND ; access the operand
    042F 1564 ;
    042F 1565 ;
    042F 1566 ;
    042F 1567 DISP_DEF_MODE: ; entrance
03 54 AD 1B E1 042F 1568 BBC #PSL_FPD,PSL(FP),1$ ; side effects are allowed - skip
    FE64 31 0434 1569 BRW ACTION_LOOP ; end of operand processing
59 14 AD41 C0 0437 1570 1$: ADDL2 REG_R0(FP)[R1],R9 ; add the register value
59 69 53 C1 043C 1571 ADDL3 R3,(R9),R9 ; R9 = the operand location
    0000 31 0440 1572 BRW ACCESS_OPERAND ; access the operand
    0443 1573 ;
    0443 1574 ;
    0443 1575 ;
    0443 1576 ACCESS_OPERAND: ; entrance
03 54 AD 1B E1 0443 1577 BBC #PSL_FPD,PSL(FP),1$ ; is the FPD bit set ?
    FE50 31 0448 1578 BRW ACTION_LOOP ; yes - don't do anything
03 01 FE50 CF 044B 1579 1$: CASEL R6,#1,#3 ; branch on the operand access
    0008 044F 1580 2$: .WORD 3$-2$ ; 1 - operand with read access
    0012 0451 1581 .WORD 4$-2$ ; 2 - operand with write access
    0012 0453 1582 .WORD 4$-2$ ; 3 - operand with modify access
    0057 0455 1583 .WORD 9$-2$ ; 4 - operand with address access
6A 69 58 28 0457 1584 3$: MOV3 R8,(R9),(R10) ; make a copy of the operand value
    5A 58 C0 045B 1585 ADDL2 R8,R10 ; update the copy index
    FE3A 31 045E 1586 BRW ACTION_LOOP ; end of operand processing
    05 52 D1 0461 1587 4$: CMPL R2,#5 ; register mode operand ?
    15 13 0464 1588 BEQL 5$ ; yes - bypass
    50 6948 9E 0466 1589 MOVAB (R9)[R8],R0 ; R0 = address following operand
    5E 50 D1 046A 1590 CMPL R0,SP ; is operand below the frame ?
    0C 1B 046D 1591 BLEQU 5$ ; yes - bypass
    50 58 AD 9E 046F 1592 MOVAB LOCAL_END(FP),R0 ; R0 = end of local storage
    59 50 D1 0473 1593 CMPL R0,R9 ; does the operand follow the frame ?
    03 1B 0476 1594 BLEQU 5$ ; yes - skip
    59 50 D0 0478 1595 MOVL R0,R9 ; change the operand address
    8B 59 D0 047B 1596 5$: MOVL R9,(R11)+ ; store the store address
    8B 58 D0 047E 1597 MOVL R8,(R11)+ ; store the store length
6B 58 00 6E 00 2C 0481 1598 MOV3 #0,(SP),#0,R8,(R11) ; clear the stored value
54 54 AD 02 04 00 04 F0 0487 1599 INSV #PSLM_Z,#0,#4,PSL(FP) ; describe a zero value
    01 FD AD 8F 048D 1600 CASEB FAULT_TYPE(FP),#1,#2 ; branch on the fault type
    000E 0492 1601 6$: .WORD 8$-6$ ; 1 - floating underflow
    0006 0494 1602 .WORD 7$-6$ ; 2 - floating overflow
    0006 0496 1603 .WORD 7$-6$ ; 3 - floating divide by zero
    54 AD 08 C8 0498 1604 7$: BISL2 #PSLM_N,PSL(FP) ; set the N bit in the PSL
    00 6B 0F E2 049C 1605 BBSS #15,(R11),8$ ; make the value a reserved value
    5B 58 C0 04A0 1606 8$: ADDL2 R8,R11 ; increment the store index
    FDF5 31 04A3 1607 BRW ACTION_LOOP ; end of operand processing
    8A 69 D0 04A6 1608 9$: MOVL (R9),(R10)+ ; copy the operand address
    FDEF 31 04A9 1609 BRW ACTION_LOOP ; end of operand processing
    04AC 1610 ;

```

				04AC	1612	:	
				04AC	1613	:	Process a Word Branch Displacement Operand
				04AC	1614	:	entered by branching
				04AC	1615	:	parameters: (See the discussion of ACTION_LOOP.)
				04AC	1616	:	
				04AC	1617	:	Discussion
				04AC	1618	:	This routine process a word branch displacement operand
				04AC	1619	:	and stores the branch address in BRANCH_ADDRESS.
				04AC	1620	:	
				04AC	1621	:	
				04AC	1622	:	
				04AC	1623	:	
				04AC	1624	:	
				04AC	1625	:	WORD_BRANCH: ; entrance
				04B0	1626	:	@REG_PC(FP),R0 ; R0 = branch displacement
				04B4	1627	:	#2,REG_PC(FP) ; increment the PC
EC AD	50	AD	50	02	C0	:	R0,REG_PC(FP),BRANCH_ADDRESS(FP) ; save the branch location
				50	C1	:	ACTION_LOOP ; end of operand processing
				FDDE	31	:	
				04BA	1628	:	
				04BD	1629	:	

			04BD	1631	:			
			04BD	1632	:			
			04BD	1633	:			
			04BD	1634	:			
			04BD	1635	:			
			04BD	1636	:			
			04BD	1637	:			
			04BD	1638	:			
			04BD	1639	:			
			04BD	1640	:			
			04BD	1641	:			
			04BD	1642	:			
			04BD	1643	:			
			04BD	1644	:			
			04BD	1645	:			
			04BD	1646	:			
			04BD	1647	:			
			04BD	1648	:			
			04BD	1649	:			
			04BD	1650	:			
			04BD	1651	:			
			04BD	1652	:	CHECK_ACB:		
54	AD	01	CA	04BD	1652	BICL2	#PSLM_C,PSL(FP)	: entrance
	04	FE	E9	04C1	1653	BLBC	CARRY_BIT(FP),1\$: clear the C bit in the PSL
54	AD	01	C8	04C5	1654	BISL2	#PSLM_C,PSL(FP)	: was the carry bit originally set ?
FD	AD	01	91	04C9	1655	1\$:	CMPB	#1,FAULT_TYPE(FP)
			FA	04CD	1656		BEQL	1\$
			FDC9	04CF	1657		BRW	ACTION_LOOP
50	C4	AD48	9E	04D2	1658	2\$:	MOVAB	READ_AREA(FP)[R8],R0
			1E	04D7	1659		BSBB	ACB_TEST
			0A	04D9	1660		BLSS	3\$
50		58	C2	04DB	1661		SUBL2	R8,R0
			17	04DE	1662		BSBB	ACB_TEST
			0D	04E0	1663		BGTR	4\$
			FDB6	04E2	1664		BRW	ACTION_LOOP
50		58	C2	04E5	1665	3\$:	SUBL2	R8,R0
			0D	04E8	1666		BSBB	ACB_TEST
			03	04EA	1667		BLSS	4\$
			FDAC	04EC	1668		BRW	ACTION_LOOP
50	AD	EC	AD	04EF	1669	4\$:	MOVL	BRANCH_ADDRESS(FP),REG_PC(FP)
			FDA4	04F4	1670		BRW	ACTION_LOOP
				04F7	1671		:	: finish up

CHECK_ACB - Perform Post Instruction Checking for ACBx

entered by branching

parameters: (See the discussion of ACTION_LOOP.)

Discussion

This routine performs the post instruction checking required to simulate floating arithmetic traps for the ACBx instructions. First the carry bit is set equal to the value of that bit when the instruction was started. If the exception was an overflow, then the checking is complete. If the exception was an underflow, then the step and limit in the read area are tested to see whether the jump should be taken under the assumption that the index is zero. If the jump should be taken, then the the branch destination address is stored into the simulated PC.

: entrance
: clear the C bit in the PSL
: was the carry bit originally set ?
: yes - make sure it's set in the PSL
: floating underflow ?
: yes - skip
: no - finish up
: R0 = location of second operand
: test the addend
: it's negative - bypass
: R0 = location of first operand
: test the limit
: it's positive - bypass
: finish up
: R0 = location of first operand
: test the limit
: it's negative - skip
: finish up
: set up the branch
: finish up

56	5D	D0	053E	1762	5\$:	MOVL	FP,R6	:	R6 = copy of frame pointer		
	58	D4	0541	1763		CLRL	R8	:	condition XQT_POLY to generate fault		
063C	'CF	00	FB	0543	1764	CALLS	#0,W^XQT_POLY	:	generate the fault		
	58	D4	0548	1765		CLRL	R8	:	clear the exception code area		
03	59	04	D0	054A	1766	MOVL	#PSLM_Z,R9	:	R9 = condition code for zero		
01	57	CF	054D	1767		CASEL	R7,#1,#3	:	branch on the data type		
		0008	'	0551	1768	6\$:	.WORD	7\$-6\$	1 - floating		
		001E	'	0553	1769		.WORD	8\$-6\$	2 - double floating		
		001E	'	0555	1770		.WORD	8\$-6\$	3 - grand floating		
		0034	'	0557	1771		.WORD	9\$-6\$	4 - huge floating		
	50	D4	0559	1772	7\$:	CLRL	R0	:	clear the result so far		
	51	D4	055B	1773		CLRL	R1	:	clear the argument		
53	20	AD	04	C1	055D	1774	ADDL3	#4,REG_R3(FP),R3	:	R3 = location of coefficients	
52	1C	AD	01	83	0562	1775	SUBB3	#1,REG_R2(FP),R2	:	insert remaining coefficients in R2	
			34	13	0567	1776	BEQL	11\$:	the count is zero - bypass	
	51	18	AD	D0	0569	1777	MOVL	REG_R1(FP),R1	:	R1 = the argument	
			26	11	056D	1778	BRB	10\$:	bypass	
			50	7C	056F	1779	8\$:	CLRQ	R0	:	clear the result so far
			54	7C	0571	1780		CLRQ	R4	:	clear the argument
53	20	AD	08	C1	0573	1781	ADDL3	#8,REG_R3(FP),R3	:	R3 = location of coefficients	
52	1C	AD	01	83	0578	1782	SUBB3	#1,REG_R2(FP),R2	:	insert remaining coefficients in R2	
			1E	13	057D	1783	BEQL	11\$:	the count is zero - bypass	
	54	24	AD	7D	057F	1784	MOVQ	REG_R4(FP),R4	:	R4,R5 = the argument	
			10	11	0583	1785	BRB	10\$:	bypass	
			50	7C	0585	1786	9\$:	CLRQ	R0	:	clear first part of result so far
			51	7C	0587	1787		CLRQ	R1	:	clear second part of result so far
55	28	AD	10	C1	0589	1788	ADDL3	#16,REG_R5(FP),R5	:	R5 = location of coefficients	
54	24	AD	01	83	058E	1789	SUBB3	#1,REG_R4(FP),R4	:	insert remaining coefficients in R4	
			08	13	0593	1790	BEQL	11\$:	the count is zero - bypass	
	58	01	D0	0595	1791	10\$:	MOVL	#1,R8	:	request the FPD bit this time	
063C	'CF	00	FB	0598	1792		CALLS	#0,W^XQT_POLY	:	execute appropriate POLY instruction	
03	01	57	CF	059D	1793	11\$:	CASEL	R7,#1,#3	:	branch on the data type	
		0008	'	05A1	1794	12\$:	.WORD	13\$-12\$	1 - floating		
		0016	'	05A3	1795		.WORD	14\$-12\$	2 - double floating		
		0016	'	05A5	1796		.WORD	14\$-12\$	3 - grand floating		
		0028	'	05A7	1797		.WORD	15\$-12\$	4 - huge floating		
	14	AD	50	7D	05A9	1798	13\$:	MOVQ	R0,REG_R0(FP)	:	store the result so and argument
	1C	AD	52	90	05AD	1799		MOVB	R2,REG_R2(FP)	:	store the remaining coefficients
	20	AD	53	D0	05B1	1800		MOVL	R3,REG_R3(FP)	:	store location of coefficients
			22	11	05B5	1801		BRB	16\$:	bypass
	14	AD	50	7D	05B7	1802	14\$:	MOVQ	R0,REG_R0(FP)	:	store the result so far
	1C	AD	52	90	05BB	1803		MOVB	R2,REG_R2(FP)	:	store the remaining coefficients
	20	AD	53	D0	05BF	1804		MOVL	R3,REG_R3(FP)	:	store location of coefficients
	24	AD	54	7D	05C3	1805		MOVQ	R4,REG_R4(FP)	:	store the argument
			10	11	05C7	1806		BRB	16\$:	bypass
	14	AD	50	7D	05C9	1807	15\$:	MOVQ	R0,REG_R0(FP)	:	store first part of result so far
	1C	AD	52	7D	05CD	1808		MOVQ	R2,REG_R2(FP)	:	store second part of result so far
	24	AD	54	90	05D1	1809		MOVB	R4,REG_R4(FP)	:	store the remaining coefficients
	28	AD	55	D0	05D5	1810		MOVL	R5,REG_R5(FP)	:	store location of coefficients
		58	0C	D1	05D9	1811	16\$:	CMPL	#SS\$_ACCVIO,R8	:	was there an access violation ?
			03	12	05DC	1812		BNEQ	17\$:	no - skip
		0136		31	05DE	1813		BRW	ACCESS FAULT	:	process the access violation fault
58	00000454	8F	D1	05E1	1814	17\$:	CMPL	#SS\$_ROPRAND,R8	:	was there a reserved operand ?	
			03	12	05E8	1815		BNEQ	18\$:	no - skip
		0140		31	05EA	1816		BRW	OPERAND FAULT	:	process the reserved operand fault
58	0000048C	8F	D1	05ED	1817	18\$:	CMPL	#SS\$_FLTQVF,R8	:	was there a overflow trap ?	
			12	12	05F4	1818		BNEQ	20\$:	no - skip

54 AD	04	00	0C	F0	05F6	1819	INSV	#PSLM_NZ,#0,#4,PSL(FP)	: describe a reserved operand
		57	04	D1	05FC	1820	CMPL	#HUGE_TYPE,R7	: is the data type huge ?
			04	12	05FF	1821	BNEQ	19\$: no - skip
	4C AD		10	C0	0601	1822	ADDL2	#16,REG_SP(FP)	: unstack the argument
			0146	31	0605	1823	BRW	OVERFLOW_TRAP	: process the overflow trap
58	000004B4		0F	D1	0608	1824	CMPL	#SS\$_FLT0VF_F,R8	: was there a overflow fault ?
			03	12	060F	1825	BNEQ	21\$: no - skip
			FF0C	31	0611	1826	BRW	3\$: process the overflow fault
54 AD	04	00	59	F0	0614	1827	INSV	R9,#0,#4,PSL(FP)	: set the condition codes
	03	01	57	CF	061A	1828	CASEL	R7,#1,#3	: branch on the data type
			0008'		061E	1829	.WORD	23\$-22\$: 1 - floating
			0008'		0620	1830	.WORD	23\$-22\$: 2 - double floating
			0008'		0622	1831	.WORD	23\$-22\$: 3 - grand floating
			0011'		0624	1832	.WORD	24\$-22\$: 4 - huge floating
1C AD	18	08	00	F0	0626	1833	INSV	#0,#8,#24,REG_R2(FP)	: perform remaining clear
			0111	31	062C	1834	BRW	UNDERFLOW_TRAP	: process the floating underflow
24 AD	18	08	00	F0	062F	1835	INSV	#0,#8,#24,REG_R4(FP)	: perform remaining clear
	4C AD		10	C0	0635	1836	ADDL2	#16,REG_SP(FP)	: unstack the argument
			0104	31	0639	1837	BRW	UNDERFLOW_TRAP	: process the floating underflow
					063C	1838	:		


```

B1'AF 01 08 55FD 0697 1897 12$: POLYG #1.0,#1,B^GRAND_TABLE ; execute the POLYG instruction
          59 DC 069D 1898 MOVPSL R9 ; R9 = resulting PSL value
          04 069F 1899 RET ; return
C1'AF 01 08 75FD 06A0 1900 13$: POLYH #1.0,#1,B^HUGE_TABLE ; execute the POLYH instruction
          59 DC 06A6 1901 MOVPSL R9 ; R9 = resulting PSL value
          04 06A8 1902 RET ; return
          06A9 1903 ;
          06A9 1904 ; Coefficients for POLYF Instruction
          06A9 1905 ;
          06A9 1906 FLOAT_TABLE:
          FFFF7FFF 06A9 1907 .LONG ^X FFFF7FFF ; largest floating value
          FFFF7FFF 06AD 1908 .LONG ^X FFFF7FFF ; largest floating value
          06B1 1909 ;
          06B1 1910 ; Coefficients for POLYD and POLYG Instructions
          06B1 1911 ;
          06B1 1912 DOUBLE_TABLE:
          06B1 1913 GRAND_TABLE:
          FFFF7FFF 06B1 1914 .LONG ^X FFFF7FFF ; largest double or grand value
          FFFFFFFF 06B5 1915 .LONG ^X FFFFFFFF
          FFFF7FFF 06B9 1916 .LONG ^X FFFF7FFF ; largest double or grand value
          FFFFFFFF 06BD 1917 .LONG ^X FFFFFFFF
          06C1 1918 ;
          06C1 1919 ; Coefficients for POLYH Instruction
          06C1 1920 ;
          06C1 1921 HUGE_TABLE:
          FFFF7FFF 06C1 1922 .LONG ^X FFFF7FFF ; largest huge floating value
          FFFFFFFF 06C5 1923 .LONG ^X FFFFFFFF
          FFFFFFFF 06C9 1924 .LONG ^X FFFFFFFF
          FFFFFFFF 06CD 1925 .LONG ^X FFFFFFFF
          FFFF7FFF 06D1 1926 .LONG ^X FFFF7FFF ; largest huge floating value
          FFFFFFFF 06D5 1927 .LONG ^X FFFFFFFF
          FFFFFFFF 06D9 1928 .LONG ^X FFFFFFFF
          FFFFFFFF 06DD 1929 .LONG ^X FFFFFFFF
          06E1 1930 ;

```

				06E1	1932	:	
				06E1	1933	:	
				06E1	1934	:	
				06E1	1935	:	
				06E1	1936	:	
				06E1	1937	:	
				06E1	1938	:	
				06E1	1939	:	
				06E1	1940	:	
				06E1	1941	:	
				06E1	1942	:	
				06E1	1943	:	
				06E1	1944	:	
				06E1	1945	:	
				06E1	1946	:	
				06E1	1947	:	
				06E1	1948	:	
				06E1	1949	:	
				06E1	1950	:	
				06E1	1951	:	
				06E1	1952	:	
				06E1	1953	:	
			0000	06E3	1954	:	
50	04	AC	7D	06E7	1955	:	
	08	A1	D5	06EA	1956	:	
			25	12	06EC	1957	
58	04	A0	D0	06F0	1958	:	
	0000043C	8F	D1	06F7	1959	:	
			18	11	06F9	1960	
	58	0C	D1	06FC	1961	:	
			12	06FE	1962	:	
FB	A6	08	A0	90	0703	1963	
EB	A6	0C	A0	D0	0708	1964	
			7E	7C	070A	1965	1\$:
00000000	'GF	02	FB	32	0711	1966	2\$:
50	0918	8F	04	0716	1967		
			04	0717	1968		


```

POLY_HANDLER:
    .WORD 0
    MOVQ 4(AP),R0
    TSTL 8(R1)
    BNEQ 2$
    MOVL 4(R0),R8
    CMPL #SS$_OPCDEC,R8
    BRB 2$
    CMPL #SS$_ACCVIO,R8
    BNEQ 1$
    MOVB 8(R0),ACCVIO_REASON(R6)
    MOVL 12(R0),ACCVIO_ADDRESS(R6)
    CLRQ -(SP)
    CALLS #2,G*SYSSUNWIND
    CVTWL #SS$_RESIGNAL,R0
    RET
    ;
    ; entrance
    ; entry mask
    ; R0 and R1 = locations of arrays
    ; condition from establisher frame ?
    ; no - bypass
    ; R8 = condition code
    ; opcode reserved to DEC ?
    ; yes - it's for the Emulator
    ; access violation ?
    ; no - bypass
    ; save the reason mask
    ; save the accessed address
    ; default PC and level for unwind
    ; unwind the frame for XQT_POLY
    ; specify condition not handled
    
```

```

0717 1970
0717 1971
0717 1972
0717 1973
0717 1974
0717 1975
0717 1976
0717 1977
0717 1978
0717 1979
0717 1980
0717 1981
0717 1982
0717 1983
0717 1984
0717 1985
0717 1986
0717 1987
0717 1988
0717 1989
0717 1990 ACCESS_FAULT:
E8 AD DD 0717 1991 PUSHL ACCVIO_ADDRESS(FP) ; entrance
FB AD DD 071A 1992 PUSHL ACCVIO_REASON(FP) ; push the accessed address
OC DD 071D 1993 PUSHL #SS$_ACCVIO ; push the reason mask
03 DD 071F 1994 PUSHL #3 ; push the condition code
50 AD F0 AD D0 0721 1995 MOVL INST_ADDRESS(FP),REG_PC(FP) ; push the number of parameters
00 54 AD 1E E5 0726 1996 BBCC #PSL_TP,PSL(FP),1$ ; restore the instruction address
38 11 072B 1997 1$: BRB SIGNAL_START ; clear the trace pending bit
072D 1998 ; signal the condition
072D 1999
072D 2000
072D 2001 OPERAND_FAULT:
7E 0454 8F 32 072D 2002 CVTWL #SS$_ROPRAND,-(SP) ; entrance
01 DD 0732 2003 PUSHL #1 ; push the condition code
50 AD F0 AD D0 0734 2004 MOVL INST_ADDRESS(FP),REG_PC(FP) ; push the number of parameters
00 54 AD 1E E5 0739 2005 BBCC #PSL_TP,PSL(FP),1$ ; restore the instruction address
25 11 073E 2006 1$: BRB SIGNAL_START ; clear the trace pending bit
0740 2007 ; signal the condition
0740 2008
0740 2009
0740 2010 UNDERFLOW_TRAP:
00 54 AD 1B E5 0740 2011 BBCC #PSL_FPD,PSL(FP),1$ ; entrance
7E 049C 8F 32 0745 2012 1$: CVTWL #SS$_FLTUND,-(SP) ; clear FPD in the PSL
01 DD 074A 2013 PUSHL #1 ; push the condition code
17 11 074C 2014 BRB SIGNAL_START ; push the number of parameters
074E 2015 ; signal the condition
074E 2016
074E 2017
074E 2018 OVERFLOW_TRAP:
00 54 AD 1B E5 074E 2019 BBCC #PSL_FPD,PSL(FP),1$ ; entrance
7E 048C 8F 32 0753 2020 1$: CVTWL #SS$_FLTOVF,-(SP) ; clear FPD in the PSL
01 DD 0758 2021 PUSHL #1 ; push the condition code
09 11 075A 2022 BRB SIGNAL_START ; push the number of parameters
075C 2023 ; signal the condition
075C 2024
075C 2025
075C 2026 DIVIDE_TRAP: ; entrance

```

Condition Signaling Routines

General Discussion

The following routines signal the various types of conditions that the input fault types can be converted into. This is done by pushing an abbreviated form of the signal vector (with the PC, PSL pair missing) and branching to SIGNAL_START. If the signaled condition is a trap then the FPD bit is cleared in the PSL. If the signaled condition is a fault, then the TP bit is cleared in the PSL and the simulated PC is changed to the instruction location. For access violations, the reason mask and the accessed address are taken from the designated cells in the frame.

Signal an Access Violation

```

; entrance
; push the accessed address
; push the reason mask
; push the condition code
; push the number of parameters
; restore the instruction address
; clear the trace pending bit
; signal the condition

```

Signal a Reserved Operand Fault

```

; entrance
; push the condition code
; push the number of parameters
; restore the instruction address
; clear the trace pending bit
; signal the condition

```

Signal a Floating Underflow Trap

```

; entrance
; clear FPD in the PSL
; push the condition code
; push the number of parameters
; signal the condition

```

Signal a Floating Overflow Trap

```

; entrance
; clear FPD in the PSL
; push the condition code
; push the number of parameters
; signal the condition

```

Signal a Floating/Decimal Divide by Zero Trap

```

; entrance

```

LIBSSIM_TRAP
1-003

E 4
- Simulate floating trap
LIBSSIM_TRAP - Convert Floating Faults t 16-SEP-1984 00:18:50 VAX/VMS Macro V04-00 Page 45
6-SEP-1984 11:10:54 [LIBRTL.SRC]LIBSIMTRA.MAR;1 (19)

7E	0494	8F	32	075C	2027	CVTWL	#SS\$_FLTDIV,-(SP)	:	push the condition code
		01	DD	0761	2028	PUSHL	#1	:	push the number of parameters
		00	11	0763	2029	BRB	SIGNAL_START	:	signal the condition
				0765	2030	:			

```

0765 2032
0765 2033
0765 2034
0765 2035
0765 2036
0765 2037
0765 2038
0765 2039
0765 2040
0765 2041
0765 2042
0765 2043
0765 2044
0765 2045
0765 2046
0765 2047
0765 2048
0765 2049
0765 2050
0765 2051
0765 2052
0765 2053
0765 2054
0765 2055
0765 2056
0765 2057
0765 2058
0765 2059
0765 2060
0765 2061
0765 2062
0765 2063
0765 2064
0765 2065
0765 2066
0765 2067
0765 2068
0765 2069
0769 2070
076D 2071
0770 2072
0774 2073
0777 2074
077B 2075
077F 2076
0782 2077
0786 2078
0789 2079
078D 2080
0790 2081
0793 2082
0797 2083
079A 2084
079E 2085
07A1 2086
07A6 2087
07AB 2088

```

SIGNAL_START - Build the Parameter Blocks for SIGNAL

entered by branching

parameters: (SP) = Truncated Signal Array Size (M)
4(SP) = Condition Code
8(SP) = First Signal Argument
:
:
4*<M-1>(SP) = Last Signal Argument

Discussion

This routine builds the signal and mechanism arrays for a condition generated by the converter. It is entered with the signal array for the condition except for the PC and PSL pair pushed onto the converter's stack (with the pushed array length correspondingly shortened). The signal array, mechanism array, and the handler parameter block are then constructed on the user's emulated stack. The routine then removes the converter frame from the stack and enters the signal dispatching loop at SIGNAL.

- Notes: 1. The precise format of the information pushed onto the user's stack is given in the description of SIGNAL below.
2. The method of getting out of the converter consists of adjusting the alignment bits and the parameter count for the converter's frame so that the stack pointer upon return is just where we want it to be. Before returning, the return PC in the frame is changed to the place we want to branch to.

```

56 4C AD DO 0765 2069 SIGNAL_START:
76 50 AD 7D 0769 2070      MOVL  REG_SP(FP),R6
57 8E DO 076D 2071      MOVQ  REG_PC(FP),-(R6)
58 57 02 78 0770 2072      MOVL  (SP)+,R7
56 58 C2 0774 2073      ASHL  #2,R7,R8
66 6E 58 28 0777 2074      SUBL2 R8,R6
76 57 02 C1 077B 2075      MOVC3 R8,(SP),(R6)
76 76 01 DO 077F 2076      ADDL3 #2,R7,-(R6)
76 14 AD 7D 0782 2077      MOVL  #1,-(R6)
76 76 03 CE 0786 2078      MOVQ  REG_R0(FP),-(R6)
76 48 AD DO 0789 2079      MNEGL #3,-(R6)
76 76 04 DO 078D 2080      MOVL  REG_FP(FP),-(R6)
76 56 DO 0790 2081      MOVL  #4,-(R6)
76 1C A6 9E 0793 2082      MOVL  R6,-(R6)
76 76 02 DO 0797 2083      MOVAB 28(R6),-(R6)
50 48 AD 9E 079A 2084      MOVL  #2,-(R6)
08 AD 44 AD 7D 079E 2085      MOVAB FRAME_END+4(FP),R0
10 AD C0 AF 9E 07A1 2086      SUBL2 R0,R6
51 56 02 00 EF 07A6 2087      MOVQ  REG_AP(FP),SAVE_AP(FP)
                                MOVAB B^SIGNAL,SAVE_PC(FP)
                                EXTZV #0,#2,R6,R1

```

```

: entrance
: R6 = user's SP
: push the user's PC and PSL
: R7 = number of signal parameters
: R8 = number of bytes to move
: extend the user stack
: move the parameters
: push the signal array length
: push code for SIGNAL (vs. STOP)
: push the user's R0 and R1
: push -3 (depth number)
: push the user's FP
: push the mechanism array length
: push the mechanism array location
: push the signal array location
: push the handler parameter count
: R0 = end of call frame + 4
: R6 = distance to user's SP
: put AP and FP back into place
: store the return PC
: R1 = stack alignment bits

```

LIB Sym
CHA
DSC
DSC
DSC
LIB
LIB
STR

PSE

\$AB
_LI

Pha

Ini
Com
Pas
Sym
Pas
Sym
Pse
Cro
Ass

The
804
The
163
8 p

Mac

_S2
190
The
MAC

LIBSSIM_TRAP
1-003

- Simulate floating trap

LIBSSIM_TRAP - Convert Floating

G 4

16-SEP-1984 00:18:50

VAX/VMS Macro V04-00

Page 47

6-SEP-1984 11:10:54

[LIBRTL.SRC]LIBSIMTRA.MAR;1

(20)

06 AD 02 0E 51
FC A0 56 FE 8F

F0 07B0 2089
C0 07B6 2090
78 07B9 2091
04 07BF 2092
07C0 2093

INSV
ADDL2
ASHL
RET
;

R1,#MASK_ALIGN,#2,SAVE_MASK(FP) ; store them
R1,R0 ; compute the argument count location
#-2,R6,-4(R0) ; store the argument count
; return (to SIGNAL)

**F

```

07C0 2095
07C0 2096
07C0 2097
07C0 2098
07C0 2099
07C0 2100
07C0 2101
07C0 2102
07C0 2103
07C0 2104
07C0 2105
07C0 2106
07C0 2107
07C0 2108
07C0 2109
07C0 2110
07C0 2111
07C0 2112
07C0 2113
07C0 2114
07C0 2115
07C0 2116
07C0 2117
07C0 2118
07C0 2119
07C0 2120
07C0 2121
07C0 2122
07C0 2123
07C0 2124
07C0 2125
07C0 2126
07C0 2127
07C0 2128
07C0 2129
07C0 2130
07C0 2131
07C0 2132
07C0 2133
07C0 2134
07C0 2135
07C0 2136
07C0 2137
07C0 2138
07C0 2139
07C0 2140
07C0 2141
07C0 2142
07C0 2143
07C0 2144
07C0 2145
07C0 2146
07C0 2147
07C6 2148

```

SIGNAL - Signal the Condition
entered by branching
parameters: (Described in Note 3)

Discussion

Following is a description of the information which is assumed to be pushed onto the stack when the routine SIGNAL is entered. The values are all longwords.

Handler Parameter Block:

- (SP) 2 (handler parameter block length)
- 4(SP) signal array location
- 8(SP) mechanism array location

Mechanism Array:

- 12(SP) 4 (mechanism array length)
- 16(SP) user's FP (establisher frame)
- 20(SP) -3 (establisher depth)
- 24(SP) user's R0
- 28(SP) user's R1

Information Not Part of any Array:

- 32(SP) 1 (code for SIGNAL)

Signal Array:

- 36(SP) signal array length M
- 40(SP) condition code
- 44(SP) first signal argument
- .
- .
- <4*M>+28(SP) last signal argument
- <4*M>+32(SP) user's PC
- <4*M>+36(SP) user's PSL

The user's stack pointer should coincide with the address <4*M>+40(SP).

Jump to the VMS routine which looks for handlers to call. Execution will not return to us.

```

00000000'GF 17 SIGNAL: JMP G^SYSSRCHANDLER ; entrance

```

LIBSSIM_TRAP
1-003

I 4
- Simulate floating trap 16-SEP-1984 00:18:50 VAX/VMS Macro V04-00 Page 49
LIBSSIM_TRAP - Convert Floating Faults t 6-SEP-1984 11:10:54 [LIBRTL.SRC]LIBSIMTRA.MAR;1 (22)

```
07C6 2150 : *****  
07C6 2151 : *  
07C6 2152 : *  
07C6 2153 : * End of VAX/VMS Common RTL File LIBSIMTRA.MAR *  
07C6 2154 : *  
07C6 2155 : *  
07C6 2156 : *****  
07C6 2157 :  
07C6 2158 .END
```

LIB
1-00

LIBSSIM TRAP
Symbol Table

- Simulate floating trap

J 4

16-SEP-1984 00:18:50 VAX/VMS Macro V04-00 Page 50
6-SEP-1984 11:10:54 [LIBRTL.SRC]LIBSIMTRA.MAR;1 (22)

ACB_CHECK	= 00000007			GRAND_TABLE	000006B1 R	01
ACB_DOUBLE	= 00000027			GRAND_TYPE	= 00000003	
ACB_FLOAT	= 00000017			HANDLER	= 00000000	
ACB_GRAND	= 00000037			HUGE_TABLE	000006C1 R	01
ACB_HUGE	= 00000047			HUGE_TYPE	= 00000004	
ACB_TEST	000004F7 R	01		INCR_DEF_MODE	000003CC R	01
ACCESS_FAULT	00000717 R	01		INCR_MODE	000003B1 R	01
ACCESS_OPERAND	00000443 R	01		INDEX_MODE	0000034F R	01
ACCVIO_ADDRESS	= FFFFFFFE8			INST_ADDRESS	= FFFFFFFF0	
ACCVIO_REASON	= FFFFFFFFB			LIBSSIM_TRAP	00000000 RG	01
ACTION_0	000001C7 R	01		LITERAL_MODE	00000302 R	01
ACTION_1	000001CF R	01		LOCAL_END	= 00000058	
ACTION_2	000001D7 R	01		LOCAL_START	= FFFFFFFA0	
ACTION_3	000001E0 R	01		LONG_DEF_MODE	0000041A R	01
ACTION_4	000001ED R	01		LONG_DISP_MODE	00000410 R	01
ACTION_5	000001FD R	01		LONG_TYPE	= 00000007	
ACTION_6	0000020C R	01		MASK_ALIGN	= 0000000E	
ACTION_7	00000218 R	01		MODIFY_ACCESS	= 00000003	
ACTION_COUNT	= FFFFFFFFC			MODIFY_DOUBLE	= 00000023	
ACTION_F	00000221 R	01		MODIFY_FLOAT	= 00000013	
ACTION_LOOP	0000029B R	01		MODIFY_GRAND	= 00000033	
ACTION_PTR	= FFFFFFFF4			MODIFY_HUGE	= 00000043	
ADDRESS_ACCESS	= 00000004			OPERAND	000002CB R	01
ADDRESS_BYTE	= 00000054			OPERAND_FAULT	0000072D R	01
BRANCH_ADDRESS	= FFFFFFFEC			OVERFLOW_TRAP	0000074E R	01
BRANCH_WORD	= 00000005			PATTERNS	00000225 R	01
BYTE_DEF_MODE	000003F3 R	01		PATTERN_ACBD	00000282 R	01
BYTE_DISP_MODE	000003EA R	01		PATTERN_ACBF	0000027C R	01
BYTE_TYPE	= 00000005			PATTERN_ACBG	00000288 R	01
CALL_ARGS	= 00000028			PATTERN_ACBH	0000028E R	01
CARRY_BIT	= FFFFFFFFE			PATTERN_ADDD2	00000228 R	01
CHECK_ACB	000004BD R	01		PATTERN_ADDD3	00000235 R	01
CHECK_POLY	00000511 R	01		PATTERN_ADDF2	00000225 R	01
CLASS_0	000001C3 R	01		PATTERN_ADDF3	00000231 R	01
CLASS_1	000001CB R	01		PATTERN_ADDG2	0000022B R	01
CLASS_2	000001D3 R	01		PATTERN_ADDG3	00000239 R	01
CLASS_3	000001DB R	01		PATTERN_ADDH2	0000022E R	01
CLASS_4	000001E5 R	01		PATTERN_ADDH3	0000023D R	01
CLASS_5	000001F5 R	01		PATTERN_CVTDF	00000241 R	01
CLASS_6	00000205 R	01		PATTERN_CVTGF	00000244 R	01
CLASS_7	00000213 R	01		PATTERN_CVTHD	0000024A R	01
CLASS_F	0000021D R	01		PATTERN_CVTHF	00000247 R	01
CLASS_TAB	000001A3 R	01		PATTERN_CVTHG	0000024D R	01
CONVERT_HANDLER	00000178 R	01		PATTERN_DIVD2	00000228 R	01
DATA_LENGTHS	00000294 R	01		PATTERN_DIVD3	00000235 R	01
DECR_MODE	0000039B R	01		PATTERN_DIVF2	00000225 R	01
DISP_DEF_MODE	0000042F R	01		PATTERN_DIVF3	00000231 R	01
DISP_MODE	00000424 R	01		PATTERN_DIVG2	0000022B R	01
DIVIDE_TRAP	0000075C R	01		PATTERN_DIVG3	00000239 R	01
DOUBLE_TABLE	000006B1 R	01		PATTERN_DIVH2	0000022E R	01
DOUBLE_TYPE	= 00000002			PATTERN_DIVH3	0000023D R	01
FAULT_TO_TRAP	0000007B R	01		PATTERN_EMODD	00000256 R	01
FAULT_TYPE	= FFFFFFFFD			PATTERN_EMODF	00000250 R	01
FINISH	00000158 R	01		PATTERN_EMODG	0000025C R	01
FLOAT_TABLE	000006A9 R	01		PATTERN_EMODH	00000262 R	01
FLOAT_TYPE	= 00000001			PATTERN_MULD2	00000228 R	01
FRAME_END	= 00000044			PATTERN_MULD3	00000235 R	01

LIBSSIM TRAP
Symbol Table

- Simulate floating trap

K 4

PATTERN_MULF2	00000225	R	01	REG_R5	= 00000028		
PATTERN_MULF3	00000231	R	01	REG_SP	= 0000004C		
PATTERN_MULG2	00000228	R	01	SAVE_ALIGN	= FFFFFFFF		
PATTERN_MULG3	00000239	R	01	SAVE_AP	= 00000008		
PATTERN_MULH2	0000022E	R	01	SAVE_MASK	= 00000006		
PATTERN_MULH3	0000023D	R	01	SAVE_PC	= 00000010		
PATTERN_POLYD	0000026D	R	01	SIGNAL	000007C0	R	01
PATTERN_POLYF	00000268	R	01	SIGNAL_START	00000765	R	01
PATTERN_POLYG	00000272	R	01	SS\$_ACCVIO	= 0000000C		
PATTERN_POLYH	00000277	R	01	SS\$_FLTDIV	= 00000494		
PATTERN_SUBD2	00000228	R	01	SS\$_FLTDIV_F	= 000004BC		
PATTERN_SUBD3	00000235	R	01	SS\$_FLTQVF	= 0000048C		
PATTERN_SUBF2	00000225	R	01	SS\$_FLTQVF_F	= 000004B4		
PATTERN_SUBF3	00000231	R	01	SS\$_FLTUND	= 0000049C		
FATTERN_SUBG2	0000022B	R	01	SS\$_FLTUND_F	= 000004C4		
PATTERN_SUBG3	00000239	R	01	SS\$_OPCDEC	= 0000043C		
PATTERN_SUBH2	0000022E	R	01	SS\$_RESIGNAL	= 00000918		
PATTERN_SUBH3	0000023D	R	01	SS\$_ROPRAND	= 00000454		
POLY_CHECK	= 00000006			SS\$_UNWIND	= 00000920		
POLY_DOUBLE	= 00000026			SYSSCALL_HANDL	*****	X	00
POLY_FLOAT	= 00000016			SYSSSRCHANDLER	*****	X	00
POLY_GRAND	= 00000036			SYSSUNWIND	*****	X	00
POLY_HANDLER	000006E1	R	01	UNDERFLOW TRAP	00000740	R	01
POLY_HUGE	= 00000046			UNUSED_BYTE_1	= FFFF0FFA		
PSL	= 00000054			UNUSED_BYTE_2	= FFFFFFF9		
PSLM_C	= 00000001			UNUSED_BYTE_3	= FFFFFFF8		
PSLM_N	= 00000008			WORD_BRANCH	000004AC	R	01
PSLM_NZ	= 0000000C			WORD_DEF_MODE	00000406	R	01
PSLM_V	= 00000002			WORD_DISP_MODE	000003FC	R	01
PSLM_Z	= 00000004			WORD_TYPE	= 00000006		
PSLM_ZVC	= 00000007			WRITE_ACCESS	= 00000002		
PSL_C	= 00000000			WRITE_AREA	= FFFFFFFA0		
PSL_FPD	= 0000001B			WRITE_DOUBLE	= 00000022		
PSL_FU	= 00000006			WRITE_FLOAT	= 00000012		
PSL_N	= 00000003			WRITE_GRAND	= 00000032		
PSL_T	= 00000004			WRITE_HUGE	= 00000042		
PSL_TP	= 0000001E			WRITE_LONG	= 00000072		
PSL_V	= 00000001			XQT_POLY	0000063C	R	01
PSL_Z	= 00000002						
READ_ACCESS	= 00000001						
READ_AREA	= FFFFFFFC4						
READ_BYTE	= 00000051						
READ_DOUBLE	= 00000021						
READ_FLOAT	= 00000011						
READ_GRAND	= 00000031						
READ_HUGE	= 00000041						
READ_WORD	= 00000061						
REGISTER_MODE	0000038A	R	01				
REG_AP	= 00000044						
REG_DEF_MODE	00000392	R	01				
REG_FP	= 00000048						
REG_PC	= 00000050						
REG_R0	= 00000014						
REG_R1	= 00000018						
REG_R2	= 0000001C						
REG_R3	= 00000020						
REG_R4	= 00000024						

! Psect synopsis !

PSECT name	Allocation	PSECT No.	Attributes
ABS	00000000 (0.)	00 (0.)	NOPIC USR CON ABS LCL NOSHR NOEXE NORD NOWRT NOVEC BYTE
LIB\$CODE	000007C6 (1990.)	01 (1.)	PIC USR CON REL LCL SHR EXE RD NOWRT NOVEC LONG
\$ABSS	00000000 (0.)	02 (2.)	NOPIC USR CON ABS LCL NOSHR EXE RD WRT NOVEC BYTE

! Performance indicators !

Phase	Page faults	CPU Time	Elapsed Time
Initialization	29	00:00:00.03	00:00:03.42
Command processing	108	00:00:00.30	00:00:03.38
Pass 1	289	00:00:06.02	00:00:27.09
Symbol table sort	0	00:00:00.75	00:00:02.32
Pass 2	387	00:00:02.88	00:00:12.42
Symbol table output	19	00:00:00.12	00:00:00.90
Psect synopsis output	1	00:00:00.02	00:00:00.02
Cross-reference output	0	00:00:00.00	00:00:00.00
Assembler run totals	835	00:00:10.13	00:00:49.55

The working set limit was 1950 pages.
56568 bytes (111 pages) of virtual memory were used to buffer the intermediate code.
There were 50 pages of symbol table space allocated to hold 724 non-local and 98 local symbols.
2158 source lines were read in Pass 1, producing 20 object records in Pass 2.
9 pages of virtual memory were used to define 8 macros.

! Macro library statistics !

Macro library name	Macros defined
_\$255\$DUA28:[SYSLIB]STARLET.MLB;2	5

582 GETS were required to define 5 macros.

There were no errors, warnings or information messages.

MACRO/ENABLE=SUPPRESSION/DISABLE=(GLOBAL,TRACEBACK)/LIS=LIS\$:LIBSIMTRA/OBJ=OBJ\$:LIBSIMTRA MSRC\$:LIBSIMTRA/UPDATE=(ENH\$:LIBSIMTRA)

LIBSPAWN
LIS

LIBSTATUM
LIS

LIBTRAAZE
LIS

LIBSPANC
LIS

LIBSYMBOL
LIS

LIBTRNLOG
LIS

LIBSKPC
LIS

LIBTIMER
LIS

LIBTPARSE
LIS

LIBTRIMF1
LIS

LIBSTRET
LIS

LIBTRAE2A
LIS