```
# SPIM S20 MIPS simulator.
# The default trap handler for spim.
#
# Copyright (C) 1990-1995 James Larus, larus@cs.wisc.edu.
# ALL RIGHTS RESERVED.
#
# SPIM is distributed under the following conditions:
#
# You may make copies of SPIM for your own use and modify those copies.
#
# All copies of SPIM must retain my name and copyright notice.
#
# You may not sell SPIM or distributed SPIM in conjunction with a commerical
# product or service without the expressed written consent of James Larus.
#
# THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND WITHOUT ANY EXPRESS OR
# IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
# PURPOSE.
#

# $Header: /u/l/a/larus/Software/SPIM/RCS/trap.handler,
# v 1.23 1997/07/09 21:45:2

########################################################################
# NOTE: Comments added and expanded by Neal Wagner, April 4, 1999
#   ("Text" below refers to Patterson and Hennessy, _Computer
#    Organization and Design_, Morgan Kaufmann.)
#
# INTERRUPT HNADLING IN MIPS:
# Coprocessor 0 has extra registers useful in handling exceptions
# There are four useful coprocessor 0 registers:
#-------------------------------------------------------------------|
#  REG NAME | NUMBER |   USAGE                                      |
#-------------------------------------------------------------------|
#  BadVAddr |   8    | Memory addr at which addr exception occurred |
#  Status   |  12    | Interrupt mask and enable bits               |
#  Cause    |  13    | Exception type and pending interrupt bits    |
#  EPC      |  14    | Address of instruction that caused exception |
#-----------|--------|----------------------------------------------|
# Details:
#   Status register: has an interrupt mask with a bit for each of
#      five interrupt levels.  If a bit is one, interrupts at that
#      level are allowed.  If a bit is zero, interrupts at that level
#      are disabled.  The low order 6 bits of the Status register
#      implement a three-level stack for the "kernel/user" and
#      "interrupt enable" bits.  The "kernel/user" bit is 0 if the
#      program was runningin the kernel when the interrupt occurred
#      and 1 if it was in user mode.  If the "interrupt enable" bit is 1,
#      interrupts are allowed.  If it is 0, they are disabled.  At an
#      interrupt, these six bits are shifted left by two bits.
#   Cause register: The value in bits 2-5 of the Cause register
#      describes the particular type of exception.  The error messages
#      below describe these values.  Thus a 7 in bits 2-5 corresponds
#      to message __e7_ below, or a "bad address in data/stack read".
#
# There are special machine instructions for accessing these
# coprocessor 0 registers:
#      mfc0  Rdest, CPsrc: "move from coprocessor 0" moves data
#         from the special coprocessor 0 register CPsrc into the
#         general purpose register Rdest.
#      mtc0  Rsrc, CPdest: "move to coprocessor 0" moves data
#         from the general purpose register Rsrc into the special
#         coprocessor 0 register CPdest.
# (There are also coprocessor load and store instructions.)
#
```

```
# ACTIONS BY THE TRAP HANDLER CODE BELOW:
#  Branch to address 0x80000080 and execute handler there:
#  1. Save $a0 and $v0 in s0 and s1.
#  2. Move Cause and EPC into registers $k0 and $k1.
#  3. Do action such as print an error message.
#  4. Restore $a0 and $v0.
#  5. Execute "rfe" instruction, which restores the previous interrupt
#     mask and kernel bits.
#  6. Return to the program by jumping to the instruction following
#     the one that caused the exception.
#
########################################################################

# Define the exception handling code.  This must go first!
        .kdata  # Store subsequent data items in the kernel data
                #  segment (text, p. A-52)
__m1_:  .asciiz "  Exception "
__m2_:  .asciiz " occurred and ignored\n"
__e0_:  .asciiz "  [Interrupt] "
__e1_:  .asciiz ""
__e2_:  .asciiz ""
__e3_:  .asciiz ""
__e4_:  .asciiz "  [Unaligned address in inst/data fetch] "
__e5_:  .asciiz "  [Unaligned address in store] "
__e6_:  .asciiz "  [Bad address in text read] "
__e7_:  .asciiz "  [Bad address in data/stack read] "
__e8_:  .asciiz "  [Error in syscall] "
__e9_:  .asciiz "  [Breakpoint] "
__e10_: .asciiz "  [Reserved instruction] "
__e11_: .asciiz ""
__e12_: .asciiz "  [Arithmetic overflow] "
__e13_: .asciiz "  [Inexact floating point result] "
__e14_: .asciiz "  [Invalid floating point result] "
__e15_: .asciiz "  [Divide by 0] "
__e16_: .asciiz "  [Floating point overflow] "
__e17_: .asciiz "  [Floating point underflow] "
__excp: .word __e0_,__e1_,__e2_,__e3_,__e4_,__e5_,__e6_,__e7_,__e8_,__e9_
        .word __e10_,__e11_,__e12_,__e13_,__e14_,__e15_,__e16_,__e17_
s1:     .word 0
s2:     .word 0

        .ktext 0x80000080  # Store subsequent items in the kernel text
                           #  segment, starting at 0x80000080 (text, A-52)
        .set noat       # Disable warnings about $at (text, p. A-52)
        # Because we are running in the kernel, we can use $k0/$k1 without
        # saving their old values.
        move $k1 $at    # Save $at in the kernel register $k1
        .set at         # Re-enable warnings about $at
        sw $v0 s1       # Not re-entrent and we can’t trust $sp.
        sw $a0 s2       #  so save $v0 and $a0 at addresses s1 and s2.

        mfc0 $k0 $13    # Cause
        sgt $v0 $k0 0x44 # ignore interrupt exceptions
        bgtz $v0 ret
        addu $0 $0 0
        li $v0 4        # syscall 4 (print_str)
        la $a0 __m1_    # print " Exception "
        syscall
        li $v0 1        # syscall 1 (print_int)
        srl $a0 $k0 2   # shift Cause reg
        syscall         # print exception number
        li $v0 4        # syscall 4 (print_str)
        lw $a0 __excp($k0)
        syscall         # print proper message
        bne $k0 0x18 ok_pc # Bad PC requires special checks
        mfc0 $a0, $14   # Move EPC (copr. reg. $14) into $a0
```

```
        and $a0, $a0, 0x3 # Is EPC word-aligned?
        beq $a0, 0, ok_pc
        li $v0 10       # Exit on really bad PC (out of text)
        syscall

ok_pc:
        li $v0 4        # syscall 4 (print_str)
        la $a0 __m2_
        syscall
        mtc0 $0, $13    # Clear Cause register, moving 0 into $13
ret:    lw $v0 s1
        lw $a0 s2
        mfc0 $k0 $14    # Move EPC into register $k0
        .set noat
        move $at $k1    # Restore $at
        .set at
        rfe             # Return from exception handler (restore status reg)
        addiu $k0 $k0 4 # Add 4 to get next instruction
        jr $k0          # Return to program


# Standard startup code.  Invoke the routine main with no arguments.

        .text
        .globl __start
__start:
        lw $a0, 0($sp)  # argc
        addiu $a1, $sp, 4 # argv
        addiu $a2, $a1, 4 # envp
        sll $v0, $a0, 2
        addu $a2, $a2, $v0
        jal main
        li $v0 10
        syscall         # syscall 10 (exit)
```