MIPS interrupts

Recall from lecture that interrupts are events that demand the processor's attention. Unlike exceptions, interrupts are normal events that must be handled without affecting any active programs. Since interrupts can happen at any time, there is no way for the active programs to prepare for the interrupt (*e.g.*, by saving registers that the interrupt might squash). It is important to note **that calling conventions do not apply when handling interrupts**: the interrupt is not being "called" by the active program—it is interrupting the active program. Thus, the interrupt handler code must ensure that it does not squash any registers that the program may be using.

Consider the following C pseudo-code for the interrupt handler:

```
void
interrupt_handler() {
  // save assembler temporary (so we don't accidentally overwrite it)
  // save $a0, $a1 registers (so we have some registers to work with)

  int cause_register = get_cause_register();  // read a coprocessor register
  if (((cause_register >> 2) & 0xf) != 0) {
    // handle exception
    return;
  }

  // otherwise it was an interrupt

  while (1) {
    cause_register = get_cause_register();  // it could have changed
    if (cause_register == 0) {
      break;  // no more unhandled interrupts
    }

    if (cause_register & 0x1000) {  // bonk interrupt (we ran into a wall)
      // handle bonk interrupt
      acknowledge_bonk_interrupt();
      continue;
    }

    if (cause_register & 0x8000) {  // handle other interrupt
      // ...
    }
  }

  // restore $a0, $a1
  // restore assembler temporary
  return_from_exception();
}
```

From the attached exception handler, you can see the MIPS translation of this code and more.

Question 1: Saving registers

In order to preserve registers, the interrupt handler must first save every register it intends to use in memory. Should it use the stack for this?

Solution: No! A possible reason for entering the interrupt handler may be because of an exception caused by a corrupted stack-pointer (the `$sp` register). Hence, registers are saved in a statically allocated chunk of global memory (in the kernel-data segment) as follows:

```
.kdata
chunkIH: .space 8 # space for 2 registers, for the interrupt handler
.ktext 0x80000080
interrupt_handler:
# save all registers to chunkIH
...
# restore all registers from chunkIH
# return from interrupt handler
```

Question 2: Saving registers to chunkIH

By convention, the registers `$k0` and `$k1` are used only by the interrupt handler (*i.e.*, the interrupt handler is free to squash these registers without affecting any active programs). What is wrong with the following code to save additional registers to `chunkIH`?

```
.ktext 0x80000080
interrupt_handler:
      la     $k0, chunkIH     # k0 = base address of chunkIH
      sw     $t0, 0($k0)      # save t0
      sw     $t1, 4($k0)      # save t1
```

Solution: The load-address (`la`) command is a pseudo-instruction, which uses the `$at` register. It is possible that the active program was itself interrupted while performing a pseudo-instruction, in which case `$at` contains useful data that gets squashed by the interrupt handler. Hence, even `$at` must be preserved by the interrupt handler:

```
interrupt_handler:
.set noat                      # turn off assembler warnings
      move   $k1, $at          # first save at
.set at                        # turn warnings back on
      la     $k0, chunkIH      # load address of available chunk
```

SPIMbot Memory-mapped I/O and Interrupts

SPIMbot can tell you its current x-coordinate (`lw` from `0xffff0020`) and y-coordinate (`lw` from `0xffff0024`). You can set SPIMbot's speed (`sw` to `0xffff0010`) and angle (`sw` angle to `0xffff0014`; and then `sw` 1 to `0xffff0018` for absolute angle or `sw` 0 for relative angle). Finally, you can read and set a timer (`lw`/`sw` from/to `0xffff001c`). In addition to the bonk interrupt (acknowledgment address `0xffff0060`), SPIMbot also has a timer interrupt (acknowledgment address `0xffff006c`) that interrupts the program when the timer goes off.

Answer these questions for the code on the next page:

1. What happens if SPIMbot hits a wall?
2. What happens on a timer interrupt?
3. What path should SPIMbot take if it doesn't hit a wall?
4. What happens on an exception?
5. The interrupt handler has a bug. Specifically, it squashes two registers. Find the bug and fix it.

```
.text
main:                                   # ENABLE INTERRUPTS
      li      $t4, 0x8000               # timer interrupt enable bit
      or      $t4, $t4, 0x1000          # bonk interrupt bit
      or      $t4, $t4, 1               # global interrupt enable
      mtc0    $t4, $12                  # set interrupt mask (Status register)

                                        # REQUEST TIMER INTERRUPT
      lw      $v0, 0xffff001c($0)       # read current time
      add     $v0, $v0, 50              # add 50 to current time
      sw      $v0, 0xffff001c($0)       # request timer interrupt in 50 cycles

      li      $a0, 10
      sw      $a0, 0xffff0010($zero)    # drive

infinite:
      j      infinite


.kdata                  # interrupt handler data (separated just for readability)
chunkIH:     .space 8      # space for two registers
non_intrpt_str:   .asciiz "Non-interrupt exception\n"
unhandled_str:    .asciiz "Unhandled interrupt type\n"

.ktext 0x80000080
interrupt_handler:
.set noat
      move      $k1, $at            # Save $at
.set at
      la      $k0, chunkIH
      sw      $a0, 0($k0)           # Get some free registers
      sw      $a1, 4($k0)           # by storing them to a global variable

      mfc0    $k0, $13              # Get Cause register
      srl     $a0, $k0, 2
      and     $a0, $a0, 0xf         # ExcCode field
      bne     $a0, 0, non_intrpt

interrupt_dispatch:                 # Interrupt:
      mfc0    $k0, $13              # Get Cause register, again
      beq     $k0, $zero, done      # handled all outstanding interrupts

      and     $a0, $k0, 0x1000      # is there a bonk interrupt?
      bne     $a0, 0, bonk_interrupt

      and     $a0, $k0, 0x8000      # is there a timer interrupt?
      bne     $a0, 0, timer_interrupt

                              # add dispatch for other interrupt types here.

      li      $v0, 4                # Unhandled interrupt types
```

```
      la      $a0, unhandled_str
      syscall
      j       done

bonk_interrupt:
      sw      $zero, 0xffff0010($zero)      # ???
      sw      $a1, 0xffff0060($zero)        # acknowledge interrupt

      j       interrupt_dispatch            # see if other interrupts are waiting

timer_interrupt:
      sw      $a1, 0xffff006c($zero)        # acknowledge interrupt

      li      $t0, -90                      # ???
      sw      $t0, 0xffff0014($zero)        # ???
      sw      $zero, 0xffff0018($zero)      # ???

      lw      $v0, 0xffff001c($0)           # current time
      add     $v0, $v0, 10000
      sw      $v0, 0xffff001c($0)           # request timer in 10000

      j       interrupt_dispatch            # see if other interrupts are waiting

non_intrpt:                           # was some non-interrupt
      li      $v0, 4
      la      $a0, non_intrpt_str
      syscall                         # print out an error message
      j       done

done:
      la      $k0, chunkIH
      lw      $a0, 0($k0)             # Get some free registers
      lw      $a1, 4($k0)             # by storing them to a global variable
      mfc0    $k0 $14                 # Exception Program Counter (PC)
.set noat
      move    $at $k1                 # Restore $at
.set at
      rfe                             # Return from exception handler
      jr      $k0
      nop
```