"The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be. ... The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program."
– F. Brooks ("The Mythical Man Month", pages 7-8)

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live." – M. Golding

Learning Objectives

1. Using programmed I/O to interact with I/O devices in MIPS assembly
2. Understanding interrupt service routines

Work that needs to be handed in (via SVN)

1. `problem1.s`: An interrupt handling routine that periodically updates SPIMbot's orientation to drive toward the public apple.
2. `problem4.s` from Lab8: **This should be checked into your Lab8 directory.** See the Lab8 handout.

Guidelines

- You'll want to use `/class/cs232/Linux/bin/QtSpimbot` for both assignments.
- Same procedure as Labs 7 and 8. Correctly implement function interfaces. Use any MIPS instructions or pseudo-instructions you want. We may try to break your code.
- You will find it useful to refer to Appendix A.7 of the book, the example interrupt service routines (bonk.c and bonk.s) on the lecture notes page, as well as the SPIMbot documentation on the assignments page. Also, we provide an electronic version of the code on pages 4 and 5 as a file in your SVN. See:
  - `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`
  - `https://wiki.engr.illinois.edu/display/cs398sp13/SPIMbot+documentation`
  - `example.s`

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 overwrite). 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 overwrite 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 dont 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 0x80000180
interrupt_handler:
# save all registers to chunkIH
...
# restore all registers from chunkIH
# return from interrupt handler
```

Question 2: Saving registers to `chunkIH` (IH = Interrupt Handler)

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

```
.ktext 0x80000180
interrupt_handler:      la     $k0, chunkIH
                        sw     $t0, 0($k0)
                        sw     $t1, 4($k0)
# k0 = base address of chunkIH
# save t0, 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 overwritten 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 angle (`sw` the angle to `0xffff0014`; and then `sw` 1 to `0xffff0018` for absolute angle or `sw` 0 for relative angle). Non-snake SPIMbots can have their speed set (`sw` to `0xffff0010`). Furthermore, 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 overwrites 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 0x80000180
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, 50000
      sw      $v0, 0xffff001c($0)      # request timer in 50000

      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)              # Restore saved registers
      lw      $a1, 4($k0)
.set noat
      move    $at, $k1                 # Restore $at
.set at
      eret                             # Return from exception handler
```

Problem 1: Timer-based navigation [25 points]

In this problem, your job is to take the "snake motion planning" code (problem1) and the associated SPIMbot control code `p1_spimbot.s` from Lab7 and implement it as a timer interrupt. This will allow your SPIMbot program to solve puzzles in its main program, while periodically adjusting the course of the snake to eat apples.

Specifically, your code should do the following:

1. acknowledge the interrupt
2. read the (X,Y) location of your snake's head,
3. read the (X,Y) location of the public apple,
4. run the `new_angle` code to compute the desired angle; we suggest that you inline this code (i.e., put the code into the interrupt handler rather than make it a function that the interrupt handler calls so you can minimize the registers that need to be saved),
5. if the desired angle is 180 away from the previous angle, then set the angle to the desired angle + 90 degrees and wait 5000 cycles. this is to avoid the snake doubling back and running into itself.
6. set the snake's angle to the desired angle
7. save the desired angle as the previous angle
8. request another timer interrupt in 500 cycles

Below is C pseudo-code that outlines what needs to be implemented.

```
ACKNOWLEDGE_TIMER_INTERRUPT();
int my_x = HEAD_X(), my_y = HEAD_Y();
int apple_x = APPLE_X(), apple_y = APPLE_Y(), desired_angle;

if (my_x != apple_x) {    // are we already aligned in x?
  if (my_x < apple_x) {   // if not, are we too far to left?
    desired_angle = 0;    // then face to the right
  } else {
    desired_angle = 180;  // otherwise face to the left
} else {                  // otherwise my_x == apple_x
  if (my_y < apple_y) {   // are we above the apple?
    desired_angle = 90;   // then face down
  } else {
    desired_angle = 270;  // otherwise face up
  }
}
if (abs(prev_angle - desired_angle)) == 180) {
  SET_ABSOLUTE_ANGLE(desired_angle + 90);
  for (int i = 2500 ; i != 0 ; i --) { }  // wait 5000 cycles
}
SET_ABSOLUTE_ANGLE(desired_angle);
prev_angle = desired_angle;
SET_TIMER(GET_TIMER() + 500);
```

For the most part, this code can be constructed from modifying the example interrupt handler code (`example.s`) using the Lab 7 code. A few notes:

1. The code we've provided in `p1_main.s` enables interrupts and requests the first timer interrupt. Do not modify this code.
2. We've also allocated a global variable `prev_angle` for your use; you must use this memory variable for holding `prev_angle` across interrupts.
3. Be sure that you save and restore registers properly in your interrupt handler. We will be testing your code with a different version of `p1_main.s` to verify that you save and restore registers properly in your interrupt handler.