Project 2: Vectored Interrupts (4%)

ENEE 447: Operating Systems — Spring 2012
Assigned: Monday, Feb 8; Due: Friday, Feb 19

Purpose
This project has you implement vectored interrupts on the Raspberry Pi. Vectored interrupts are one of the most important facilities that hardware offers, because they allow a wide range of asynchronous operation to occur — whenever an interrupt occurs, the PC is redirected to a completely different location, which allows the operating system to split its attention between multiple things.

Vectored Interrupts in ARM
The ARM implementation puts the vector table at the very start of memory, and it must contain a set of jump instructions as opposed to jump addresses. A typical layout might look like the following:

```
.globl _start
_start:
    @ jump table:
    ldr pc, res_handler          @ RESET handler - runs in SVC mode
    ldr pc, und_handler          @ UNDEFINED INSTR handler - runs in UND mode
    ldr pc, swi_handler          @ SWI (TRAP) handler - runs in SVC mode
    ldr pc, pre_handler          @ PREFETCH ABORT handler - runs in ABT mode
    ldr pc, dat_handler          @ DATA ABORT handler - runs in ABT mode
    ldr pc, hyp_handler          @ HYP MODE handler - runs in HYP mode
    ldr pc, irq_handler          @ IRQ INTERRUPT handler - runs in IRQ mode
    ldr pc, fiq_handler          @ FIQ INTERRUPT handler - runs in FIQ mode

@ pointers to handler functions:
res_handler: .word <name of handler function>
und_handler: .word <name of handler function>
swi_handler: .word <name of handler function>
pre_handler: .word <name of handler function>
dat_handler: .word <name of handler function>
hyp_handler: .word <name of handler function>
irq_handler: .word <name of handler function>
fiq_handler: .word <name of handler function>
```

So, whenever the system takes a RESET interrupt, the number 0x00000000 is loaded into the program counter, which causes the processor to jump to address zero. At address zero is an instruction

```
    ldr pc, res_handler
```

that tells the hardware to load the PC with whatever value is found at location res_handler. At that location is a data word holding the address of the first instruction in the corresponding handler function. So this is effectively a jump to that handler function.

Similarly, whenever the system takes a SWI interrupt (which is caused by the svc assembly-code instruction), the number 0x00000008 is loaded into the program counter, which causes the processor to jump to address 0x08 (the third word in memory). At address 0x08 is an instruction

```
    ldr pc, swi_handler
```

that tells the hardware to load the PC with whatever value is found at location swi_handler. At that location is a data word holding the address of the first instruction in the corresponding handler function. So this is effectively a jump to that handler function.

And so forth.
Why, you might ask, don’t they simply put a bunch of branch statements at the top, like this:

```assembly
.globl _start
_start:
    @ jump table:
    b <name of handler function> @ RESET handler - runs in SVC mode
    b <name of handler function> @ UNDEFINED INSTR handler - runs in UND mode
    b <name of handler function> @ SWI (TRAP) handler - runs in SVC mode
    b <name of handler function> @ PREFETCH ABORT handler - runs in ABT mode
    b <name of handler function> @ DATA ABORT handler - runs in ABT mode
    b <name of handler function> @ HYP MODR handler - runs in HYP mode
    b <name of handler function> @ IRQ INTERRUPT handler - runs in IRQ mode
    b <name of handler function> @ FIQ INTERRUPT handle - runs in FIQ mode
```

This also works (I have checked it out), but the previous form is the way that I have seen people implement it in ARM documents and in on-line code examples. The second form is slightly faster, because it involves one fewer memory reference. My guess is that the second form is not used because it is less flexible: the branch statements are PC-relative, and so the branch targets (the handler functions) must lie within a certain distance of the vector table itself. This means that you couldn’t locate the kernel and all its handlers at the top of memory (e.g., at 0xF0000000 and above). The first form, which uses an indirect jump through those `.word` directives, allows one to jump to any location in the 32-bit address space, including jumps to the top of memory.

In all of our projects, and in the operating system you are building, the kernel and its interrupt handlers will all lie in a small region starting at memory address zero, so the flexibility offered by the first form is not necessary. Your implementation can take whatever form you want.

### Implement Vectored Interrupts

Your task is to implement the ARM vector table (see the example jump tables shown above) and two different types of interrupt handlers:

- one that software invokes intentionally and therefore synchronously: the SWI/TRAP handler, which is invoked through the `svc` assembly-code instruction, and
- one that happens asynchronously: the IRQ handler, which is invoked through any number of means, including writing to mailboxes and as a result of count-down timers reaching zero, etc.

Your trap handler will simply blink the green LED, once, and your IRQ handler will simply blink the red LED, once. In other words, in your trap-handler code, you should have the following:

```assembly
    bl blink_green
```

and in your IRQ handler code you should have the following:

```assembly
    bl blink_red
```

We will give you code that drives all of this; your job is just to set up the jump table how you want it and write the handlers. The boot code you are given starts up the processor and branches into two directions: core0 runs a program called “kernel” in privileged mode, and core1 runs a program called “userspace” in user mode. Cores 2 and 3 just hang. This will help you to avoid scenarios where the various cores all stomp on each other running the same code. 😊

### Build It, Load It, Run It

Once you have it working, show us.