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:

```
.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 MODE 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.

**A Note on Modes and Stacks**

Note that each of the vectors runs in a different mode (except for two that both run in ABT mode). Recall the register-file arrangement in the ARM architecture:

<table>
<thead>
<tr>
<th>User/System</th>
<th>FIQ</th>
<th>IRQ</th>
<th>SVC</th>
<th>Undef</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>r8</td>
<td>r8_fiq</td>
<td>r8_fiq</td>
<td>r8_fiq</td>
<td>r8_fiq</td>
<td>r8_fiq</td>
</tr>
<tr>
<td>r9</td>
<td>r9_fiq</td>
<td>r9_fiq</td>
<td>r9_fiq</td>
<td>r9_fiq</td>
<td>r9_fiq</td>
</tr>
<tr>
<td>r10</td>
<td>r10_fiq</td>
<td>r10_fiq</td>
<td>r10_fiq</td>
<td>r10_fiq</td>
<td>r10_fiq</td>
</tr>
<tr>
<td>r11</td>
<td>r11_fiq</td>
<td>r11_fiq</td>
<td>r11_fiq</td>
<td>r11_fiq</td>
<td>r11_fiq</td>
</tr>
<tr>
<td>r12</td>
<td>r12_fiq</td>
<td>r12_fiq</td>
<td>r12_fiq</td>
<td>r12_fiq</td>
<td>r12_fiq</td>
</tr>
<tr>
<td>r13/SP</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
<td>r13_fiq</td>
</tr>
<tr>
<td>r14/LR</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
<td>r14_fiq</td>
</tr>
<tr>
<td>r15/PC</td>
<td>r15/PC</td>
<td>r15/PC</td>
<td>r15/PC</td>
<td>r15/PC</td>
<td>r15/PC</td>
</tr>
<tr>
<td>cpsr</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>spsr_fiq</td>
<td>spsr_irq</td>
<td>spsr_svc</td>
<td>spsr_undef</td>
<td>spsr_abort</td>
</tr>
</tbody>
</table>
When a mode is invoked, its corresponding register set becomes visible. So, for instance, each mode has its own banked stack pointer and link register — the sp/lr registers, r13 and r14. In addition, the FIQ mode also has its own private r8–r12 registers. Why this is interesting is that you can set up a separate stack for each mode that becomes visible when that mode is invoked. The code is as follows:

```asm
.equ USR_mode, 0x10
.equ FIQ_mode, 0x11
.equ IRQ_mode, 0x12
.equ SVC_mode, 0x13
.equ HYP_mode, 0x1A
.equ SYS_mode, 0x1F
.equ No_Int, 0xC0

mov r2, # No_Int | IRQ_mode
msr cpsr_c, r2
mov sp, # IRQSTACK1
mov r2, # No_Int | FIQ_mode
msr cpsr_c, r2
mov sp, # FIQSTACK1
mov r2, # No_Int | SVC_mode
msr cpsr_c, r2
mov sp, # KSTACK1
```

This has been set up for you in the file `1_boot.s`.

**CPSR and SPSR**

In addition, when the hardware throws an exception and jumps to the corresponding vector, the CPSR (the Current Program Status Register) is saved to the mode’s SPSR (the Saved Program Status Register), and when the hardware returns from exception, that is the copy that is moved back into the CPSR. Thus, the interrupt handler can make changes to the CPSR by writing to its local SPSR before returning from exception. The registers have the following format and meaning:

**Format of the CPSR and SPSRs**

The CPSR and SPSR bit assignments are:

<table>
<thead>
<tr>
<th>bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
<td>Negative condition flag</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
<td>Zero condition flag</td>
</tr>
<tr>
<td>29</td>
<td>C</td>
<td>Carry condition flag</td>
</tr>
<tr>
<td>28</td>
<td>V</td>
<td>Overflow condition flag</td>
</tr>
<tr>
<td>27</td>
<td>Q</td>
<td>Cumulative saturation bit</td>
</tr>
<tr>
<td>26</td>
<td>Reserved, RAZ/SBZP</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>GE[3:0]</td>
<td>General Execution bits</td>
</tr>
<tr>
<td>24</td>
<td>IT[7:2]</td>
<td>If-Then execution state bits for the Thumb IT (If-Then) instruction</td>
</tr>
<tr>
<td>23</td>
<td>I[1:0]</td>
<td>Instruction set state, privilege level</td>
</tr>
<tr>
<td>22</td>
<td>E</td>
<td>Reserved</td>
</tr>
<tr>
<td>21</td>
<td>A</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>F</td>
<td>Reserved</td>
</tr>
<tr>
<td>19</td>
<td>T</td>
<td>Jazelle bit</td>
</tr>
<tr>
<td>16</td>
<td>M[4:0]</td>
<td>Miscellaneous bits</td>
</tr>
</tbody>
</table>

**Condition flags, bits[31:28]**

Set on the result of instruction execution. The flags are:

- N, bit[31]: Negative condition flag
- Z, bit[30]: Zero condition flag
- C, bit[29]: Carry condition flag
- V, bit[28]: Overflow condition flag

The condition flags can be read or written in any mode, and are described in *The Application Program Status Register (APSR)* on page A2-49.

**Q, bit[27]**

Cumulative saturation bit. This bit can be read or written in any mode, and is described in *The Application Program Status Register (APSR)* on page A2-49.

**IT[7:0], bits[15:10, 26:25]**

If-Then execution state bits for the Thumb IT (If-Then) instruction. *IT block state register, ITSTATE on page A2-51* describes the encoding of these bits. CPSR.IT[7:0] are the IT[7:0] bits described there. For more information, see *IT on page A8-390*.

For details of how these bits can be accessed see *Accessing the execution state bits on page B1-1150*.

**J, bit[24]**

Jazelle bit, see the description of the T bit, bit[5].

**Bits[23:20]**

Reserved. RAZ/SBZP.
GE[3:0], bins[19:16]
Greater than or Equal flags, for the parallel addition and subtraction (SIMD) instructions described in Parallel addition and subtraction instructions on page A4-171. The GE[3:0] field can be read or written in any mode, and is described in The Application Program Status Register (APSR) on page A2-49.

E, bit[9]
Endianness execution state bit. Controls the load and store endianness for data accesses:
0 Little-endian operation
1 Big-endian operation.

Instruction fetches ignore this bit. Endianness mapping register, ENDIANSTATE on page A2-53 describes the encoding of this bit.

CPSR.E is the ENDIANSTATE bit described there.

For details of how this bit can be accessed see Accessing the execution state bits on page B1-1150. When the reset value of the CPSR.E bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

Mask bits, bins[8:6]
These bits are:


The possible values of each bit are:
0 Exception not masked.
1 Exception masked.

The A bit has no effect on any Data Abort exception generated by a Watchpoint debug event, even if that exception is asynchronous. For more information see Debug exception on Watchpoint debug event on page C4-2091.

In an implementation that does not include the Security Extensions, setting a mask bit masks the corresponding exception, meaning it cannot be taken. However, the Security Extensions and Virtualization Extensions significantly alter the behaviour and effect of these bits, see Effects of the Security Extensions on the CPSR A and F bits on page B1-1151 and Asynchronous exception masking on page B1-1184.

The mask bits can be written only at PL1 or higher. Their values can be read in any mode, but ARM deprecates any use of their values, or attempt to change them, by software executing at PL0.

Updates to the F bit are restricted if Non-maskable FIQs (NMFIs) are supported, see Non-maskable FIQs on page B1-1151.

T, bit[5]
Thumb execution state bit. This bit and the J execution state bit, bit[24], determine the instruction set state of the processor, ARM, Thumb, Jazelle, or ThumbEE. Instruction set state register, ISETSTATE on page A2-50 describes the encoding of these bits. CPSR.J and CPSR.T are the same bits as ISETSTATE.J and ISETSTATE.T respectively. For more information, see Instruction set states on page B1-1155.

For details of how these bits can be accessed see Accessing the execution state bits on page B1-1150.

M[4:0], bins[4:0]
Mode field. This field determines the current mode of the processor. The permitted values of this field are listed in Table B1-1 on page B1-1139. All other values of M[4:0] are reserved. The effect of setting M[4:0] to a reserved value is UNPREDICTABLE.

Note that, for the purposes of this project, you will not need to interact with the CPSR or SPSR at all other than using it to set up the various stacks as shown above; the format is merely provided here so that you can see what the status register does and how it does it.

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

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. :D

**Build It, Load It, Run It**

Once you have it working, show us.