SYSTEMS II — PROJECT 1
Revision 0 — 2015-Mar-21
Interrupts and their handling for beginning a kernel

1 Overview and motivation

For this project, we assume that you have a working BIOS that can load and boot some arbitrary code. Of course, the purpose of the BIOS is to load and boot not just any old code, but specifically a *kernel*, which then serves both to control the sharing of the hardware **and** to abstract the use of that hardware.

Here, you will begin a fledgling kernel that, once loaded and executed by the BIOS, will establish its control of the processor and thus all of the system's hardware. That kernel will then load a single user-level process, execute it, and then demonstrate that the kernel retains control when that user-level process does something that such "regular" processes should not be allowed to do.

2 Interrupts

2.1 Codes

For a kernel to establish control of the CPU and the hardware overall, it must establish its procedures as the ones to call when CPU interrupts occur. The K-SYSTEM ISA enumerates the following interrupts:

0. `INVALID_ADDRESS`: Some operand specified a memory address that is invalid. Typically used when an invalid or impermissible *virtual* address cannot be translated.

1. `INVALID_REGISTER`: Some operand specified a register number that is invalid.

2. `BUS_ERROR`: An operand provided an address that yielded an address on the bus that was invalid. The bus may have received an address for which there is no responding device, or the bus may have refused to process a misaligned address.[1]

3. `CLOCK_ALARM`: A periodic alarm generated when the *cycle counter* matches the *alarm register*. (See the `SETALM` instruction described in Project-0.

4. `DIVIDE_BY_ZERO`: Occurs when one of the arithmetic division instructions receives a denominator operand whose value is zero.

5. `OVERFLOW`: Occurs when a *signed* arithmetic operation yields an overflowed result.

6. `INVALID_INSTRUCTION`: If an instruction contains an invalid opcode, or if an operand has invalid status bits, then this interrupt occurs.

[1] A 32-bit system will expect any request for a word-sized value (e.g., **not** a `COPYB` instruction) to be *word-aligned*—that is, the address A, given 4-byte words, should satifying the property that $A \ mod \ 4 \equiv 0$.

7. `PERMISSION_VIOLATION`: A supervisor-only instruction (see the listing of available opcodes in Project-0) was issued while the processor was in user mode.

8. `INVALID_SHIFT_AMOUNT`: When one of the arithmetic *shift* instructions is used, the number of bits to shift can be no more than the word size.

9. `SYSTEM_CALL`: A special case of the `INVALID_INSTRUCTION` interrupt reserved for the use of a particular invalid opcode used for system call vectoring.

2.2 Vectoring

When an interrupt occurs, the processor performs a specific sequence of steps:

1. **Elevate permission mode:** Set the processor into supervisor mode.

2. **Preserve state:** Store into the *interrupt buffer* the `IP` **at the moment of the interrupt** and any auxiliary information about the interrupt (e.g., the virtual address that triggered an `INVALID_ADDRESS` interrupt). The interrupt buffer is a main memory space—at least two words in size—that the processor finds via the *interrupt buffer register (`IB`)*. (See the description of the `SETIBR` instruction in Project-0.)

3. **Vector to interrupt handler:** The processor uses the interrupt code to find the corresponding handler procedure. Specifically, the *trap table* is an array of pointers to the entry points of interrupt handling procedures. The processor looks up the correct entry in this table by calculating ...

   $$t_e = t_b + c|w|$$

   ... where t_e is the address of the correct trap table entry, which is obtained from the t_b *trap base*—the starting address of the trap table—as well as c, the interrupt code, and $|w|$, the word size. In short, the interrupt code is an *index* into the array of word-sized addresses. The *trap base* is found via the *trap base register (`TB`)*, which must be set with the `SETTBR` instruction described in Project-0.

   Once the handler finds the correct table entry address (t_e), then it can grab the value from that address, thus pulling the address to which the processor then jumps in order to handle the interrupt. Since these addresses are to kernel code, the kernel therefore maintains control of the processor and other hardware.

3 Your assignment

You must write a first version of your kernel in assembly code (`kernel.asm`) such that, once loaded and executed by the BIOS, it does the following:

1. Create a *default interrupt handler*, which is a function that does little other than halt the processor.

2. Create a *trap table*; initialize all of its entries to point to the default interrupt handler.
3. Set the `TB` register to point to the trap table.
4. Create a two-word *interrupt buffer*.
5. Set the `IB` register to point to the interrupt buffer.
6. Copy the *user-level program* (e.g., `add-two-numbers`, assumed to be the 3^{rd} ROM as given at the command line for the simulator) into a free space in main memory (RAM).
7. `JUMPMD` to the copied user-level program's first machine code instruction.
8. Ensure that the user-level program runs. When it does something that is illegal in user-mode (e.g., attempt the `HALT` instruction), be sure that an interrupt occurs and is correctly handled by a vector to your default interrupt handler.

Once again, observe that you are encouraged to use the *DMA portal* of the bus controller to carry out any copying steps (e.g., the user-level program into a free space in RAM).

4 How to submit your work

Use the CS submission systems to submit your work. Specifically, you will need to submit your `kernel.asm` file; you may optionally submit an updated `bios.asm` file if you have changed it from your Project-0 submission.

This assignment is due at **11:59 pm** on **Monday, March** 23^{rd}**.**