CS 50 Homework 4:
Trap Handling

In your experience programming homeworks 2 and 3 you undoubtedly experienced several exceptions that caused your program to crash. And when that happened, you got a message like:

```
Exception [Misaligned load address]
   MCAUSE: 0x00000004
   MEPC: 0x00010058
   MTVAL: 0x7fffed96
```

When this happens it wasn't vrv's built in functionality printing out a report but a bit of code loaded as the "system file" within vrv: the default *trap handler*. As discussed in lecture the trap handler is a small piece of code that is triggered on any trap or interrupt. Its job is to figure out what to do and, when complete, either resume execution or continue onward.

For this exercise you will complete your own version of the system file, **`trap_handler.s`**. You will load it in vrv as a "custom system file" and then load a **`main.s`** (or other test routine) and **`utils.s`** as normal program files.

To enable your trap handler to call other functions, the startup routine allocates a small space for a kernel data stack at **`kstack`** and stores the pointer to the top of this stack in **`mscratch`**. That way, when your trap handler does a swap between **`mscratch`** and **`sp`** the trap handler can now just use the stack pointer normally: decrementing it, storing stuff on the stack, calling other functions, etc.

Your custom trap handler will need to first swap **`sp`** and **`mscratch`**, decrement its newly acquired stack pointer and then use the space to save all the registers on the stack. It should then examine the type of exception and the instruction triggering it by examining **`mcause`** and the instruction referred to in **`mepc`**.

If the instruction is not a **`lw`** and the exception is not a misaligned load address it will jump to the existing **`terminate`** routine which is the old functionality copied from the default system file, printing out the information about the exception.

If it is a misaligned **`lw`**, your trap handler should instead process the load as a series of 4 single byte loads starting at the address stored in **`mtval`** (which for load/store traps contains the address which the program attempted to access), storing the result into the stack space for the destination register. It should then increment **`mepc`** to the next instruction and, finally, restore all registers (which, in the process will make sure that the destination register has the correct value in it), swap **`mscratch`** and **`sp`**, and execute **`mret`** to return control to the next instruction.

This assignment will be due on Friday May 24th at 10 PM Davis time.

Expected Homework Outcomes

1. Understand the process of trap handling
2. Building software that decodes RISC-V instructions

Hints

You are strongly encouraged to build your own copy of the RISC-V simulator. Instructions are available here:
https://docs.google.com/document/d/1gibQ88EyITDgLlui_FIv1qaE8c6iIb2NHsmJ_bTZ-Fc/edit

The **`utils.s`** functions include **`malloc`**, **`strcmp`**, **`printstr`**, **`printhex`** and **`assert`**. You can use these in building your own main functions that test different operations.

You will need to write test code in **`main.s`** to actually test the functionality of your trap handler, as the provided **`main.s`** is far from comprehensive. This test code should ensure that no registers are changed and that the trap is properly processed. You will probably also need to write other independent tests with their own main function, since instructions other than **`lw`** and causes other than misaligned load will need to keep the current termination functionality.

In running your code in vrv or qvrv you will need to specify your trap_handler.s as a ***system file*** rather than a user file. You will also want to turn on viewing of “Kernel Text” in the text view option and “Kernel Memory” in the data view option, allowing you to see the code and memory used by your trap handler. That way you can observe and set breakpoints in your trap handling code. Once this is enabled just scroll to the bottom for the code/memory to get to the kernel’s section of code and memory.

In building your trap handler’s preamble and postamble you should probably save ALL registers from **`x0`** to **`x31`**, and use the numerical versions rather than the mnemonics for the saving and restoration process only. (And yes, it is OK to needlessly save **`x0`** and **`sp`**). Now you don’t have to worry about keeping track of what registers you saved and which ones you did not.

For my implementation I just wrote a tiny python script for saving/loading to save on typing and reduce errors and I cut and pasted the results into my program.

Don’t forget the difference between signed and unsigned loads for single bytes.

The autograder is not yet released but will include multiple tests including invalid loads that aren’t misaligned (which should be terminated), misaligned half-word loads (which should be terminated), all possible misalignments, and code to verify that the trap handler doesn’t disrupt any registers. The autograder will notify you of which tests you fail.