LSU EE 4720 Homework 5 Solution Due: 27 March 2013

Problem 1: In the following execution of MIPS code the `lw` instruction raises a *TLB miss* exception and the handler is called. A TLB miss is not an error, it indicates that the TLB needs to be updated, which is what the handler will do.

Execution is shown up to the first instruction of the handler. Alert students will recognize that there is something wrong in the execution below: it shows the execution of a deferred exception for an instruction, the `lw`, that should raise a precise exception.

```
# Cycle        0  1  2  3  4  5  6  7  8  9
 sh r1, 0(r3)  IF ID EX ME WB
 lw r1, 0(r2)     IF ID EX M*x
 addi r2, r2, 4      IF ID EX ME WB
 sw r7, 0(r8)           IF ID EX ME WB
 and  r4, r1, r6           IF ID EX ME WB
 or r10, r11, r12


HANDLER:
# Cycle        0  1  2  3  4  5  6  7  8  9
 sw r31,0x100(r0)               IF ID EX ME WB
 ... # Additional handler code here.
 eret
```

(*a*) Show the execution of the `eret` instruction and the instructions that execute after the `eret`. Assume that `eret` reaches `IF` in cycle number 100. The execution should be for a deferred exception, even though memory instruction exceptions should be—must be—precise. A correct solution to this part will result in incorrect execution of the code.

Solution appears below. Note that the cycle after 100 is 101, but is written as 1 to save space.

In a deferred exception the handler starts several instructions *after* the faulting instruction. In the example above the `addi`, `sw`, and `and` execute before the handler starts. The return point would then be after the `and` instruction, that is what is shown below.

Also, notice that `eret` does not have a delay slot.

The solution below assumes that there is a connection from the register file (actually coprocessor set 0, which contains the exception return address) to the `IF`-stage multiplexor. That would enable the first user instruction to reach `IF` when `eret` is in `EX`.

```
# Cycle        0  1  2  3  4  5  6  7  8  9  ... 100  1  2  3  4  5  6  7  8
 sh r1, 0(r3)  IF ID EX ME WB
 lw r1, 0(r2)     IF ID EX M*x
 addi r2, r2, 4      IF ID EX ME WB
 sw r7, 0(r8)           IF ID EX ME WB
 and  r4, r1, r6           IF ID EX ME WB
 or r10, r11, r12                                     IF ID EX ME WB  SOLUTION


HANDLER:
# Cycle        0  1  2  3  4  5  6  7  8  9  ... 100  1  2  3  4  5  6  7  8
```

```
 sw r31,0x100(r0)            IF ID EX ME WB
 ... # Additional handler code here.
 eret                                              IF ID EX ME WB   SOLUTION
 xor                                                  IFx
# Cycle        0  1  2  3  4  5  6  7  8  9  ... 100  1  2  3  4  5  6  7  8
```

(*b*) Suppose the execution above is for a computer on Mars, meaning that there is no fast or cheap way of replacing the hardware, and there is no way to turn on precise exceptions for the `lw`. Happily, it is possible to re-write the handler. Explain what the handler would have to do so that the code above executes correctly. The handler will know the address of the faulting instruction. Optional: explain why the `sw r7` is nothing to worry about, at least in the execution above.

Because the `lw` did not make it to writeback, `r1` and `r4` will have incorrect values when the handler starts. Re-write the handler so that it puts the correct values in `r1` and `r4` and then returns to the `or` instruction (as it might for a deferred exception).

The handler will update the TLB, as the original handler did. But then it will load the word from memory that the `lw` would have loaded, using address `r2-4`, and put it in `r1`. and then recompute `r4`. The `sw r7` would only be a problem if it wrote the same address as the `lw`, making it impossible to retrieve the prior word at that address. However, if the addresses were the same the `sw r7` would also raise an exception, so they must be different. After this, execution can return to the `or` and continue as though nothing happened.

(*c*) Show the execution of the code above, but this time for a system in which `lw` raises a precise exception. Start at cycle 0 with the `sh` instruction, and have the `lw` raising once again a TLB miss exception. The execution should be in two parts, first from the `sh` up to the first instruction of the handler, then jump ahead to cycle 100 with `eret` in `IF` and continue with whatever instructions remain.

Solution appears below. Since the exception is precise the faulting instruction *`lw`) and those that follow it are squashed and all instructions before the faulting instruction finishes normally. The handler has the option of re-executing the faulting instruction or skipping it. For a TLB miss the usual practice is to re-execute it.

```
SOLUTION
# Cycle        0  1  2  3  4  5  6  7  8  9  ... 100  1  2  3  4  5  6  7  8
 sh r1, 0(r3)  IF ID EX ME WB
 lw r1, 0(r2)     IF ID EX M*x                           IF ID EX ME WB
 addi r2, r2, 4      IF ID EXx                              IF ID EX ME WB
 sw r7, 0(r8)           IF IDx                                 IF ID EX ME WB
 and  r4, r1, r6           IFx                                    IF ID EX ME WB
 or r10, r11, r12                                                    IF ID EX ME WB


HANDLER:
# Cycle        0  1  2  3  4  5  6  7  8  9  ... 100  1  2  3  4  5  6  7  8
 sw r31,0x100(r0)            IF ID EX ME WB
 ... # Additional handler code here.
 eret                                              IF ID EX ME WB
 xor                                                  IFx
# Cycle        0  1  2  3  4  5  6  7  8  9  ... 100  1  2  3  4  5  6  7  8
```

Problem 2: Solve Spring 2012 Final Exam Problem 2, which asks for the execution of MIPS floating-point instructions on our FP implementation.

See the posted final exam solution.

Problem 3: Solve Spring 2012 Final Exam Problem 1 (yes, this is out of order). In this problem parts of the FP multiply unit are used to implement the MIPS integer `mul` instruction. Note that the `mul` writes integer registers, unlike `mult` which writes the `hi` and `lo` registers. In other words, **do not** use `hi` and `lo` registers in your solution.

See the posted final exam solution.