LSU EE 4720 Homework 4 Solution Due: 4 December 2006

Problem 1: The floating point pipeline in the MIPS implementation illustrated below must sometimes stall instructions to avoid the WF structural hazard. The WF structural hazard could be avoided by requiring all instructions that use WF to go through the same number of stages. Note that instructions that use WB all pass through five stages, even though some instructions, such as `xor`, could write back earlier.

Redesign the illustrated implementation so that the WF structural hazard is eliminated by having WF instructions (consider `add.d`, `sub.d`, `mul.d`, and `lwc1`) all pass through the same number of stages. The functional units themselves shouldn't change (still six multiply steps and four add steps) but their positions might change.

(*a*) Show the possibly relocated functional units and their connections. Don't forget connections for the `lwc1` instruction.

The add unit steps now share stages with like-numbered multiply unit steps, see the illustration below. After `A4` the FP add result is ready but it continues down the pipeline so that it reaches `WF` two cycles later than usual. There is no way there can be a `WF` structural hazard with a `add.d` and a `mul.d` because such a `mul.d` would have to be in `ID` in the same cycle as an `add.d`.

The result of a `lwc1` joins the FP pipeline at the `A3` stage. As with the `add.d`, the `lwc1` cannot have a structural hazard with a `mul.d` or `add.d`. The add and load results are combined using a mux in the `M5` stage. This reduces the number of pipeline latches used and also simplifies the control logic.

(*b*) Show any changes to the logic generating the `fd`, `we`, and `xw` signals. *Note: The original assignment did not ask for* `xw` *changes.*

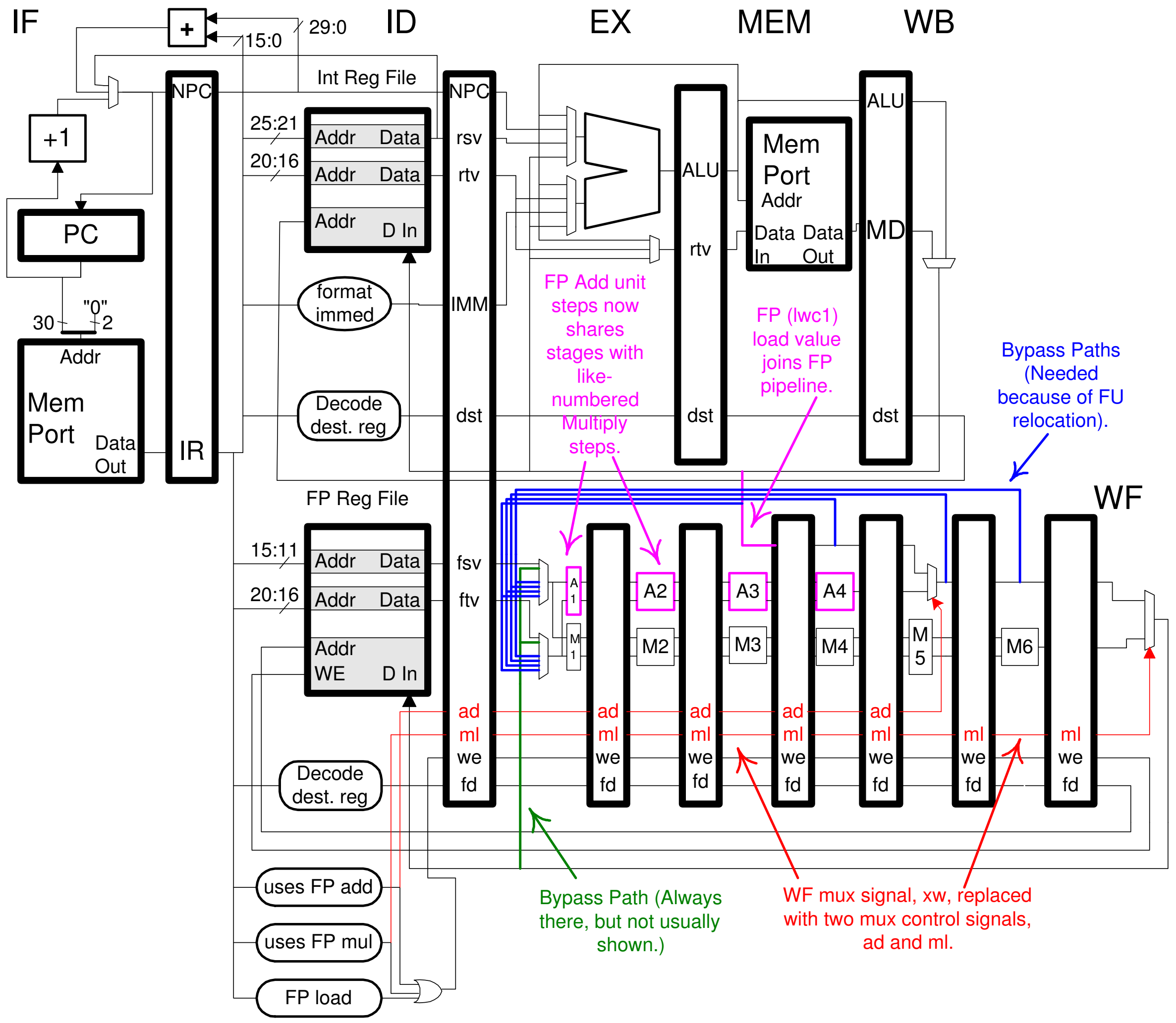
Because there can be no `WF` structural hazards all of the structural hazard logic has been eliminated. Because the `fd` and `we` signals now only enter in `ID` the logic is much simpler. As before, `we` is set to `1` if any instruction using `WF` is present, which for this implementation means an instruction using the FP Add Unit, FP Multiply Unit, or a FP load instruction. The `fd` signal is just the output of $\boxed{\text{Decode dest. reg.}}$ logic.

The `WF` three-input mux in the original design has been replaced by two two-input muxen: one in `M5` and one in `WF`. This simplifies the control logic.

(*c*) Show bypass paths needed to avoid stalls between any pair of floating point instructions mentioned above.

The solution below shows the pre-existing (but not usually shown) bypass path in green, and new bypass paths for this problem in blue. In the sample execution below bypasses are used in cycle 4 (`ldc1` to `add.d`) and cycle 8 (`add.d` to `sub.d`).

```
# Sample Execution:
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 ldc1 f2, 0(r1)   IF ID EX ME _3 _4 _5 _6 WF
 add.d f0, f2, f4    IF ID -> A1 A2 A3 A4 _5 _6 WF
 sub.d f6, f0, f8       IF -> ID -------> A1 A2 A3 A4 _5 _6 WF
```


IF
ID
EX
MEM
WB
WF
Int Reg File
FP Reg File
FP Add unit steps now shares stages with like-numbered Multiply steps.
FP (lwc1) load value joins FP pipeline.
Bypass Paths (Needed because of FU relocation).
Bypass Path (Always there, but not usually shown.)
WF mux signal, xw, replaced with two mux control signals, ad and ml.
uses FP add
uses FP mul
FP load
Decode dest. reg
format immed
Mem Port
PC
NPC
IR
A2
A3
A4
M2
M3
M4
M5
M6

Problem 2: Consider the changes to avoid structural hazard stalls from the previous problem. Provide an argument, either for making the changes and or against making the changes. For your argument use whatever cost and performance estimates can be made from the previous problem. Add to that the results of fictitious code analysis experiments and alternative ways of using silicon area to improve performance.

The code analysis experiments might look at the dynamic instruction stream of selected programs. For these experiments explain what programs were used and what you looked for in the instruction stream. Make up results to bolster your argument.

For the alternative ways of using silicon area, consider other ways of avoiding the structural hazard stalls, or other ways of improving performance. This does not have to be very detailed, but it must be specific. (For example, "use the silicon area for pipeline improvement" is too vague.)

The argument should be about a page and built on a few specific elements, rather than meandering long-winded generalities.

It's not worth it. Improvement is limited to a handful of programs, while the cost is substantial.

The benefit is only practically realized for a few FP-dense programs. Normally our compilers will schedule (arrange) FP instructions so that WF structural hazards are avoided. When analyzing SPECcpu2000 FP programs, we found only one program in which WF structural hazards remained after compiler scheduling. Even so, the resulting stalls caused only a 10% increase in execution time and a competent program could have re-cast that region of code to avoid any stalls. *Note: The remarks above about "our compilers" are fictional.*

The modified hardware includes three new pipeline latches (M3/M4 to M5/M6) plus their bypass connections.

The added cost might be used to increase the L2 cache size, benefiting all programs. Another option would be to add a second FP register write port, though that would add to the complexity of the control logic.

Problem 3: In the previous problem structural hazards were avoided by having all WF instructions pass through the same number of stages. If both WB and WF instructions passed through the same number of stages then, were it not for stores, it would *easily* be possible for floating-point instructions to raise precise exceptions without added stalls (even if exceptions could not be detected until M6).

(*a*) For this part, ignore store instructions. Explain why having all instructions pass through the same number of stages makes it easier to implement precise exceptions (without added stalls, etc.) for floating point instructions.

Implementing precise exceptions for FP instructions is difficult because FP instructions write back out of order, so that should it be necessary to squash instructions following a faulting instruction some of those following instructions may have already written back. If all instructions use the same number of stages then instructions will write back in order and so it will always be possible to squash instructions following a faulting instruction.

(*b*) For this part, include store instructions. Explain how store instructions preclude precise exceptions for the implementation outlined above, or at least for a simple one.

Consider first an implementation in which the memory port is in M2 (as in the solution to the first problem). After a store instruction passes through M2 it will have written memory and so it will be too late to squash it, precluding precise exceptions.

If instead, the memory port is in M6 (the stage before WB) it will be possible to have precise exceptions but instructions dependent on loads will have to stall. See the example below.

```
# Alternate solution, ME in stage _6.
#
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 ldc1 f2, 0(r1)   IF ID EX _2 _3 _4 _5 ME WF
 add.d f0, f2, f4    IF ID -------------> A1 A2 A3 A4 _5 _6 WF
 sub.d f6, f0, f8       IF -------------> ID -------> A1 A2 A3 A4 _5 _6 WF
```

(*c*) For this part, include store instructions. Do something about stores so that the all-instructions-use-the-same-number-of-stages implementation can provide precise exceptions to floating point instructions. It is okay if the modified implementation adds stalls around loads and stores. A good solution balances cost with performance.

If your solution is costly say so and justify it. If your solution is low cost but lowers performance say so and show the execution of code samples that encounter stalls.

One solution would be the ME in stage M6 variation shown above. Though the cost of this solution is low its need for frequent stalls would make performance too low.

Another solution would be to have the memory port provide the overwritten data on a store. That is, when a store instruction is in ME the Data Out of the memory port will hold the data which the store is about to overwrite. That data will proceed down the pipeline and ordinarily will just be discarded when the instruction reached WB. Going down the pipeline with the replaced data will be the effective address (this would require a new set of pipeline latches). If an instruction raises an exception, these replaced data and effective addresses will be used to write memory, undoing the effect of the stores.