LSU EE 4720 **Homework 5** Solution **Due: 21 April 2004**

Problem 1: One question when extending an ISA from 32 to 64 bits is what to do about the shift instructions. Because of the way that the shift instructions are encoded in MIPS two new shifts (of each type) were added to MIPS-64.

(*a*) What do you think the MIPS-32 `sra` instruction should do in MIPS-64? Remember that an implementation of MIPS-64 must run MIPS-32 code correctly. Please answer this question before answering the next parts (but feel free to look at the questions). *Hint: Any serious answer will get full credit. A smart-alec answer will get full credit only if it's particularly witty.*

Just shift the lower 32 bits, sign extend using bit position 31. Leave the high 32 bits unchanged. For example:

```
# My idea for how sra on a 64-bit machine should work.
# Before $r1 = 0x8888 8888 8888 8888
sra $r1, $r1, 1
# After  $r1 = 0x8888 8888 C444 4444  (Spaces added for clarity.)
```

(*b*) Give two reasons why the MIPS-32 `sra` (not `srav`) instruction could not be used for all right arithmetic shifts needed in a 64-bit program.

It can't specify a shift of more than 32 bits since the sa field is only five bits. If it sign extended using bit 63 as the sign it would not work for 32-bit code, if it sign extended on bit 31 it would not be appropriate for 64-bit code.

(*c*) What are the new MIPS-64 shift right arithmetic instructions? Give the mnemonics.

`DSRA` and `DSRA32`.

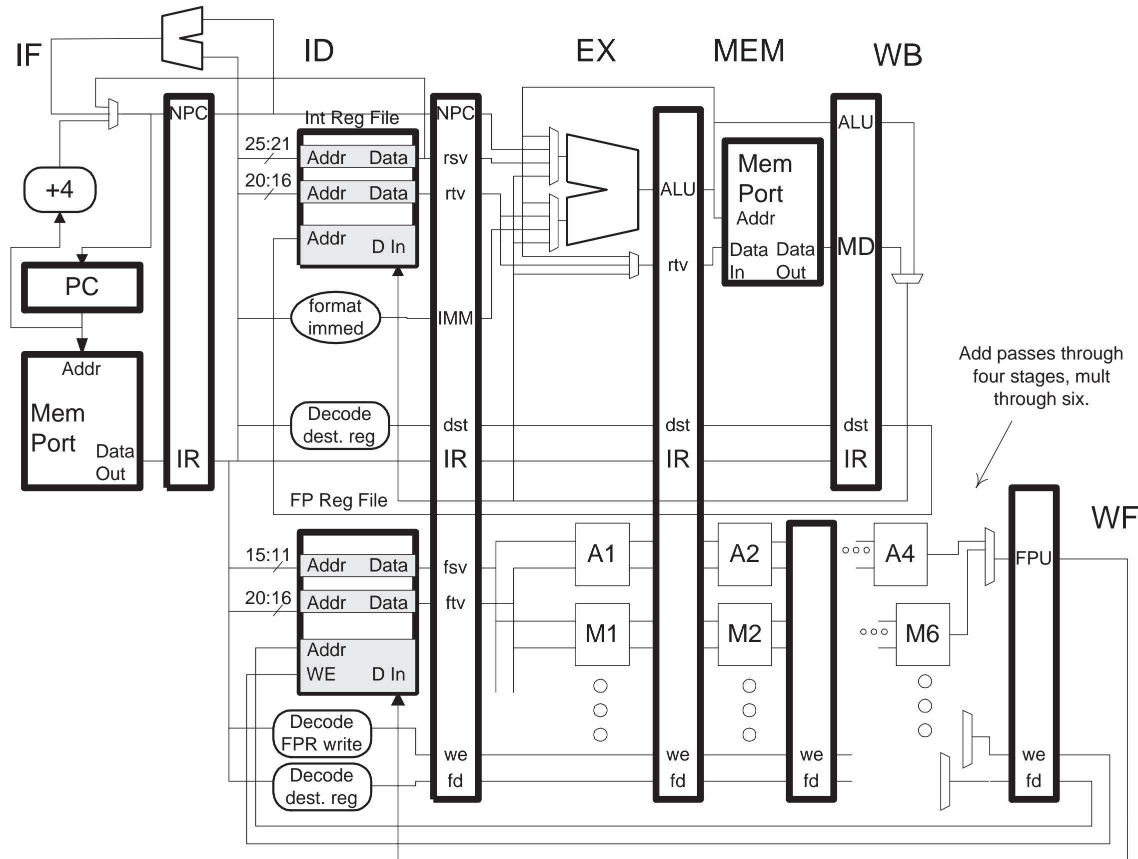
(*d*) Why were two (as opposed to one) new shift instructions of each type added to MIPS-64?

Because they use the existing five-bit sa field which can't specify a full range of shifts.

Problem 2: Do the Problem 2 (a) through (d) from the Fall 2003 EE 4720 final exam (the one on floating point instructions). (`http://www.ece.lsu.edu/ee4720/2003f/fe.pdf`) Do not look at the solution until after you have solved the problem or gave it a good try.

See `http://www.ece.lsu.edu/ee4720/2003f/fe_sol.pdf`.

Problem 3: In the diagram below the `we` pipeline latches carry write enable signals for use in floating point writeback. If the functional units were arranged differently the `we` pipeline latches could be used as a reservation register (for detecting WF structural hazards).



(*a*) Redraw the diagram with that arrangement. *Hint: Try to use the* `we` *signal in the diagram above for a reservation register. Figure out why that won't work and come up with a solution.*

See the next page.

(*b*) Suppose the ID stage has boxes [uses FP ADD] and [uses FP MUL] to detect which (if any) floating point functional unit an instruction would use. Design the control logic to generate a stall signal if there would be a write float structural hazard.

See the next page.

(*c*) Add the connections necessary for a `lwc1` instruction. Include the connections needed to detect a WF structural hazard (as was done for ADD and MUL in the previous part).

See the next page.

The multiply and add functional units are rearranged so that they finish "together" rather than start "together." In the original arrangement a particular `we` pipeline latch is always a fixed distance from `ID`, but their distance from `WF` depends upon which instruction they carry. For example, the first `we` latch is always one cycle from `ID`, if it is carrying an `add` instruction it is four cycles from `WF` but if it is carrying a `mul` it is six cycles from `WF`. In a reservation register a particular bit position describes an instruction a particular distance from `WF` and so the `we` pipeline latches in the original diagram cannot be used as a reservation register.

In the modified pipeline, below, the `we` is a fixed distance from `WF` and so it can, and is, used as a reservation register. Instructions using the FP add check the second `we` pipeline latch and if it holds a 1 an ID stall signal is asserted, otherwise the add (in particular, the `we` signal, register number (`fd`), and a control signal for the `WF` multiplexor) is inserted into the pipeline. Note that the operands themselves enter whether or not ID is stalled, if stalled then later at `WF` the result will be ignored. Similar logic is shown for the `lwc1` (and other FP loads) instruction. Note that instructions using the multiply unit never check the reservation register, that is because they take the longest (we're ignoring divide) and so no other instruction can use `WF` at the same time.

The diagram also shows the control logic for the `WF` multiplexor.