LSU EE 4720 **Homework 3** Solution **Due: 9 April 2008**

For answers to these questions consult the SPECcpu2006 Run and Reporting Rules (which can be found at `spec.org`*).*

Problem 1: One way testers can stretch the rules is by using compiler optimizations that give good performance when they work correctly but are too error prone for non-experimental use.

(*a*) Why would it be a bad idea for SPEC to limit allowable compiler optimizations to those that are already known to be safe? (Say, dead-code elimination based on a SPEC-provided analysis technique.)

(In the question "known to be safe" is not the same as "safe." A tester might have good reason to believe that an optimization is safe but such an optimization is not known by SPEC to be safe and so, based on the question above [but not in real life], could not be used.)

Limiting allowable optimizations to a short, conservative list would discourage development of new compiler optimization techniques. Compiler optimization is a perfectly legitimate way of achieving performance goals (and is in fact preferred over elaborate hardware techniques).

(*b*) Rather than dictate allowable optimizations the rules instead explain that if it's good enough for your customers it's good enough for SPEC, though not in those words. Find the section in the run and reporting rules where this rule is given.

Section 1.3.2.

(*c*) For at least three bullet items in the section (from the last part) explain what sort of unscrupulous action the bullet item is supposed to prevent.

- be specified using customer-recognizable names

The compiler is available and the company would sell it to any customer than can provide its name, but the name is kept secret from the customer.

- be generally available within certain time frames

The compiler is never made available.

- provide documentation

The compiler can't be used because it is undocumented.

- provide an option for customer support

The compiler can't be used because there is no support if a customer can't figure out how to use it.

- be of production quality

The optimizations are too buggy for reasonable use.

- provide a suitable environment for programming

The optimizations can only be used for very narrow purposes (the particular benchmarks).

Problem 2: When preparing a run of the SPEC benchmark the tester provides, among other things, libraries (such as the C standard library that contains routines such as strlen, malloc, printf). It is in the testers interest to make sure these library routines run as fast as possible and is free to do so within the SPEC rules.

Section 2.1.2 stipulates that one can't use flags that substitute library routines for routines *defined in the benchmark*.

In addition to base and peak, imagine a third metric called *swap*, in which the rule in Section 2.1.2 didn't apply. Testers could abuse the swap metric by substituting routines that merely return the correct value (since input data is known in advance), but for this question suppose testers

comply with the spirit of the SPEC rules and substitute routines which provide higher performance for any input data.

(*a*) Comparing the peak scores to the base scores shows the additional performance that can be obtained by a suitably motivated and resourced expert. Explain what might be learned by comparing swap scores to base and peak scores. (That is, where might the higher performance be coming from.)

If the benchmarks were well-written the swap result might show performance obtainable by structuring the computation for the implementation. For example, the code might be re-written so that it could use packed-operand (sometimes called multimedia) instructions, something a compiler couldn't always do because, for one reason, it doesn't know if using lower-precision and saturating arithmetic is okay.

If the benchmarks are not well written the swap result might show how poorly written they were. (That is, the re-writing would benefit many systems, not just the one it was re-written for.)

(*b*) Provide an argument that the swap metric is a good test of a system that complements base and peak.

The compiler is not smart enough to use some special instructions, such as packed-operand instructions, in many programs and so neither base nor peak would show the true potential of the system.

(*c*) Provide an argument that swap doesn't really tell you anything about the system (CPU, memory, compiler and other build items).

The swapped routines might improve the performance of any system and so the swap result would just show how many skilled programmers the tester was able to use to prepare the test.

Problem 3: For exceptions the handler needs to know the address of the faulting instruction both so that it can examine the instruction and so that it knows where to return to in case the instruction needs to be re-executed or skipped. *For answers to this question consult the ARM and MIPS32 (Volume 3) ISA manuals on the course references page.*

A programmer-friendly ISA would provide the handler with the address of the faulting instruction, however in both MIPS32 and ARM may provide an address *near* the faulting instruction.

(*a*) In which registers do MIPS and ARM A32 write the approximate faulting instruction address? (For MIPS give the register number as well as its name.)

MIPS writes the address to register `c14` (co-processor 0 register 14), named `EPC`. ARM writes the address to `r14`.

(*b*) The address that MIPS provides may be that of the faulting instruction, or it may not be. When is this done, and what is the other address?

If the faulting instruction is in the delay slot of a CTI (control-transfer instruction) then register `c14` is written with the branch address. The handler will be able to determine which instruction raised the exception, but it will return to the CTI, executing it a second time.

(*c*) ARM A32 also does not provide consistent addresses. What addresses does it provide? Give a credible reason for the differences in addresses.

Let `PC` denote the address of the faulting instruction if a load or store raised the exception `r14` is written with `PC+8`, for most other instructions it writes `PC+4`.

One reason for this inconsistency is that the implementation is expected to branch to the handler as soon as the exception is discovered, and for loads or stores the discovery might be one cycle later. The implementation does not bother sending an instruction PC down the pipeline so the exception mechanism uses the current PC value.

Note that ARM implementations would have to write these addresses whether or not the reasoning above is correct.