

## Re: non load/store architecture?

---

*Source:* <http://coding.derkeiler.com/Archive/General/comp.arch.embedded/2006-12/msg00772.html>

---

- *From:* David Brown <[david@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:david@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Thu, 14 Dec 2006 15:44:28 +0100
- 

Brandon J. Van Every wrote:

David Brown wrote:

Where does that leave us? A pure RISC architecture is best when aiming for maximal ipc, and can be run at higher clock speeds as each step is simpler. But the ColdFire code is more compact, leading to lower bandwidth requirements on the instruction bus, and it does more per instruction, giving better performance for the same ipc.

Working on OpenGL device driver optimization for the DEC Alpha, I never saw instruction cache misses. Only data cache. Performance code is in small loops, not huge hulking one-shots. I say "more compact code improves performance" is theory, and not observable in practice. More compact data, on the other hand, matters a great deal.

The advantages of compact code will depend a lot on the rest of the device, such as the types, sizes, and speeds of the cache(s) and databuses. Also relevant to c.a.e., though not a speed issue, is that the size of the code has a direct bearing on the cost for typical embedded systems (i.e., flash code store).

Being low power is a good indicator of an efficient ISA – the x86 is not low power, and neither are most fast RISC cores as they need high clock speeds. Remember, what's important for a processor is the work done per clock, not just instructions per clock, and in comparison to a pure RISC architecture, the ColdFire sacrifices a little ipc for a lot more work done per instruction.

The units of work I've always cared about are FPU adds, multiplies, and divides. There isn't more arithmetic work to do per instruction. You could do more load/store work, but assuming you hit your primary data cache, that's not your bottleneck anyways. The arithmetic is. As I said above, instruction cache bloat doesn't matter in tightly looping code. Or, I'd wager, in loosely looping code either. Instruction caches are pretty big compared to the looping code. If all your code

## Re: non load/store architecture?

is one-shot then you've got completely different system caching issues, nothing to do with the CPU.

For the type of code you are talking about, that's true. As always, there are no correct answers as it all depends on the application. In particular, if you are doing heavy FP work the the FP units are likely to be the bottleneck, and individual instructions shuffling data around or doing simple arithmetic (the most common type of instruction in most code) don't matter much. But in a lot of code it does matter. Take the simple C code "x = 123456;", where "x" is a 32-bit global variable. On the ColdFire, this compiles to:

```
move.l #123456,%d0
move.l %d0,x
```

Two instructions, each 6 bytes long, each executing in 1 clock (plus a write access to memory).

On the PPC, this compiles to:

```
lis %r0,0x1
ori %r0,%r0,57920
lis %r9,x@ha
stw %r0,x@l(%r9)
```

(See what I mean about ColdFire code being nice and clear?)

That's four instructions, each 4 bytes, each executing in 1 clock (plus a write access to memory).

The ColdFire generates more compact code, running at twice the speed for the same clock. That's what I mean by greater work done per clock.

Now I suppose if you design CPUs with almost no cache, you might care about instructions being small. But then, you're not designing a performance CPU anyways. So who's gonna care about the performance? "Good" won't mean optimization, it'll mean low power or cheap to manufacture or something.

We are clearly coming from this from different experiences, if OpenGL drivers on an Alpha are typical for your programming, while I work mostly with smaller processors (the ColdFire I am using at the moment has no cache – all its flash and sram are internal, with single cycle access). But performance is very important to small systems – high performance means you can use slower clock speeds, leading to lower power, lower EMI, and cheaper components. It might not be the most important factor, but it is still there.

Even on cached processors, small code means better use of the cache. Critical loops will (should!) fit within even a small instruction cache, but programs consist of more than their critical loops. A complete instruction cache miss might mean a stall of a hundred or more clock cycles (which might be worth twice that in instruction counts on a superscaler processor) – there is a reason why more expensive processors have larger instruction caches. More compact code gives the same benefits of a larger cache.

Re: non load/store architecture?

Re: non load/store architecture?

Cheers,  
Brandon Van Every