RETROSPECTIVE:

Very Long Instruction Word Architectures and the ELI- 512

Joseph A. Fisher

Hewlett-Packard Laboratories
Cambridge, Massachusetts
jfisher@hpl.hp.com

VLIW Architectures and Region Scheduling

In this paper I introduced the term VLIW. VLIW was motivated by a compiler technique, and, for many readers, this paper was their introduction to "region scheduling" as well. I had put forward the first region scheduling algorithm, called Trace Scheduling, a few years before. Since region scheduling is a compiler technique, it is of interest to fewer people, but it enables superscalars and VLIWs with lots of instruction-level parallelism (ILP). Because I could see the power of region scheduling, I first began to think about VLIWs. I was fortunate in that this allowed me to coin the term Instruction-level parallelism, and to work out a lot of the original details and terminology of ILP, before many others believed it was important.

How VLIWs Came About

VLIWs came from my work as a graduate student at Courant. Ralph Grishman (my advisor) and I built PUMA, a CDC-6600 emulator that Ralph had designed (it worked well and eventually several were made and used to replace some big, old supercomputers of their day). One of my jobs was "chief tool builder" — I read the literature, learned the state of the art in ECAD algorithms, and then wrote tools to do most of the wire routing, partitioning, chip layout, simulation, and so on. You couldn't buy tools like that at a university then. The tools I wrote worked very well, but I was really frustrated with how hard it was to write, and especially maintain, the 64-bit horizontal microcode PUMA used (PUMA stood for "Processing Unit with Microprogrammed Arithmetic"). I originally thought of the problem that I solved with region scheduling as a hardware design problem. What I wanted to do was convert vertical (serial) microcode into horizontal microcode. The analogy to chip layout is clear:

- Chip layout involves converting a 1-dimension representation (a chip list) into a 2-dimensional representation (a placement); I was converting serial operations into 2- dimensional horizontal microcode

- Chip layout has nodes (chips) connected by edges (wire); I had operations connected by a data precedence relation.

- Chip layout tries to minimize wire length given the constraints of wiring; I had to minimize schedule length given the constraints of data precedence.

Because of this analogy, I was surprised when I realized that this was really a part of compiling, relying more upon compiler tools than CAD techniques. Fortunately, I was at Courant, which was then compiler heaven. (The holy scrolls were copies of Cocke & Schwartz, which included a catalog of optimizations; Ken Kennedy had just gotten his degree there.) The few people looking at this problem elsewhere were scheduling basic blocks, and then trying to iteratively improve the schedule by moving operations from block-to-block afterwards. The key insight was to recognize that this locked you to too many bad decisions, and you should instead look at a lot of code at once — for example a long execution trace. Trace Scheduling lets you do that, and frees you up to generate far more ILP. Since then, new region selection algorithms have suggested other regions of choice, often a more limited subset of a trace, reducing the complexity. Some of the region scheduling regimens that have gotten the most attention include Percolation, Superblock, Hyperblock and Trace-2.

Now that more and more ILP is present in microprocessors, region scheduling has become the technology of choice in high-end compilers.

Why Not VLIWs?

Given Trace Scheduling, I wondered why you couldn't build a RISC-style CPU with lots of ILP, and thus run really fast. Indeed, the farther my group and I went (I was at Yale by then), the more it seemed obvious that you could, and that it would be a good thing at least some of the time. I then learned a couple of things. First, you can get more people, a LOT more people, to come to your talk if you promise them bizarre sounding hardware instead of a compiler technique. Second, many people seemed to think that it would be very hard to build such a thing.

The first effect was good and bad. It's probably mostly my fault that a lot of people think of VLIWs as a weird kind of beast, rather than one of the natural alternatives once you start thinking about lots of ILP. I think that if I had presented these ideas in the context of a 2-issue compiled LIW, or a superscalar of any size, people would have thought more soberly about the idea. But because of region scheduling, I felt there was a real use for systems that could issue 7 (or many more) operations per cycle. This much ILP and weird long instructions to boot; it was too much for most people to accept. It was great for me professionally: I put forward a lunatic-seeming proposal, and then had it turn out to be practical, at least for some important uses, and not bad at worst in any case.

The second effect absolutely amazed me, and it still does. Why would this be impossible to build? I'd ask them, and I guess the lack of answers meant that you couldn't because no one ever had. (Really, in 1998 it seems amazing that people actually believed you couldn't build such a CPU at all.) Whether you'd WANT to build one seemed legitimately controversial and it still does, whatever I personally believe. The real question was: having built it, what now? Does this region scheduling stuff really work well enough to justify it? Answer: sometimes. Anyhow, I can't portray strongly enough the way in which this concept was greeted by the many minicomputer manufacturers John O'Donnell and I approached in trying to convince them to build one. I recently ran across a wonderful article in the San Francisco Chronicle (7/9/97, first Business Section page), quoting Ray Simar, Texas Instruments' program manager for their new generation of DSP chips:

The theoretical breakthrough came out of Yale University in the early 1980's. "I remember looking at the idea and saying these guys were nuts," Simar recalled. "I thought there was no way it would work in the real world." But three years ago, when Simar was given the job of coming up with a great leap forward in DSP, he revisited the Yale work with new appreciation. "At the end of the day what we thought was ridiculous was the best solution," he said.

It's just amazing to read someone being as forthright as that. Indeed, that is how people felt.

Some Naivete

There were a lot of aspects of this paper that seem naive from the perspective of 1998. Some of these were genuine naivete, others were simply artifacts of their time. The most significant is the lack of any mention of object-code compatibility as an issue for VLIWs. This issue was not on my radar screen. Being compatible with another company's binaries was an oddity, and most manufacturers changed architectures willingly. (If you copied someone else's, you sort of weren't even a legitimate computer vendor in some people's eyes, you were a "clone manufacturer".) People sometimes brought it up, but not often until Multiflow, when it was considered a big issue. (At Multiflow, we were "upward" compatible, but not as much as we'd have liked to be. You could run Trace 7 code on a Trace 14, but that was it. The 28, the one with 1024 bit instructions, you really needed to recompile for to run correctly.)

I never dreamed that there might someday be techniques that operated at run time that might solve the binary compatibility problem and change a lot of our other architectural considerations. (I've called these walk-time techniques, but nobody else seems to.)

A truly naive thing is that I described VLIW as an architecture, without really being conscious of the distinction between architectures and implementations. I didn't know to say it then, but VLIW is really a design philosophy, much like RISC, CISC, superscalar, vector processor, etc. To me, the part of VLIW that mattered then and matters now is the philosophy that one should get a lot of ILP in a processor without asking the hardware to do much to locate and schedule it. As with anything of this sort, there's a spectrum. I think of an implementation that expects operations to have been arranged so it can trivially put them in parallel at run-time as a good embodiment of this philosophy. I think of processors that significantly rearrange code, mapping architectural registers into physical registers, etc., and in general thinking instead of computing the answer, as embodiments of opposite. Does the object code actually have to have wide instructions in it for an implementation to be a VLIW? Not to me, and I'm not really interested in the question, any more than I want to know whether a particular processor is really RISC. I can tell the extent to which they follow this philosophy, and I think it's a good design philosophy.

At least two more truly naive things appear in this paper. First, I really ignored the whole question of exceptions. As anyone who builds real CPUs knows, you spend more time handling that problem than any other. Exceptions are a real pain for out-of-order processors. They're probably worse for superscalars than VLIWs, but no fun either way. Second, the memory system I thought was desirable, was, instead, baroque and silly. John Cocke tried to straighten me out. He suggsted that it would be enough to try to avoid references to the same bank, and that it would be a good idea to ignore this back-door nonsense. But I didn't listen.

Some Terminology

Finally, it's worth clarifying some terminology to put this paper in a more modern context. When John Ruttenberg and I laid out the basics of the high-level ELI architecture, we decided that register banks had to be split. We termed the combination of register banks and the functional units that took their operands from them a "cluster". That term has mostly stuck, but lately there have also been references to "split register bank architectures".

This paper addressed the problem of telling whether indirect references are to the same address or not, and called that problem "anti-aliasing". Because that term already had such strong meaning in the graphics world, I later renamed it "memory disambiguation", and that ugly term stuck. (Yale Patt always complains that his term, "the unknown memory address problem", was better. This shows again that short and ugly beats long and careful every time.) VLIW itself is another short and ugly term that stuck. I tried SPIE (for Static Parallel Instruction Execution). That turned out to be a conference name, which I found out when I used the term in a grant proposal and got on all the wrong mailing lists. I eventually came up with VLIW. I figured if VLSI could stick, why not VLIW? VLIW unfortunately emphasizes the long instruction implementation detail over the explicitly parallel instruction design philosophy that is really the key aspect.

One last thing I'd like to mention is that Mary Claire vanLeunen (author of the still wonderful "A Handbook for Scholars", revised ed., Oxford University Press, New York, 1992) taught me to write in the course of editing this paper, for which I'm still very grateful. Several others had tried; some had helped a lot, but this was where it took. "It's a lot like programming. Your goal is transparency," she told me. I figured that if she knew that about programming — not so many people knew that about programming then — she was undoubtedly right about writing. Recently, because of her lessons, I had reviews of a paper that said, "This is such a clearly written paper, it should be published on those grounds alone," and, from another reviewer, "This paper was written in an annoyingly juvenile style." Right!