Internet Nuggets

Mark Thorson
mmm@cup.portal.com
March, 1995

This column consists of selected traffic from the *comp.arch* newsgroup, a forum for discussion of computer architecture on Internet—an international computer network.

As always, the opinions expressed in this column are the personal views of the authors, and do not necessarily represent the institutions to which they are affiliated.

Text which sets the context of a message appears in italics; this is usually text the author has quoted from earlier messages. The code-like expressions below the authors' names are their addresses on Internet.

Superscalar vs. VLIW

Marc Tremblay
tremblay@eng.sun.com

Superscalar seems unable to handle more than four simultaneous operations, of which only two or three can be integer. In principle, VLIW could handle an arbitrary number of parallel operations, limited only by the ability of compilers to generate those operations...

...does everyone agree that future systems will all be VLIW because superscalar can't handle the vastly higher transistor count and I/O count of next-generation silicon technology?

(My opening statement while on a panel on VLIW at the recent Microprocessor Forum basically addressed this issue.)

One of the main differences between VLIW machines and superscalar processors is the logic needed for resource allocation and dependency checking. In order to maintain binary compatibility, superscalar processors do the checking/dispatching in hardware while VLIW machines give up hardware binary compatibility and let the software organize instructions (and NOPs...).

The resource allocation and dependency checking logic represents about 5% of the total area on UltraSPARC, a 4-way superscalar processor. Most of the work required to do the dispatch is done in a single stage. Other stages are involved to speed up the task, for instance the previous stage adds 32 bits of "pre-decoded" information to each instruction when they enter the instruction buffer.

What happens to this logic if let's say we decide to go to an 8-way machine? The reason I choose 8-way is that several papers have claimed that given the current state of the software (both programming languages and compilers), a machine offering more parallelism would not be used very effectively. Notice that the complexity of the grouping logic is quadratic with respect to the maximum number of instructions dispatched. So, for an 8-way superscalar processor:

1. The area grows by 4X (with respect to 4-way).
2. Timing becomes prohibitive for a single stage.

So, what can be done?

1. Area: 5% X 4 = 20%. New process technology combined with a larger die, brings this percentage down to ~8%.
2. Timing: dedicate two or even three stages to the grouping logic.

a) New branch prediction algorithms (2-levels, using branch correlation) can now reasonably reach 96% accuracy on SPECint92 (vs. 86% for current 2-bit counter algorithms used on UltraSPARC, Alpha, PPC 620, etc.).

b) Consequently, each extra stage in the front of the pipeline, i.e. before the branch is resolved, now only cost around 0.8% (20% conditional branches, 4% mis-predicted, 1.0 CPU CPI => 0.8% overhead).

Now would a company give up hardware binary compatibility for saving 8% of the die area and gain 1.6% in performance (assuming two extra stages)? *No way.*

Unless:

There is not much software based on the previous generation (so compatibility is not a strategic advantage).

The current instruction set makes it very difficult to go to a wider machine (e.g. variable length instructions).

So, superscalar processors should be around for a long time. Now if people start writing code in a different way (we all know how long that takes), and if compilers can take advantage of it, then building a 20-wide machine may require dropping hardware checking -:).

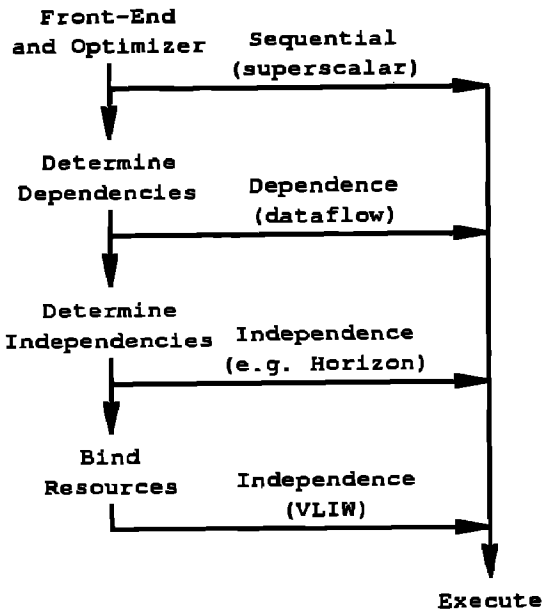
Notice that for a 4- to 8-wide processor, let it be VLIW or superscalar, many other parts will also limit the cycle time (e.g. large multi-ported register files, large multi-ported caches, complex bypass logic, etc.). And we haven't even started addressing bigger bottlenecks, specifically, memory bandwidth, both for instructions and data.

Superscalar vs. VLIW

John Setel O'Donnell
jod@equator.com

The best reference as an overview of the issues is Rau and Fisher's paper "Instruction-Level Parallel Processing: History, Overview and Perspective" in *The Journal of Supercomputing* 7:1/2, 1993. This issue is also available as a book, *Instruction-Level Parallelism*, from Kluwer.

Fisher and Rau propose a model to categorize the division of labor in instruction-level parallel execution. They point out that there's a natural flow of tasks required, and that different architectures address these tasks with a different division of labor:



No matter what the architecture style, when you build a new processor model you're very tempted to change the resources in the execute unit to take best advantage of the technology you're working with. In a VLIW, that change is exposed, possibly requiring recompilation to use the new machine.

Computer architecture is the contract between the hardware guys and the software guys about who'll handle what; it's a good contract only to the extent that it's good for each of them. A decade ago, the RISC architectures had the characteristics that VLIW architecture has today: they were incompatible with what had gone before, but they resulted in the smallest, fastest-time-to-market chips (remember Bill Joy talking about the semiconductor process advantage of getting to market sooner?), and the best cost-performance. Why? Because the architecture matched the implementation. SPARC, MIPS, etc. all exactly matched the pipeline structure of the chips then being built.

The RISC architectures were (unfortunately or fortunately, depending on your point of view) kept relatively static while the hardware evolved radically, so that machines now have a very high degree of "superscalar pressure": the in-order sequence of operations no longer remotely resembles what happens at runtime.

This does mean you can run your existing software right away (the schedule hit was taken during chip development—most/all superscalars so far have gone 20 months over original schedule), but it means the chip carries a substantial penalty in complexity.

By the way, today's superscalars have not significantly simplified the job of compilers; the high-performance compilers model what the machine does at runtime and do instruction scheduling to try create sequences with the minimum number of runtime-detected conflicts.

In the "pure" VLIW approach (Multiflow-style VLIW), the architecture totally reflects the underlying implementation. This removes the complexity and area penalty from the chip, removes the N^2 complex scheduling logic, and opens a much wider range of implementation choices which allow scaling to higher degrees of parallelism.

This has good effects: VLIWs can scale to higher clock frequencies (c.f. Alpha, the most VLIW-style implementation). It also has the effect that you must recompile for each new implementation; at Multiflow we had eight different ISAs for the six models we shipped (and the two that didn't make it out the door). In some applications/markets this is of little or no concern (e.g. in embedded systems, PDA's, multimedia: no reason to care a whit about *object* code compatibility); in others, x86 is the really important object code to be compatible with.

The HP/Intel project has the challenge of retaining x86 (and PA?) compatibility while bringing "VLIW-ness" to the processor. This might mean a machine with the full superscalar scheduling logic which can somehow be bypassed in "VLIW mode"—imagine an AMD-style front end for the x86 side and a VLIW mode where the compiler output goes directly to the decoded cache. However, it seems likely that the range of architectural choices will be greatly restricted. For example, growing the number of register files as the number of functional units increases—a key technique to maintain simplicity and clock rate—is obviously out.

Fisher and Rau have publicly described their "new, improved, not-your-father's-Oldsmobile" ideas for compatibility among members of a VLIW product family. They've described an architecture approach—"split issue"—for handling latency variation among implementations, but have only discussed emulation—left-to-right semantics and piecewise execution—as a means for handling increases in width as you describe above.

(If the manufacturer had been really clever and had *planned* a 4-way and an 8-way VLIW, the 4-way machine could natively execute the 8-way's code at half speed, so the 8-way could be the compiler default.)

The best scaling comes when you let the implementors start anew in each generation of the process technology and reexamine the tradeoffs they made last time. Obviously, in doing such a project the compiler writers are first-class members of the chip architecture team, because almost every tradeoff crosses over into the compiler.

Someone recently quipped on comp.arch that the *real* problem with VLIW architectures is that they require the hardware and software folks to work together, and that they're doomed on that score alone. Until companies that design chips place the same responsibility for performance, clock rate, etc. on their compiler writers as they do on their chip designers—and the same rewards for success—he's probably right.

Three broad classes of problems exist in VLIW self-compatibility:

1. Latency management. If we generate a schedule for a VLIW of a given width, with given latencies, and then later want to run that code on another VLIW of the same width but with different latencies, we have a problem. If the real latencies are longer than expected, we could solve the problem by stalling the machine—but bad performance might result and we

might need N^2 dependency checkers just like a superscalar machine. If the real latencies are shorter than expected, we have a problem; the data is written early and results might be incorrect.

Multiflow handled this last issue by declaring registers "dead" between the cycle a pipelined operation targeting the register was issued and the cycle it completed; this increases register pressure but makes for simple context switching.

Rau proposes that we solve this one by "split issue": when issuing the floating-point add, you just specify its operands. Later, you issue a "grab fpadd output into rdest" operation. This operation will stall if the hardware isn't done yet, and has a FIFO backing it up to absorb early-arriving data.

2. Functional unit management. E.g., build a 4-wide and an 8-wide. Fisher discussed a couple of ideas for this one:

   a) Piecewise execution. Make the operations in the VLIW correctly issuable sequentially in left-to-right order. This means that an operation targeting R1 must be issued left of one reading R1 if they're in the same instruction; otherwise the results of simultaneous vs. sequential issue would be different. (The Multiflow machines did *not* have this restriction on their semantics.) Then any machine can execute "pieces" of the code of any other machine; in particular, the 4-wide can issue 8-wide code.

   This is not an unrealistic situation; you can imagine desktops wanting to run the same code as the big hairy server machines. This gets downward but not upward compatibility.

   b) Install-time recompilation. Have "fat binaries" a lá PowerMAC with more than one version of the object code in the file. Bury the final register allocation and instruction scheduling stuff in the OS, and invoke it when a program is "installed". This way, stuff you use often you take the one-time hit to get full performance, stuff you use less often runs, but not at peak performance.

3. Resource management (other than issue slots and registers). The Multiflow VLIWs achieved their cost/performance partly by software management of bus, memory bank, and register bank conflicts. Limited (saturable) interconnect was built, but built without locks or arbiters. Software was responsible for ensuring that , for example, pipelined operations started three instructions ago didn't oversubscribe the write ports on a register bank when coupled with the short-latency ops being issued in this instruction.

   No solution on this one; you can't have this kind of thing in the architecture if you want any compatibility. This isn't such a big deal.
