Joseph A. Fisher, Paolo Faraboschi, and Cliff Young

VLIW Processors: From Blue Sky To Best Buy

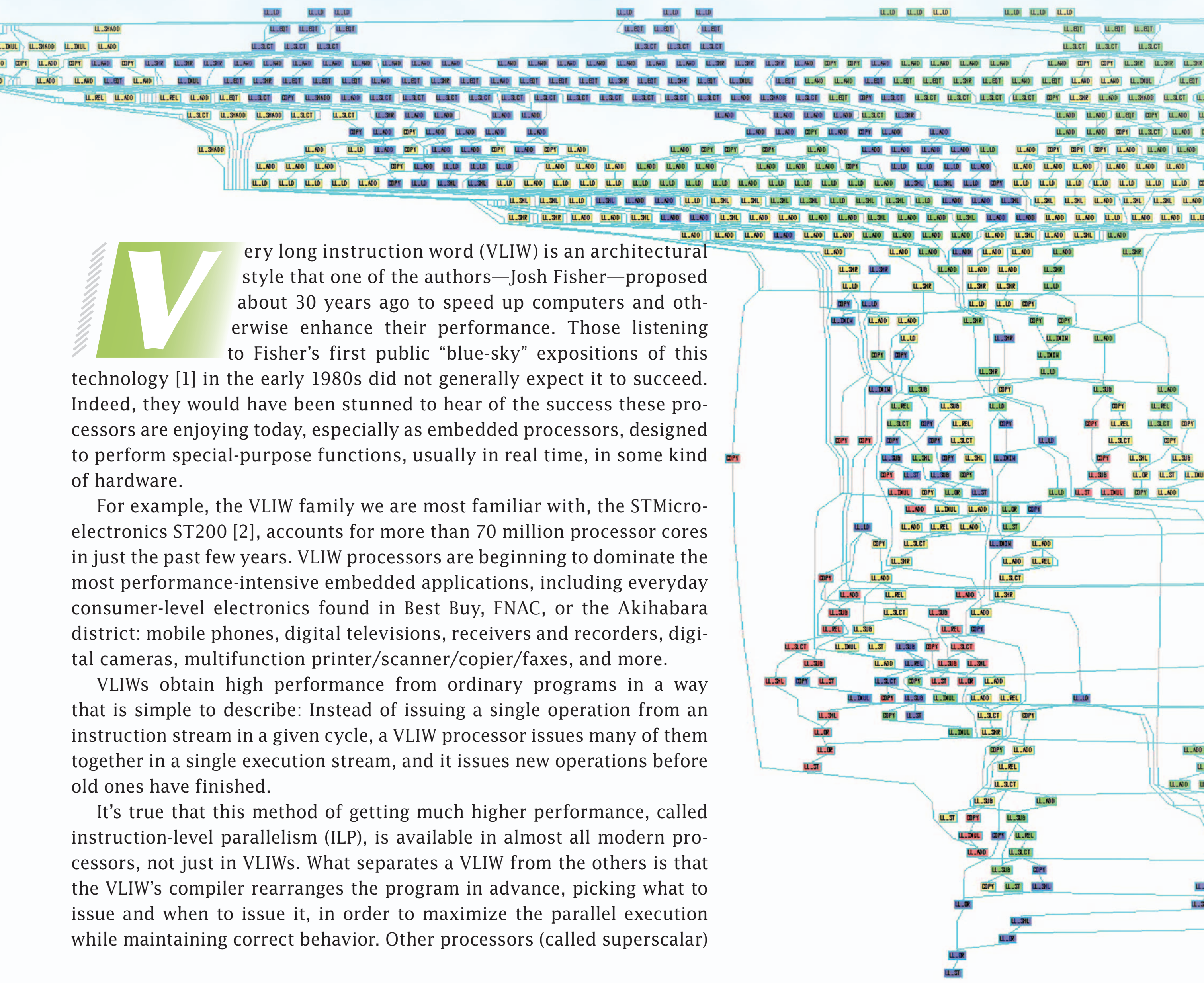


IMAGE COURTESY OF JOSEPH A. FISHER

Very long instruction word (VLIW) is an architectural style that one of the authors—Josh Fisher—proposed about 30 years ago to speed up computers and otherwise enhance their performance. Those listening to Fisher's first public "blue-sky" expositions of this technology [1] in the early 1980s did not generally expect it to succeed. Indeed, they would have been stunned to hear of the success these processors are enjoying today, especially as embedded processors, designed to perform special-purpose functions, usually in real time, in some kind of hardware.

For example, the VLIW family we are most familiar with, the STMicroelectronics ST200 [2], accounts for more than 70 million processor cores in just the past few years. VLIW processors are beginning to dominate the most performance-intensive embedded applications, including everyday consumer-level electronics found in Best Buy, FNAC, or the Akihabara district: mobile phones, digital televisions, receivers and recorders, digital cameras, multifunction printer/scanner/copier/faxes, and more.

VLIWs obtain high performance from ordinary programs in a way that is simple to describe: Instead of issuing a single operation from an instruction stream in a given cycle, a VLIW processor issues many of them together in a single execution stream, and it issues new operations before old ones have finished.

It's true that this method of getting much higher performance, called instruction-level parallelism (ILP), is available in almost all modern processors, not just in VLIWs. What separates a VLIW from the others is that the VLIW's compiler rearranges the program in advance, picking what to issue and when to issue it, in order to maximize the parallel execution while maintaining correct behavior. Other processors (called superscalar)

Digital Object Identifier 10.1109/MSSC.2009.932433

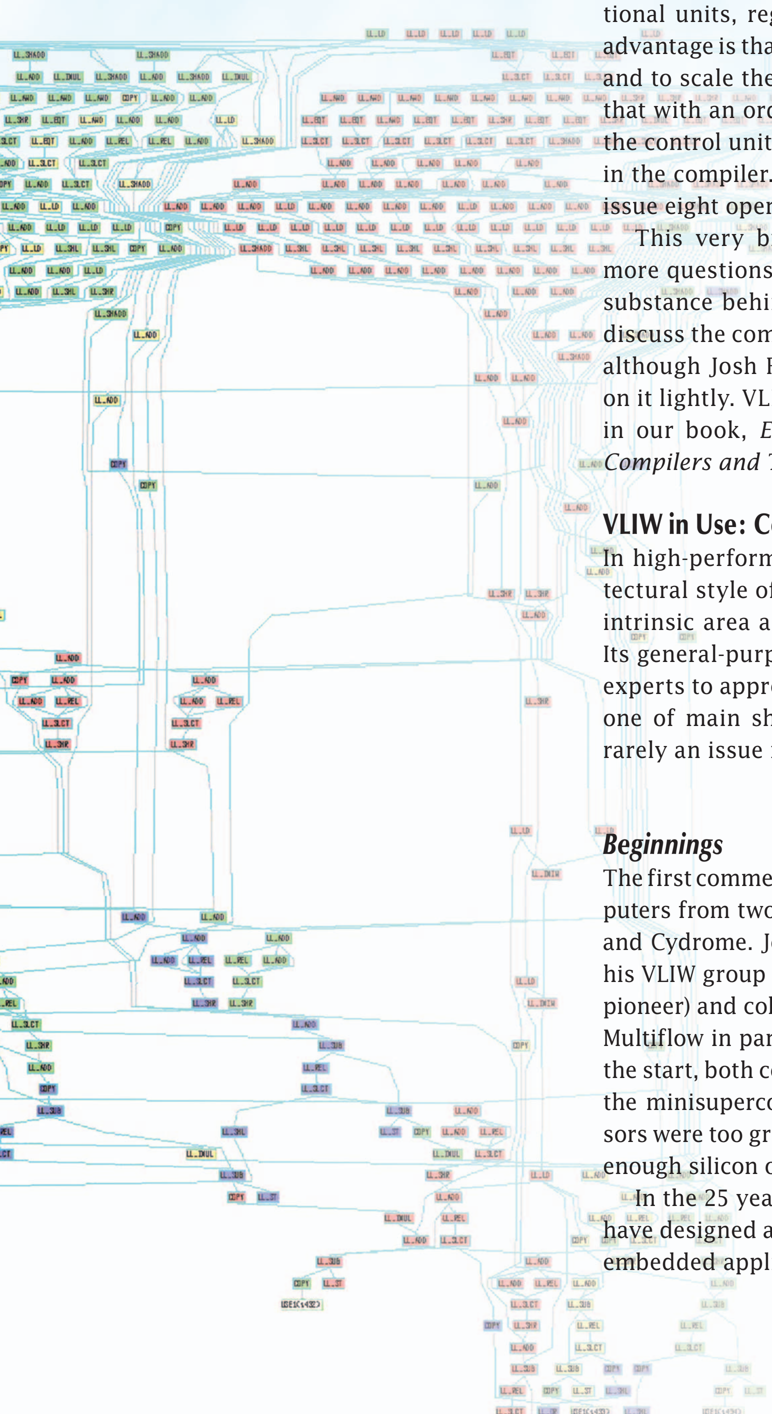
rely on the hardware to do this while the program runs (see Figure 1). Although the tradeoffs are complex, VLIWs can achieve far higher performance, offering much more ILP with much lower silicon and power costs.

Since the VLIW compiler presents many operations to be issued at once, it usually is asked to bundle them into a single, very long, instruction word—hence the description VLIW. Ideally, a VLIW-style architecture should be developed in the presence of a malleable, VLIW-targeting compiler, one that can generate code for candidate architectures. Several VLIW architectures have been developed this way, and have been quite successful. This is a sound design practice, since the compiler is doing a major part of the job done by the hardware in other processors.

Originally dismissed as impractical, VLIW is now the technology of choice in countless embedded processor applications.

While any real-world architecture is a highly complex object, the basic block diagram of a VLIW is relatively simple, consisting of a set of functional units, registers, and paths to memory. Thus one important VLIW advantage is that it gives architecture designers the ability to make changes and to scale the architecture to different sizes. The hardest part of doing that with an ordinary architecture is dealing with the implied changes in the control unit. With a VLIW, that part is usually done by changing tables in the compiler. Figure 2 is a simplified block diagram of a VLIW that can issue eight operations per cycle.

This very brief description of the VLIW style will hopefully leave more questions than answers in the reader's mind, since there is a lot of substance behind the ideas we've just discussed. Moreover, it does not discuss the compilation process—which is the core of the VLIW concept—although Josh Fisher's narrative (the second half of the article) touches on it lightly. VLIW and its compilation are covered in much greater depth in our book, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools* [3].



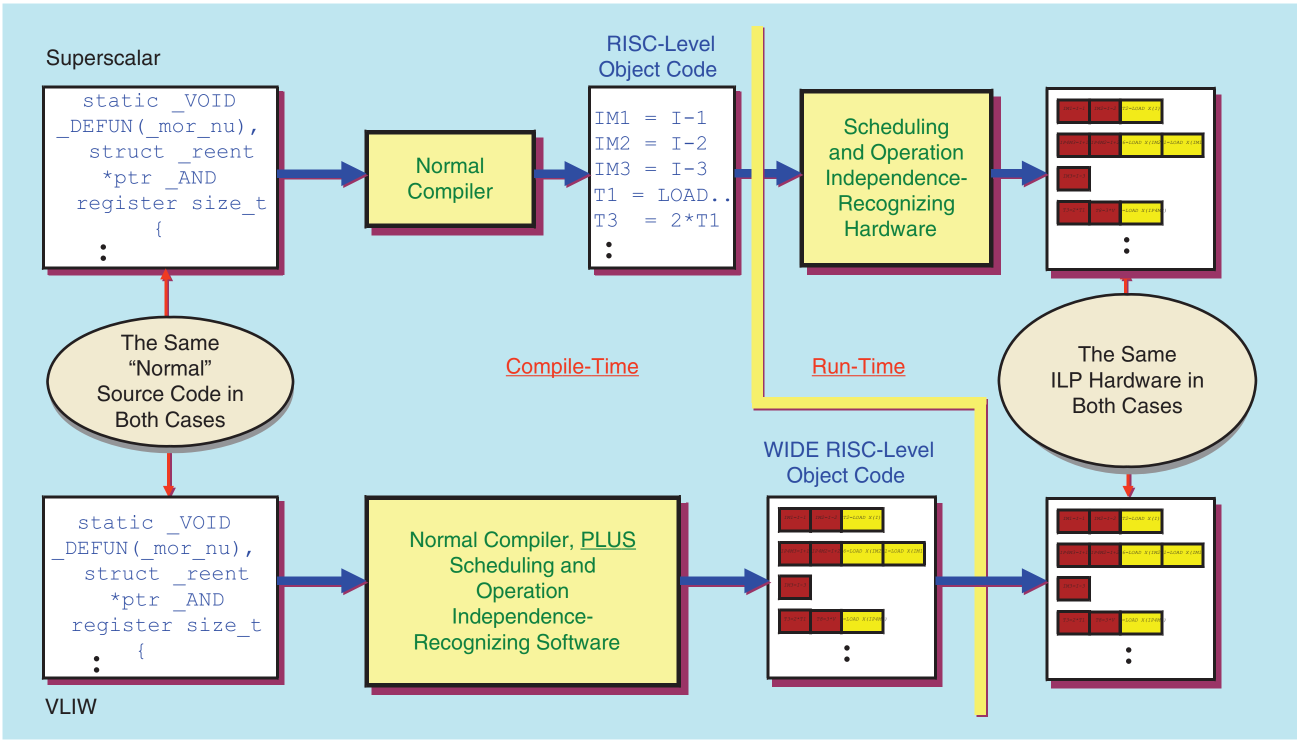
VLIW in Use: Commercial Successes

In high-performance embedded computing, VLIW has become the architectural style of choice. The economics of embedded computing turn the intrinsic area and power efficiency of VLIW into compelling arguments. Its general-purpose, compiler-driven programmability enables even nonexperts to approach the processor's performance peak. At the same time, one of main shortcomings of VLIW, its lack of binary compatibility, is rarely an issue in embedded computing.

Beginnings

The first commercial VLIW-style architectures were scientific minisupercomputers from two start-up companies in the mid-1980s: Multiflow Computer and Cydrome. Josh Fisher started Multiflow in 1984 with two members of his VLIW group at Yale University. B. Ramakrishna (Bob) Rau (another VLIW pioneer) and colleagues started Cydrome at about the same time. Although Multiflow in particular had an impressive degree of commercial success at the start, both companies failed after five or six years, along with the rest of the minisupercomputer industry. The market advantages of microprocessors were too great for their competitors to overcome, and there was neither enough silicon on a chip for those architectures, nor access to it.

In the 25 years since Multiflow and Cydrome, dozens of manufacturers have designed and sold VLIW processors. The greatest success has been in embedded applications, with VLIWs dominating digital video and cellular



Superscalar
RISC-Level Object Code
static _VOID _DEFUN(_mor_nu), struct _reent *ptr _AND register size_t {
Normal Compiler
IM1 = I-1
IM2 = I-2
IM3 = I-3
T1 = LOAD..
T3 = 2*T1
Scheduling and Operation Independence-Recognizing Hardware
The Same "Normal" Source Code in Both Cases
Compile-Time
Run-Time
The Same ILP Hardware in Both Cases
WIDE RISC-Level Object Code
Normal Compiler, PLUS Scheduling and Operation Independence-Recognizing Software
VLIW

FIGURE 1: VLIW versus superscalar: The execution hardware of both methodologies is instruction-level parallel and approximately equivalent. The distinction is in the scheduling: Superscalars perform this in hardware, whereas VLIWs let the compiler rearrange the code so that the hardware can execute it without changing it. (From [3].)

base stations and playing a major role in media-intensive cell phones and personal digital assistants (PDAs). Many other embedded applications are turning toward VLIW, especially as media-rich applications begin to dominate computing.

Digital Video

In digital video, STMicroelectronics (STM) has shipped over 40 million systems-on-chip (SoCs) containing a VLIW processor from the ST200 family. In fact, since many of these SoCs contain multiple ST200s (the STi7200 contains four ST231s), they have shipped in excess of 70 million of these VLIW processors. These have been used primarily to do audio and video functions in set-top boxes, inside products such as satellite receivers, digital video recorders, cable receivers, televisions, automobile multimedia, and more.

Hewlett-Packard developed the original ST200 architecture in collaboration with STM, in Josh Fisher's laboratory in Cambridge, Massachusetts, with Paolo Faraboschi as the lead architect. Fred Homewood of STM led the microarchitecture, and STM did the hardware design and contributed to the architectural effort. The ST200 was a true collaboration of teams with expertise in compilers, architecture and hardware elements, and software tools.

STM has stressed that the success of this technology has been a result of all of the claimed virtues of VLIWs, for example:

- The processors carry out their computations in about the same number of cycles as earlier digital signal processor (DSP) circuits, but because of the simplicity of the hardware, they cycle at more than two times the frequency.
- The ease of hardware implementation has made subsequent generations of the processor far easier to build than expected. Originally, the designers thought they would need a custom data-path design with custom register files. But they discovered that their tools were capable of doing the layout with "preplacement." Subsequently they were able to drop the custom register file (the ST231 is fully synthesized, apart from the RAM) and use automatic layout. Recently the tools have improved further, yielding a significantly smaller core and improved frequency in the latest designs—so much so that the designers wondered if they'd omitted some part of the core.
- The very clean architecture—that is, instructions are either executed or not, the CPU has a well-defined state, and so forth—has made it easy to port operating systems, and STM has found it easy to get good performance without major source code changes. The ST200 family has, surprisingly, also been used as a Linux host, even though it was designed as a slave processor.

Wireless

According to Ray Simar, Texas Instruments' VLIW architect, "Four out of five base stations in use today use VLIW technology; with the newer designs using multiple VLIW cores per SoC device. At the same time wireless handsets are being driven to VLIW solutions as the demand for video and image processing increases."

There are an estimated 3–4 billion mobile subscribers worldwide, and many analysts say that the percentage of new handsets that are "smart phones" will approach 50%—and this does not include ordinary handsets that do video and image processing. Given that Texas Instruments is the dominant force in base station and handset electronics, the analysts' prediction augurs well for the future of VLIW processors in wireless. (Another article in this issue goes into TI's use of VLIW technology in greater detail.) Moreover, TI uses VLIW technology in many other applications beyond wireless.

Printing, Scanning, Copying, and Faxing

Hewlett-Packard will introduce consumer inkjet multifunction products that incorporate the ST231 VLIW microprocessor in 2009. This is from the same processor family that has had such great success in digital video.

The following is especially interesting because it addresses the migration from dedicated, fixed-function hardware to the far more flexible use of a real processor. The ST231's performance and ease-of-use advantages strongly influenced HP's decision to use the microprocessor in its multifunction devices. Patrick Chase, imaging systems architect, HP Imaging and Printing Group, explains HP's choice this way:

> HP imaging products rely heavily upon custom ASICs [application-specific integrated circuits] for core imaging functionality. These ASICs traditionally rely on fixed-function hardware for most data processing, with a host CPU for control. In order to increase both the range of products served by each ASIC design and its lifespan in HP's product portfolio, we identified a need to shift the emphasis of our architectures in the direction of more programmable computing resources. We therefore identified an initial set of features to be implemented by a high-performance imaging processor and established two key goals for our investigation: First, that the incremental cost of the processor not be significantly higher than the cost of fixed-function hardware implementing the initial set of candidate features. Second, that the processor be programmable using common tools and languages such as C/C++.
>
> We evaluated a range of alternatives, including traditional DSPs and more general-purpose processors in both SIMD (single instruction, multiple data) and VLIW idioms. Each alternative was benchmarked across a suite of applications, which were selected both to represent HP's intended workloads and to exhibit a wide range of dataflow characteristics and therefore a wide range of applicable optimization techniques. Subject-area experts separately evaluated toolset quality and ease of programming, ease of integration, and vendor capability.
>
> HP chose the ST-231 VLIW CPU because it performed uniformly well across all of the benchmarked applications, because it integrated well into our existing ASIC architectures, and because its development environment was most similar to conventional CPUs for the embedded space. It has met all performance and capability expectations, and we believe that its flexibility (particularly its ability to function as a host processor) will continue to enable new features and product categories.

First Implementation as a Microprocessor

As chip circuit densities started to become sufficient for VLIWs, Philips Laboratories began to design the LIFE-1 VLIW processor in 1987. After several years of internal development, Philips introduced the processor commercially in 1996 as the TM-1, later renamed the TriMedia TM1000. Although Philips initially found it hard to

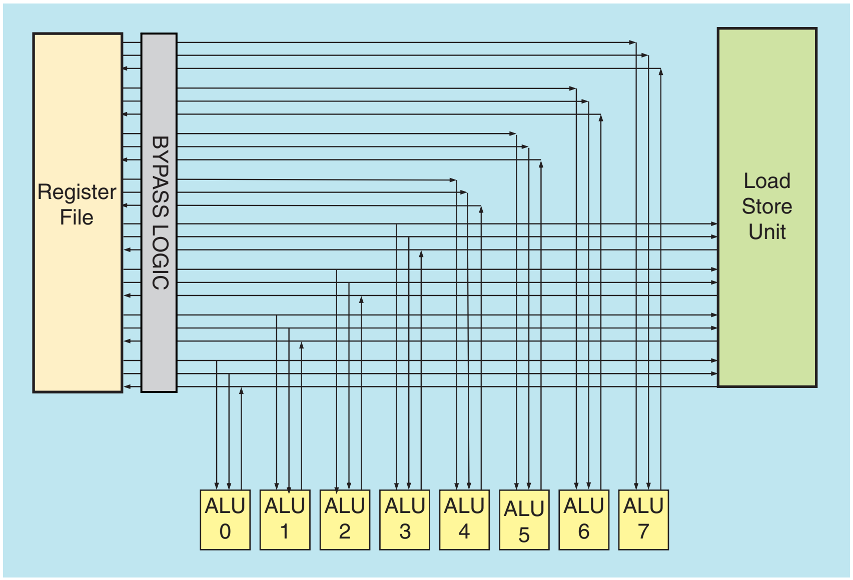


FIGURE 2: The basic connections of an eight-unit VLIW data path, where eight arithmetic logic units (ALUs) are connected to a single multiported register file. Many details of a real data path are omitted in this schematic (such as pipeline staging registers, control paths, and so on), and it is not representative of the physical layout. Because of the obvious wire congestion in the center, the register files are often partitioned, with a subset of functional units getting fastest access to a corresponding subset of the registers. (From [3].)

market the TriMedia, the company stayed with it, and eventually it became a major product line for Philips. Today the descendants of the TriMedia form the high-performance heart of NXP's media systems-on-chip (NXP is the spin-off of the former Philips Semiconductor Division). The NXP VLIW platforms are popular in digital TVs and other digital video products, as well as in cellular handsets and other applications.

Digital Cameras

Another noteworthy VLIW processor is the FR-V family from Fujitsu. The FR-V is designed for the usual media processing applications but is notable for its use in Leica digital single-lens reflex (SLR) cameras (and is rumored to be used by several other makers of high-end digital SLRs). It is offered as a platform in many digital media applications, although reliable information about its use is scarce. Some of the implementations of the FR-V are eight-unit-wide VLIWs.

Other Embedded VLIW Applications

The applications just described are probably the most important, though given the closely guarded nature of embedded core development, it is impossible to be sure. It is known that VLIW processors have been used successfully in products such as hearing aids, graphics boards, and network communications devices.

Virtually every one of the large semiconductor companies has had a VLIW processor core among its products at some point in the past decade, as have many "fabless" start-ups.

"Fabless" (Intellectual-Property-Based) VLIW Designs

Fabless semiconductor suppliers design and market semiconductor chips but do not actually manufacture them—they outsource that operation to semiconductor foundries, or "fabs." Many fabless start-ups and more established companies are offering VLIW designs for systems and semiconductor companies to utilize. Some notable ones are Jazz (from Improv Systems), CEVA, Silicon Hive, and Tensilica.

VLIW in Servers, Desktops, and Notebooks

In the second half of the 1990s, there were two prominent attempts to use VLIW technology to supply processors for general use. One was by the company Transmeta, which used a VLIW processor to emulate an Intel x86. While this was technically successful and had excellent cost/performance characteristics, the perceived advantages were not enough to overcome Intel and AMD's strength in the marketplace.

The second attempt came from Intel itself, in the form of the Itanium processor family. The underlying "EPIC" design style borrows many characteristics from VLIWs, while adding many new complex features of its own. The original versions of the Itanium came out of Hewlett-Packard Labs. After Cydrome went out of business, Bob Rau joined HP Labs and worked on the project that became the Intel Itanium (Josh Fisher did as well but moved into embedded computing before long). Rau strongly influenced that project, and the Itanium contains many artifacts reminiscent of Cydrome's technology. The Itanium, while it is now the fourth most popular server architecture, has not come close to meeting industry projections for its volume.

Josh Fisher's Narrative

[*The rest of this article is a personal reminiscence by coauthor Josh Fisher about the invention of VLIW.*]

VLIWs came from my work as a graduate student at the Courant Institute of Mathematical Sciences at New York University. It is useful to see that work as the combination of two things:

1) solving a difficult, practical, nuts-and-bolts problem
2) thinking creatively about the implications of what I invented, without being bound to the domain of the problem.

The lesson is that it is all too easy to stop at the first step, understandably satisfied with the quality of a good solution. I was fortunate, though, that I was in one of the rare places where, in the late 1970s, architecture and compiling met. Besides, I had to produce a Ph.D. thesis [4], one of the few circumstances in which big thoughts from the young and inexperienced are tolerated, even encouraged. Being able to free my mind, and put my solution into a bigger and more valuable context, was a luxury even then, and I am lucky to have had that opportunity.

The Hard Problem and Trace Scheduling

I was trying to solve a problem (automatically producing horizontal microcode from sequential source code) that arose in the hardware project I had joined. In the pre-Internet age, it was hard to find the research community for this, and when I found it, it was a small group of mostly hardware engineering researchers. I also first thought of this as a hardware problem, analogous to laying out our hundreds of medium-scale integrated chips on boards. In the retrospective in this issue, I explain the analogy, which is very persuasive but completely wrong.

In any event, most of the work published on this problem showed little recognition of either compiler technology or computational complexity. Thus there were many beneficial compiler mechanisms that weren't being used, and I could see that the proposed "optimal" solutions were certainly not that (for example, they would have proved that $p = np$ as a side effect).

After about a year of looking at what the process was like by hand, I proposed a solution to that problem that I thought was far better than anything seen so far (and probably was). It appeared to work well. I called it "trace scheduling." It, or something that resembles it, is now common in compilers for high-performance architectures. The advantage of trace scheduling is that it circumvents the limitation of looking for instruction-level parallelism (ILP) only within sections of code without branches in or out, except at the top and bottom of the section (these are called basic blocks). The problem is that very little ILP exists within basic blocks, and no method before trace scheduling was effective at finding more than a little ILP.

The basic idea in trace scheduling is to select code originating in a large region (typically many basic blocks) before scheduling. You then schedule operations from the entire region as if it were one big basic block (which often contains a lot of ILP), possibly adding extra operations so that the semantics of the program remain unchanged, despite the violence done to the flow of control. As you can imagine, this is a very complex process. Figure 3 illustrates the selection of a large region of code.

Instruction-Level Parallelism as a Concept

But here's step 2: Compacting horizontal microcode was a fine application, and I was very happy and proud to have come up with a good method for doing so. But the real payoff was that I started wondering why we shouldn't build general-purpose architectures, rather than arcane microcode, that took advantage of what seemed like a potentially very large performance gain. My new compiler technique, if it turned out to be as powerful as I hoped, might enable that.

I abstracted the kind of parallelism that horizontal microcode presented. I coined the term "instruction-level parallelism" and felt that it was a good description for what was already found in lots of disparate kinds of processors: superscalars (not yet called that), DSPs, horizontal microcode, and dataflow processors, at least. No one was thinking about that kind of parallelism as a subject before, though there was an active research and/or development community around each type of processor.

At Yale

Shortly after I started work as a professor of computer science at Yale, I was hired as a consultant by GE R&D to try to apply my techniques to one of the DSP machines they were using, a Floating Point Systems AP-120. I saw very clearly that my compiler technique, indeed most complex compiler tasks, would have a very difficult time dealing with a target like that. It just wasn't compiler friendly and, like all similar systems, it could get good performance only on tight inner loops of hand code. What was needed was something very different—something with a similar style of hardware, but in a real system processor. I called these new processors VLIWs, an abbreviation for a name that is quite visual and conveys the idea succinctly. VLIWs don't *have* to have long instruction words, but most do.

To me, VLIWs were

- architectures that were structured like horizontal microcode but
- were clean and general-purpose in nature: designed as the target of a compiler that found and specified the ILP and as the processor of a real computer system—in fact; they could even run an OS.

There had been many prior processors that met the first criterion above—I've already mentioned a few—leading some to question of whether VLIWs didn't exist for a long time before 1980 (you can go back to Alan Turing's ACE report of 1946 to find some static ILP [5]). But that misses the point of what a VLIW is.

I thought these proposed processors might become the general-purpose processors that offered the most bang for the buck. I certainly didn't think the hardware would be next to impossible to build, as most people seemed to when they heard about them—quite the opposite. Using them well would be another matter, but I knew that they would be relatively straightforward to build compared to other complex architectures—easier, not harder.

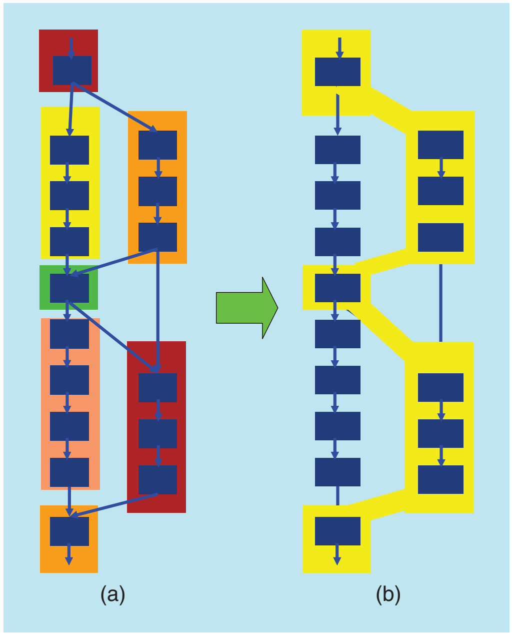


FIGURE 3: From basic blocks to traces. It was clear that compacting only (a) basic blocks was exposing too few operations to fill all usable slots of instruction-level parallel hardware. Trace scheduling, which considers larger regions, such as (b) traces, opened up a new field of research. (From [3].)

The Bulldog Compiler

By 1980, I had exhausted all objections in my mind and thought there was a good chance that VLIWs would make excellent general-purpose processors. So I started a project to build a compiler.

The compiler succeeded in doing quality trace scheduling, as described beautifully in John Ellis' book *Bulldog: A Compiler for VLIW Architectures* [6]. Ellis had re-engineered the compiler and experimented with it and then won the ACM Best Thesis Award. His experiments convinced me it was time to try to build a VLIW, as targeted by that compiler.

A VERY LONG WORD

This single instruction was issued in a single cycle on the Trace-28:

```
cl0 ialu0e st.64 sb0.r4,r3,zero
cl0 falu0 mpy.f64 lfb.r12,r4,r10
cl0 ialu01 dld.64 fb0.r14,r3,17#128
cl0 ialu11 cgt.s32 li1bb.r3,r32,6#3
cl0 falu1 mpy.f64 lfb.r46,r34,r42
cl1 ialu0e st.64 sb1.r4,r2,6#8
cl1 falu0 mpy.f64 lfb.r12,r4,r10
cl1 ialu01 dld.64 fb1.r14,r2,17#136
cl1 ialu11 cgt.s32 li1bb.r5,r32,6#2
cl1 falu1 mpy.f64 lfb.r46,r34,r42
cl2 ialu0e dld.64 fb2.r12,r2,17#144
cl2 falu0 mpy.f64 lfb.r0,r4,r10
cl2 ialu01 st.64 sb2.r4,r2,6#16
cl2 ialu11 cgt.s32 li1bb.r5,r32,6#1
cl2 falu1 mpy.f64 lfb.r46,r34,r42
cl3 ialu0e dld.64 fb3.r12,r2,17#152
cl3 falu0 mpy.f64 lfb.r0,r4,r10
cl3 ialu01 st.64 sb3.r4,r2,6#24
cl3 ialu11 cgt.s32 li1bb.r5,r32,zero
cl3 falu1 mpy.f64 lfb.r46,r34,r42
cl0 br true and r3 L9?3
cl1 br true and r5 L10?3
cl2 br true and r5 L11?3
```

My Yale team and I then started a hardware/architecture project, called *ELI*, as outlined in a 1983 ISCA paper reprinted in this issue [1]. At that point, we also arranged visits from all of the manufacturers of computers we could convince to visit us, in the hope of doing joint development. Our unusual technology was promising enough that most of them came, but they were generally unconvinced of the potential of VLIWs. I think most of them thought we were smart, but a little crazy and unrealistic. We might have been crazy, but I don't think there was anything unrealistic about building a VLIW, especially starting from the Bulldog compiler. It was just very different and thus, somehow, must have seemed impossible to our visitors.

Multiflow

In 1984, John O'Donnell, John Ruttenberg (both of whom had worked for me at Yale), and I gave up on the joint development route and started Multiflow Computer. The history of Multiflow has been well documented (the Wikipedia entry is a summary I wrote that I think captures the history best—at least the March 2009 version does), and two other articles in this issue and the Summer 2009 issue ("The VLIW Architecture of Joseph A. Fisher–Part II") discuss the development of the Multiflow Trace. We really surprised a lot of people by delivering a system that was easier to use than those coming from the major manufacturers of large scientific systems—our customers told us so—and our performance/cost on several major applications was as good as or better than anyone else's. I believe that Multiflow's computers were the most novel ever to be broadly sold, programmed, and used as if they were normal computers (other novel computers either required novel programming or represented more incremental steps beyond existing computers).

After Multiflow, many people still argued against the desirability of VLIWs, but no one ever argued against their practicality, even though almost no one thought they were practical before we delivered one. Multiflow's biggest system, the Trace-28, issued 28 operations per cycle. That was more than almost anyone could use—the sweet spot seemed, and seems, to be about eight per cycle—but it was impressive when it clicked.

By way of example, "A Very Long Word" shows a noncontrived, compiler-generated single instruction, containing 24 operations, from the popular linear algebra routine DAXPY. Although many embedded applications (for example, signal and media processing) contain this much and more ILP, we know of no one who has proposed such a wide CPU for embedded processors. This might make sense in the future; only time will tell.

Why Didn't Anyone Do This Before?

I have found it really instructive to ponder this question. I don't have the space here, but the book *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools* [3] describes my pondering at length.

The short story is that ILP was being used in the 1960s and 1970s by three different communities:

- the horizontal microcode community—usually hardware engineers
- the DSP community—usually experts in a scientific application domain
- what we would today call the supercomputer community.

In the last case, there was a great deal of computer science and software education among those trying to speed up scalar processing (using the superscalar hardware that the biggest computers of the time had). They were genuinely sophisticated about performance and hoped to get a lot of ILP (again, without having a term for it). It is remarkable to read:

> If this approach is carried further in the coming years, processors may contain tens or hundreds of independent arithmetic units. The problem of efficient utilization of such a processor then becomes a major problem. Ultimately, the problem of efficiency will fall on the compilers and the compiler writers. [7]

The compiler experts, too, were aware of the potential, but saw the limitation: "In particular, stores must be inhibited while fetches are in progress, and vice versa. This feature makes it infeasible to extend this design to include parallelism on a very large scale" [8]. Compilers did do the job of finding ILP in short, straight-line pieces of code as early as 1964. Indeed, a discussion was reported in *Communications of the ACM* in 1964—involving such computer pioneers as Alan Perlis, John Backus, Robert Floyd, and Maurice Wilkes—after the presentation of a paper on the subject [9]. The summary of the discussion was that

> The paper did present an ingenious method of rearranging the linear structure of a calculation and of departing from a linear structure and carrying out operations in parallel. The demanding bookkeeping required to assure that the proper operands enter into the calculation seemed, at least at first glance, to have been solved. [9]

Unfortunately, the movement toward general-purpose ILP soon died out in this community. Anecdotally, it is believed that it was killed by a series of experiments to measure the maximum ILP available in programs (the first of the so-called MaxPar experiments), most notably experiments done by Riseman and Foster [10]. Ironically, they found nearly unbounded amounts of ILP, but the straightforward hardware-oriented methods they imagined would be able to exploit it required impractical quantities of hardware because the hardware would have to be continually replicated to deal with branches. The community took this to mean that there was not much ILP available rather than that it was there, but they were not looking at the problem the right way.

They really missed the possibility of software techniques, such as trace scheduling and its descendants, and sophisticated software pipelining, which would deliver a good part of the available ILP. These techniques unlocked the available ILP and made practical VLIWs with eight or more simultaneously issued operations a reality.

Today the performance, cost, and ease of use of VLIW is unquestioned, and one by one the compute-intensive applications are taking advantage of these strengths. It is easy to predict that embedded processors will find their way into more and places, places we cannot today imagine, often with very stringent performance, size, power, and cost requirements. VLIW, especially with its track record, appears to be a very good match for such applications.

References

[1] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. 10th Annu. Int. Symp. Computer Architecture*, June 1983, pp. 140–150.

[2] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A technology platform for customizable VLIW embedded processing," in *Proc. 27th Annu. Int. Symp. Computer Architecture*, June 2000, pp. 203–213.

[3] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Tools and Compilers.* San Francisco, CA: Morgan Kaufmann, 2005.

[4] J. A. Fisher, "The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources," Ph.D. dissertation, Univ., New York, Tech. Rep. COO-3077-161, Oct. 1979.

[5] B. E. Carpenter and R. W. Duran, Eds., *A. M. Turing's ACE Report of 1946 and Other Papers*. Cambridge, MA: MIT Press, 1986.

[6] J. R. Ellis, "Bulldog: A compiler for VLIW architectures," Ph.D. dissertation, Yale Univ., New Haven, 1985.

[7] H. S. Stone, "One-pass compilation of arithmetic expressions for a parallel processor," *Commun. ACM*, vol. 10, no. 4, pp. 220–223, Apr. 1967.

[8] J. T. Schwartz, "Large parallel computers," *J. ACM*, vol. 13, no. 1, pp. 25–32, Jan. 1966.

[9] R. W. Allard, K. A. Wolf, and R. A. Zemlin, "Some effects of the 6600 computer on language structures," *Commun. ACM*, vol. 7, no. 2, pp. 112–119, Feb. 1964.

[10] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. Comput.*, vol. 21, no. 12, pp. 1405–1411, Dec. 1972.

About the Authors

Joseph A. Fisher (josh@vliw.org) is a Hewlett-Packard senior fellow (retired). While a earning his Ph.D. at the Courant Institute of New York University (1979), and as a member of the computer science faculty at Yale University (1979–1984) he developed many of the basic techniques of compiler-controlled instruction-level parallelism and coined several fundamental terms in the field, including VLIW and the term instruction-level parallelism itself. At Yale, he won a National Science Foundation President's Young Investigator Award. In 1984, he cofounded Multiflow Computer, which built VLIW minisupercomputers and changed the way the industry perceived the practicality of VLIW architectures. In 1987, he won the Eli Whitney Connecticut Entrepreneur of the Year Award. He joined HP Labs in 1990 and in 1994 established HP Labs Cambridge to engage in research in embedded VLIW and related topics. Under his leadership, HP researchers codeveloped with STMicroelectronics the Lx family of VLIW processors, now used broadly in digital media applications. In 2003, he received the IEEE/ACM Eckert-Mauchly award.

Paolo Faraboschi (paolo@vliw.org) is a distinguished technologist in the Exascale Computing Lab at HP Labs and leads the Barcelona Research Office, whose focus is system-level modeling and simulation. He was the technical lead of the custom-fit processors effort at HP Labs Cambridge and the principal architect of the Lx/ST200 family of VLIW cores. His research interest is the boundary of hardware and software: highly parallel systems, VLIW architectures, compilers, and embedded processors. Before joining HP in 1994, he received a Ph.D. (1993) and M.S. (1989) in electrical engineering from the University of Genoa, Italy. He is an active member of the computer architecture community, and he was recently program cochair for MICRO41 (2008).

Cliff Young (cliff@vliw.org) is a research scientist and member of the technical staff at D.E. Shaw Research. He works on projects involving special-purpose, high-performance computers for computational biochemistry, including Anton, a special-purpose supercomputer for molecular dynamics, which began operation in 2008. Previously, he was a member of technical staff at Bell Laboratories in Murray Hill, New Jersey. He received A.B., S.M., and Ph.D. degrees in computer science from Harvard University in 1989, 1995, and 1998, respectively.