A Computer with Its Head Cut Off

Brian Hayes

When Silicon Valley wants to look good, it measures itself against Detroit. The comparison goes like this: If automotive technology had kept pace with computer technology over the past few decades, you would now be driving a V-32 instead of a V-8, and it would have a top speed of 10,000 miles per hour. Or you could have an economy car that weighs 30 pounds and gets a thousand miles to a gallon of gas. In either case the sticker price of a new car would be less than $50.

In response to all this goading, Detroit grumbles: Yes, but would you really want to drive a car that crashes twice a day?

Although weighing cars against computers is not quite fair to either industry, the comparison does illuminate the extraordinary record of recent progress in microelectronics and computer technology. The performance of computer systems has been doubling every two years for 20 years or more, so that a computer today is a thousand times more powerful than one built in the mid-1970s. It is an impressive record, perhaps unprecedented in the history of human ingenuity. And yet looking at that steeply rising performance curve prompts a nervous question: Where will the next doubling come from, and the doubling after that?

Here I want to describe one candidate architecture for the next generation of high-performance computer systems. It goes by the very long and unwieldy name of very-long-instruction-word architecture, or VLIW. It is not a new idea, but it is newly fashionable, partly because of reports that two major manufacturers in the computer industry, Intel and Hewlett-Packard, plan to collaborate on VLIW designs. I would not presume to judge whether VLIW is the best bet for the future of computing, but it is being considered seriously enough to merit wider discussion.

Computer Architecture

Much of the performance increase in computer systems has come from advances in the fabrication of semiconductor devices. Individual transistors switch faster, and so circuits can be run at a higher speed. But improvements in the chipmaker's art cannot account for all of the performance gains in the past 20 years. Another contribution has come from changes in computer architecture—the way the basic building blocks of the computer are organized and interconnected.

The central processing unit of a computer has two main subsystems. On the one hand there is the data-manipulating machinery, which includes adders, multipliers, comparators and the like, as well as registers for the temporary storage of data. On the other hand is the instruction-manipulating machinery, which decodes and interprets the instructions that make up a computer program. These two parts of the machine are called the data path and the control path. A third component, the system clock, acts like a metronome beating out a steady rhythm to synchronize activity throughout the processor.

The evolution of computer architecture can be viewed as a kind of dialectical contest between the data path and the control path, with each section of the processor competing for its share of resources. Ultimately the competition is for silicon "real estate": If the control apparatus grows larger, there is less room for the data path, and vice versa.

In a very simple computer it is easy to see exactly how the control path steers computations in the data path. Suppose the data path includes function units for just four operations, such as addition, subtraction, multiplication and division. A program instruction can specify one of these operations by activating the appropriate function unit and turning off all the rest. The most efficient encoding would set aside two binary digits (bits) to specify the operation; the two bits have four possible values (00, 01, 10 and 11), which can be matched up in any convenient way with the four arithmetic operations. A circuit called a two-into-four decoder takes the two bits as input and activates one of four output lines accordingly.

Early computer designers labored to pack as much capability as possible into each instruction and thus gave a major share of resources to the control path. There were at least three reasons for adopting this strategy. First, more-powerful instructions would make life easier for the programmer. Second, they promised higher performance: If each instruction could accomplish more, then the computer could complete a task with fewer instructions. The third reason is related to the sec-

Brian Hayes is a former editor of American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@mercury.interpath.net.*

ond: Programs made up of fewer instructions take up less room in memory. The principal drawback of the strategy was the grotesque complexity of the decoder and control unit needed to implement very complex instructions.

A solution to this problem was devised remarkably early in the history of electronic digital computing. In 1951 Maurice V. Wilkes of the University of Cambridge proposed the idea of microprogramming, which turns the control unit into a miniature computer within the computer. Instead of building a tangled web of hardware gates and latches to translate instruction words into control actions, the designer creates a sequence of microinstructions, or microcode, whose bits correspond more or less directly to the necessary control signals. The microcode is stored in a special memory array within the processor. The original program instruction—now called a macroinstruction—becomes a pointer into this memory, selecting which sequence of microinstructions is to be executed.

Microprogramming was slow to catch on until it was adopted by IBM in the System/360 series of mainframe computers in the 1960s. Thereafter it became very popular, and by 1980 virtually all computers were microprogrammed. Two styles of microcode evolved. Horizontal microprograms are short and fat; in the limiting case the microprogram for each macroinstruction consists of a single wide microinstruction with one bit for each control signal in the processor. Vertical microprograms are tall and skinny, with narrower microinstructions that need some degree of decoding before they can be supplied to the control circuitry.

The RISC Rebellion

In the late 1970s a rebellion was launched against complex microprogrammed architectures. The ringleaders were John Cocke of IBM, David A. Patterson of the University of California at Berkeley and John L. Hennessy of Stanford University. They found that the complicated instructions designers had struggled to build into the hardware were actually slowing the system down. Some of these instructions required dozens of clock cycles to complete their execution; moreover, the intricacy of the tasks to be accomplished within each cycle set a limit on the clock rate. Computers, it seemed, could be made more powerful by removing some of their most sophisticated features. Patterson named the new stripped-down architecture the Reduced Instruction Set Computer, or RISC.

As a rule, RISC chips have no microcode; the instruction format is simple enough for direct hardware decoding. The simplified control path leaves more room for the data path and also allows higher clock rates. Perhaps most important, RISC processors attempt to maintain an average execution rate of one instruction per clock cycle.

The invention of RISC architecture is often seen as a reaction against microprogramming, but, paradoxically, it can also be viewed as the apotheosis of microcode. In a sense, what has been removed from a RISC is not the microcode but the macrocode; the instructions recognized by the processor are closer to vertical microinstructions than to traditional macroinstructions. Think of a RISC processor as a lobotomized computer, one that is missing the higher functions needed for decoding and interpreting macroinstructions.

What about ease of programming? Although writing software directly in the instruction set of a RISC processor would be quite difficult, this drawback has ceased to be an issue because almost all programming is now done in higher-level languages. Only the compiler—the program that translates the higher-level notation into the native language of the processor—has to know about the primitive instructions recognized by RISC hardware.

RISC chips have come to totally dominate the market for scientific workstation computers. They are also starting to show up in mass-market machines, notably those based on the PowerPC chip developed by IBM, Motorola and Apple Computer.

The Superscalar Solution

A RISC chip that completes an instruction in every clock cycle would seem to represent a limiting case. How could it be made to compute any faster, except by boosting the clock rate? The answer is that it must complete multiple instructions per cycle.

The most direct way of extending RISC principles to gain still greater speed is called a superscalar architecture. The basic idea is to build a processor whose data path includes multiple function units—several adders, say, and a couple of multipliers—and then modify the control path to keep all the units busy as much as possible. Thus the processor might be performing an addition, a multiplication and a numeric comparison all at the

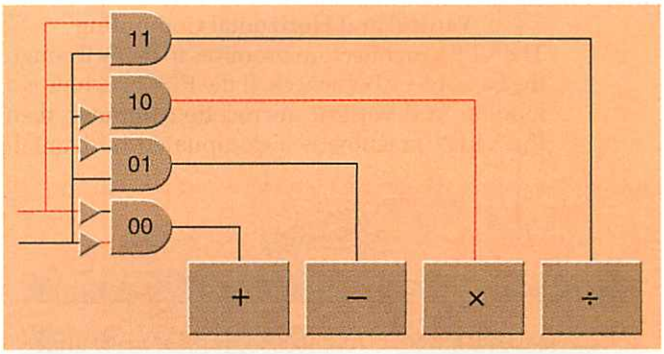


Figure 1. Hard-wired two-into-four decoder controls four function units.

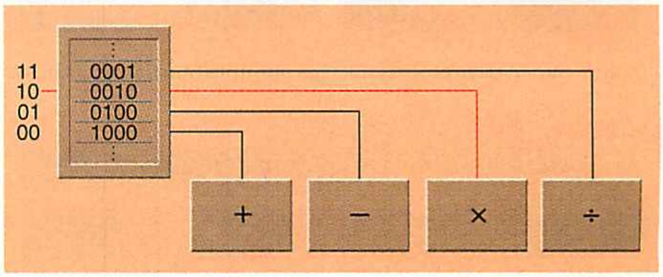


Figure 2. Basic structure of a microprogrammed control unit.

same time. To achieve this kind of parallelism, the control unit must be able to analyze a sequence of instructions and decide when some of them can be issued simultaneously. For example, if the operation $c = a + b$ is followed by $d = b \times 2$, there is no obvious reason the second instruction cannot be started at the same time as the first. But conflicts often block parallel execution. If the second operation reads $d = c \times 2$, it cannot be executed until the first instruction completes, because only then will the value of c be known. In a superscalar chip the control path must detect such dependencies and schedule the instructions accordingly.

Several superscalar processors are now in successful production. The current state of the art is the Alpha 21164 chip from Digital Equipment Corporation, which can execute four arithmetic instructions at once. It runs at a clock rate of 300 megahertz and is said to have a peak throughput of more than a billion instructions per second.

The potential of superscalar techniques has surely not been exhausted, and yet it appears difficult to achieve much more than fourfold parallelism. For one thing, the complexity of the instruction-scheduling circuits increases as the square of the number of instructions being scheduled, so that the control path would soon become unwieldy. There may also be a more fundamental limit. Scheduling algorithms view a program as a set of "basic blocks," which are sequences of instructions that are always executed together; the program cannot branch into a basic block except at the top and it cannot branch out except at the bottom. Superscalar processors look mainly at instructions within the same basic block as candidates for concurrent execution. But the potential parallelism in basic blocks is severely limited. A typical basic block is only about a dozen instructions long, and it cannot be sped up by more than a factor of two or three.

Vertical and Horizontal Computing

The VLIW architecture promises to break through the basic-block bottleneck. If the RISC machine is a lobotomized vertical-microcode computer, then the VLIW machine is a decapitated horizontal-microcode computer. Even the rudimentary instruction-decoding mechanism of a typical RISC has been removed. A VLIW instruction is a long string of bits—from a few hundred bits to a thousand or more—that directly controls an entire complement of function units. Each bit turns on or off a particular element of the data path. Parallel execution is arranged simply by setting the instruction bits that activate several function units at the same time. The hardware does no instruction scheduling; all decisions about when and where instructions are executed must be made when the program is compiled.

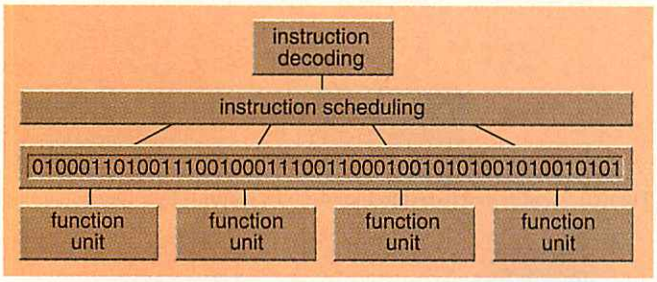


Figure 3. Superscalar processor keeps multiple function units busy.

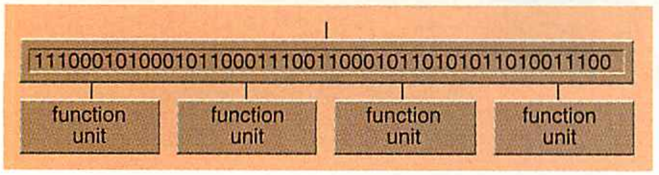


Figure 4. The bits of a VLIW control the function units directly.

The enabling technology for VLIW computers is not an innovation in hardware engineering but rather a set of compiler techniques. The compiler must decide which operations can be dispatched to the various function units at any given time and pack them together into one long instruction word. To find worthwhile amounts of parallelism, it must look beyond the boundaries of basic blocks for instructions to schedule together. This is a delicate task: When the scope of the scheduling algorithm extends beyond branch points and merge points in the program text, the compiler cannot even know which instructions will be executed and which will be bypassed on any given pass through the program.

An algorithm for VLIW compilation, called trace scheduling, was devised in about 1980 by Joseph A. Fisher, who was then at Yale University. The idea is to select paths through the program that could conceivably be followed during a specific run, and ideally to start with the likeliest paths. Each such path, called a program trace, can include multiple basic blocks, with branches and merges, but it cannot have any loops (branches backward to an earlier point on the same trace). The scheduler deals with the entire trace as if it were a single basic block, moving operations around as necessary to pack them into the minimum number of long instruction words. The only constraints on the scheduling are data dependencies (where one instruction needs the result of another) and a requirement that branch instructions not be executed out of order.

Moving instructions between basic blocks can alter the meaning of a program. For example, suppose the addition $c = c + 1$ is performed just above a branch instruction, so that the incremented value of c is available in either arm of the branch. If the scheduling algorithm moves the addition operation below the branch, c will have the wrong value in the arm of the branch not included in the trace. To avoid this error the $c = c + 1$ operation must be duplicated and inserted in both arms of the branch. Fisher formulated a complete set of rules for making such adjustments, which he termed "bookkeeping."

Another compilation strategy suitable for VLIW computers emerged from the work of Wen-Mei Hwu and his colleagues at the University of Illinois at Urbana-Champaign. Hwu's method also gathers basic blocks into larger units, in this case called superblocks. But then the program code is rearranged and duplicated as necessary to eliminate merge points, where another path of execution enters the interior of the superblock. The absence of merge points allows a more aggressive approach to scheduling the instructions.

A technique called predicated execution, which can eliminate many branch instructions, has proved a valuable adjunct to VLIW compiler algorithms. In a conventional computer the statement *if* $s \neq 0$ *then* $s = 1/s$ would be represented by a conditional branch that bypasses the division instruction if the inequality proves false. With predicated execution, the comparison stores the value *false* in a predicate register associated with the division instruction; both instructions can be executed at once, but the result of the division is discarded if its predicate is false.

In 1984 Fisher and several colleagues founded a company to manufacture a VLIW computer called the Multiflow Trace. Their largest model had 28 function units and an instruction word of 1,024 bits. At about the same time B. Ramakrishna Rau, another pioneer of VLIW studies, formed a company called Cydrome that built a VLIW minisupercomputer. Neither Multiflow nor Cydrome proved commercially successful, but their failure had more to do with the declining market for large-scale computer systems than with VLIW architecture. Interest in VLIW systems is now reviving, but in a different context: as a candidate for the next evolutionary development in microprocessors for scientific workstations and perhaps for general-purpose personal computers. Both Fisher and Rau are now in the research division of Hewlett-Packard, a leading maker of workstations.

What Comes Next

The next generation of high-performance computer systems will surely have to rely on some form of parallelism. In comparing the superscalar and the VLIW approaches to instruction-level parallelism, there are advantages and disadvantages on both sides. Scheduling at run time, as in the superscalar processor, offers the advantage that everything is known about the path of execution by the time that scheduling decisions have to be made. On the other hand, the analysis of dependencies and the search for instructions that can be executed concurrently must be carried out in haste as the program is running. If the scheduling algorithm became too elaborate, it would consume more time than parallel execution would save.

The VLIW scheduler has the advantage that it runs only once, before execution begins, and hence it can dedicate almost unlimited computational resources to the task. The drawback is that certain details of the program's structure cannot be deduced at compile time. For example, two variables named x and y might appear to be distinct, and yet they could be "aliases" referring to the same memory location; if the compiler cannot prove that x and y are independent, it cannot schedule concurrent operations on them. This problem does not

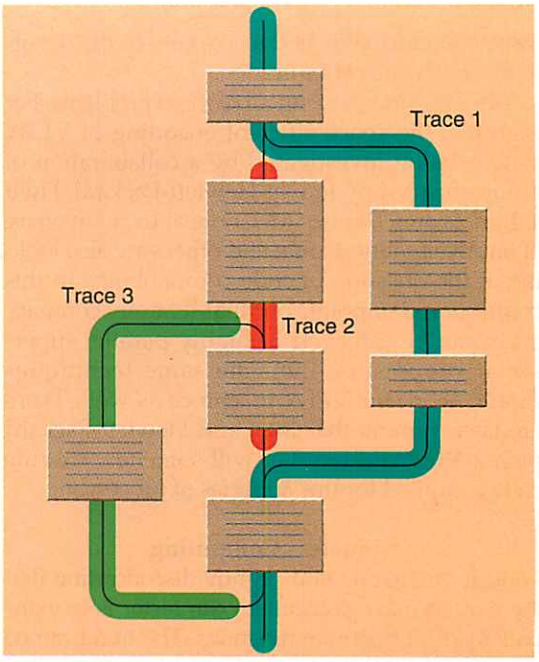


Figure 5. Program traces span multiple basic blocks.

arise in run-time scheduling, since by then the memory addresses themselves are available. Even with such limitations, however, a VLIW compiler should be able to find much more parallelism than a superscalar run-time scheduler.

The prospect of widespread adoption of VLIW technology raises a number of other interesting questions. One issue that is often mentioned is "code bloat." In principle, instructions 1,024 bits wide might take up no more space than instructions 32 bits wide, because there would be proportionately fewer of the wide instructions. In practice, even after adopting various tricks for code compression, VLIW programs would probably be larger than equivalent RISC programs.

Most of the compiling algorithms for VLIW machines have been developed with scientific computing in mind; they are tuned for numerically intensive applications such as Fourier transforms and matrix inversions, where programs are often constructed out of simple loops. A general-purpose computer must also handle other kinds of code efficiently. Indeed, it must be able switch instantly from one kind of task to another. It is not yet clear how well VLIW systems will adapt to this environment.

Perhaps the most worrisome problem connected with VLIW machines is the difficulty of providing a growth path and upward compatibility. Existing processor chips come in families that differ in performance but share the same instruction set. The best-known example is the Intel family: Many programs compiled 15 years ago for the Intel 8086 still run on the current 486 and Pentium designs. The same concept is very attractive for VLIW chips, since more powerful members of a family could be created simply by adding more function units, or by making the units faster. It looks very difficult, however, to achieve compatibility across such a family without recompiling programs; the VLIW instruction format is tied too closely to the details of the hardware configuration.

Work is under way on all of these problems. For example, the space-efficient encoding of VLIW code is being investigated by a collaboration of groups headed by Rau at Hewlett-Packard, Hwu at Illinois and Thomas M. Conte at the University of South Carolina. Conte and others are also looking at the question of code compatibility. In this connection Rau points out that if family compatibility can be achieved in highly parallel superscalar processors, then the same techniques should work for VLIW machines as well. There are even rumors that Intel and Hewlett-Packard plan a VLIW design that will somehow execute code compiled for the '86 series of processors.

Spineless Computing

I find it fascinating and slightly disconcerting that the way to make computers run faster is to extirpate their higher brain functions. The trend can be carried at least one step further. A concept called transport-triggered architecture (TTA), proposed by Henk Corporaal of the University of Delft, breaks down computations into even smaller pieces than RISC or VLIW operations. Addition in an ordinary computer appears to be an atomic operation but is actually a multistep procedure. First the operands are transferred to the two inputs of an adder, then the addition itself is done, and finally the result is moved to some destination register. A TTA processor would have no "add" instruction but would give the compiler control of these individual actions. To continue the gruesome metaphor of lobotomized and decapitated processors, the TTA machine lacks even the spinal-cord neurons that handle primitive reflexes; the compiler must tell each muscle fiber when to twitch.

Who would have guessed that this is an image of the ultimate computer?

Bibliography

Colwell, Robert P., Robert P. Nix, John J. O'Donnell, David B. Papworth and Paul K. Rodman. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers* 37:967–979.

Conte, Thomas M. 1995. Superscalar and VLIW processors. To appear in: A. Y. Zomaya (ed.): *Handbook of Parallel and Distributed Computing*. New York: McGraw-Hill.

Ellis, John R. 1985. *Bulldog: A Compiler for VLIW Architectures*. Cambridge: MIT Press.

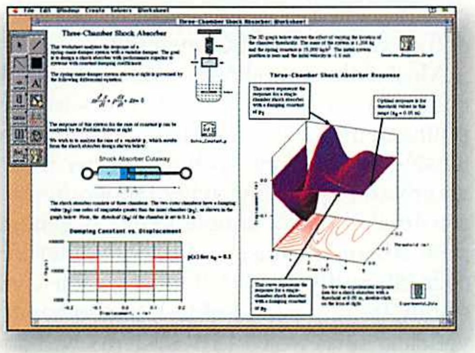
Fisher, Joseph A., and B. Ramakrishna Rau (eds). 1993. Special issue on instruction-level parallelism. *The Journal of Supercomputing* 7.

Pountain, Dick. 1995. Transport-triggered architectures. *Byte* 20(2):151–152.

From the makers of LabVIEW®

HiQ®

Interactive Math for Macintosh and Power Macintosh



Three-Chamber Shock Absorber
Three-Chamber Shock Absorber Response
Damping Constant vs. Displacement

Accelerated for Power Macintosh

Interactively analyze and document in the HiQ environment – much more than a command-line interface.

Solve Math Problems Interactively

- Data Fitting
- Data Visualization
- Eigenvalue
- Expression Evaluation
- Linear Algebra
- Numerical Integration
- Nonlinear Systems
- Optimization
- Ordinary Differential Equations
- Probability and Statistics
- Root Solving
- Signal Processing and much more ...

Call today for Your FREE HiQ Demo (800) 433-3488
(Includes Interactive Data Fitter)
E-mail address: info@natinst.com

NATIONAL INSTRUMENTS™
The Software is the Instrument®
6504 Bridge Point Parkway
Austin, TX 78730-5039 USA
Tel: (512) 794-0100
Fax: (512) 794-8411

Circle 41 on Reader Service Card

SCIENTIST®

the Experimental Data Fitting package from MicroMath

SCIENTIST for Windows was designed from the ground up to fit model equations to experimental data. The culmination of over eight years of software development, SCIENTIST integrates and extends four previous MicroMath products into one powerful package devoted to simulation and data fitting. Fit simple equations, or fit combinations or systems of nonlinear algebraic equations, differential equations and Laplace transforms—or fit a variety of interpolating or least squares splines and polynomials. Also includes 3D line and surface plotting, and a built-in worksheet with dimensional analysis, error propagation, unit conversion/factoring and expression evaluation using IEEE specs.

Requires Windows. Call toll-free for more information or to place order, or Fax for quickest service. $395/Visa/MC accepted. 60 day money back guarantee.

MicroMath®
1-800-942-MATH
FAX 1-801-943-0299

Circle 16 on Reader Service Card