VLIW: The Wave of the Future?

Processor Design Style Could Be Faster, Cheaper Than RISC

by Linley Gwennap

After years of obscurity, VLIW (very long instruction word) architectures have recently become a hot topic. Hewlett-Packard has announced plans *(see **0717MSB.PDF**)* to deploy a VLIW microprocessor as early as 1997. Intel and IBM are also said to be developing such designs, and other companies are investigating this idea. As RISC and CISC microprocessors reach their limits, VLIW could be the road to better performance—or just a cul-de-sac.

In a sense, VLIW is a natural successor to RISC. It moves complexity from the hardware to the compiler, allowing simpler, faster processors. The CISC-to-RISC transition eliminated the decoding and execution of long, complex instructions in hardware. The objective of VLIW is to eliminate the complicated instruction scheduling and parallel dispatch that occurs in most modern microprocessors. In theory, a VLIW processor should be faster and less expensive than a comparable RISC chip. The downside of VLIW is the need for a powerful compiler and a new method of software distribution. Both VLIW and future superscalar processors could be limited by a lack of instruction-level parallelism in typical programs.

Limitations of CISC and RISC

Figure 1 shows a set of operations as they might be presented to a typical CISC, RISC, or VLIW processor. (For this discussion, an operation is a simple task such as load, branch, or add.) A CISC instruction can contain one or more operations and is encoded as one or more bytes; the number of operations, however, is generally less than the number of bytes. The complex, variable-length instructions keep code size small, as memory was expensive in the 1970s when most CISC architectures were designed. These original CISC chips executed one operation every several cycles.

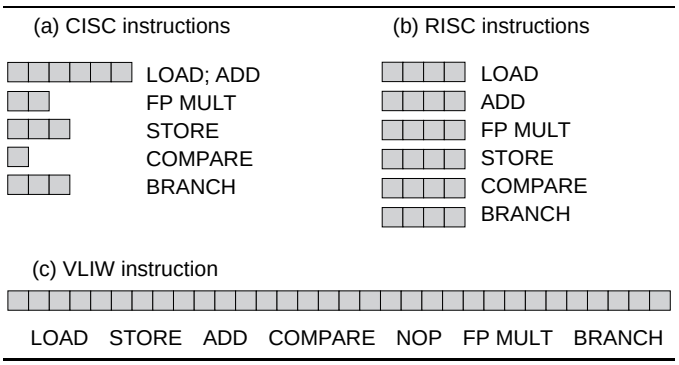

Figure 1. While CISC instructions have variable lengths, RISC instructions are fixed in length. VLIW instructions encode many operations in a single, long instruction word.

In the 1980s, processor designs took advantage of falling memory prices by implementing fixed-length instructions (generally 32 bits wide) that each encode a single operation. These RISC chips rely on the compiler to generate regular, easy-to-decode instructions, simplifying the hardware decode unit. RISC processors typically execute one operation per cycle, simplifying the implementation of pipelining and increasing their clock rate.

Over the past few years, several superscalar microprocessors have appeared. These chips fetch and decode two or more instructions at a time. Before issuing these instructions for simultaneous execution, however, a superscalar processor must first check for potential problems. These include data dependencies, instruction dependencies (branches), and resource conflicts.

Data dependencies occur when one instruction requires the results of a previous instruction; in this case, the two instructions must be issued on separate cycles to generate the correct result. Conditional branches prevent the CPU from knowing which instruction path should be executed; this information is not available until the condition is resolved. Resource conflicts occur when two instructions require the same piece of hardware, such as a particular register or an integer adder.

IBM's Power2 *(see **071301.PDF**)* and other modern processors use multiported register files and duplicated function units to reduce the number of resource conflicts. Cyrix's M1 *(see **071401.PDF**)* and other forthcoming processors will implement branch prediction, speculative execution, and register renaming to resolve conditional branches and reduce data dependencies. By 1995, these techniques will be widely used in processors capable of decoding and executing four to six instructions per cycle.

As the maximum number of instructions to be issued per cycle increases, the number of interactions that must be checked increases geometrically, since each instruction must be compared to every other instruction that could potentially be issued that cycle. This creates very complex hardware designs, particularly in the decode logic, which can increase the cycle time (at 71 MHz, Power2 is significantly slower than many simpler RISC processors) or increase the pipeline depth (Pentium requires two stages to decode instructions); in either case, performance is reduced. Furthermore, this complex logic consumes a large area on the die and is the source of many bugs during the development process.

Due to these factors, many experts wonder whether

current architectures will be able to go beyond 4–6 instructions per cycle *(see **071604.PDF**)*. If this is indeed a limit, performance of these architectures may increase at a much slower rate after 1995, relying mainly on improvements in circuit technology.

The Advantages of VLIW

Many modern compilers are designed to arrange instructions so they can be executed most efficiently by a superscalar target processor. The processor, however, must correctly execute code generated by older compilers for scalar processors and thus must check the instructions again and arrange them for execution. Current instruction sets have few or no provisions to pass scheduling information (such as dependencies and branch frequency) from the compiler to the processor.

A VLIW design eliminates the need for complex instruction-scheduling logic on the chip by moving scheduling entirely into the compiler. Each VLIW instruction consists of a number of independent operations that can be safely executed by the CPU in a single cycle. A VLIW compiler must have intimate knowledge of the number and type of function units in the target processor, as well as the latencies of these units and any other unusual hardware features (interleaved memory, for example). The compiler then groups operations to be executed on each cycle.

Figure 2 shows how a typical VLIW instruction presents a number of operations as a single, very long instruction (hence the name). These operations can flow directly to the various function units with a minimum of decoding. A NOP is sent to unused units; a pure VLIW processor has no interlocks. For cycles in which no processing takes place, a very wide NOP must be issued; this could happen while waiting for memory, for example.

Eliminating the hardware instruction scheduler and simplifying decode and execution will result in a smaller design than a comparable RISC or CISC processor. Moving functions such as branch prediction and register renaming into the compiler further simplifies the hardware. The cycle time of this simpler design should also improve. These differences will increase as superscalar processors become even more complex.

While VLIW hardware becomes simpler, the compiler becomes more challenging. Modern compilers are already aware of the intricacies of processor designs and perform extensive instruction scheduling; a VLIW compiler takes this work a step further. Even for today's relatively simple superscalar CPUs, the compiler can do a better job of scheduling instructions than the hardware dispatcher; this gap could grow as processors implement more function units. With VLIW, users should be able to trade a longer compile time for speedier program execution, an attractive proposition for most.

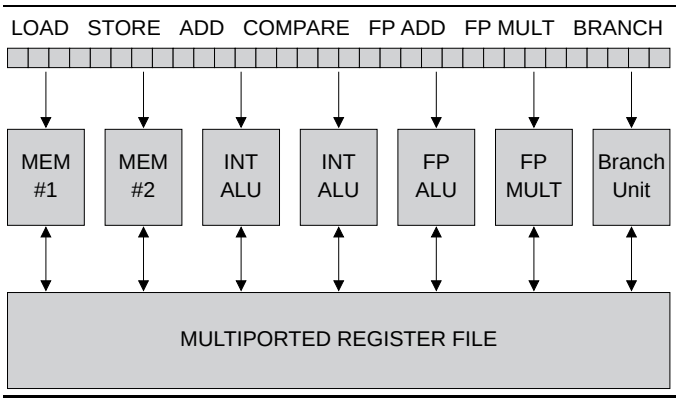

Figure 2. A very long instruction word contains multiple operations, each of which can be routed directly to a single function unit.

VLIW Evolved from Mainframes

VLIW is hardly new; like most computer architecture techniques, it was first pioneered in mainframes, appeared later in minicomputers, and ultimately ended up in microprocessors. According to VLIW guru Josh Fisher, a similar technique called horizontal microcode was considered by computer pioneer Alan Turing in 1946 and detailed by Maurice Wilkes in 1951. Contrasting the single wide VLIW instruction in Figure 1 with the presentation of operations to a RISC or CISC machine shows the derivation of the term horizontal.

During the 1970s, a number of companies built special-purpose computers using horizontal microcode. The development of fast writable memory led some vendors, notably Floating Point Systems, to implement a writable control store and hence a programmable VLIW machine. Because of their specialized designs, however, these systems proved unsuited for general applications.

In the 1980s, a few small companies attempted to deliver general-purpose VLIW systems to the minisupercomputer market. Most notable were Multiflow, Cydrome, and Culler. These companies were able to complete entire system designs and ship a number of units, but all were ultimately unsuccessful. (Fisher was the principal technologist at Multiflow, and Bob Rau led the Cydrome effort; both are now at HP.)

These vendors may have had the right idea at the wrong time. The Multiflow Trace and Cydrome Cydra 5 were constructed from multiple gate arrays, as the designs were too complex for then-current VLSI implementations. The multichip designs hampered communication and reduced cycle time. Neither system implemented a data cache due to technology limitations. As a result, the Multiflow machine delivered about the same performance on general-purpose integer code as the MIPS R2000, a contemporary RISC microprocessor. On scientific code, it was able to exploit the higher levels of parallelism to achieve more than twice the performance of the R2000, but comparably priced vector processors delivered similar results.

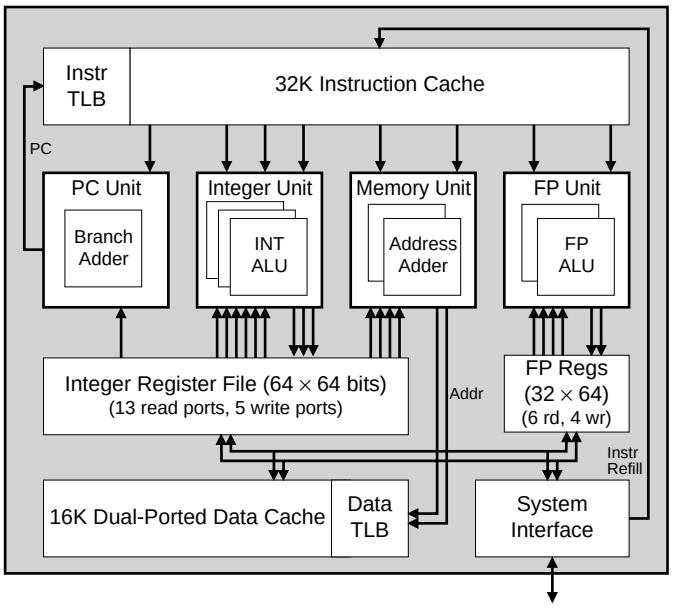

Figure 3. A hypothetical VLIW microprocessor with eight function units could be built with current VLSI technology.

The Multiflow and Cydrome Processors

The Multiflow machine came in three flavors: the basic system used a 256-bit instruction word, while larger versions used a 512-bit or 1024-bit width. The wider designs were simply combinations of the basic system, dubbed the Trace 7/300.

The 7/300 contained two integer ALUs, two floating-point ALUs, and a branch unit. The integer units were fast enough to execute two operations per 130-ns cycle, however, so the system could perform a total of four integer operations per cycle, two of which could be loads or stores. The 256-bit instruction was thus divided into seven 32-bit operations; the remaining 32 bits were used to encode immediate values.

Each 32-bit operation used a three-operand RISC-like encoding; all loads and stores were explicitly encoded as separate operations. The 7/300 contained a total of 64 (32-bit) integer registers and 32 (64-bit) floating-point registers. Due to the physical implementation, these registers were divided among the function units, but all units could access all registers (with time delays for remote accesses).

Because of the wide instructions and explicit NOPs, programs could be quite large. To conserve space on disk and in main memory, code was stored in a compressed format with NOPs removed. The processor automatically expanded the instructions as they were loaded into the 8K instruction cache. As an additional space saver, a single instruction could encode a multicycle NOP.

The memory system allowed up to 512M of main memory with up to 64-way interleaving. The Trace had no data cache, so all memory references went directly to main memory. The compiler was required to observe all latencies, take into account the interleaving, and schedule the eight memory buses. Conflicting memory references on a single bus or memory card caused an undefined program error.

To increase utilization of the function units, the compiler could speculatively execute operations that followed conditional branches. This speculative execution required minimal support from the hardware, consisting mainly of a set of speculative load opcodes that handled exceptions differently. The Trace compiler also performed branch prediction, loop unrolling, and a variety of other techniques to locate and issue independent operations in parallel.

Many of the compiler tactics developed by Multiflow are quite applicable to standard superscalar processors as well; as noted previously, compilers for these CPUs do a significant amount of instruction scheduling. Nearly every high-performance processor vendor has licensed the Trace compiler technology, including HP, Intel, Digital, Silicon Graphics, and Fujitsu.

Cydrome's VLIW processor was similar to the 7/300 with its 256-bit instructions that each contained seven operations. Unlike the regular operation coding of the Trace machine, the Cydrome operations used a varying number of bits as needed. For example, a memory access with displacement had a 44-bit encoding, while a simple integer operation required 39 bits. This format was called MultiOp.

To conserve space, Cydrome also implemented a second instruction format called UniOp. In this mode, a 256-bit instruction contained six 40-bit operations that were each issued on separate cycles. A simple decoder routed the operation to the correct function unit while issuing NOPs to the other units. The instruction stream could arbitrarily mix MultiOp and UniOp forms. The compiler typically generated MultiOp instructions for inner loops where parallelism was high, while the UniOp format was used for the rest of the code.

VLIW Microprocessors, Past and Future

By the late 1980s, IC manufacturing improvements made it possible to implement a VLIW processor on a single chip. Intel's i860 (see MPR 3/1/89, p. 1) was arguably the first such device. It executes a pair of 32-bit instructions on each cycle; unlike true superscalar code, these instructions must be paired and aligned for the chip's two function units (integer and FP). Philips built a more aggressive VLIW microprocessor called LIFE (see MPR 8/8/90, p. 6) that incorporated six function units, but the design was not commercially successful.

Figure 3 shows a hypothetical VLIW processor that can execute up to eight instructions per cycle: three integer math, two floating-point math, two load/stores, and one branch. It has 64 integer registers and 32 floating-

point registers, all 64 bits wide. The instructions are 256 bits wide, with each operation using a 32-bit RISC-like encoding. (This would require stealing a few bits to address the extra integer registers.) The data cache is fully dual-ported and 16K in size; the instruction cache size is 32K (1K instructions).

This processor could be built easily using current IC processes. As a demonstration, consider connecting two R4600 processors: the resulting chip is about 150 mm^2 in a 0.65-micron process, still relatively small compared to Pentium or SuperSparc. The new processor fits all the above parameters except for the third integer unit. Also, the register files would have to expand significantly to handle the extra read and write ports. These additional features could probably be inserted by removing the extra RISC control logic not needed by a VLIW design. If not, the die size could grow a bit.

The cycle time of this hypothetical processor would be limited only by the speed of the function units. As demonstrated by the R4400 and Alpha processors, it is possible to build ALUs that cycle at 150–200 MHz in manufacturable microprocessors. The new CPU should easily match the instructions-per-cycle of IBM's Power2 (which has fewer function units); at that rate, the 150-MHz VLIW chip, with a comparable memory system, should exceed 200 SPECint92 and 300 SPECfp92.

VLIW Breaks Binary Compatibility

So why hasn't anyone built this processor? For one thing, it involves designing an architecture from the ground up. Like any new architecture, a VLIW design would be incompatible with existing binaries, which has been a formidable barrier to its adoption. The RISC-like operation encodings make it appear that a VLIW processor could be designed to execute existing RISC instructions in a mode similar to Cydrome's UniOp, but the RISC binaries would need some additional encodings to differentiate them from VLIW instructions. Furthermore, using an existing instruction set as the basis for a VLIW design would restrict the ability to add features, such as extra integer registers, that will be needed in these future processors.

Another difficulty with a VLIW design is that the compiler cannot know what data will be used when a program is executed. As a result, certain instruction groupings cannot be guaranteed to be safe, and a pure VLIW system would have to schedule for the worst case. A compromise solution is to implement simple scoreboarding in hardware to detect such conflicts; this allows the compiler to schedule instructions more aggressively while taking a performance hit in rare situations.

The biggest problem with VLIW, one that Multiflow and Cydrome were encountering just as they ran out of cash, is that each successive VLIW machine is generally not compatible with binaries for the previous design. If programs written for the 386 would not run on the 486, the 486 would be much less successful than it is today. Similarly, VLIW machines will find it difficult to succeed until the binary compatibility problem can be solved.

The best idea so far is to distribute software in an intermediate format instead of in binary form. When a program is loaded onto a system, an installer would transform the intermediate code into a binary targeted for the specific processor in that system. The trick is to define a distribution format that does not reveal the source code but contains enough information to generate the necessary binaries.

This concept is similar to the architecture-neutral distribution format (ANDF) proposed by OSF, but would be much simpler to implement. Multiple generations of VLIW processors could share the same operation encodings, so the installer would simple rearrange the operations based on the number and latencies of the function units in the target processor.

A true ANDF would allow programs to be installed on current RISC or CISC systems and then reinstalled on future VLIW processors. Alternatively, future VLIW systems could retain compatibility with existing processors via software emulation or binary translation.

The Fundamental Issue of ILP

As shown by the example, it is possible that a VLIW processor built today could outperform contemporary microprocessors by a significant margin. That 50–100% gap may not be enough to incite vendors to move to a new, unproven technology. As hardware instruction scheduling becomes more complex, however, the gap between traditional implementations and VLIW designs could grow, leading to a changeover at some point.

One counterargument is that the limits of instruction-level parallelism (ILP) will nullify the benefits of building processors with eight or more execution units. For these processors, the instruction scheduler (be it in hardware or software) must find enough data-independent instructions to take advantage of the large number of function units. Some studies show that, even with infinite function units, ILP peaks at three to five operations per cycle for typical (nonscientific) code. If this is true, there is no point in building CPUs, VLIW or not, with more function units—the future may instead lie in multiple independent CPUs on a chip.

The biggest problem is that most programs encounter a branch every six instructions or so. To improve ILP, the scheduler must use accurate branch prediction and speculative execution to look beyond the next branch. Loop unrolling and trace analysis assist in this process. These techniques are in practice today; Fisher is optimistic that future compiler writers will develop new techniques to further increase ILP. If he's right, there could be a bright future for VLIW microprocessors. ♦

