
INTRODUCTION AND MOTIVATION

Currently, in the mid 1990s, IC fabrication technology is advanced enough to allow unprecedented implementations of computer architectures on a single chip. Also, the current rate of process advancement allows implementations to be improved at a rate that is satisfying for most of the markets these implementations serve. In particular, the vendors of general-purpose microprocessors are competing for sockets in desktop personal computers (including workstations) by pushing the envelopes of clock rate (raw operating speed) and parallel execution.

The market for desktop microprocessors is proving to be extremely dynamic. In particular, the x86 market has surprised many observers by attaining performance levels and price/performance levels that many thought were out of reach. The reason for the pessimism about the x86 was its architecture (instruction set). Indeed, with the advent of RISC architectures, the x86 is now recognized as a deficient instruction set.

Instruction set compatibility is at the heart of the desktop microprocessor market. Because the application programs that end users purchase are delivered in binary (directly executable by the microprocessor) form, the end users' desire to protect their software investments creates tremendous *instruction-set* inertia.

There is a different market, though, that is much less affected by instruction-set inertia. This market is typically called the embedded market, and it is characterized by products containing factory-installed software that runs on a microprocessor whose instruction set is not readily evident to the end user. Although the vendor of the product containing the embedded microprocessor has an investment in the embedded software, just like end users with their applications, there is considerably more freedom to migrate embedded software to a new microprocessor with a different instruction set. To overcome this lower level of instruction-set inertia, all it takes is a sufficiently better set of implementation characteristics, particularly absolute performance and/or price-performance.

This lower level of instruction-set inertia gives the vendors of embedded microprocessors the freedom and initiative to seek out new instruction sets. The relative success of RISC microprocessors in the high-end of the embedded market is an example of innovation by microprocessor vendors that produced a benefit large enough to overcome the market's inertia. To the vendors' disappointment, the benefits of RISCs have not been sufficient to overcome the instruction-set inertia of the mainstream desktop computer market.

Because of advances in IC fabrication technology and advances in high-level language compiler technology, it now appears that microprocessor vendors are compelled by the potential benefits of another change in microprocessor instruction sets. As before, the embedded market is likely to be first to accept this change.

The new direction in microprocessor architecture is toward VLIW (very long instruction word) instruction sets. VLIW architectures are characterized by instructions that each specify several independent operations. This is compared to RISC instructions that typically specify one operation and CISC instructions that typically specify several dependent operations. VLIW instructions are necessarily longer than RISC or CISC instructions, thus the name.

WHY VLIW?

The key to higher performance in microprocessors for a broad range of applications is the ability to exploit fine-grain, instruction-level parallelism. Some methods for exploiting fine-grain parallelism include:

- + pipelining
- + multiple processors
- + superscalar implementation
- + specifying multiple independent operations per instruction

Pipelining is now universally implemented in high-performance processors. Little more can be gained by improving the implementation of a single pipeline.

Using multiple processors improves performance for only a restricted set of applications.

Superscalar implementations can improve performance for all types of applications. Superscalar (*super*: beyond; *scalar*: one dimensional) means the ability to fetch, issue to execution units, and complete more than one instruction at a time. Superscalar implementations are required when architectural compatibility must be preserved, and they will be used for entrenched architectures with legacy software, such as the x86 architecture that dominates the desktop computer market.

Specifying multiple operations per instruction creates a very-long instruction word architecture or VLIW. A VLIW implementation has capabilities very similar to those of a superscalar processor—issuing and completing more than one operation at a time—with one important exception: the VLIW *hardware* is not responsible for discovering opportunities to execute multiple operations concurrently. For the VLIW implementation, the long instruction word already encodes the concurrent operations. This explicit encoding leads to dramatically reduced hardware complexity compared to a high-degree superscalar implementation of a RISC or CISC.

The big advantage of VLIW, then, is that a highly concurrent (parallel) implementation is much simpler and cheaper to build than equivalently concurrent RISC or CISC chips. VLIW is a simpler way to build a superscalar microprocessor.

ARCHITECTURE VS. IMPLEMENTATION

The word *architecture* in the context of computer science is often misused. Used accurately, architecture refers to the instruction set and resources available to someone who writes programs. The architecture is what is described in a definition document, often called a *user's manual*. Thus, architecture contains instruction formats, instruction semantics (operation definitions), registers, memory addressing modes, characteristics of the address space (linear, segmented, special address regions), and anything else a programmer would need to know.

An implementation is the hardware design that realizes the operations specified by the architecture. The implementation determines the characteristics of a microprocessor that are most often measured: price, performance, power consumption, heat dissipation, numbers of pins, operating frequency, and so on.

Architecture and implementation are separate, but they do interact. As many researchers into computer architecture discovered between the mid 1970s and 1980s, architecture can have a dramatic effect on the quality of an implementation. In the mid 1980s, IC process technology could fabricate a microcoded implementation of a CISC instruction set and a tiny cache or MMU. For about the same cost, this same process technology could fabricate a pipelined implementation of a simple RISC instruction set (including

Philips Semiconductors

Introduction to VLIW Computer Architecture

large register file) with an MMU. At the time, however, chip technology was not dense enough to build a cost-effective, pipelined implementation of a CISC instruction set. As a result, pipelined RISC chips enjoyed a dramatic performance advantage and made a compelling case for RISC architectures.

The important point is that a range of implementations of any architecture can be built, but architecture influences the quality and cost-effectiveness of those implementations. The influence is exerted largely in the trade-offs that must be made to accommodate complexity associated with the instruction set. The more chip area spent on logic to decode instructions and implement irregularities, the less that can be devoted to performance-enhancing features.

ARCHITECTURE COMPARISON: CISC, RISC, AND VLIW

From the larger perspective, RISC, CISC, and VLIW architectures have more similarities than differences. The differences that exist, however, have profound effects on the implementations of these architectures.

Obviously these architectures all use the traditional state-machine model of computation: Each instruction effects an incremental change in the state (memory, registers) of the computer, and the hardware fetches and executes instructions sequentially until a branch instruction causes the flow of control to change.


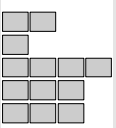
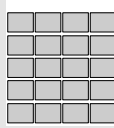

ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FORMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFERENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRUCTIONS  = 1 BYTE			

TABLE I

The differences between RISC, CISC, and VLIW are in the formats and semantics of the instructions. Table I compares architecture characteristics.

CISC instructions vary in size, often specify a sequence of operations, and can require serial (slow) decoding algorithms. CISCs tend to have few registers, and the registers may be special-purpose, which restricts the ways in which they can be used. Memory references are typically combined with other operations (such as *add memory to register*). CISC instruction sets are designed to take advantage of microcode.

RISC instructions specify simple operations, are fixed in size, and are easy (quick) to decode. RISC architectures have a relatively large number of general-purpose registers. Instructions can reference main

Philips Semiconductors

Introduction to VLIW Computer Architecture

memory only through simple *load-register-from-memory* and *store-register-to-memory* operations. RISC instruction sets do not need microcode and are designed to simplify pipelining.

VLIW instructions are like RISC instructions except that they are longer to allow them to specify multiple, independent simple operations. A VLIW instruction can be thought of as several RISC instructions joined together. VLIW architectures tend to be RISC-like in most attributes.

Figure 1 shows a C-language code fragment containing small function definition. This function adds a local variable to a parameter passed from the caller of the function.

The implementation of this function in CISC, RISC, and VLIW code is also shown. This example is extremely unfair to the RISC and VLIW machines, but it illustrates the differences between the architectures.

The CISC code consists of one instruction because the CISC architecture has an add instruction that can encode a memory address for the destination. So, the CISC instruction adds the local variable in register r2 to the memory-based parameter. The encoding of this CISC instruction might take four bytes on some hypothetical machine.

The RISC code is artificially inefficient. Normally, a good compiler would pass the parameter in a register, which would make the RISC code consist of only a single register-to-register add instruction. For the sake of illustration, however, the code will consist of three instructions as shown. These three instructions load the parameter to a register, add it to the local variable already in a register, and then store the result back to memory. Each RISC instruction requires four bytes.

The VLIW code is similarly hampered by poor register allocation. The example VLIW architecture shown has the ability to simultaneously issue three operations. The first slot (group of four bytes) is for branch instructions, the middle slot is for ALU instructions, and the last slot is for the load/store unit. Since the three RISC operations needed to implement the code fragment are dependent, it is not possible to pack the load and add in the same VLIW instruction. Thus, three separate VLIW instructions are necessary.

With the code fragment as shown, the VLIW instruction is depressingly inefficient from the point of view of code density. In a real program situation, the compiler for the VLIW would use several program optimization techniques to fill all three slots in all three instructions. It is instructive to contemplate the performance each machine might achieve for this code. We need to assume that each machine has an efficient, pipelined implementation.

```
function (j)
long j; {
long i;

    j = j + i;
}
```

CISC

```
add 4[r1] <- r2
```

addMR	4	r1	r5
-------	---	----	----

RISC

```
load r5 <- 4[r1]
add r5 <- r5 + r2
store 4[r1] <- r5
```

load	r5	r1	4
add	r5	r5	r2
store	r5	r1	4

VLIW

```
load r5 <- 4[r1]
add r5 <- r5 + r2
store 4[r1] <- r5
```

-	-	-	-	-	-	-	-	load	r5	r1	4
-	-	-	-	add	r5	r5	r2	-	-	-	-
-	-	-	-	-	-	-	-	store	r5	r1	4

FIGURE 1

Philips Semiconductors

Introduction to VLIW Computer Architecture

A CISC machine such as the 486 or Pentium would be able to execute the code fragment in three cycles.

A RISC machine would be able to execute the fragment in three cycles as well, but the cycles would likely be faster than on the CISC.

The VLIW machine, assuming three fully-packed instructions, would effectively execute the code for this fragment in one cycle. To see this, observe that the fragment requires three out of nine slots, for one-third use of resources. One-third of three cycles is one cycle.

To be even more accurate, we can assume good register allocation as shown in Figure 2. This example may actually be giving the CISC machine a slight unfair advantage since it will not be possible to allocate parameters to registers on the CISC as often as is possible for the RISC and VLIW.

The CISC and RISC machines with good register allocation would take one cycle for one register-to-register instruction, but notice that the RISC code size is now much more in line with that of the CISC. Again assuming fully packed instructions, the VLIW execution time would also gain a factor of three benefit from good register allocation, yielding an effective execution time for the fragment of one-third of a cycle!

Note that these comparisons have been between scalar (one-instruction per cycle maximum) RISC and CISC implementations and a relatively narrow VLIW. While it would be more realistic to compare superscalar RISCs and CISCs against a wider VLIW, such a comparison is more complicated. Suffice it to say that the conclusions would be roughly the same.

```
function (j)
long j; {
long i;

    j = j + i;
}
```

CISC

add r3 <- r2

addRR	r1	r5
-------	----	----

RISC

add r3 <- r3 + r2

add	r3	r3	r2
-----	----	----	----

VLIW

add r3 <- r3 + r2

-	-	-	-	add	r3	r3	r2	-	-	-	-
---	---	---	---	-----	----	----	----	---	---	---	---

FIGURE 2

IMPLEMENTATION COMPARISON: SUPERSCALAR CISC, SUPERSCALAR RISC, VLIW

The differences between CISC, RISC, and VLIW architectures manifest themselves in their respective implementations. Comparing high-performance implementations of each is the most telling.

High-performance RISC and CISC designs are called superscalar implementations. Superscalar in this context simply means "beyond scalar" where scalar means one operations at a time. Thus, superscalar means more than one operation at a time.

Most CISC instruction sets were designed with the idea that an implementation will fetch one instruction, execute its operations fully, then move on to the next instruction. The assumed execution model was thus serial in nature.

RISC architects were aware of the advantages and peculiarities of pipelined processor implementations, and so designed RISC instruction sets with a pipelined execution model in mind. In contrast to the assumed CISC execution model, the idea for the RISC execution model is that an implementation will fetch one instruction, issue it into the pipeline, and then move on to the next instruction before the previous one has completed its trip through the pipeline.

Philips Semiconductors

Introduction to VLIW Computer Architecture

The assumed RISC execution model—a pipeline—overlaps phases of execution for several instructions simultaneously, but like the CISC execution model, it is scalar; that is, at most one instruction is issued at once.

For either CISC or RISC to reach higher levels of performance than provided by a single pipeline, a superscalar implementation must be constructed. The nature of a superscalar implementation is that it fetches, issues, and completes more than one CISC or RISC instruction per cycle.

Some more recent RISC architectures have been designed with superscalar implementations in mind. The most notable examples are the DEC Alpha and IBMPOWER (from which PowerPC is derived). Nonetheless, superscalar RISC and superscalar CISC implementations share fundamental complexities: the need for the hardware to discover and exploit instruction-level parallelism.

Figure 3 shows a crude high-level block diagram of a superscalar RISC or CISC processor implementation. The implementation consists of a collection of execution units (integer ALUs, floating-point ALUs, load/store units, branch units, etc.) that are fed operations from an instruction dispatcher and operands from a register file.

The execution units have reservation stations to buffer waiting operations that have been issued but are not yet executed. The operations may be waiting on operands that are not yet available.

The instruction dispatcher examines a window of instructions contained in a buffer. The dispatcher looks at the instructions in the window and decides which ones can be dispatched to execution units. It tries to dispatch as many instructions at once as is possible, i.e. it attempts to discover maximal amounts of instruction-level parallelism. Higher degrees of superscalar execution, i.e., more execution units, require wider windows and a more sophisticated dispatcher.

It is conceptually simple—though expensive—to build an implementation with lots of execution units and an aggressive dispatcher, but it is not currently profitable to do so. The reason has more to do with software than hardware.

The compilers for RISC and CISC processors produce code with certain goals in mind. These goals are typically to minimize code size and run time. For scalar and very simple superscalar processor implementation, these goals are mostly compatible.

For high-performance superscalar implementations, on the other hand, the goal of minimizing code size limits the performance that the superscalar implementation can achieve. Performance is limited because minimizing code size results in frequent conditional branches, about every six instructions. Conceptually, the

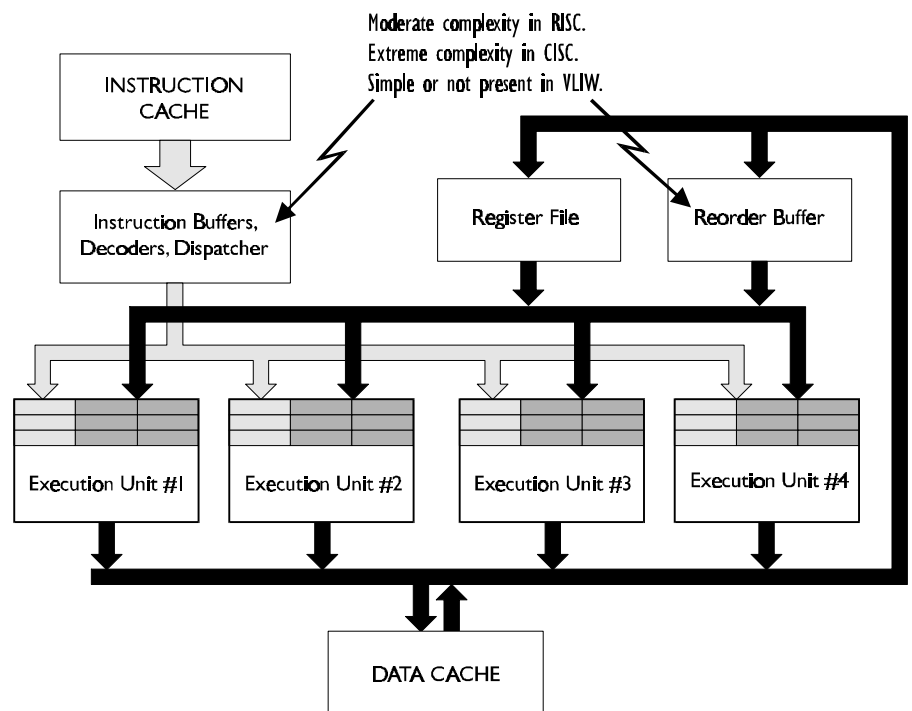


FIGURE 3

processor must wait until the branch is resolved before it can begin to look for parallelism at the target of the branch.

To avoid waiting for conditional branches to be resolved, high-performance superscalar implementations implement branch prediction. With branch prediction, the processor makes an early guess about the outcome of the branch and begins looking for parallelism along the predicted path. The act of dispatching and executing instructions from a predicted—but unconfirmed—path is called speculative execution.

Unfortunately, branch prediction is not 100% accurate. Thus, with speculative execution, it is necessary to be able to *undo* the effects of speculatively executed instructions in the case of a mispredicted branch. Some implementations, such as Intel's superscalar Pentium, simply prevent instructions along the predicted path from progressing far enough to modify any visible processor state, but to gain the most from speculative execution, it is necessary to allow instructions along the predicted path to execute fully.

To be able to undo the effects of full, speculative execution, a hardware structure called a reorder buffer can be employed. This structure is an adjunct to the register file that keeps track of all the results produced by instructions that have recently been executed or that have been dispatched to execution units but have not yet completed. The reorder buffer provides a place for results of speculatively executed instruction (and it solves other problems as well). When a conditional branch is, in fact, resolved, the results of the speculatively executed instructions can be either dropped from the reorder buffer (branch mispredicted) or written from the buffer to the register file (branch predicted correctly).

The major differences between high-performance superscalar implementations of RISCs and CISCs are a matter of degree. The instruction-decode and -dispatch logic in an aggressive superscalar RISC is simpler than that of an aggressive superscalar CISC, but the logic is required in both cases. The same is true for the reorder buffer.

SOFTWARE INSTEAD OF HARDWARE: IMPLEMENTATION ADVANTAGES OF VLIW

A VLIW implementation achieves the same effect as a superscalar RISC or CISC implementation, but the VLIW design does so without the two most complex parts of a high-performance superscalar design.

Because VLIW instructions explicitly specify several independent operations—that is, they explicitly, specify parallelism—it is not necessary to have decoding and dispatching hardware that tries to reconstruct parallelism from a serial instruction stream. Instead of having hardware attempt to discover parallelism, VLIW processors rely on the compiler that generates the VLIW code to explicitly specify parallelism. Relying on the compiler has advantages.

First, the compiler has the ability to look at much larger windows of instructions than the hardware. For a superscalar processor, a larger hardware window implies a larger amount of logic and therefore chip area. At some point, there simply is not enough of either, and window size is constrained. Worse, even before a simple limit on the amount of hardware is reached, complexity may adversely affect the speed of the logic, thus the window size is constrained to avoid reducing the clock speed of the chip. Software windows can be arbitrarily large. Thus, looking for parallelism in a software window is likely to yield better results.

Second, the compiler has knowledge of the source code of the program. Source code typically contains important information about program behavior that can be used to help express maximum parallelism at the instruction-set level. A powerful technique called trace-driven compilation can be employed to dramatically improve the quality of code output by the compiler. Trace-driven compilation first produces a suboptimal, but correct, VLIW program. The program has embedded routines that take note of program behavior. The recorded program behavior—which branches are taken, how often, etc.—is then used by the compiler during a second compilation to produce code that takes advantage of accurate knowledge of

program behavior. Thus, with trace-driven compilation, the compiler has access to some of the dynamic information that would be apparent to the hardware dispatch logic in a superscalar processor.

Third, with sufficient registers, it is possible to mimic the functions of the superscalar implementation's reorder buffer. The purpose of the reorder buffer is to allow a superscalar processor to speculatively execute instructions and then be able to quickly discard the speculative results if necessary. With sufficient registers, a VLIW machine can place the results of speculatively executed instructions in temporary registers. The compiler knows how many instructions will be speculatively executed, so it simply uses the temporary registers along the speculated (predicted) path and ignores the values in those registers along the path that will be taken if the branch turn out to have been mispredicted.

Figure 4 shows a generic VLIW implementation, without the complex reorder buffer and decoding and dispatching logic.

THE ADVANTAGE OF COMPILER COMPLEXITY OVER HARDWARE COMPLEXITY

While a VLIW architecture reduces hardware complexity over a superscalar implementation, a much more complex compiler is required. Extracting maximum performance from a superscalar RISC or CISC implementation does require sophisticated compiler techniques, but the level of sophistication in a VLIW compiler is significantly higher.

VLIW simply moves complexity from hardware into software. Luckily, this trade-off has a significant side benefit: the complexity is paid for only once, when the compiler is written instead of every time a chip is fabricated. Among the possible benefits is a smaller chip, which leads to increased profits for the microprocessor vendor and/or cheaper prices for the customers that use the microprocessors. Complexity is usually easier to deal with in a software design than in a hardware design. Thus, the chip may cost less to design, be quicker to design, and may require less debugging, all of which are factors that can make the design cheaper. Also, improvements to the compiler can be made after chips have been fabricated; improvements to superscalar dispatch hardware require changes to the microprocessor, which naturally incurs all the expenses of turning a chip design.

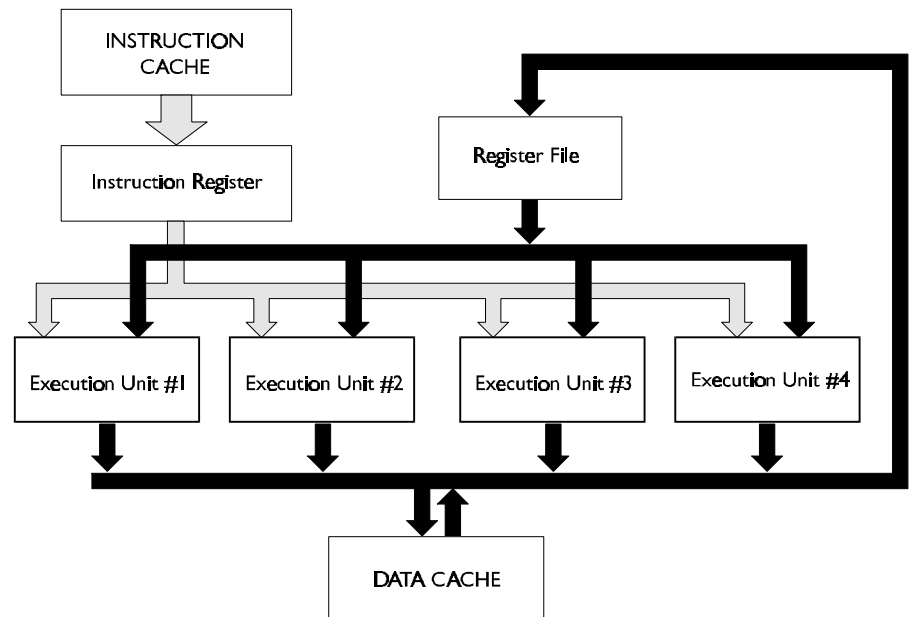


FIGURE 4

PRACTICAL VLIW ARCHITECTURES AND IMPLEMENTATIONS

The simplest VLIW instruction format encodes an operation for every execution unit in the machine. This makes sense under the assumption that every instruction will always have something useful for every execution unit to do. Unfortunately, despite the best efforts of the best compiler algorithms, it is typically not possible to pack every instruction with work for all execution units. Also, in a VLIW machine that has both integer and floating-point execution units, the best compiler would not be able to keep the floating-point units busy during the execution of an integer-only application.

Philips Semiconductors

Introduction to VLIW Computer Architecture

The problem with instructions that do not make full use of all execution units is that they waste precious processor resources: instruction memory space, instruction cache space, and bus bandwidth.

There are at least two solutions to reducing the waste of resources due to sparse instructions. First, instructions can be compressed with a more highly-encoded representation. Any number of techniques, such as Huffman encoding to allocate the fewest bits to the most frequently used operations, can be used.

Second, it is possible to define an instruction word that encodes fewer operations than the number of available execution units. Imagine a VLIW machine with ten execution units but an instruction word that can describe only five operations. In this scheme, a unit number is encoded along with the operation; the unit number specifies to which execution unit the operation should be sent. The benefit is better utilization of resources. A potential problem is that the shorter instruction prohibits the machine from issuing the maximum possible number of operations at any one time. To prevent this problem from limiting performance, the size of the instruction word can be tuned based on analysis of simulations of program behavior.

Of course, it is completely reasonable to combine these two techniques: use compression on shorter-than-maximum-length instructions.

HISTORICAL PERSPECTIVE

VLIW is not a new computer architecture. Horizontal microcode, a processor implementation technique in use for decades, defines a specialized, low-level VLIW architecture. This low-level architecture runs a microprogram that interprets (emulates) a higher-level (user-visible) instruction set. The VLIW nature of the horizontal microinstructions is used to attain a high-performance interpretation of the high-level instruction set by executing several low-level steps concurrently. Each horizontal microcode instruction encodes many irregular, specialized operations that are directed at primitive logic blocks inside a processor. From the outside, the horizontally microcoded processor appears to be directly running the emulated instruction set.

In the 1980s, a few small companies attempted to commercialize VLIW architectures in the general-purpose market. Unfortunately, they were ultimately unsuccessful. Multiflow is the most well known. Multiflow's founders were academicians who did pioneering, fundamental research into VLIW compilation techniques. Multiflow's computers worked, but the company was probably about a decade ahead of its time. The Multiflow machines, built from discrete parts, could not keep pace with the rapid advances in single-chip microprocessors. Using today's technology, they would have a better chance at being competitive.

In the early 1990s, Intel introduced the i860 RISC microprocessor. This simple chip had two modes of operation: a scalar mode and a VLIW mode. In the VLIW mode, the processor always fetched two instructions and assumed that one was an integer instruction and the other floating-point. A single program could switch (somewhat painfully) between the scalar and VLIW modes, thus implementing a crude form of code compression. Ultimately, the i860 failed in the market. The chip was positioned to compete with other general-purpose microprocessors for desktop computers, but it had compilers of insufficient quality to satisfy the needs of this market.