x86 Has Plenty of Performance Headroom

Aggressive Superscalar Techniques Just Beginning to Appear

by Brian Case

Ever since Intel introduced the 386 in 1985, pundits have predicted that the x86 architecture had reached the end of the line in delivering competitive performance. So far, these predictions have been premature.

Over the past nine years, Intel has vehemently denied that x86 performance would fall behind that of any other architecture—especially any RISC architecture. While no x86 has ever been an absolute performance leader—it has always been behind implementations of other architectures—Intel has adequately backed up its bravado with excellent, competitive implementations in the 486 and Pentium.

Now, with the recent Intel/HP agreement *(see **080801.PDF**)*, Intel has had a change of heart and apparently believes that the x86 architecture will be too limiting for implementations beyond another generation or two. It is strange that now, just as implementation technology is leveling the playing field between RISCs and the x86, Intel is pulling away from its flagship architecture. It is true that keeping the x86 competitive requires complex implementations, but why would Intel suddenly be afraid of complexity?

The key to high performance is parallelism, and given enough hardware, available parallelism is controlled fundamentally by the program, not the processor architecture. Techniques like out-of-order execution and register renaming will allow superscalar x86 implementations to exploit available program parallelism as effectively as advanced implementations of other architectures (but the x86 will require more logic).

A VLIW (Very Long Instruction Word) architecture—which Intel and HP are rumored to be developing—does allow program parallelism to be exploited with less hardware complexity and, therefore, possibly a faster processor cycle time *(see **080205.PDF**)*. A faster cycle time could give the VLIW machine higher performance if all other implementation aspects remain equal, but the performance advantage may not be worth the costs of changing architectures. As evidence, note that the performance advantage of RISCs has not been enough to cause a significant shift in the market.

Unless there is a truly fundamental change in software technology—e.g., the way programs are written—x86 implementations can remain competitive well past the end of the decade. Even if Intel moves away from the x86, there will still be several microprocessor vendors willing to supply market demand.

This article is a tutorial introduction to some of the techniques that will be used in forthcoming superscalar x86 microprocessors. The Cyrix M1 *(see **071401.PDF**)* and NexGen's 586 *(see **080403.PDF**)* are harbingers of what is to come. We expect AMD's K5 and Intel's P6 to further exploit advanced superscalar techniques.

Aggressive Superscalar Design

Although the M1 pipeline improves on Pentium's—it allows ALU operations with a memory operand to be pipelined and executed in a single cycle and provides a modest performance benefit—the key to higher performance is not in pipeline tweaks. The future for high-performance processor designs is a better superscalar processor organization that can exploit out-of-order instruction issue and execution.

Out-of-order techniques are important because instructions tend to depend on previous instructions for results. Some instructions must be executed in a serial order because of the way they depend on—or conflict with—each other. Without out-of-order issue and execution, instruction dependencies force the resources of an aggressive superscalar implementation to sit idle more often than necessary.

Pentium does not permit general out-of-order issue or execution. The M1 design goes much farther, allowing out-of-order issue and completion when one pipe is stalled due to a cache miss, but it too is limited by its "pipeline-centric" organization. That is, the instruction fetching, decoding, and execution resources are all connected together in a more-or-less synchronous pipeline.

What is needed for more advanced implementations is an asynchronous, decoupled organization that separates execution resources from each other and from instruction fetching. The instruction-fetching hardware takes care of following the flow of control and feeding operations to the execution units. The execution units maintain queues of waiting operations and execute them when all operands are available. Thus, the execution side of an advanced superscalar processor acts like a dataflow machine—monitoring operand availability and "firing" operations when all needed operands become valid.

Some of the latest superscalar microprocessors implement this decoupled organization, most notably the Power2, PowerPC 604, and NexGen 586. Before discussing advanced superscalar organizations, we will examine a couple of techniques—register renaming and branch prediction—that are key to realizing maximum benefit from a decoupled superscalar organization.

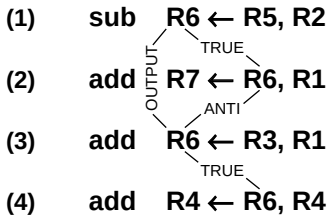

Figure 1. This short instruction sequence illustrates the three types of register data dependencies.

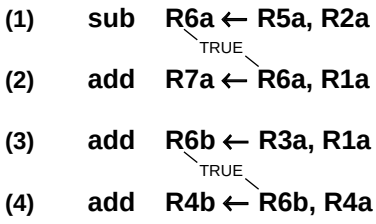

Figure 2. In concept, register renaming transforms the instruction sequence of Figure 1 into this sequence with renamed registers.

Dependencies Reduce Overlap

There are three kinds of operand data dependencies to consider: true dependencies, antidependencies, and output dependencies.

True dependencies (also called read-after-write or r-a-w dependencies) reflect the necessary data flow of the program. In Figure 1, instruction (2) has a true dependence on instruction (1) because (2) uses the result of (1). Clearly, (2) cannot be executed until the result of (1) is available.

There is nothing a processor design can do to eliminate true dependencies. The performance-limiting effects of true dependencies can be mitigated with short operation latencies (e.g., single-cycle ALUs) and data-forwarding logic (to deliver the result of an ALU computation directly to the ALU inputs instead of waiting for the result to propagate through the register file).

Antidependencies (also called write-after-read, or w-a-r dependencies) result from reusing registers. In Figure 1, instruction (3) has an antidependency on instruction (2) because (3) comes after (2) and (3) computes a result into one of the source registers of (2). Clearly, (3) cannot be executed before the operands for (2) are available. Once the operands for (2) are fetched, however, (3) can be executed even if (2) has not yet been.

Output dependencies (also called write-after-write, or w-a-w dependencies) occur when instructions compute results into the same register. In Figure 1, instruction (3) has an output dependency on instruction (1). If (1) were executed after (3), later uses of R6 would fetch an old, stale value computed by (1) instead of the value computed by (3).

Data dependencies have the effect of enforcing a strict ordering on instruction execution, which is counter to the goal of out-of-order superscalar execution. In Figure 1, the three kinds of dependencies just discussed require that the four instructions be executed serially; as it stands, no simultaneous, superscalar execution is possible for this sequence.

Register Renaming Breaks Dependencies

Register renaming can be used to eliminate false (anti- and output) dependencies, thereby creating more opportunities for out-of-order execution. With register renaming (and sufficient instruction lookahead), it is possible to execute the four instructions in Figure 1 at a rate of two instructions per cycle. (This is a contrived example: note that it is also possible to execute these four instructions in two cycles—the two pairs (1),(2) and (3),(4)—with a single three-input ALU.)

Conceptually, register renaming assigns a unique register to the result of every instruction that writes to a register and causes subsequent instructions that use a result to reference the renamed register instead of the original register. This technique eliminates output dependencies, since no two instructions write to the same register. Register renaming also eliminates antidependencies because a given instruction cannot overwrite the input operands of an earlier instruction.

In effect, register renaming would change the instruction sequence of Figure 1 to that shown in Figure 2. The renamed sequence exposes the fact that instruction (3) can be issued and executed in parallel with either instruction (1) or (2). Thus, since these are single-cycle integer operations, a superscalar processor with register renaming and at least two integer execution units would be able to execute the four-instruction sequence in as few as two cycles.

In practice, the hardware that implements renaming reuses register names as appropriate. When no more references to a renamed register are possible (because the program overwrites the original register with a new value and the value in the renamed register is not required for misprediction recovery, see below), that renamed register can be used again. Thus, in Figure 2, after instruction (2) executes (actually, after it has fetched its operands), R6a could be used again.

It is interesting to note that traditional register-allocation algorithms used in high-level language compilers tend to create instruction sequences rich in anti- and output dependencies, because traditional algorithms attempt to use as few registers as possible to hold all the program's register values. From the compiler's point of view, it makes sense to cram as many values into registers as possible to reduce the chance of running out of registers. The x86 architecture—with only eight general-purpose registers—demands stingy allocation.

Newer compilers for superscalar RISC implementations use different register-allocation strategies that avoid as many dependencies as possible. Unfortunately, the x86 has so few registers that better algorithms are ineffective, and legacy binaries (compiled with older register-allocation algorithms) will always be a concern for any successful architecture.

Thus, a superscalar processor with out-of-order issue and execution needs register renaming to expose the instruction parallelism hidden behind false dependencies. This is especially true for implementations of register-poor architectures like the x86.

Existing implementations with register renaming include the PowerPC 603 *(see **071402.PDF**)* and 604 *(see **080501.PDF**)*, the Power1 (MPR 8/21/91, p. 10) and Power2 *(see **071301.PDF**)*, and NexGen's 586. The Power1 and Power2 implementations rename registers only on floating-point loads; the other chips implement renaming more fully. The forthcoming M1 will also use renaming, and AMD's K5 and Intel's P6 superscalar x86 implementations should add to the list.

Implementing Register Renaming

Register renaming can be implemented in a number of ways, but conceptually it is simply a mapping from a small logical (architectural) register space onto a larger physical (implementation-dependent) register set. The size of the logical register set is fixed by the architecture, but the physical register set is sized to match the out-of-order capabilities of the implementation.

The concept of register renaming adds a level of indirection between the register numbers specified in an instruction and the register file address inputs. This level of indirection can be implemented with a small memory that produces a physical register number when it is presented with an architectural register number. Figure 3 shows the idea.

In practice, register renaming hardware may be more complex than this. Depending on the implementation, the renaming mechanism may have to provide tags in lieu of register values when a renamed register value has not yet been computed. One of many possible implementations, the reorder buffer, is discussed below.

Register renaming serves two purposes in an out-of-order superscalar machine: to eliminate false dependencies and to support speculative execution. Speculative execution—the result of branch prediction—requires a mechanism to recover quickly from incorrect speculation. Register renaming allows speculatively computed values to be discarded simply by changing the architectural-to-physical register mapping.

Branch Prediction Finds Parallelism

With the ability to fetch several instructions per cycle and with register renaming to expose the hidden parallelism in the instruction stream, an aggressive superscalar implementation needs a large supply of available instructions for issue and execution. If true dependencies stall one or two instructions, the potential of the superscalar implementation will be wasted unless the instruction-fetch logic can look far ahead in the instruction stream.

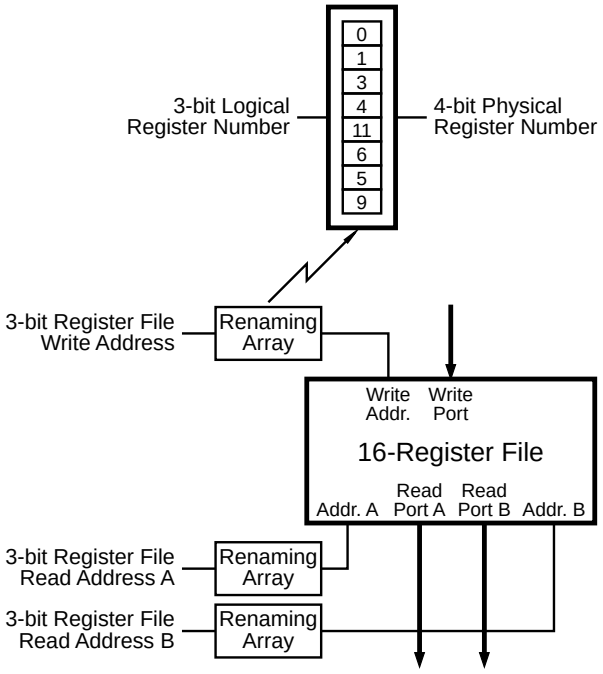

Figure 3. Register renaming creates a level of indirection between the register file and the register addresses provided by instruction decode. In this example, an eight-register architectural space is remapped to a sixteen-register physical space.

Unfortunately, on average, branches occur every four or five instructions, and many branches are conditional. To keep looking ahead for independent instructions, the instruction-fetch logic must predict the outcome of conditional branches (because the instruction that determines the outcome of the branch has probably not been executed when the branch is encountered).

The problem is not that branch prediction is difficult—Pentium and other microprocessors already implement sophisticated branch prediction—but that branch prediction is not 100% accurate. Thus, when a branch is mispredicted, aggressive look-ahead instruction-fetch combined with the dependency elimination of register renaming results in the execution of instructions that should not have been executed. In fact, an aggressive implementation may require that several conditional branches be predicted to keep the execution units busy.

In an implementation that allows up to four predicted branches, the processor can actually execute instructions from up to five different instruction streams. If the first branch was mispredicted, then many instructions that were executed should not have been.

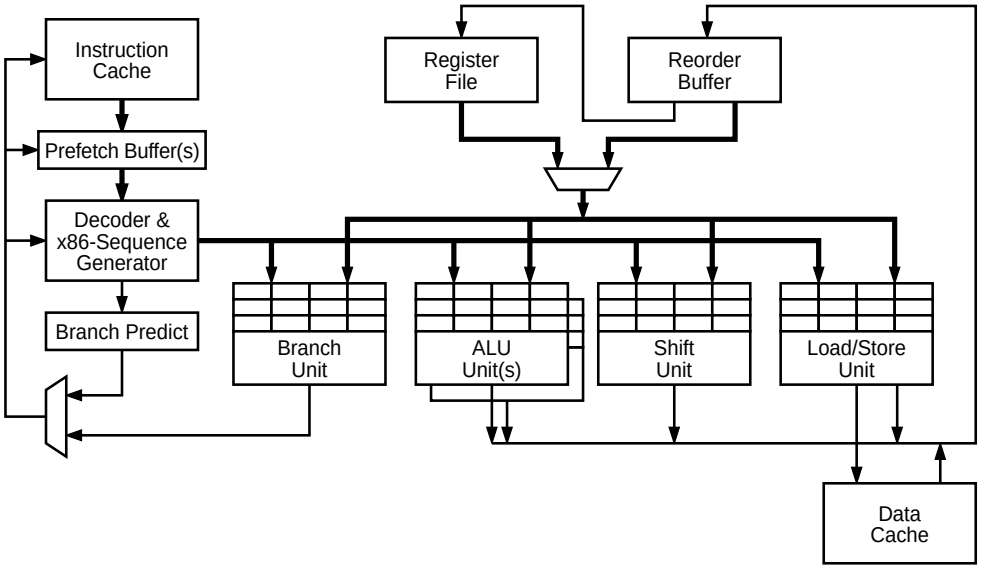

Figure 4. The high-level organization of a decoupled superscalar x86 processor. The instruction fetching and dispatching logic takes care of following branches and converting complex x86 instructions into sequences of simpler internal operations. The execution units queue operations and execute them as operands become available. The reorder buffer helps implement register renaming and recovery from mispredicted branches.

Given out-of-order execution and branch prediction, the processor must have a method to recover from mispredictions by undoing the effects of those instructions that should not have been executed, and the recovery method must be fast enough that the processor can begin fetching and executing instructions from the correct program path with minimum delay.

Register renaming can help implement the needed recovery mechanism by providing storage for the results of look-ahead (speculatively executed) instructions and, as explained below, by providing a way to discard the results if the instructions should not have been executed.

Reorder Buffer for Renaming, Prediction

One of the many ways to implement register renaming and fast recovery from mispredicted branches is with a reorder buffer [Johnson91]. A reorder buffer is a true FIFO structure that implements a central repository for information about completed and uncompleted computations.

As an operation is dispatched to an execution unit, it is allocated an entry at the top of the buffer. When the result of the operation is computed, it is written into the entry in the reorder buffer, not into the register file. Subsequent references to this result register get the needed value from the buffer, not from the register file.

The entry at the bottom of the reorder buffer FIFO is a candidate for writing its result to the register file. If the entry at the bottom has no possible pending exceptions, such as a page fault for a load or a possible misspeculation as a result of a mispredicted branch, and its result is available, then the result is written to the register file. Otherwise, the entry stays at the bottom of the FIFO until all possible exceptions are cleared or one is signalled.

Thus, as instructions are dispatched, some operands are provided by the register file and some are provided by the reorder buffer. When a register operand is needed but not yet computed, a tag identifying the reorder buffer entry is given instead. The tag is then used to identify a needed operand as soon as it is computed.

A key feature of the reorder buffer is that it can be emptied in a single cycle when a mispredicted branch is detected. Since the register file contains only computed results that are guaranteed to be "safe," the contents of the reorder buffer can simply be invalidated—much like clearing the valid bits in cache or TLB entries. Any results produced by speculative execution are thus dropped, and the processor can begin executing instructions immediately from the correct path.

The biggest drawback of the reorder buffer is that it requires associative lookup to find the most recent entry that matches a register number, which makes the reorder buffer expensive in terms of die area. Although only a modest number of entries are required (16 in the example machine in [Johnson91]), the associative lookup requires lots of comparators and wiring, but careful design should limit its impact on basic processor cycle time.

For an x86 implementation, the reorder buffer must also mark x86 instruction boundaries, since multiple reorder buffer entries are needed to store intermediate results for a single complex instruction.

Aggressive x86 Superscalar Organization

Figure 4 shows a high-level block diagram of an aggressive, decoupled superscalar x86 processor organization. The instruction fetching, decoding, and dispatching hardware operates independently of the execution resources. The fetching hardware takes care of program flow, including branch prediction.

The execution resources are decoupled from the fetching hardware by reservation stations. Each execution unit has a reservation station to buffer waiting operations. The reservation stations monitor the result buses to detect operands that are needed by waiting operations. Since operations execute as soon as their

operands are ready, operations waiting in reservation stations can be executed out of order.

The reorder buffer supplies the most recent operands to speculatively issued instructions, writing them to the register file when they are guaranteed correct.

Executing x86 Instructions

The execution units of the decoupled organization shown in Figure 4 naturally accommodate the instruction semantics of RISC instructions, which do essentially one thing: e.g., they add, shift, load, store, or branch, so each RISC instruction encodes one operation that gets dispatched to one execution unit.

Most x86 instructions are not as simple, but the superscalar core in Figure 4 can be just as effective for x86 instructions. In fact, an organization similar to that of Figure 4 is implemented by the NexGen 586.

To make use of the independent execution units for x86 instructions, the instruction-decoding logic simply dispatches multiple operations that implement the semantics of the x86 instruction. For example, an x86 register-to-register ALU operation would cause only one internal operation to be dispatched to an integer execution unit. An x86 memory-to-register ALU instruction would cause two internal operations to be dispatched: a load and an integer ALU operation. The integer ALU operation would have a true dependency on the load, so it would wait until the load completes. Because of the decoupled organization, however, operations from other x86 instructions could proceed while the ALU operation waits for the load.

An x86 register-to-memory ALU instruction might dispatch three internal operations: a load, an ALU, and a store (it is also possible to simply have a "load-operate-store" execution unit). The store would have a true dependency on the ALU operation, and the ALU operation would depend on the load. As before, these steps would not execute concurrently with each other, but they could execute along with independent steps from other x86 instructions.

Thus, the decoupled organization allows "microcode" steps from two or more x86 instructions to execute concurrently. This strategy provides a sort of CISC-to-superscalar-RISC translation on the fly. The microcode steps of several x86 instructions can execute with maximum concurrency as allowed by available execution units and true dependencies (register renaming eliminating the false dependencies). This is very different from the operation of Pentium, which overlaps only simple x86 instructions that can be handled directly by its pipelines.

The logic to perform this on-the-fly translation does not affect the execution side of the machine. It may require an extra stage of logic in the pipelined fetching, decoding, and dispatching hardware, which will reduce performance on mispredicted branches but will otherwise go unnoticed.

With Really Free Hardware...

In the (not-too?) distant future, it may be feasible to implement a processor that follows both paths of several conditional branches instead of simply predicting the branch outcomes and following the likely paths. By following both branch paths, useful work is guaranteed and no backtracking need be done to recover from a mispredicted branch. This technique was used in the IBM 370/168 and 3033.

To pursue both paths, it is possible to dedicate a copy of the processor core for each path. Since an aggressive superscalar processor can encounter several additional conditional branches while waiting for the determination of one conditional branch, it would be necessary to have perhaps eight or even sixteen copies of the processor core to follow all possible paths while waiting for the resolution of the first conditional branch.

Copying the hardware of an aggressive superscalar processor core eight or sixteen times is clearly very expensive. Better ways to get most of the same performance benefit include improved branch prediction (better hardware algorithms and more compiler support) and simply issuing instructions from both paths to the same execution units. As with the hardware described in this article, register renaming and a reorder buffer take care of marking instructions appropriately so each gets the right operands and so mispredicted instructions can be cancelled.

Although duplicating the processor core to follow multiple branch paths is not the most efficient use of hardware, chip technology a decade from now may permit such a brute-force implementation. If this type of design improves performance for important applications, it may someday make sense from a business perspective.

Superscalar Dispatch at the x86 Level

NexGen's 586 operates essentially as described above, using superscalar techniques to issue and execute internal operations, which NexGen calls "RISC86" instructions. What it does not do, however, is decode more than one x86 instruction per cycle.

The same parallel decoding hardware that allows Pentium to recognize two simple x86 instructions in a single cycle could be added to our decoupled superscalar core and thus to the 586. AMD's K5 will probably have an organization similar to the 586 coupled with the ability to decode and dispatch more than one x86 instruction per cycle. Intel's P6 and the NexGen 686 will likely do the same.

Pentium's design restricts which x86 instructions can be simultaneously issued by its decoding hardware. As the degree of superscalar issue at the x86 level increases—the K5 is a four-issue machine—it is conceivable that the restrictions could get more severe, which

would limit superscalar issue to a smaller subset of the x86 instruction set. Any new restrictions would have to be incorporated into compiler code-generation algorithms to ensure that the superscalar capabilities of advanced x86 processors are fully exploited.

Market forces, however, may prevent new issue restrictions from being implemented. It would be a market disadvantage to achieve less than maximum performance on existing binaries compiled to exploit Pentium's superscalar dispatch. AMD says that the K5 will have issue restrictions no more severe than Pentium's.

The Future Looks Complex

Once a decoupled processor organization is adopted, it is possible to expand it almost indefinitely. More execution units, more cache ports, bigger reservation stations, a larger reorder buffer, larger branch prediction tables, more operand buses, more internal operation dispatch buses, greater degrees of x86 superscalar decoding, and so on, can consume almost arbitrary amounts of hardware, whether on one chip or many. In the end, it is possible to have multiple copies of the processor core following each possible branch path (see sidebar), eliminating all backtracking.

There are two fundamental limits, however, that may be reached before such drastic measures are taken. First, the complexity of the overhead logic that orchestrates the concurrency may limit cycle time so much that a simpler design with a faster clock would deliver higher performance. Second, and more likely, most programs may not have sufficient parallelism available to keep busy a superscalar core that has, say, 16 execution units. To exploit machines with such capabilities may require a change in software technology. Or perhaps we will discover that predicting and following 16 consecutive branches provides enough parallelism.

Relative Implementation Costs Dropping

The decoupled, highly concurrent processor organizations discussed here are complex. They require a tremendous amount of logic to perform dependency checking and associative lookup and a tremendous amount of wiring to connect all the elements together with sufficient bandwidth to keep the execution units from waiting for operations and operands.

Much of the complexity, however, is in the execution side of the machine, which remains largely the same whether the x86 or a RISC instruction set is being executed. The x86 architecture complicates the execution side of the machine somewhat by requiring, among other things, four-input address adders, hardware to track multiple copies of the condition-code register, and the need to deal with the increased frequency of memory references (compared to RISCs), but a large part of the impact of the x86 is isolated in the instruction fetching, decoding, and dispatching hardware.

There is some empirical evidence to counter the argument that advanced x86 implementations are paying a declining penalty for their instruction set. In its day, the 486 was roughly—some observers would say only very roughly—comparable to contemporary RISCs in terms of die size and technology. Pentium uses more transistors and more die area and achieves a lower clock rate than its contemporaries. The rumor is that P6, to achieve competitive performance, will resort to a multichip module to accommodate off-chip caches (rumors differ over whether the off-chip caches are level-one or level-two).

If this is a trend, then it seems clear why Intel is taking steps to develop a new processor architecture: high-performance x86 implementations, at least from Intel, are getting too expensive and can never reach leadership performance. Another possibility is that Intel is trying to address a different market with P6. Perhaps enhanced Pentiums will be Intel's next-generation mainstream x86; perhaps P6 is initially intended for servers and is not meant to replace Pentium in the desktop market right away.

The bottom line is this: although RISCs have implementation advantages over the x86 for an advanced, decoupled superscalar organization, the advantages are not as compelling as they were for simple pipelined implementations.

x86 Performance Increases Will Continue

Again, as was stated in the beginning of this article, it seems strange that Intel would abandon the x86 just when implementation technology seems to be mitigating the overhead of the x86 instruction set. Perhaps, though, Intel and HP have identified—or hope to identify—a combination of software and hardware technologies that will achieve price/performance levels unattainable by implementations of traditional instruction sets.

Other possibilities abound. Perhaps Intel simply wants to re-establish a proprietary position. Perhaps Intel simply desires to instill fear, uncertainty, and doubt in the market. Maybe Intel wants to prevent, once and for all, the adoption of RISC alternatives—at least one RISC competitor is eliminated by swallowing PA-RISC—and ruin the prospects of its x86 competitors.

Regardless of Intel's actions, the x86 will continue to advance in performance through the efforts of other x86 microprocessor vendors. Although its implementations will not overtake the price/performance of RISC or VLIW (or whatever) implementations, the x86 architecture will continue to spawn competitive implementations that will yield satisfying performance gains for loyal customers. ♦

