VOL. 10, NO. 10

New Instruction Sets Are Coming

But Current Architectures May Be Able to Keep Pace

by Brian Case

In 25 years, the microprocessor has risen meteorically from its humble beginnings as the heart of a simple desk calculator. Fueled by the exponential improvement in fabrication technology, instruction throughput and system capabilities have increased astoundingly. Though some mainstream operating systems and applications do not take full advantage of them, modern high-end microprocessors provide supercomputer capabilities, including hundreds of MIPS, gigabytes of address space, and support for concurrent processes in protected virtual-memory address spaces.

Although instruction sets are evolving to support multimedia data types (video and audio data streams and 3D graphics), the basic nature of an instruction has not changed since the microprocessor's birth two and a half decades ago. Microarchitectures have changed radically, but individual instructions are still geared to implementations that fetch, decode, and execute each instruction in three separate, serial steps. Parallel microarchitectures are now capable of overlapping the fetch, decode, and execution phases of many instructions. As a result, the decoupled, autonomous units of today's high-end microprocessors can be operating on one or two dozen or more instructions simultaneously, and, because of multiple branch prediction, the instructions can be taken from widely disparate sections of a program.

From the point of view of hardware, microprocessor instructions seem to deliberately hide information about parallel execution opportunities; substantial hardware is needed to discover when instructions can be executed simultaneously. Much of the information about potential parallelism is available at compile time, when sequences of machine instructions are created.

With the implementation overhead of powerful superscalar processors growing rapidly, some designers—apparently including those at HP and Intel—think it is finally time for new architectures. The alternative—asking microprocessor designers and hardware to continue to shoulder the burden of extracting parallelism—has historical precedent on its side: wait long enough, and circuit technology provides enough resources to solve any problem.

The History of High Performance

The history—and near future—of commercial high-performance microprocessors can be organized roughly into four epochs. The epochs are defined by the interplay between integrated circuit density and processor implementation techniques. A transition from one epoch to the next occurs when circuit technology is just good enough to permit the use of one or more major new implementation techniques.

In the first epoch of microprocessor design, circuit technology was barely sufficient to permit the fabrication of a basic single-chip microprocessor. The second epoch of microprocessor design was ushered in on the coattails of RISC architectures. The transition to the third epoch began several years ago with the introduction of the first superscalar implementations.

The microprocessor industry is just beginning the transition to the fourth epoch, which is characterized by implementations that decouple instruction fetching, decoding, and execution and can execute internal operations speculatively and out of order. Very soon, every high-performance microprocessor will need a decoupled speculative out-of-order internal organization to be commercially viable.

In the fourth epoch, microprocessor designers can have it all: decoding for a complex instruction set, a rich set of execution resources, good dynamic branch prediction, register renaming, a large window (reservation station and reorder buffer) of pending internal operations, large on-chip caches, high-frequency operation, and high-pin-count packages. As the epoch progresses, chip vendors can exploit new levels of circuit density to achieve new performance levels by simply increasing the sizes of all the on-chip elements.

The Fourth Epoch: Time for a Change?

The problem with decoupled speculative out-of-order implementations is that a small increase in the capacity of the parallel-execution hardware requires a large increase in circuitry for the implementation. Because of this nonlinear relationship, some architects and designers are appealing for new instruction sets that can simplify the implementation of superscalar hardware. Based on the first three epochs, during which advancing technology squelched calls for simplification, it is easy to dismiss these appeals as another cry of wolf. Three trends, however, might contribute to the success of a new architecture.

First, the dominance of x86 may allow Intel to simply decree the success of a new instruction set. If Intel can provide a new chip that simultaneously boosts the performance of existing x86 code and offers dramatic performance rewards for shifting to a new instruction set, the new architecture might become pervasive very quickly.

Second, if the shift to virtual-machine-based languages—Java, that is—succeeds, the underlying instruction set would be hidden. With a virtual-machine-based environment, instruction-set incompatibilities are invisible, and the only apparent difference to the user is the performance and cost of a complete platform. This is the situation today: x86 processors do not, by themselves, offer the best price/performance, but when integrated into a complete platform, nothing else can compete.

Third, traditional increases in circuit densities might begin to slow down in the next few years, or at least the cost of maintaining the historical pace of advancement might become too high. Under these conditions, chip vendors may be required to make instruction-set changes to maintain the pace of price/performance improvement that end users have come to expect. A new instruction set might allow microprocessor vendors to do more with the same.

The Case for New Instruction Sets

Current state-of-the art superscalar implementations are capable of executing four instructions in a cycle, given perfect conditions. As circuit technology improves, these implementations will get smaller, which will allow operation at higher clock rates and reduce manufacturing cost.

Eventually, process technology will allow higher-degree superscalar designs to be built cost-effectively. Consider an eight-way superscalar design. At eight instructions per cycle, the optimum execution quantum of the implementation—eight instructions—is actually bigger than the average size of a basic block (a branch-free, straight-line sequence of instructions). Using typical instruction frequencies, this eight-instruction quantum would contain one or two branches and two or three memory references.

Branches and memory references are known to be the main sources of significant pipeline stalls during normal program execution. In an eight-way machine, the effect of even one stall cycle is magnified by a factor of eight. Consider that a 500-MHz processor has a 2-ns cycle time. Given that main memory has a latency of at least 50 ns, an eight-way processor that must wait for a main-memory reference can waste literally hundreds of instruction-execution opportunities. Similarly, a mispredicted branch can cost several cycles of recovery time, again wasting an impressive amount of internal performance.

Thus, two of the keys to making use of the tremendous performance potential of a high-degree superscalar design are eliminating (or predicting with 100% accuracy) branches and reducing the latency of loads and stores. While branches and memory references will always be sources of grief for designers, techniques to address these and other issues are available.

New Instructions: Architect's Shopping List

A wide range of nontraditional instruction-set features can be used to reduce the cost of a high-performance microprocessor implementation. Examples of most of these features can already be found in existing chips, as Table 1 shows, but no single architecture uses them all.

Predicated execution. Predicated execution, or conditional execution, helps eliminate conditional branches. With predicated execution, each instruction specifies the condition under which it will be executed. For example, instructions in the ARM RISC architecture (see MPR 12/18/91, p. 11) use four bits to specify a combination of condition codes that must be true for the instruction to take effect.

Characteristic	Current Architectures
Predicated execution	ARM, Trimedia
More registers	Titan, 29000, Trimedia
Speculative loads	SPARC V9, Trimedia
Data prefetch hints	SPARC V8, PA-RISC 2.0, Alpha
Instruction prefetch hints	SPARC V9
Static branch prediction hints	PowerPC, PA-RISC 2.0, MIPS
Multiway branches	none
Dependency hints	none

Table 1. Many of the advanced features proposed for a next-generation instruction set are available in today's processors.

A more general form of predication is called guarding and is implemented, for example, in the Philips Trimedia VLIW microprocessor *(see* **091506.PDF***)*. An operation in a Trimedia instruction specifies one of 128 general-purpose registers as the guard; if the LSB of the guard register is 1, the operation takes effect; if the LSB is 0, the operation is effectively a NOP.

Guarding is more powerful than the ARM technique because a large number of arbitrarily complex guard conditions can be precomputed and then used to guard a large block of branch-free code.

While it is true that an instruction with a false guard condition wastes an issue slot, a guarded instruction cannot disturb the flow of other instructions, unlike a conditional branch. Also, a guarded instruction does not consume an entry in the branch-prediction unit and does not have the potential of being mispredicted.

On the other hand, a machine with guarding might need a guard-prediction unit in addition to a branch-prediction unit to prevent guarded instructions from waiting while guard conditions are computed.

More registers. One of the biggest sources of complexity in modern superscalar implementations is the logic that supports speculative and out-of-order execution. Microprocessor designers are already starting to complain that this hardware does not scale well to higher orders of multiple dispatch and issue.

The logic for register renaming, which helps eliminate false dependencies between instructions, is a part of the hardware that supports speculation and out-of-order execution. False dependencies arise because too few architectural registers are available to the compiler to allow assignment of a different register to each unique result value within a basic block. With enough architectural registers, an optimizing compiler can schedule instructions so register renaming is not needed.

Loop unrolling is an optimization that can benefit from a large register file. Unrolling can eliminate conditional branches and create a long basic block that can be efficiently scheduled by an optimizing compiler. Static scheduling (code reorganization), however, requires storage to hold temporary results. Without enough registers, loads and stores are needed to save temporary values. These loads and stores naturally reduce the benefit of the unrolling.

Another use for temporary storage in a superscalar processor is in support of general speculative, out-of-order execution. The amount of temporary storage needed is directly related to the amount of speculation allowed.

Instead of requiring hardware to discover instructions that can be speculatively executed, an optimizing compiler can generate the instructions along the two possible paths of a conditional branch so they use disjoint subsets of a large register file. To support multiple branch speculations, the compiler can ensure that all instructions along several paths use disjoint subsets.

If the compiler has enough registers available to guarantee that results generated along the paths will not collide, an implementation is free to speculatively execute instructions down all paths, and no special tracking and resolving hardware (e.g., a reorder buffer for register renaming) is required to prevent the instructions on the paths from interfering. When the branch conditions are resolved, the code down the correct paths naturally used only the appropriate values, and the code that begins after the end of the correct path will naturally use the correct values.

Eventually, the registers used in one of the disjoint subsets must be reused. Therefore, to fully support this kind of static speculation, a synchronization instruction might be needed to stop the processor from proceeding past the end of a speculative section of code until all speculated branches needed to reach that point are resolved.

Speculative loads and data-prefetch hints. As the internal performance of microprocessors increases, the disparity between register-to-register operations and memory operations becomes more apparent. To prevent a processor from wasting potential performance, the impact of memory references must be reduced as much as possible.

Reordering register-to-register instructions ahead of conditional branches is easy, given enough temporary storage to hold the speculatively computed result. Load operations, however, as traditionally defined, can have irreversible side effects such as causing a page fault or protection fault.

A new architecture could define a speculative load operation that would not generate any side effects if the speculation were eventually identified as incorrect. This well-behaved load operation could be moved ahead of conditional branches by either a compiler or speculative-execution hardware.

The Trimedia VLIW architecture includes speculative loads. Speculative loads allow a processor to continue executing past load operations, but if an instruction that uses the result of the load is encountered and the result is not available, the processor may be forced to stall while it waits for the value from cache or main memory. Data-prefetch operations would allow a compiler to notify a processor's memory system of the address or address range of needed data well ahead of an actual reference to that data, reducing or eliminating such stalls.

As superscalar processors increase in degree, the performance loss from each cycle of stall increases. A data-prefetch operation issued early enough can help hide a long-latency fetch from a distant cache or main memory. If a speculative load or a data-prefetch hint can save even one or two cycles in a loop, a potentially large performance gain can be reaped.

Instruction-prefetch hints. Giving the processor an instruction-prefetch hint is not the same as speculatively executing a conditional branch. An instruction-prefetch hint is used to tell the instruction-fetching hardware the address of a new flow of control well ahead of a branch; speculatively executing a branch is valuable only if the instructions at the speculated target can be fetched immediately. An instruction-prefetch hint is one way of helping to make sure the instructions at a branch target are available from the first-level cache when the branch is executed.

As with data-prefetch hints, notifying hardware well in advance of the need for a block of instructions can hide a long-latency fetch from caches or memory. Again, saving even one or two cycles, let alone dozens, can bring a processor significantly closer to its performance potential.

Branch-direction hints. Dynamic branch prediction, as implemented in most modern microprocessors, works well most of the time. Pathologically bad cases do arise, however, and a compiler may know the likely outcome of the first execution of a conditional branch before dynamic prediction has collected a branch history. An implementation may benefit by combining dynamic branch prediction with an instruction set that encodes static branch-direction hints.

Multiway branches. No, architects are not advocating a direct implementation of the FORTRAN computed-branch construct. Multiway branches are useful in an architecture that uses predicated execution to combine the two paths of a single conditional branch into a single branch-free sequence of instructions. In the case when the two paths themselves end with conditional branches, there can be four possible paths out of the combined stream of predicated instructions. A multiway branch at the end of the stream is an efficient way to transfer control to the correct path.

Conceptually, a multiway branch would specify two or more branch targets and a way of selecting the proper target. The specific semantics would be determined by balancing the needs of optimizing compiler analysis algorithms and implementation simplicity.

One possibility would be to specify two source registers containing indirect target addresses and a third source register with a selection condition. Such a branch could have three targets: two explicit destinations and the fall-through path. Another possibility is a 64-bit instruction format that would allow three 16-bit relative offsets plus a source register whose value would select one of the three offsets or the fall-through path.

Dependency hints. Part of the hardware that supports speculative out-of-order execution is logic that checks for dependencies between pending operations in the reservation station(s). Dependency information in the instruction stream can be provided by an optimizing compiler and could simplify implementations by eliminating the need for hardware to recompute interoperation dependencies.

Effect on Instruction Sets

An instruction set combining most of the traditional RISC tenets with the above ideas would be the basis for a powerful architecture that would increase the efficiency of a processor implementation, but some sacrifices would be required.

One sacrifice would be increased dependence on compiler technology. To even reasonably exploit a machine incorporating all the ideas listed above would require a relatively sophisticated compiler. This is not a severe deficiency, however, because a sophisticated compiler is already required to even approach the performance potential of existing high-end microprocessors.

Perhaps the biggest sacrifice would be compromises in the instruction format. To combine three-address, register-to-register operations (a basic RISC tenet) with a large number of registers and guarded execution would require four bit-hungry register specifiers. With 128 registers, the four register specifiers alone consume 28 bits. To encode a reasonable number of operations would require more than 32 bits for basic arithmetic instructions.

One solution is to step backward to two-address, destructive operations. With one less register specifier, a 128-register machine would still have 11 bits in a 32-bit format for encoding basic arithmetic operations.

If instructions longer than 32 bits are acceptable, it might make sense to step backward in another way and create complex instructions that—don't faint—combine arithmetic operations and memory references. It is also possible to encode two arbitrary and possibly dependent arithmetic operations in a single long 64-bit instruction. The precedent for this can be found in an architecture proposed by compiler-researcher Bill Wulf several years ago. (SuperSPARC could internally cascade two dependent operations in a single cycle because its ALUs were disproportionately faster than its caches.) Another instruction that can benefit from a long format is the multiway branch.

In a machine with some instructions longer than 32 bits, it probably makes sense to allow two or three different instruction formats, perhaps 64 bits, 32 bits, and 16 bits. This concept is a logical extension from RISC-like architectures such as NEC's V800 *(see* **071406.PDF***)*, which intermixes 32- and 16-bit instructions. An architecture with multiple instruction sizes complicates the implementation of instruction fetching and decoding, but given current technology—and the Pentium Pro and AMD K5 as proof of concept—the complexity is not overwhelming. With just two or three power-of-two instruction sizes, the design challenges should be only a fraction of those encountered in an x86 implementation.

Multithreading Support Can Be a Big Win

Potentially one of the biggest parallelism-discovering wins is support for multiple threads of execution within a single program. By definition, separate threads are independent and therefore inherently parallel. By fetching instructions from separate threads, a processor can easily find parallelism and keep its execution resources busy. While one thread is stalled waiting for a load instruction to return a value from memory, the independent instructions from another thread can be executing.

Multithreading support requires, at least conceptually, multiple program counters, one for each thread. Thus, architectural support for multiple threads might benefit from some changes to the instruction set, but the changes are in the form of additional instructions, not modifications to the fundamental instruction semantics or formats. Thus, it should be possible to add multithreading support to existing architectures, but it may make more sense to add it to a new architecture where the integration into the instruction set and implementation can be as clean as possible.

The first implementation of the MicroUnity architecture *(see* **091402.PDF***)* has built-in multithreading to accommodate its extreme pipelining.

Existing Architectures Will Still Thrive

While the new features listed above are many and impressive, the current state of the art in superscalar design defines a clear path to steadily improving the performance of existing architectures over the next few years. Increasing the width of the fetcher and decoder, the size of the window (reservation station) of pending instructions, and the number of execution units should allow the processor to find more independent instructions and execute more instructions per cycle. It is true that significantly more hardware will be required to achieve each modest increment in performance, but circuit technology seems ready to provide the needed additional resources for at least two or three more generations.

Also, the appeal of binary compatibility with existing applications cannot be overstated. The entrenchment of the PC standard and the fact that customers continued to buy Sun machines over the past few years despite an embarrassing price/performance ratio support the costly development of compatible chips.

A New ISA: Coming in Your Next PC?

While samples of the Merced chip are not expected until 1H98, all indications are that HP and Intel are serious about releasing a new architecture. Since most of the non-x86 CPU vendors already have relatively new architectures and are fighting for market share, it seems unlikely that a new architecture will emerge from any of the RISC vendors. Thus, Merced represents the first real chance to integrate all these ideas into a single product.

There are, however, disturbing parallels between the arguments being made now for the new architectural features and the arguments that were used to "prove" RISCs would succeed over competing architectures.

First, the technical arguments in favor of a new architecture are couched in terms of better performance and simpler, cheaper implementations. Just as with RISC, these arguments are irrefutable. The problem is that, as with RISC, the differences in performance and implementation cost may not be significant enough. RISC promoters used to predict when competing architectures would run out of gas and used to draw performance graphs that had the slope of RISCs rising at a faster rate than for other architectures.

History has shown that gas is plentiful and that the performance curves are essentially architecture-independent. The question now is, "how much gas is left, and are the curves still architecture-independent?" If the x86 is really starting to run out of gas, then the new architecture has a chance at success.

Second, as evidenced by SPARC workstations and PCs in the late 1980s, performance alone does not drive buying decisions. RISC promoters used to advance the theory that the performance of RISC chips would spawn new applications that would make RISC-based computers irresistible to users. Unfortunately, it takes more than a 20% to 50% performance boost to create new application possibilities. The PowerPC-based Macintoshes did not deliver compelling new applications. A new architecture like Merced will probably need a performance advantage of a factor of three or more to enable compelling new applications.

While such a large performance advantage seems unlikely, Merced should have two advantages never enjoyed by RISCs. First, Merced will have the full attention of Intel, a force more powerful than any in microprocessor history. RISCs were, in general, promoted by upstarts that were often handicapped by lagging process technology. Second, Merced can offer a value proposition never tried by RISCs: the ability to deliver competitive performance on x86 binaries plus a boost on recompiled code. M

Brian Case helped design the architecture of the 29000 at AMD and is currently an independent consultant specializing in microprocessor design issues. He can be reached via e-mail at bcase@best.com.