VIEWPOINT

What's Next for Microprocessor Design?

Some Variant of Multiprocessing Seems Likely

by Brian Case

Over the past two decades, microprocessor designers have ceaselessly adopted existing advanced techniques, most of which were pioneered in mainframes and minicomputers. The most advanced technique in use today—decoupled superscalar organization *(see* ***081102.PDF****)*—was first implemented in mainframe CPUs. Cutting-edge microprocessors, however, are actually implementing decoupled designs that go beyond mainframes in sophistication. Furthermore, their single-chip implementations permit higher clock rates.

Instead of asking "What existing technique should we borrow next?" the microprocessor-design community suddenly finds itself asking "Is there anything left we haven't done yet?" For all intents and purposes, the answer is "no." When it comes to mainframes and microprocessors, the student seems to have become the teacher.

One technique that has been the backbone of the biggest mainframe CPUs has not been subsumed by single-chip microprocessors: multiprocessing. Multiprocessor systems based on microprocessors have been around for years, but it has never made economic or technical sense to build a general-purpose, single-chip multiprocessor. Are the economic and technical barriers finally low enough? Are single-chip multiprocessors the way of the future?

Of course, these questions can be given definitive answers only by active microprocessor designers, but we can ponder the possibilities even without inside information. (See also ***080605.PDF*** for another discussion of single-chip multiprocessor issues.)

One-Chip Multiprocessors: Pro and Con

Single-chip multiprocessors offer one major enticement to the microprocessor designer: the ability to increase the sophistication of a design with what is essentially a macrocell, cookie-cutter approach. With multiprocessing, a faster, more expensive microprocessor can be built by simply pasting down another instance of an existing processor core. Performance can be improved without increasing the clock rate or changing the core design. Single-chip multiprocessors leverage very pleasingly the huge investment required to design a modern uniprocessor core.

Unfortunately, designing a single-chip multiprocessor is not really as simple as plopping down processor cores. Designers must address complex issues such as processor synchronization and shared access to caches and external buses. Furthermore, the bandwidth and latency of microprocessor buses is already a limiting factor on performance; external interfaces may require rethinking and redesign to accommodate the demands of multiple on-chip processors.

Perhaps the primary downside to multiprocessors in general is the lack of software: there just aren't many programs that can take advantage of multiple processors. Before this situation can improve, a multiprocessor infrastructure must be built. The software industry needs programmers, compilers, and operating systems that are multiprocessor aware.

There are signs of change along these lines. Windows NT can use multiprocessors, and even Windows 95 supports parallel-execution threads within a single program. Pervasive glueless multiprocessing in next-generation processors, including Pentium Pro (the P6), combined with increasing OS support could spur the development of a multiprocessor infrastructure within a couple of years. That means CPU designers need to begin contemplating single-chip multiprocessors now.

If building single-chip multiprocessors is done by simply adapting and shrinking current multiprocessor system design techniques, then there is little to say about the future; we'll simply have to wait for a software infrastructure and advances in IC process technology. But what if single-chip multiprocessors could be more than simple adaptations of traditional multiprocessor system design? Are there other ways to organize multiple processors on a chip?

Multiprocessing for a Single Program

An inspired adaptation of replicating multiple processor cores to form a single logical processor is reported by Manoj Franklin in his 1993 Ph.D. dissertation from the University of Wisconsin, Madison. This organization, called a multiscalar architecture, places multiple uniprocessor cores in a ring, as shown in Figure 1. Each unit is essentially a complete processor core with some extra logic to buffer and keep track of register values that propagate around the ring (counterclockwise in Figure 1). The following is a cursory description of multiscalar concepts; for a complete discussion, see Franklin's dissertation, Technical Report #1196, November 1993 (now available on the Web at *www.cs.wisc.edu:80/tr/uwmadisoncs: cs-tr-93-1196).*

Franklin's multiscalar machine executes multiple threads of control from a single program. The global control unit (GCU) dispatches contiguous blocks of instructions, called tasks, to each of the processing units. The single program being executed has been sliced up into tasks by the compiler. A task is nothing more than a sequence of instructions and can be as simple as one or a few instructions in a basic block or as complex as an entire loop. Tasks are chosen to be as independent as possible.

To dispatch a task, the GCU simply hands an initial program counter value and a special bit-vector mask to a processing unit; the masks help the units locate the most up-to-date copy of each register. The GCU uses the head and tail pointers to manage the processing units as a circular buffer.

The global control unit is capable of dispatching a new task each cycle. It does so by predicting the flow through the task list (with branch prediction) and speculatively dispatching tasks. At any given time, all tasks but the first (i.e., the least recently dispatched task at the tail of the list) are executing speculatively.

Each processing unit is free to execute the instructions in its task as quickly as possible; each unit can be a simple pipelined processor or a sophisticated superscalar processor. Register values needed because of dependencies (e.g., one task needs a value computed by an earlier task) are communicated through the ring-like communication path connecting the execution units. Given suitable tasks, all of the execution units are busy running their little parts of the whole program concurrently, and the amount of communication overhead is small. Typically, each processor takes a bite out of the program that is at least several instructions long.

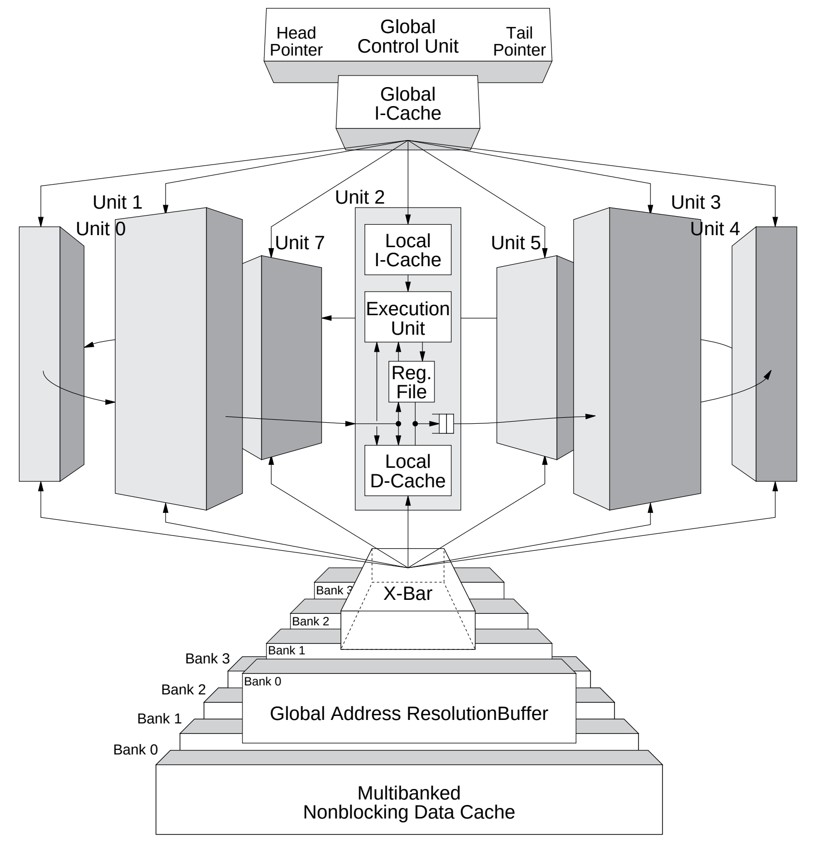

Figure 1. Block diagram of an eight-unit multiscalar processor. The execution units are traditional uniprocessor cores of any appropriate design (pipelined, superscalar, etc). The global control unit dispatches blocks of instructions ("tasks") to each unit.

Finding More Parallelism

This approach has some advantages. One problem for decoupled superscalar designs is that data dependencies can limit available parallelism; when one instruction depends on the result of a previous instruction, the second must wait for the first to be executed. One solution to this problem is a large window of pending instructions (the reservation station and reorder buffer in Pentium Pro) *(see **090202.PDF**)*. As the window size increases, the probability of finding independent, ready-to-execute instructions increases, but a large window also increases implementation complexity.

The multiscalar organization attacks the problem of dependencies in two ways. First, the compiler can collect a data-dependent sequence of instructions into a single task, which means that only one processing unit is tied up; the others can proceed with later, more parallel portions of the program. Second, since task sizes tend to be relatively large (several to tens of instructions), the multiscalar processor effectively implements a large window into the instruction stream, which helps it find maximum instruction-level parallelism.

It is possible to build a multiscalar processor that forms lists of tasks at run time, but most practical implementations would rely on a compiler to group instructions into tasks. Another possibility not mentioned in Franklin's dissertation is the use of a binary post-processor to build a list of tasks. A compiler is likely, however, to do the best job, given its knowledge of the program source code.

The processing units could use an existing instruction set. For example, it is possible to build a multiscalar x86 implementation, which would provide good backward compatibility along with benefits for new software applications.

Initial simulation results for multiscalar organizations show promise, especially for integer programs that are thought to have small amounts of parallelism. Perhaps more than anything, the multiscalar research

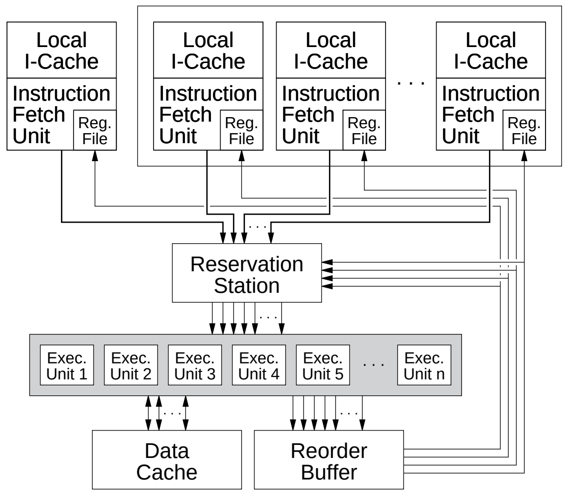

Figure 2. Decoupled superscalar organization enhanced to allow multiprocessing.

shows that a lot of parallelism can be found by looking far enough ahead into the instruction stream.

A multiscalar processor would be more complex than a traditional multiprocessor, but the multiscalar design has the advantage of being able to speed up the execution of a single program without explicit work on the part of the programmer. With traditional multiprocessors, multithreading can speed the execution of a single program, but the programmer is required to create threads explicitly.

Using One Processor for Multiprocessing

Another approach to multiprocessing does not use multiple processors at all. Instead, it builds on the strong foundation of a decoupled superscalar organization. As Figure 2 shows, this multiple-program-counter (which we'll call multi-PC) superscalar organization allows several instruction fetch units (IFUs) to dispatch instructions to a common pool of execution resources. Whereas a familiar decoupled implementation has just one I-cache/IFU/register-file block, the modified organization has several such blocks, which expands the pool of execution units, expands the sizes of the reservation station and reorder buffer, and increases the number of buses to carry operands and operations. As operations are dispatched, they must be tagged with an identifier unique to the originating IFU, so the reservation station and reorder buffer know how to forward intermediate operands from previous instructions and know which register file to write results into.

This organization requires a large number of execution units to accommodate a high rate of instruction issue. Since the reservation station contains operations from different programs or threads from one program, it is likely that there are always operations ready to execute. Consequently, the execution units will seldom sit completely idle, and the average utilization will be higher than with just one I-cache/IFU/ register-file unit.

The multi-PC superscalar design could be used to simulate Franklin's multiscalar organization, except that it lacks the inter-register-file communication ring. A similar communication path could be added, or a variation on the multi-PC organization with a single central register file might be devised.

While a multi-PC design looks attractive in a block diagram, it has one major drawback: a potential explosion in complexity. The designers of P6 and K5 have already remarked on the costs associated with routing operands and comparing tags to determine how to store and forward results. The implementation costs of the reservation station and reorder buffer are high in current designs; the multi-PC organization can only make those structures more costly.

In return for the complexity, however, this organization offers higher execution rates for single threads of execution that are inherently highly parallel. When only a single thread is executing (a worst-case scenario), the large number of execution units will result in very high throughput if that thread is highly parallel. In contrast, the worst-case throughput for the multiscalar design—when only one processing unit is able to make progress—is determined by the sophistication of the individual processing units. Since all the execution resources are divided statically among several processing units, each of the multiscalar processing units is less capable than the execution engine in the multi-PC design, which has a common pool of execution resources that can be dynamically allocated. Given a constant number of execution resources, the multi-PC design is more flexible.

There's a Multiprocessor in Your Future

In my view, some sort of single-chip multiprocessor makes good sense. Multiprocessing is the easiest way to increase instruction throughput. When will we see multiprocessor microprocessors? Only the chip vendors know for sure, but it's a safe bet that uniprocessors will rule for at least one more generation.

I suspect that single-chip multiprocessors, at least at first, will be shrunken versions of current multiprocessor systems, because the traditional multiprocessing paradigm is reasonably well understood and an infrastructure of operating system and hardware support seems to be under construction. But, as this article has shown, there are a couple of alternative approaches that deserve investigation, and others will probably be proposed. The multiscalar and multi-PC ideas show that it may be worthwhile to search for the holy grail: speeding up a single program with multiprocessing concepts. ♦

Will single-chip microprocessors be the next step, or will another technique be more popular? Let us know by sending e-mail to editor@mdr.zd.com.