Sun Reveals Secrets of “Magic”

New MAJC Architecture Has VLIW, Chip Multiprocessing Up Its Sleeve

by Tom R. Halfhill

If there were any doubts that VLIW has succeeded RISC as the most important influence on new microprocessor architectures, they vanished this month when Sun pulled the latest example out of its hat: MAJC (pronounced “magic”), the Microprocessor Architecture for Java Computing.

Sun says it conjured MAJC for the coming wave of networked information appliances—advanced set-top boxes, screen phones, automotive-navigation systems, home video-game consoles, and the like. There’s no technical reason that a MAJC chip couldn’t power a PC, workstation, or server, but Sun is pursuing the nebulous post-PC market for obvious business reasons: it’s more trendy, and the path of least resistance rarely leads through the kingdom of Wintel. Although Sun won’t disclose the first MAJC chip until Microprocessor Forum in October, it described the instruction-set architecture at the Hot Chips conference last week.

MAJC offers yet another spin on the VLIW technology explored by mainframe engineers in the 1960s and pioneered by Multiflow, Cydrome, and Culler in the 1980s (see MPR 2/14/94, p. 18). Since then, VLIW has evolved in a variety of ways, forming the basis for such architectures as Intel’s IA-64, Philips’s TriMedia, Equator’s MAP1000, Fujitsu’s FR-V, and high-end DSPs from Texas Instruments, StarCore, and Analog Devices.

Each variation takes a slightly different approach. Some companies, such as Intel and Hewlett-Packard, avoid the VLIW moniker altogether, preferring to invent stainless labels such as EPIC (explicitly parallel instruction computing) to describe their enhancements of the concept.

No matter what they’re called, all of these architectures have two things in common: bundles or packets of multiple instructions and compiler-directed parallel execution. This contrasts with today’s superscalar RISC and CISC architectures, which handle instructions as discrete units and schedule the instructions for parallel execution at run time, often by reordering them on the fly. MAJC is a typical VLIW architecture in the sense that it too relies on instruction packets and smart compilers instead of complex control logic.

Sun’s contribution to the art appears to be a Java-friendly (though not Java-specific) architecture that’s particularly amenable to multithreading and chip multiprocessing (CMP)—the integration of multiple CPU cores on a single die. This should be a happy marriage, because Java is inherently a multithreaded language for multitasking operating systems. It also means that MAJC is a highly scalable architecture designed to take advantage of deep-submicron IC processes, because a single processor can integrate up to one thousand easily cloned CPU cores.

But the name MAJC is misleading. Although it has some features to improve the performance of Java virtual machines and just-in-time (JIT) compilers, MAJC is a general-purpose architecture that can run software written in any language. Yet it has virtually nothing in common with Sun’s SPARC, a seminal RISC architecture for general-purpose computing, or Sun’s Java chips, which have a stack-based architecture and a bytecode-native instruction set.

Why Another New Architecture?

Sun says it launched the MAJC project in 1995 after identifying four trends: the growing importance of compute-intensive algorithms in application software; the rapid adoption of Java, a virtual platform that insulates application programmers from CPUs and operating systems; the additional parallelism available in multithreaded code; and the system-on-a-chip (SOC) approach to microprocessor design, which takes advantage of growing transistor budgets by integrating CPU cores with on-chip peripherals and large caches.

Marc Tremblay, MAJC’s architect, says those observations led his team to develop an architecture that can more efficiently exploit the thread-level parallelism in media-rich code and make it easier for engineers to design and verify highly integrated processors with tens or hundreds of millions of transistors.

Of those four reasons, however, perhaps two explain why Sun didn’t simply extend its existing architectures. The remaining reasons seem more questionable.

Consider the first trend Sun identifies: the growing importance of compute-intensive algorithms in general application software. As Sun points out, today’s programs often manipulate 3D graphics, digital video, and digital audio and perform other tasks in which the ratio of compute operations to memory operations is greater than in the past.

This trend has not escaped the notice of other CPU designers, and most of them are coping without creating new architectures. Virtually every RISC and CISC architecture has added multimedia or digital-signal-processing (DSP) extensions in recent years. Indeed, Sun helped lead the charge by introducing its visual instruction set (VIS; see MPR 12/5/94, p. 16) for UltraSparc in 1994, years before the x86 gained MMX, SSE, or 3DNow.

Sun retorts that media processing is much easier when a CPU has lots of visible registers. By starting with a blank slate, Sun was able to define an architecture that allows for hundreds of registers, plus some four-operand instructions that would be difficult to add to existing architectures. Still, given the obstacles to establishing a new architecture in the market, Sun doesn't need MAJC just for multimedia tricks.

Betting On Java

Nor is the popularity of Java a compelling motive. Sun cites a Dataquest projection that about 70% of Fortune 500 companies will be using Java by next year. Of those companies, about 80% will be using Java on clients, and virtually 100% will be running Java on servers. Although that speaks well of Java's penetration in large enterprises, it doesn't justify the enormous effort it will take to win acceptance for MAJC.

To begin with, even Sun says MAJC isn't aimed at corporate clients and servers. Those kinds of computers have ample memory for JIT compilers, such as Sun's HotSpot, which are improving by leaps and bounds. Today's CPUs already do a fairly good job of running Java, and they might do much better with the addition of a few Java-specific instructions—something that's likely to happen if Java becomes as universal as Sun hopes.

In the information-appliance market that Sun is targeting with MAJC, Java's role is uncertain. It might become a dominant factor, or it might be limited to a few applets on Web sites; it's too early to tell. Such products will almost certainly have less memory than PCs, so the ability of MAJC to accelerate Java with little or no help from a memory-hungry JIT compiler would definitely be valuable.

But remember, that's also why Sun invented bytecode-native Java chips. Fixing the problems that have kept Sun's PicoJava core and MicroJava-II processor from achieving any visible success seems like an easier chore than introducing another new architecture.

Java has become so critical to Sun's business strategy, however, that questioning why Sun does anything to further Java is like asking why Intel clings to a CPU architecture designed when Jimmy Carter was president, or why Microsoft battles the U.S. government to keep Windows from running without a Web browser. When a company pins its future on a strategic technology, it will do almost anything to promote or defend that technology.

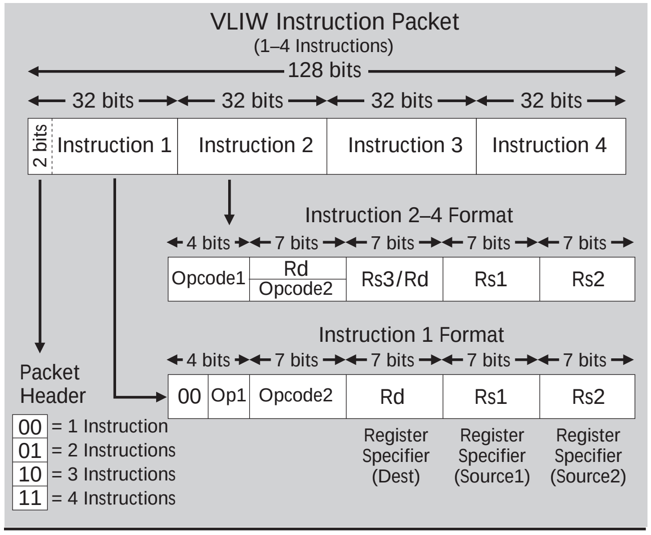

Figure 1. MAJC uses variable-size VLIW packets with 1 to 4 subinstructions. A two-bit header indicates the packet size.

Sun's Approach to VLIW

The two remaining reasons for creating MAJC make more sense: to extract more parallelism from program code and to exploit Moore's Law for implementing something other than Godzilla-sized caches.

Despite the diminishing returns of superscalar designs, Sun contends that extracting more parallelism isn't just a pipe dream. The key is to take advantage of thread- and method-level parallelism—tactics that play to Java's strengths as a multithreaded, object-oriented language.

At best, claims Sun, a CPU needs only four pipelines to wring almost all of the instruction-level parallelism out of a single execution thread (or a single-threaded process). Therefore, MAJC instruction packets can vary in size from one to four subinstructions, according to how many instructions can execute together without data dependencies. A compiler makes this determination—MAJC does no run-time reordering, unlike most superscalar architectures—and attaches a two-bit header to the packet indicating its length. At run time, a MAJC processor reads the header and dispatches one to four subinstructions to a CPU core. As Figure 1 shows, each subinstruction is 32 bits long, so packets can vary in length from 32 to 128 bits.

MAJC departs not only from the approach that Intel and HP take with IA-64 (see MPR 5/31/99, p. 1) but also from the classic VLIW architectures of the 1980s. At the heart of this departure is Sun's assertion that anything beyond four-way instruction-level parallelism is wasted on general-purpose code (though not necessarily on highly vectorized scientific code, which is not a priority for MAJC). Some CPU architects argue that designs as wide as 8 or 16 pipelines could find useful parallelism—and, indeed, that VLIW is a way to make such designs practical without the pain of complex control logic.

Intel and HP, for example, devised IA-64 to enable the design of very wide processors. Unlike MAJC, IA-64 always bundles the same number of instructions (three) in a packet that's fixed in size (128 bits). Each packet has a header that indicates how many instructions in that packet and following packets can execute without dependencies. In this way, IA-64 enables *n*-way instruction-level parallelism, because it can chain multiple instruction packets together into execution packets of any length.

Instead of spending transistors on what it believes are the wastefully diminishing returns of very wide designs, Sun would rather pursue thread-level parallelism by integrating multiple cores on a single chip. Different threads and processes should have few or no mutual data dependencies. MAJC compilers can bundle the instructions of those threads and processes into different packets, and MAJC processors can dispatch those packets to different cores. So MAJC allows four-way instruction parallelism per thread and *n*-way thread parallelism for *n*-threaded code. Sun says it would rather build a CMP processor with two four-way cores than a processor with a single eight-way core.

Although nothing inherent in IA-64 prevents thread-level parallelism or CMP, Sun claims Intel's architecture is more complex, especially with its baggage of x86 compatibility. Sun thinks MAJC will make it easier to implement CMP in a comparable IC process. Nobody will know for sure until actual implementations of both architectures appear.

Both MAJC and IA-64 avoid a drawback of some classic VLIW designs from the 1980s—the symbiotic relationship between VLIW formats and microarchitectures. In those machines, the instruction slots in each fixed-length VLIW packet mapped directly to the configuration of function units: integer slots, floating-point slots, and so forth. If the compiler couldn't fill all the slots with nondependent instructions that exactly matched the complement of function units, it padded the unusable slots with NOPs. This wasted instruction-fetch bandwidth and inflated the size of the code. It was less of a problem with the scientific code that early VLIW architectures were designed to execute, but MAJC needs to run more general-purpose code.

MAJC's variable-size VLIW format allows compilers to create shorter packets if there aren't enough nondependent instructions, so there's no need to pad any slots with NOPs. Intel achieves the same thing by divorcing the relationship between the size of instruction packets and the chains of nondependent instructions.

A Marriage Made in Hacker Heaven

Now it's obvious why MAJC is a good mate for Java. It's so easy to create threads in Java that the biggest problem is spawning too many of them—potentially leading to "deadlocks" in which multiple threads develop interdependencies that prevent them from completing their tasks.

But that's an embarrassment of riches. The good news is that MAJC should allow a CMP processor to extract more thread parallelism from multithreaded software without requiring Java programmers to write their code any differently than they do now.

Even within a thread, Sun claims, a MAJC compiler will find more parallelism in Java code than an ordinary compiler finds in procedural code. That's because MAJC supports two additional techniques: method-level parallelism and what Sun calls "vertical multithreading."

Method-level parallelism builds on Java's strengths as a thoroughly object-oriented language. All executable Java code resides in methods (functions within classes). In C++ programs, it's still possible—indeed, almost inevitable—to find procedural code mixed together with object-oriented code. Java compilers won't allow that.

MAJC assumes that methods are largely free of mutual dependencies because they are independent subroutines, even though they can access some common variables. MAJC processors will try to take advantage of this by speculatively executing multiple methods in parallel, almost as if they were explicitly coded threads.

The key difference between method- and thread-level parallelism is the speculation. It's similar to speculative execution in other CPUs, except it's more closely mapped to the higher-level programming structures of object-oriented languages like Java. Sun fancifully calls it "space-time computing," because different methods execute in their own memory spaces and perform operations that aren't synchronized until a later time.

King of the Heap

One method in a thread, which Sun calls the "head thread," executes in the standard memory heap as usual. The other methods execute in speculative heaps, which Sun calls "dimensions." The heaps are completely independent. If a speculative method creates a new object, that object appears only in its speculative heap, where it remains "unborn" from the viewpoint of the head thread and other speculative methods. To improve performance, the processor temporarily disables exception checking and thread synchronization when executing speculative methods, because run-time exceptions and thread conflicts are rare.

A checkpointing mechanism allows speculative methods to stall each other if the processor detects a dependency. It can detect read-after-write and write-after-write violations. Only after the processor resolves all dependencies does it validate the results of the speculative methods, merge their heaps, deliver their unborn objects, and create a new head thread. If the processor cannot resolve a dependency, it discards the speculative results and garbage-collects the speculative heaps without penalizing the head thread.

Method-level parallelism is even more transparent to programmers than thread-level parallelism, because it requires no extra code at all. Like speculative execution in other CPUs, it happens at run time without any effort by the programmer—although adhering to good object-oriented

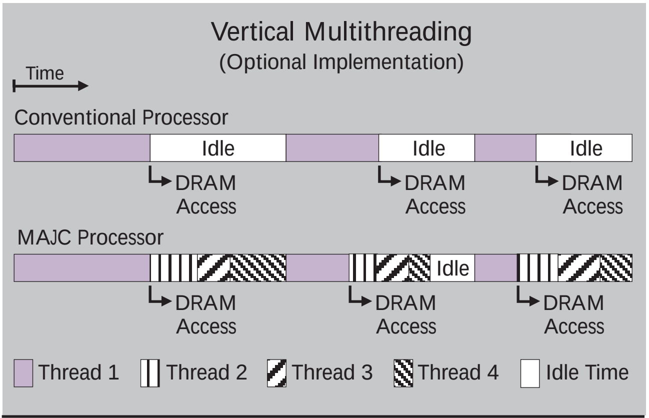

Figure 2. Vertical multithreading is Sun's technique for switching contexts among program threads during memory accesses.

design principles by writing self-contained methods would probably help.

Likewise, the technique that Sun calls vertical multithreading is also transparent to programmers. Essentially, it's Sun's term for switching contexts among different threads to fill time that a processor would waste after a cache miss. While the processor is waiting on a memory access, it can use its otherwise idle cycles to execute instructions in another thread—again based on the assumption that threads have few data dependencies. As Figure 2 shows, a MAJC processor can maintain state for up to four interwoven threads without penalizing the highest-priority thread.

Although MAJC has some instructions that make it easier for compilers to manage vertical multithreading, at the processor level it's an implementation option, not an architectural feature.

Designed for Scalability

MAJC permits anything from a one-way scalar processor (in effect, disabling VLIW by never issuing more than one subinstruction per packet) to a CMP superchip with an arbitrarily large number of four-way cores. (Actually, MAJC limits CMP to 1,024 cores per chip because cross-calls between cores use a 10-bit identifier, but transistor densities will dictate the practical limit for many years.)

To make this scalability easier to implement, MAJC function units are largely orthogonal. As Figure 3 shows, there are no integer units, floating-point units, multimedia units, or DSP units. Any function unit can handle any data type. Each unit has its own adder, multiplier, and shifter.

This eliminates the instruction steering required in some other VLIW architectures, such as TriMedia (see MPR 12/5/94, p. 12). With one exception (described below), a MAJC compiler can put any type of instruction into any packet slot, and the processor can dispatch any instruction to any pipeline. This design also reduces stalls due to resource dependencies,

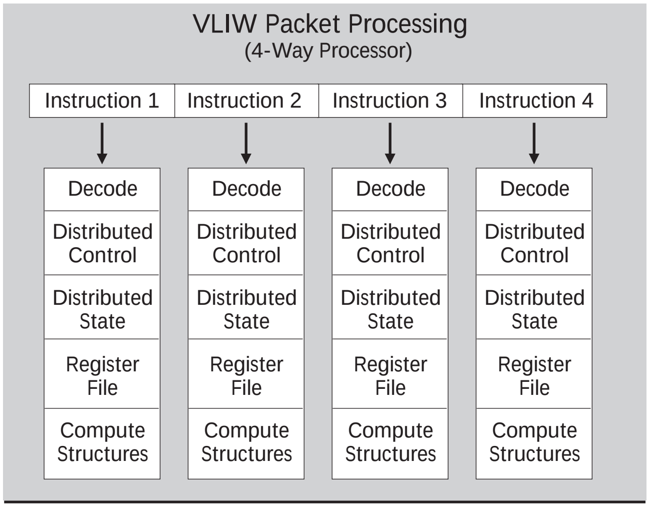

Figure 3. MAJC function units are orthogonal, capable of handling any type of instruction dispatched from any instruction slot.

because a processor with four function units always has enough execution bandwidth to handle the largest-size packet. The downside, of course, is larger function units.

There is one exception to this orthogonal design. As mentioned above, instruction packets have a two-bit header that indicates how many subinstructions follow. The header maps onto the opcode space of the first instruction, as Figure 1 shows. Therefore, the first instruction can address only three registers (two sources and a destination), not four registers (three sources and a destination).

This limitation means a compiler can't put instructions that manipulate four operands—such as a nondestructive multiply-add (MADD)—in the first instruction slot. The header consumes bits required to address the fourth register. It also means the first function unit in a core needs some extra decoding logic to interpret the header, and that it can do without some logic required to execute four-operand instructions. But these are relatively minor differences, and the other function units can be truly identical.

All of the units have 64-bit data paths. (MAJC allows 32-bit data paths for lower-cost chips.) They can handle 64-bit long-integer and double-precision floating-point instructions; 32-bit integer and single-precision floating-point instructions; 16-bit short-integer instructions; and some instructions that manipulate 8-bit values, particularly for single-instruction multiple-data (SIMD) operations. Some instructions have saturation, and others support special media types, such as 8- and 16-bit audio and graphics data. There are also DSP-type instructions; these include MADDs and operations that shuffle bytes, extract bits, count leading ones/zeroes, and find minimum/maximum values.

Variable-Size Register Files

Each function unit has its own register file, which programmers can partition as local and global registers. The instruction format allocates seven bits for register specifiers, allowing up to 128 register addresses. As Figure 4 shows, a programmable delimiter that can range from 0 to 127 in 32-step increments indicates which registers are local and which are global. When an instruction stores a value in a global register, the CPU broadcasts that value to the corresponding global registers in all other function units in that core.

If the delimiter points at R95, for example, then 96 registers (R0-R95) are globals shared by all function units. The remaining 32 registers in each unit are locals, accessible only by that unit. In a four-way core, this would make 224 registers visible to programmers and compilers—96 globals and 128 locals. This arrangement is repeated for every core. Normally the registers are 64 bits wide, although the architecture allows 32-bit implementations as well.

This register-rich design does bolster Sun's argument that MAJC is particularly well suited for media processing in ways that existing architectures would find difficult to match. For instance, a four-operand, nondestructive MADD is a useful instruction, but it needs tons of registers. A four-way

For More Information

Sun is withholding details about MAJC chips until Microprocessor Forum in October, but says it will introduce the first chip next year. For more information, go to *www.sun.com/microelectronics/MAJC*.

MAJC core can execute three MADDs per cycle, with a latency of six cycles. That means 18 pipelined MADDs could be in flight at once, requiring 72 registers to juggle their operands. Few existing architectures have that many registers to burn. Register renaming is a workaround, but at the cost of additional complexity.

With its orthogonal function units, duplicate register files, and provisions for CMP, MAJC encourages a cookie-cutter approach to chip design. Sun's concept is that engineers can concentrate on designing a powerful function unit with a fully custom layout and optimized data paths, then simply repeat that layout as many times as necessary to reach the desired level of performance. After a design team completes the first unit, it can add others in a step-and-repeat fashion with relatively little effort.

In this way, MAJC chips can leverage the expanding transistor budgets granted by the relentless march of Moore's Law to integrate more functional logic, not just larger cache arrays and superscalar interlocks. Theoretically, such a design should also be easier to verify.

This kind of design reuse inevitably wastes some transistors in surrounding structures, but it's a reasonable tradeoff in an era when transistor budgets are running surpluses. Software engineers make similar tradeoffs when they accept some code inflation in return for the greater productivity of high-level languages and object-oriented programming.

According to Sun's cycle-accurate simulations, a MAJC processor should outperform a comparable RISC processor, even without using CMP. But we expect Sun's first MAJC chip to integrate at least two cores—partly to show off what the architecture can do and partly to inspire potential licensees. At 300–500 MHz, such a chip could match the superlative performance of the Emotion Engine processor that Sony and Toshiba designed for the new PlayStation (see MPR 4/19/99, p. 1). But with its large function units and multiple cores, the MAJC chip could match the Emotion Engine's enormous die size and manufacturing cost too. That's probably why Sun says the first chip isn't intended for very low cost or battery-powered products.

Long-Range Vision

Sun is proclaiming MAJC as "the most important semiconductor architecture of the next 20 years." That seems a trifle ambitious, in view of the market's less-than-enthusiastic embrace of SPARC and bytecode-native Java chips. Although Sun has licensed both of those architectures to several partners, they have made little impact outside of Sun's own servers and workstations. (One exception is Fujitsu's SparcLite, which is found in many digital cameras.)

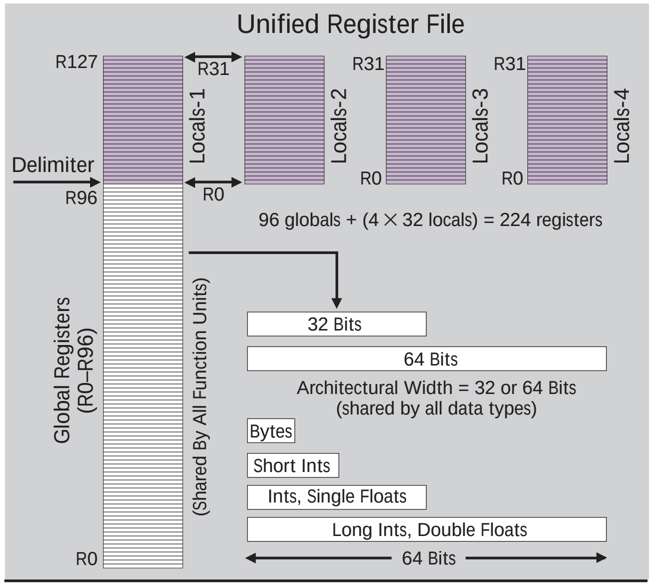

Figure 4. Each function unit has its own register file. A movable delimiter indicates which registers are global and which are local.

Perhaps Sun's community-source licenses—which allow anyone to download and study chip designs for free, paying money only for actual use—would entice more companies to adopt MAJC. Sun hasn't said whether it will offer MAJC under community licensing, but it's a logical step.

To help jump-start MAJC, Sun can leverage some in-house projects. The JavaStation, a thin client for corporate desktops, could certainly use a more powerful CPU than its sluggish MicroSparc-II. In 1997, Sun acquired Diba, a start-up company that designed information appliances. Since then, Diba has been reorganized as the Consumer Technologies Group within Sun Microelectronics, making it an obvious candidate for MAJC. Sun's PersonalJava, EmbeddedJava, and Java 2 Micro Edition are pared-down Java platforms for this market, requiring as little as 128K of RAM without sacrificing such critical features as multithreading; MAJC could give them a valuable performance boost.

Sun must also start building an infrastructure for MAJC. It needs embedded operating systems, compilers, and debugging tools. Sun has its own compiler, but the company appeared surprised when we inquired about licensing the compiler's back end to other tool vendors. Third-party tool support is critical, and VLIW compilers are notoriously difficult to write.

There's no doubt that MAJC is a highly innovative and versatile architecture. But winning acceptance for MAJC will require some clever sleight of hand. Embedded developers already have plenty of capable architectures to choose from, and MAJC competes with them all. Sun might need real magic to convince customers that MAJC has substance and won't disappear in a puff of smoke. M