JSTAR COPROCESSOR ACCELERATES JAVA

Licensable Bytecode Translator Works With Almost Any Core

By Tom R. Halfhill {3/27/00-04}

While bytecode-native Java chips continue struggling to find a market, a team led by former Sun engineers has invented a novel alternative: a coprocessor that attaches to any CPU core and translates Java bytecodes into native instructions on the fly. The coprocessor is

available now as a licensable Verilog model from JEDI Technologies, a Santa Clara–based startup founded in 1998.

JEDI's 32-bit coprocessor, called JSTAR, is architecture neutral. Programmable microcode and a bus-interface wrapper allow it to work with almost any 32- or 64-bit microprocessor—RISC, CISC, VLIW, embedded, desktop, or server. It requires no modifications to the host CPU core, operating system, or application software. CPU and ASIC designers can integrate JSTAR on a chip without altering the processor's I/O interfaces or pin compatibility. As Figure 1 shows, JSTAR drops into the instruction path between the CPU core and the primary caches or main memory.

Initially, JEDI is focusing on the embedded market. That's a logical strategy, because JSTAR allows a microprocessor to run Java programs much faster than a bytecode interpreter without the memory required for a just-in-time (JIT) compiler. Thus it addresses a critical tradeoff that impedes the widespread adoption of Java in the embedded world: to maximize performance, a Java-based product must absorb the cost, size, and power consumption of RAM for a JIT compiler; or to minimize cost, size, and power consumption, a Java-based product must do without a JIT compiler and suffer from lower performance.

JSTAR is compact and versatile. At 30,000 gates, it occupies only about 1mm^2 of die area in a typical 0.18-micron IC process. It can run at the host processor's core frequency, so its performance will scale on a linear curve with the CPU's native performance. When wedded to a

1.5V RISC core at 180MHz, its logic circuits consume about 18mW and its SRAM (for microcode storage) typically needs another 15mW. It's compatible with existing cache hierarchies, memory interfaces, and MMUs. It steps out of the way during native-code execution, and it doesn't interfere with Java native methods (subroutines that are usually written in C or C++ and compiled to native code, then called from a Java program).

Over time, JEDI hopes that JSTAR coprocessors will become as commonplace in microprocessors as FPUs, which also made their first appearance as auxiliary coprocessors. We think JSTAR is less compelling outside the embedded market,
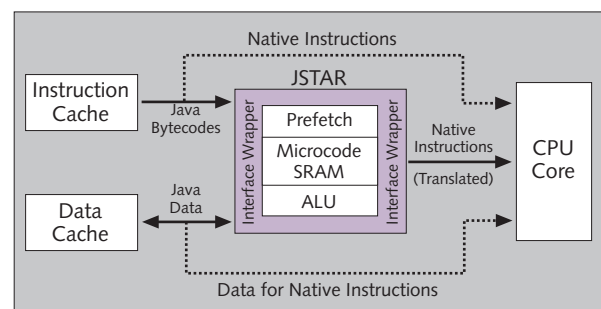
Figure 1. JSTAR intercepts Java bytecode instructions normally interpreted in software by a Java virtual machine, then translates the bytecodes into the native instructions of the host CPU. Native software executes normally.

however. Larger systems can more easily spare the CPU cycles and memory required by JIT compilers, adaptive compilers, statically compiled Java programs, and other software-based solutions that improve Java performance even more than JSTAR does. Still, that leaves a huge market for JSTAR.

One Catch: The Java Virtual Machine

In concept, JSTAR seems simple: it replaces the bytecode interpreter in a Java virtual machine (JVM) with tightly integrated hardware that translates bytecodes into native instructions at run time. But implementing the idea and integrating it with a CPU core and a JVM presented several challenges to the relatively small design team. (JSTAR is the brainchild of two former Sun engineers: Mukesh K. Patel, JEDI's president and CEO, and Jay P. Kamdar, COO and vice president of marketing. Patel was the chief architect, and the microarchitecture was designed by Tore Kellgreen and Erik Eidt.)

One challenge was to minimize JSTAR's effect on the JVM, which does more than just interpret Java bytecodes. The JVM also loads Java classes into memory dynamically, verifies class security, synchronizes multiple threads of execution (Java is inherently multithreaded), dynamically allocates and deallocates memory (automatic garbage collection), and checks for run-time exceptions (such as references outside the bounds of arrays).

Together, the JVM and the Java class libraries form a software abstraction layer called the Java run-time environment that insulates Java programs from the underlying CPU architecture and operating system. Java source files compiled into bytecodes (which are comparable to other kinds of machine-language instructions) will run on any JVM, which is the foundation of Sun's "write once, run anywhere" philosophy.

Although bytecode compatibility is universal—at least, to the degree that JVMs are compatible—different vendors implement the JVM's functions in slightly different ways. JEDI didn't want to create a special JVM solely for JSTAR, so it needed to design a coprocessor that integrates fairly well with existing JVMs. Some changes were unavoidable, because JSTAR takes over most of the work handled by the bytecode interpreter. The result is a compromise: JSTAR doesn't require a new JVM, but it does require a specially modified JVM.

Currently, JEDI is targeting four JVMs: Sun's standard JVM, Sun's KVM, Hewlett-Packard's Chai, and Transvirtual's Kaffe. The last three are optimized for embedded systems. JEDI says it takes a few programmers a few months to modify and verify a JVM for JSTAR, so unless there's a compelling reason to use a different JVM, most customers will choose one from the approved list.

The limited choice of JVMs isn't a major drawback, but it could have some complications. If any customers want to use JSTAR with a CPU core that isn't supported by one of the modified JVMs, they will have to choose between porting a JVM (which requires about five programmer-months) or switching CPUs. If a customer wants to use JSTAR in a real-

time application, the JVM must also support deterministic response. Currently, JSTAR doesn't run on PERC, a real-time "clean room" JVM from NewMonics.

In the future, some of the JVMs compatible with JSTAR may adopt the proposed real-time extensions to Java. An industry working group with representatives from several companies is currently defining those extensions, and the first reference implementation is expected this summer.

Bytecodes Go Native

The other major challenge faced by JEDI's design team was to minimize JSTAR's effect on the CPU core. This effort was more successful.

As Figure 1 shows, JSTAR is an instruction-path coprocessor that sits between the core and any memory structures that are present, such as a primary cache (unified or split), an MMU, or main memory (DRAM, SRAM, ROM, or flash ROM). JEDI supplies a Verilog wrapper that adapts JSTAR's generic interface protocols to specific CPUs. The first wrapper is designed for cores based on the popular MIPS-I architecture, and JEDI has demonstrated JSTAR running in an FPGA with a Lexra LX4180 MIPS-like core. Until JEDI creates more wrappers, customers who prefer other CPU architectures will have to adapt the wrapper themselves—not a difficult task, JEDI claims.

Because several patent applications are pending, JEDI is more than a little vague about JSTAR's microarchitecture and the mechanism that bypasses the JVM's interpreter. Nevertheless, we have some clues. JSTAR reserves several pages of virtual memory addresses and maintains its own program counter independently of the CPU's program counter. Whenever the CPU attempts to reference a memory location within the range of reserved addresses, JSTAR traps the operation and switches into "Java mode"—which is merely a different path of execution, not a special processor mode or context.

This suggests that JSTAR somehow intercepts calls by the CPU to the JVM's interpreter, perhaps by virtually relocating the interpreter loop from its actual address to the block of reserved addresses. The relocation would be virtual because there's no longer a conventional interpreter loop. When the JVM jumps to an address in reserved memory (an address that it thinks is the beginning of an interpreter loop in real memory), JSTAR takes over.

Referring to its own program counter, which points to a bytecode address in real memory, JSTAR fetches a bytecode instruction from that location, translates the bytecode into one or more native instructions, and feeds the instructions to the CPU, masquerading as the interpreter. The CPU's program counter continues walking through the "interpreter loop" until it reaches the loop terminator, then begins the next iteration at the first virtual address. Meanwhile, JSTAR continues incrementing its own program counter to point to real memory locations of bytecode instructions.

The CPU is oblivious to all this redirection—it thinks it's still fetching native instructions from valid locations in memory, as indicated by its own program counter. But the native instructions it fetches are translated bytecode instructions, and they're coming from JSTAR, not from a native program (a bytecode interpreter) in real memory.

If a Java program calls a native method, JSTAR feeds the CPU a native branch instruction that jumps to the native code. When the method is finished and branches back to the Java program, the CPU tries to invoke the interpreter, which again redirects execution to JSTAR.

Our understanding of this mechanism may be incomplete, but the result is clear: Java bytecode instructions normally executed by a JVM's interpreter are diverted to JSTAR, which translates the bytecodes into native instructions and feeds them into the CPU's instruction path. The whole process is transparent to the Java program, the CPU, and the operating system. Although the operating system must save some additional state information associated with JSTAR during a context switch—the Java program counter is one example—most operating systems provide a way to do this without modifying the kernel.

Triage for Bytecode Translation

JSTAR translates each bytecode instruction into one or more native instructions by running small microcode routines. It requires less than 6K of VLIW-style microcode stored in SRAM, which allows JEDI to easily support different CPU architectures. Only the microcode, JSTAR's interface wrapper, and the JVM have to change.

JSTAR doesn't execute every bytecode instruction in microcode. Nor do bytecode-native Java chips (see sidebar, "Java Chips Fight an Uphill Battle"). Some of the 226 instructions in the Java instruction set are too complex, or occur too infrequently in typical software, to be worth implementing in hard-wired logic or microcode. (One example is the instruction that creates an object whenever a program executes a "new" command.) In those cases, JSTAR jumps to a software routine that interprets the bytecode instruction the usual way.

This approach also allows JSTAR to handle instruction types that a host CPU doesn't natively support. For instance, if a Java program tries to execute a floating-point add instruction on a CPU without an FPU, JSTAR jumps to a routine that interprets the instruction in software, just as an unmodified JVM would.

JSTAR has some logic that offloads work from the CPU. It has a two-word prefetch buffer for bytecodes, which is useful for executing references to the Java stack and main memory in parallel. There is some logic for byte alignment, because bytecode instructions are variable in size.

More important, JSTAR helps the CPU utilize its caches more efficiently—an almost accidental side effect of the way JSTAR bypasses the interpreter. Normally, a CPU loads a Java program's instructions into the data cache because it sees the bytecodes as data for a native program—the JVM's interpreter or a JIT compiler. Only the interpreter's or compiler's instructions load into the instruction cache. But because JSTAR is an instruction-path coprocessor that translates bytecodes into native instructions between the CPU and the caches, it can fool the cache controller into fetching bytecodes into the CPU's instruction cache. At the same time, JSTAR eliminates the need to load the native instructions of an interpreter or JIT compiler into the instruction cache.

Result: Java instructions end up in the instruction cache and their operands end up in the data cache, where they belong. The system is better balanced and works the way it was designed to work.

All this trickery potentially imposes one penalty, however. JSTAR must examine every memory reference by the CPU to see if it falls within the range of virtual addresses that trigger "Java mode." This check adds a mux to the instruction path, which is often the critical path in processors with a primary instruction cache. According to JEDI, the effect is minuscule—a 180MHz processor slows to about 178MHz. That's less than a 2% penalty, which is a small price to pay for the much greater increase in Java performance.

If cache access isn't the critical path, the additional mux probably won't impair the host CPU. An ARM7 core that lacks a primary cache would hardly notice the extra gate delay. Also, an ASIC designer who's working with a synthesizable CPU core could integrate JSTAR's mux with the CPU's address-output mux, effectively eliminating the delay from the critical path.

JSTAR's Caffeine Jag

Measuring the benefits of JSTAR isn't easy, because there aren't any thorough embedded-benchmark programs for Java. JEDI has run tests with Pendragon Software's Embedded CaffeineMark 3.0 (*www.pendragon-software.com/pendragon/cm3/*). The CaffeineMark program is a fairly simple benchmark that includes an Eratosthenes sieve, various sorting loops, recursive method calls, and some floating-point math.

Figure 2 shows the results of these tests, converted into CaffeineMarks per MHz. (Raw clock-speed comparisons wouldn't be valid because the JSTAR/Lexra prototype was
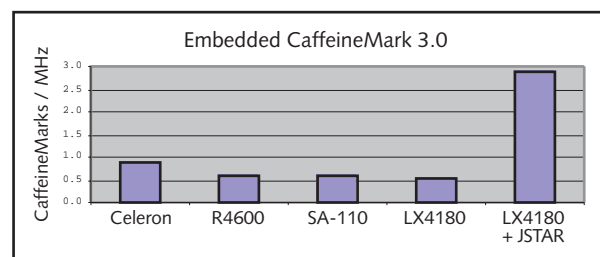
Figure 2. JSTAR runs the Embedded CaffeineMark program about five times faster than the same CPU running a bytecode interpreter at the same clock frequency.

Java Chips Fight an Uphill Battle

Bytecode-native Java processors largely eliminate the need for a software-based interpreter or a JIT compiler because they natively execute the vast majority of bytecodes in the Java instruction set. (As with JSTAR, they still interpret a few of the most complex and infrequently used bytecode instructions in software.) So far, however, Java chips haven't been very popular with embedded-system designers. This is partly due to shortcomings of the chips themselves, but previous attempts to sell language-specific processors weren't much more successful.

The earliest champion of Java chips was Sun Microelectronics, which made a fanfare in 1997 with its PicoJava core (see *MPR 11/17/97-02*, "MicroJava Pushes Bytecode Performance"). Sun licensed the core to several companies. Unfortunately, the performance and power consumption of PicoJava-based chips was disappointing, and they've had trouble competing against other kinds of embedded processors. Sun gradually scaled down the PicoJava project and canceled plans to sell MicroJava chips itself. Indeed, the turmoil at Sun over the direction of Java-chip development drove several engineers away, including the founders of JEDI.

Java processors from other vendors haven't fared much better. Patriot Scientific's PSC1000 has scored a few design wins, but not always for its ability to natively run Java (see *MPR 4/19/99-en*, "Japanese Go ShBoom"). The PSC1000 wasn't originally designed as a Java chip—it was a stack-oriented Forth processor that Patriot adapted to Java.

Another Java chip with a strange history is Rockwell Collins' JEM1 (see *MPR 10/27/97-en*, "Rockwell Unearths Java JEM"). It too was based on an existing architecture adapted to Java. Rockwell ultimately decided not to use JEM1 and licensed it to aJile Systems, a startup founded in 1999 by some former employees of Rockwell, Sun, and Centaur. Starting with a second-generation core known as JEM2, aJile has produced the aJ-PC104, a single-board Java processor intended for real-time applications, and the aJ-100, a bytecode-native chip with a real-time embedded kernel.

Finally, there's the Imsys GP1000, an unusual embedded processor from Sweden (see *MPR 12/28/98-03*, "GP1000 Has Rewritable Microcode"). It's yet another example of a chip that wasn't originally designed to run Java. Halfway through the project, Imsys realized it could write a version of microcode that natively executes most of the Java bytecodes. Because the GP1000's microcode is rewritable, the project didn't commit Imsys to producing a Java-only chip. The GP1000 also has some other features that dovetail nicely with Java. Imsys has some design wins for the GP1000, but the company says its customers aren't ready to disclose how they're using the chip's Java capabilities in future products.

---

running in an FPGA.) With JSTAR, the R3000-class LX4180 core runs the benchmark tests about five times faster than it does without JSTAR. Intel's Celeron processor fared better in this comparison than the other embedded processors, mainly because its relatively powerful FPU aced the floating-point test.

Figure 2 compares JSTAR's performance to interpreters, not JIT compilers. JIT compilers can boost performance even more than JSTAR does because they can apply some classic compiler optimizations while observing a program's run-time behavior. For instance, a JIT compiler can inline a frequently executed method to eliminate the method call, and it can ignore references to dead variables. JSTAR simply translates bytecode instructions into native instructions one by one, no matter how frequently it encounters the same sequence of instructions or how useless the instructions are.

Although JIT compilers can accelerate Java performance by an order of magnitude or more, they require too much memory to be practical for many embedded applications. A typical JIT compiler not only requires about 500K of memory for itself but also some additional memory for the translated code it caches. JSTAR doesn't cache any translated code, so only the bytecode image of the Java program resides in memory—and bytecode is much denser than RISC code or even CISC code.

To demonstrate the density of bytecode, JEDI compared the size of the classic Eight Queens algorithm (which arranges eight queens on a chessboard so they can't capture each other) as compiled in Java bytecode, x86 native code, and MIPS native code. The bytecode version is only 281 bytes long, the x86 version is 902 bytes, and the MIPS version is 2,196 bytes. If those numbers are surprising, remember that Sun originally designed Java for a networked embedded application (interactive TV), so code density was a high priority.

Code density is just one reason that Java would be more useful for embedded-software development if it weren't for Java's relatively slow performance or large memory requirements. Additional reasons are Java's inherent concurrency, security, networking capabilities, productivity, and code safety—and, of course, cross-platform compatibility.

"Write once, run anywhere" should be even more important for embedded developers than for PC developers, because there are many more architectures to support in the embedded market. If JSTAR can solve Java's problems, embedded-Java developers could preserve their software investments while switching CPU architectures to gain any advantage in performance, power consumption, or cost.

Not Just Another Java Chip

Unfortunately, JSTAR doesn't solve all of Java's problems. It does boost performance closer to JIT-compiler levels without the memory bloat of a compiler or static compilation. That should open some doors that were closed before. But JSTAR still needs a JVM, which also gobbles memory. Other companies are attacking that problem in different ways, usually by stripping the JVM to bare essentials. (Transvirtual's Kaffe can be as small as 100K.)

Even with the help of JSTAR, Java programs won't run as fast as native software. JVMs spend only about half their time interpreting bytecodes. The rest of their time is consumed by thread synchronization, garbage collection, exception handling, and other tasks required of a virtual machine. JSTAR does little or nothing to reduce that overhead.

Nor does JSTAR remove another obstacle to using Java for embedded applications: real-time response. JSTAR doesn't make the problem much worse, though. Except for the small amount of additional state the operating system must save during a context switch, JSTAR adds no latency to interrupt handling.

JSTAR does offer an interesting alternative to bytecodenative Java chips. Its biggest advantage is that it's not a chip—it's a licensable, synthesizable coprocessor that integrates with almost any CPU core. That means designers don't have to live with decisions made by other designers. Customers who prefer the MIPS architecture because their older software is written for MIPS don't have to rewrite everything in Java to use JSTAR. They can keep using their existing native software and development tools, and probably their favorite operating system too, while writing new software in Java. The same goes for customers who prefer ARM or any other architecture that can be made to work with JSTAR.

Another significant advantage of JSTAR is that its performance scales linearly with the clock speed of the host CPU. It's not limited to the fixed levels of performance offered by Java-chip vendors. If an off-the-shelf Java chip is fast enough for a given embedded application, then it's probably a better solution than JSTAR. But if the application demands higher performance without a JIT compiler, JSTAR's only speed limit is the fastest available licensable core on the fastest available IC process.

Other Solutions May Overtake JSTAR

What may be more limited, however, is JSTAR's window of opportunity. JSTAR may become less attractive as embedded processors get faster, memory gets cheaper, and JIT compilers continue to improve. If an embedded system

can spare the memory for a JIT compiler and the cost of a powerful CPU, it can achieve higher Java performance without JSTAR. Already, we think JSTAR makes little sense in a PC or server processor.

JEDI disagrees, arguing that server processors can use JSTAR to keep up with the increasing demand for Web services, especially as the number of non-PC information appliances continues to grow. Palmtop computers, cellular phones, home video-game consoles, and other devices that are gaining Internet capability far outsell PCs. Imagine what would happen, says JEDI, if millions of people with Web-enabled cell phones tried to check the value of their investments after the New York stock market closed each day. The resulting "flash crowds" could easily overwhelm a Web site. Multithreaded Java servlets—the server-side counterparts to Java applets—can handle numerous users simultaneously, and they are lighter-weight tasks than some CGI (common gateway interface) processes written in Perl. In that situation, a server processor could make good use of JSTAR.

It's not a bad argument. But an optimized JVM with an adaptive JIT compiler such as Sun's HotSpot can accelerate Java performance even more than JSTAR can. HotSpot runs on today's off-the-shelf processors, and it's free. The hidden cost is CPU power and memory. To run HotSpot, Sun recommends at least 48M of RAM for an x86-based Windows NT system and 64MB for a SPARC-V8 or -V9 Solaris system. Sun obtained the most impressive benchmark results on systems with 256–512M of RAM. Although that's not an unreasonable amount of memory for a Web server at a busy site, most embedded systems have more stringent power-consumption, size, and cost requirements.

That's why we think JSTAR's window of opportunity is open wider in the embedded market than anywhere else. And because progress tends to come more slowly to the embedded market than to the PC and server markets, that window will stay open longer than a quick calculation based on Moore's Law implies. JSTAR may not be the perfect Java solution, but it's a good solution, and we think it has a brighter future than Java chips. ◇