Cahners
MICROPROCESSOR REPORT
www.MPRonline.com
THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

EMBEDDED JAVA CHIPS GET REAL

Bytecode-Native aJ-100 Handles Real-Time Processing

By Tom R. Halfhill {8/7/00-02}

Java and real-time processing usually go together like coffee and ketchup. But a Silicon Valley startup, aJile Systems, has a new Java chip that handles interrupts in real time and doesn't need a third-party RTOS. It also allows embedded-system developers to write all their software in Java—even device drivers and other low-level code that normally would be written in C or assembly language.

AJile's aJ-100 microprocessor is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. JEM2 is an enhanced version of JEM1, created in 1997 by the Rockwell-Collins Advanced Architecture Microprocessor group (see *MPR 10/27/97-en*, "Rockwell Unearths Java JEM"). Rockwell-Collins originally developed JEM for avionics applications by adapting an existing design for a stack-based embedded processor.

Sun's Java virtual machine (JVM) uses a stack-based architecture, and the JEM cores have a writable microcode store, so the Java adaptation was fairly straightforward. The Rockwell-Collins team widened the core's datapaths to 32 bits, added some new datapaths to support Java data types, modified the instruction-fetch unit, and wrote a new microcode library that executes Java bytecodes. (Bytecodes are executable Java instructions created by a Java compiler or assembler. They are comparable to the machine instructions created by other compilers and assemblers.) By executing bytecodes natively, Java processors don't need a bytecode interpreter or a memory-hogging just-in-time (JIT) compiler.

Although Rockwell-Collins is using JEM2 for internal research and development, the company decided not to sell the chip on the merchant market. Instead, it exclusively licensed the design to aJile, which was founded last year by engineers from Rockwell-Collins, Centaur Technologies, Sun Microsystems, and IDT. AJile has wrapped additional on-chip memory and peripherals around the JEM2 core to create a more-integrated chip for low-power (<100mW) and real-time (<1 microsecond) embedded applications.

AJile's goal is to penetrate embedded markets that until now have been off limits to Java because of high cost, low performance, and excessive power consumption. By programming in Java, developers can write more-portable code while leveraging Java's productivity advantages over C/C++ and assembly language.

A Compact CISC Core

Development boards based on a JEM2 chip with an FPGA are available now, and samples of the aJ-100 are scheduled to be available this fall. Production is expected to begin by the end of the year. Fabricated in a 0.25-micron CMOS process, the fully static core runs at 100MHz at 2.5V (with 5V-tolerant I/O), consumes less than 1mW per MHz, and occupies less than 1mm^2 of die area. With integrated peripherals and 48K of on-chip SRAM, the total die area is less than 16mm^2. The chip is packaged in a 176-pin LQFP and will cost $15 in 10,000-unit quantities.

As Figure 1 shows, the aJ-100 core is a relatively simple 32-bit CISC machine with a single ALU, 24 general-purpose registers, 16K of microcode ROM, 16K of microcode SRAM, and some on-chip stack control and power management. The memory bus is 32 bits wide internally and runs at the processor's core speed. To reduce the pin count, the current

version of the aJ-100 brings out only 28 address lines. That still permits the aJ-100 to address 32MB of physical memory, hardly a limitation for its intended applications.

The ROM control store contains the microcode that executes Java bytecode instructions and IEEE-754 floating-point routines. No Java processor announced to date natively executes the entire set of 226 bytecode instructions, and the aJ-100 is no exception. The reason is that some of the most complex instructions take an indeterminately long time to execute and are relatively rare in Java programs. Java processors trap those bytecodes at run time and execute them in software.

Compared with other Java chips, the aJ-100 executes a larger percentage of bytecode instructions (99%) without resorting to software traps. In fact, the aJ-100 natively executes all but two bytecode instructions: MULTIANEWARRAY (allocate a new multidimensional array) and ATHROW (throw an exception or error). In comparison, Sun's picoJava core executes about 170 bytecode instructions in hard-wired logic, implements about 30 in microcode, and traps the remaining instructions in software.

To handle ATHROW and MULTIANEWARRAY, the aJ-100 has a low-level process running in executive or supervisor mode on its own stack. (Note that this mode is not part of the standard JVM specification.) On a trap, the processor switches to executive mode and begins executing a trap handler. This handler, like all other software running on the aJ-100, is written in Java and may call other Java methods. To execute MULTIANEWARRAY, for example, the trap handler repeatedly calls the code that allocates memory for single-dimensional arrays, since multidimensional arrays in Java are simply multiple instances of single-dimensional arrays. When this work is finished, the trap handler returns control to the original user-mode process. In the case of MULTIANEWARRAY, the trap handler leaves a return argument on the user-mode stack—an object pointer to the new array. Normal execution resumes with the instruction in the user-mode process that follows the instruction that triggered the trap.

In addition to the 16K ROM control store, the aJ-100 has another control store with 16K of SRAM. This allows aJile to add new microcode for extended bytecode instructions. For example, aJile could define an instruction that performs a square-root function within a critical loop. Application programmers wouldn't have to directly manipulate, or even know about, this instruction, because aJile provides a special linker called JEM Builder that maps the instruction to a Java method. (Methods are the object-oriented counterparts to functions, procedures, and subroutines in other programming languages.) In this example, JEM Builder would redirect the square-root method in Java's Math class (java.lang.Math.sqrt) to call the custom square-root instruction instead of invoking the usual math routine. A reserved opcode in the bytecode instruction set, IMPDEP2, traps the method invocation at run time and redirects the call.

By defining extended bytecodes, aJile can customize the aJ-100 for specific customers and applications. AJile will probably offer this option as an additional service instead of training customers to use a microassembler themselves. In this and other respects, aJile doesn't try to match the level of configurability offered by core vendors such as ARC Cores and Tensilica. But customization via extended bytecodes is a valuable feature, and it's similar to the capabilities of a pair of Java chips from Imsys in Sweden: the GP1000 and the Cjip (see *MPR 12/28/98-03*, "GP1000 Processor Has Rewritable Microcode"). Like the aJ-100, the GP1000 and the Cjip implement most Java bytecode instructions in microcode that's partitioned into ROM and SRAM control stores. This allows Imsys to create custom instructions at the microcode level. Unlike the aJ-100, the GP1000 and the Cjip are not exclusively Java chips; alternate microcode libraries implement proprietary instruction sets for software development in C, C++, Forth, and assembly language.

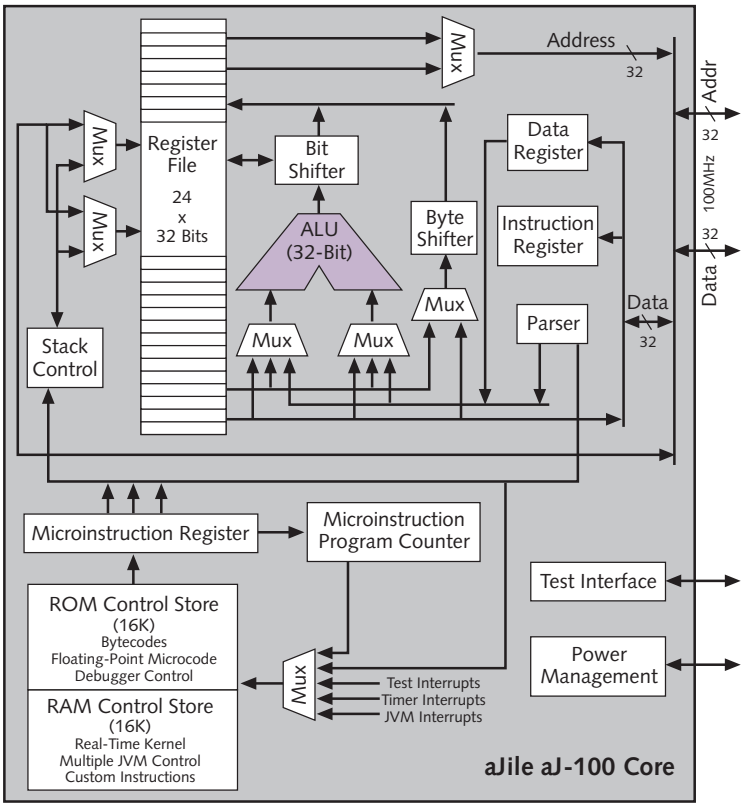

Figure 1. The aJ-100 is an integrated processor based on the JEM2 core originally designed by Rockwell-Collins. The core is a fairly simple 32-bit stack-based CISC processor with a writable microcode store.

Tossing Out the RTOS

Besides providing a way to add extended bytecodes, the SRAM control store in the aJ-100 contains a

microcoded real-time kernel and some control code for running multiple JVMs as separate processes. These unusual features are substitutes for a conventional RTOS, which on any other processor would manage the real-time interrupt processing and multitasking. Figure 2 shows the elements (shaded in purple) rendered unnecessary by the aJ-100's microcoded bytecode execution and real-time kernel.

The JEM Builder linker allows programmers to assign any Java method to an interrupt in a 32-entry vector table. When an event triggers an interrupt, the aJ-100 suspends the currently running thread, saves the thread's state, and switches to the thread that contains the linked method in the interrupt vector table. The context switch takes no more than 500ns at 100MHz. (AJile expects the response time will drop to 300ns in future versions of the aJ-100.) When the processor is done handling the interrupt, it switches back to the highest-priority thread that's ready to resume execution.

The primary secrets behind the aJ-100's real-time processing are its ability to run multiple JVMs and its real-time kernel that natively supports Java multithreading. In the past, the real-time response of embedded software written in Java was too slow and unpredictable, because a central feature of the language is automatic garbage collection.

Unlike C/C++, Java doesn't require programmers to explicitly allocate and deallocate memory for data structures. Instead, the JVM automatically allocates memory for new objects and automatically frees the memory when the program no longer references the object. (Java is a thoroughly object-oriented language; except for a few primitive variable types, all data structures are objects.) But the garbage collector that automatically frees memory is relatively slow and nondeterministic. It plays havoc with programs that require fast context switching to handle interrupts in real time.

By running multiple JVMs as separate processes, the aJ-100 allows software developers to assign different tasks to different JVMs. Moreover, different JVMs can observe different rules for memory management. The JVM dedicated to real-time processing can disable automatic garbage collection altogether, while other tasks that don't require real-time response can run on another JVM that has a normal garbage collector.

Partitioning an application across multiple JVMs is also a safety feature. Each JVM is a separate process that runs inside its own region of protected memory, and each JVM has its own threads, interrupts, timers, and time slices (in programmable increments that default to 10 microseconds). If one JVM crashes, it shouldn't take down the whole system. To keep a crashed or misbehaving JVM from interfering with other processes, the aJ-100 uses watchdog timers to ensure that each JVM gets the time slice due it.

Multiple JVMs normally don't share memory on the heap, since this would compromise system stability and security. However, developers can set up the JVMs with overlapping regions of memory to share data or code. For the convenience of programmers, JEM Builder provides an abstract interface to the common memory (GlobalMemoryDescriptor), so different JVMs can share global objects. For maximum security, though, it's safer for the JVMs to communicate via standard socket interfaces.

Memory Achieves Immortality

To make everything work, aJile had to define some extended bytecodes (such as IPEEK and IPOKE) that directly manipulate memory. AJile also had to implement Java's threading primitives (such as java.lang.Thread.yield) in microcode. But as is the case with the custom instructions described above, the extended bytecodes and real-time kernel are invisible to Java programmers. AJile's linker maps those functions to Java classes and methods.

Extending the bytecode instruction set and disabling the garbage collector might sound like renegade behavior. Other companies are taking a similar approach, however. A loose alliance is developing an official real-time Java specification that should be finished later this year (see *www.rtg.org*). Participating companies include aJile, Ada Core Technologies, IBM, Sun Microsystems, Microware, QNX, and Nortel. AJile's CTO and cofounder, David Hardin, is a coauthor of the specification.

The official real-time Java specification will work with any CPU, not just Java chips. It stops short of the explicit memory management required by C/C++, but it allows programmers to choose from three types of memory when creating a new object: normal (for regular nondeterministic garbage collection); immortal (allocated memory that is never deallocated); and scoped (when an object falls out of scope, the garbage collector reclaims the memory immediately, instead of whenever it feels like doing so). Because the official

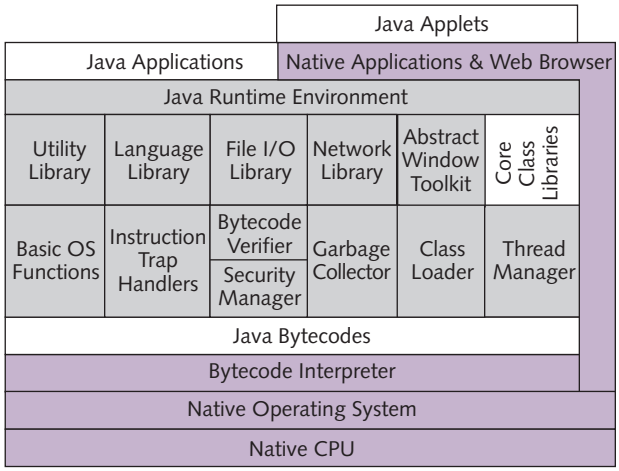

Figure 2. Normally, a Java run-time environment requires an interpreter to translate Java bytecodes into the CPU's native machine code. The aJ-100 eliminates that layer and the other elements shown here in purple by virtue of its bytecode-native instruction set and microcoded real-time kernel.

real-time Java specification seems destined to become an industry standard, aJile plans to implement it when it's final.

Even now, aJile's approach to real-time processing allows programmers to write device drivers and other low-level system software in Java instead of C or assembly language. Developers can use the same high-level, object-oriented programming model for a whole embedded application, top to bottom. For instance, aJile has a device-driver class called EthernetController with simple methods for sending and receiving packets.

Extending Java's object model to low-level programming also allows developers to use standard Java tools, such as Symantec Visual Cafe, Inprise JBuilder, or IBM VisualAge. AJile's linker strips unused classes out of a project to save memory and builds an optimized application that can run from ROM. The linker's ability to work with standard development tools saves aJile the trouble of creating proprietary tools and gives customers the option of using any tools they like.

Integration Aids Embedded Design

By adding some on-chip memory and peripherals to the JEM2 core, aJile has created a well-integrated solution. As Figure 3 shows, the aJ-100 has 32K of SRAM on chip for storing data. (This is in addition to the 16K of SRAM and 16K of ROM for the microcode control stores described above.)

AJile has also adapted the Advanced Microcontroller Bus Architecture (AMBA) to the aJ-100. As seen in Figure 3, a 16-bit AMBA bus separates lower-speed peripherals from higher-speed devices on the 32-bit processor bus. The slower peripherals include the interrupt controller, three 16-bit timer/counters, 40 one-bit general-purpose I/O (GPIO) ports, two 16550-compatible UARTs (which support the IrDA physical-layer protocol), and a serial peripheral interface (SPI). The aJ-100 has eight general-purpose chip selects with programmable wait states and address setup and hold.

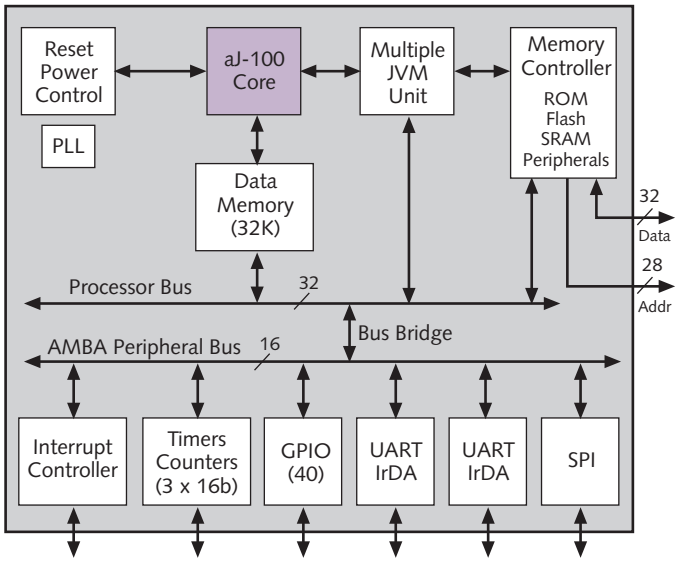

Figure 3. The aJ-100 surrounds the processor core with several on-chip peripherals on two internal buses, including a 16-bit AMBA bus.

Hanging off the 32-bit processor bus is a memory controller for external SRAM, ROM, flash ROM, and peripherals. It supports bus widths of 8, 16, and 32 bits. However, this controller doesn't support DRAM, so an external memory controller is necessary for some designs.

Since March, aJile has been shipping an evaluation board called the aJ-PC104. It has a JEM2 processor, an FPGA to implement the additional aJ-100 functions, 1M of flash memory, 1M of SRAM, and a 10Base-T Ethernet port. It also includes JEM Builder (aJile's optimizing linker), a Charade system debugger with JTAG, and a JVM based on Sun's Java 1.1.8 platform. This fall, aJile plans to upgrade the JVM to comply with Sun's Java 2 Micro Edition (J2ME). The JVM is valuable because it means developers don't have to license anything from Sun. The aJ-PC104 development system costs $599 and allows developers to start working on projects before samples of the aJ-100 arrive next quarter.

Crossing the Design-Win Desert

The aJ-100 is a clever product, but its prospects are anything but assured. Since Sun announced the first Java chips in 1996, embedded developers haven't exactly been beating down the doors. Several companies initially signed up as Sun licensees, then retreated from their plans to sell Java chips based on Sun's troubled picoJava core. Other companies, such as Imsys, created independent designs and hedged their bets by not making the processors too Java-centric.

Meanwhile, Java has achieved success as a virtual platform for applets on Web pages and business applications on servers. It's catching on much more slowly in the embedded market because the garbage collector interferes with real-time processing and the run-time environment needs too much memory. To get decent performance, a JIT compiler is almost mandatory, but that devours even more memory (about 500K plus enough RAM to cache the recompiled code) and makes response times even more nondeterministic (due to the JIT compiler's caching behavior).

Still, the embedded market for Java is not as barren as some critics imply. The Imsys GP1000 and Patriot Scientific PSC1000 have scored a few design wins (see *MPR 4/24/00-04*, "Patriot Scientific Allies With ProSyst"), Sony is using Java in some digital-video cameras (albeit for the user interface, not for control functions), and more than 20 million Java-based smart cards shipped last year. Java is expected to play a key role in third-generation (3G) wireless phones, and vendors like Motorola are working on integrated chips for 3G handsets that make Java performance a high priority. Embedded developers are not averse to using Java if they can overcome the obstacles.

But there's a difference between using Java and using a Java chip. If an embedded system can stand enough memory for a JIT compiler, a conventional embedded processor is a

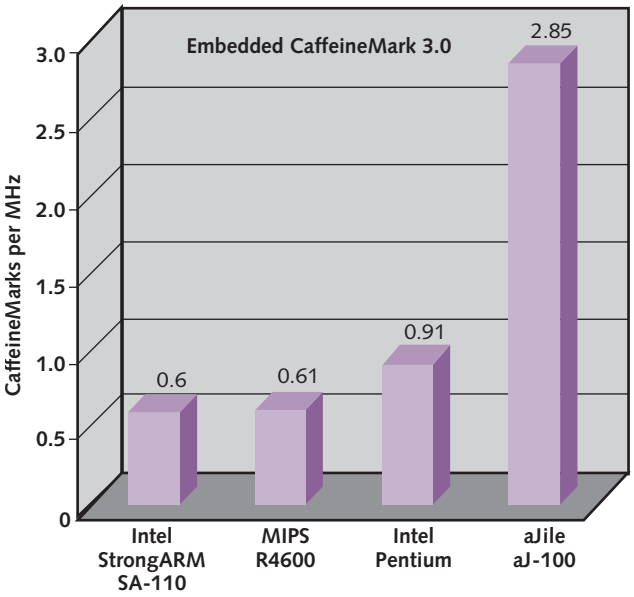

Figure 4. Native bytecode execution gives the aJ-100 a clear performance advantage over other types of microprocessors. Note, however, that the performance comparisons in this chart are relative, not absolute (CaffeineMarks per MHz), and that the other microprocessors were running the tests with an interpreter, not a much faster JIT compiler. The other processors would outperform the aJ-100 by sheer weight of clock speed, but they would also consume much more power.

workable alternative. As Figure 4 shows, aJile's tests indicate that the aJ-100 is nearly five times faster than a StrongARM SA-110—if the SA-110 is limited to using an interpreter and to running at the same 100MHz frequency as the aJ-100. With a JIT compiler, the SA-110 would be about two times faster than the aJ-100 at comparable clock speeds (though it would also consume three times as much power). After factoring in the SA-110's higher maximum clock frequency (233MHz vs. 100MHz), we find that StrongARM packs an even stiffer punch.

If memory is tight and real-time response vital—a typical scenario for embedded systems—the aJ-100 looks a lot better. It doesn't need an interpreter, JIT compiler, or third-party RTOS, and the JVM is smaller because the multithreading primitives are in microcode. Less memory also means lower power consumption. And no ordinary microprocessor and JVM can yet match the aJ-100's 500ns thread switching.

To get there, however, aJile had to create a special JVM unique to the aJ-100. Real-time Java code written for the aJ-100 is 100% Java but isn't yet "write once, run anywhere," because other JVMs and processors don't have the same features. Thus, aJ-100 developers may get the productivity advantages of writing in Java but not the effortless cross-platform compatibility that Sun promises.

When the official real-time Java specification is final—and when aJile adopts it—Java code written for the aJ-100 should be portable to other platforms, whether or not those platforms are Java chips. As other processor vendors adopt the real-time Java specification, the aJ-100 will have to compete on its merits as an embedded processor, not just as a Java processor with fast interrupt handling.

More competition will come from JSTAR, a bytecode-translation coprocessor that anybody can license and integrate with a CPU core (see *MPR 3/27/00-04*, "JSTAR Coprocessor Accelerates Java"). JSTAR, like the aJ-100, boosts Java performance while eliminating the need for a JIT compiler. JSTAR has the added advantage of compatibility with multiple CPU architectures, so developers can preserve their investment in native code while writing new code in Java. And JSTAR's performance scales with the frequency of the host processor: mating it with a fast RISC core could produce a chip that easily outraces the aJ-100. According to JEDI's CaffeineMark tests, however, the combination of JSTAR and a Lexra LX4180 would yield about 33% less relative performance (CaffeineMarks per MHz) than an aJ-100. Also, JSTAR requires the development of an ASIC, so it's not an off-the-shelf solution like the aJ-100. Implementing JSTAR in an FPGA would provide a quicker path to market but would cost more.

Obviously, the aJ-100 is a niche product. That niche could expand as more embedded developers adopt Java. But it could also shrink if clock frequencies climb and memory costs fall to the point where Java interpreters and JIT compilers become minor obstacles for embedded systems. We think Java chips still must prove their worthiness; the aJ-100 adds fuel to the debate, but it doesn't settle the argument.

Price & Availability

Production volumes of aJile's aJ-100 processor are scheduled to be available in 4Q00 at a price of $15 per chip in 10,000-unit quantities. The aJ-PC104 development system—which uses a JEM2 processor and an FPGA to provide the functionality of the aJ-100—is available now for $599. For more information, see *www.ajile.com*.