Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

Octera Throws a Javalon

Java-like Synthesizable Processor Targets Deeply Embedded Systems

By Tom R. Halfhill {3/17/03-02}

Only a few years ago, engineers despaired of using Java for real-time embedded applications because of its obesity and nondeterministic behavior. Java seemed more suitable for desktop PCs and servers that could satisfy its appetite for megabytes and megahertz. Yet, from the very beginning, Sun had conceived Java as a programming language and virtual platform for an embedded application—interactive TV set-top boxes. Perhaps it is inevitable that Java is returning to its roots.

Today, Java is showing up in all kinds of embedded systems, from PDAs to next-generation mobile phones. Of course, it's also popular on servers. In fact, Java is successful just about everywhere except the platform that Sun wanted to conquer most: the desktop PC. The irresistible force of Sun is still having trouble overcoming the immovable object in Redmond—but that's another story.

Now some companies are pushing Java into a very different frontier: deeply embedded applications that demand hard real-time performance. Examples might include motor controllers, industrial machinery, smartcards, automotive telematics, and other mundane systems that aren't as sexy as Web-browsing, game-playing cell phones but nevertheless keep the world turning. These applications require high reliability, frugal memory consumption, low cost, low power, and fast response to real-time interrupts—qualities not usually associated with Java.

One of the first pioneers to explore this territory was aJile Systems, which makes a standard-part Java microcontroller. (See *MPR 8/7/00-02*, "Embedded Java Chips Get Real.") Another contender is the Imsys Cjip, a rewritable-microcode chip that has instruction sets for Java, C, and Forth. (See *MPR 8/14/00-04*, "Imsys Hedges Bets On Java.") Now, San Diego–based Octera is introducing Javalon-1, a synthesizable microprocessor core that natively executes Java bytecode instructions. Javalon-1 is the first member of a small family of cores that will have minor variations on the same basic design. Chip designers can use Javalon-1 as the basis for a self-sufficient microcontroller or as a slave to another microprocessor core on an SoC or ASIC.

Javalon-1's memory requirements are unusually low (about 5K of firmware, not counting application memory). It doesn't need a Java virtual machine (JVM), a Java bytecode interpreter, or a just-in-time (JIT) bytecode compiler, because it natively executes Java bytecode instructions in hardware or software. It doesn't use a garbage collector to manage memory, so it's deterministic and has relatively low interrupt latencies (about 100 clock cycles, worst-case). And it's suitable for small embedded systems because it's compact (about 25,000 logic gates). Octera is wrapping up the design work now and plans to begin licensing Javalon-1 in 3Q03.

What's the catch? Javalon-1 isn't officially a Java processor, because it doesn't fully comply with Sun's Java specifications. In general, it follows the Java 2 Micro Edition (J2ME) and Connected Limited Device Configuration (CLDC) guidelines, but it makes several compromises to achieve its fast interrupt response, efficient memory usage, and low gate count. Javalon-1 is more accurately described as a Java-like processor or as a processor that executes Java bytecodes without fully supporting a Sun-standard Java platform. Some of Octera's trade-offs are sure to be controversial.

Instruction	Description
aaload	Load ref from array
aastore	Store into ref array
aconst_null	Push null
aload	Load ref from local var
aload_<n>	Load ref from local var
anewarray	Create new array of ref
areturn	Return ref from method
arraylength	Get length of array
astore	Store ref into local var
astore_<n>	Store ref into local var
athrow	Throw exception or error
baload	Load byte/boolean from array
bastore	Store into byte/boolean array
bipush	Push byte
breakpoint*	Breakpoint for debugger
caload	Load char from array
castore	Store into char array
checkcast	Check object type
d2f	Convert double to float
d2i	Convert double to int
d2l	Convert double to long
dadd	Add double
daload	Load double from array
dastore	Store into double array
dcmp<op>	Compare double
dconst_<d>	Push double
ddiv	Divide double
dload	Load double from local var
dload_<n>	Load double from local var
dmul	Multiply double
dneg	Negate double
drem	Remainder double
dreturn	Return double from method
dstore	Store double into local var
dstore_<n>	Store double into local var
dsub	Subtract double
dup	Duplicate top stack operand
dup_x1	Duplicate top stack operand, insert 2 values down
dup_x2	Duplicate top stack operand, insert 2 or 3 values down
dup2	Duplicate top 1 or 2 stack values
dup2_x1	Duplicate top 1 or 2 stack values, insert 2 or 3 values down
dup2_x2	Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values down
f2d	Convert float to double
f2i	Convert float to int
f2l	Convert float to long
fadd	Add float
faload	Load float from array
fastore	Store into float array
fcmp<op>	Compare float
fconst_<f>	Push float
fdiv	Divide float
fload	Load float from local var
fload_<n>	Load float from local var
fmul	Multiply float
fneg	Negate float
frem	Remainder float
freturn	Return float from method
fstore	Store float into local var
fstore_<n>	Store float into local var
fsub	Subtract float
getfield	Get field from object
getstatic	Get static field from class
goto	Branch always
goto_w	Branch always (wide index)
i2b	Convert int to byte
i2c	Convert int to char
i2d	Convert int to double
i2f	Convert int to float
i2l	Convert int to long
i2s	Convert int to short
iadd	Add int
iaload	Load int from array
iand	Boolean AND int
iastore	Store into int array
iconst_<i>	Push int
idiv	Divide int
if_acmp(cond)	Branch if ref compare true
if_icmp(cond)	Branch if int compare true
if(cond)	Branch if int=0
ifnonnull	Branch if ref not null
ifnull	Branch if ref=null
iinc	Increment local var by const
iload	Load int from local var
iload_<n>	Load int from local var
impdep1*	Implementation-specific trap
impdep2*	Implementation-specific trap
imul	Multiply int
ineg	Negate int
instanceof	Compare object type
invokeinterface	Invoke interface method
invokespecial	Invoke instance method with special handling
invokestatic	Invoke static method
invokevirtual	Invoke instance method
ior	Boolean OR int
irem	Remainder int
ireturn	Return int from method
ishl	Shift left int
ishr	Arithmetic shift right int
istore	Store int into local var
istore_<n>	Store int into local var
isub	Subtract int
iushr	Logical shift right int
ixor	Boolean XOR int
jsr	Jump to subroutine
jsr_w	Jump to subroutine (wide index)
l2d	Convert long to double
l2f	Convert long to float
l2i	Convert long to int
ladd	Add long
laload	Load long from array
land	Boolean AND long
lastore	Store into long array
lcmp	Compare long
lconst_<l>	Push long
ldc	Push from runtime const pool
ldc_<w>	Push from runtime const pool, wide index
ldc2_<w>	Push long or double from run-time const pool, wide index
ldiv	Divide long
lload	Load long from local var
lload_<n>	Load long from local var
lmul	Multiply long
lneg	Negate long
lookupswitch	Access jump table by key match and jump
lor	Boolean OR long
lrem	Remainder long
lreturn	Return long from method
lshl	Shift left long
lshr	Arithmetic shift right long
lstore	Store long into local var
lstore_<n>	Store long into local var
lsub	Subtract long
lushr	Arithmetic shift right long
lxor	Boolean XOR long
monitorenter	Enter monitor for object
monitorexit	Exit monitor for object
multianewarray	Create new multidim array
new	Create new object
newarray	Create new array
nop	No operation
pop	Pop top stack operand
pop2	Pop top 1 or 2 stack operands
putfield	Set field in object
putstatic	Set static field in class
ret	Return from subroutine
return	Return void from method
saload	Load short from array
sastore	Store into short array
sipush	Push short
swap	Swap top two stack operands
tableswitch	Access jump table by index and jump
wide	Extend local var index by bytes

Table 1. The Java bytecode instruction set has 204 instructions, including three reserved instruction slots. This table shows only 151 mnemonics, because some instructions (such as `aload_<n>`) have multiple numbered mnemonics (`aload_1`, `aload_2`, etc.) for different operand types. Operand types are *byte* (8 bits), *short* (16-bit integer), *int* (32-bit integer), *long* (64-bit integer), *float* (32-bit floating-point), and *double* (64-bit floating-point). Gray cells in this table indicate floating-point instructions not supported by Javalon-1. *These opcodes are reserved for debuggers and implementation-specific instructions; they never appear in standard Java class files.

Genesis of a Core

If you've never heard of Octera, it's probably because Javalon-1 is the company's first venture into intellectual-property (IP) licensing. Octera is primarily an embedded-systems design center with experience in ASICs, SoCs, and board-level design. Some of Octera's founders and managers have been working with stack-based computers since the days of Burroughs and Unisys, so they feel at home with Java's retro stack architecture. They have also noticed that universities are turning out thousands of young programmers who are more comfortable with Java than with assembly language and C, the longtime staples of embedded-system development. Octera thinks the time is ripe for a microprocessor core that can execute Java bytecodes in hard real-time applications.

Javalon-1 is the product of a four-year design effort. The processor has a 32-bit stack-based architecture created from scratch to run Java bytecodes and nothing else. Most microprocessors, of course, have a native machine language unrelated to any particular high-level programming language. In contrast, Javalon-1's machine language is derived from the Java bytecode instruction set—the processor has no other machine language.

Bytecodes are the virtual machine instructions in Java class files that an interpreter or JIT compiler normally translates into native executable code at run time. As their name implies, bytecodes are 8 bits long, and they manipulate operands ranging in size from 8 to 64 bits. Table 1 lists all the instructions in the Java instruction-set architecture (ISA), some of which aren't supported by Javalon-1.

The Java ISA is unusual when compared with other microprocessor ISAs, mainly because Sun designed it primarily as a software virtual machine, not for implementation in logic as a conventional microprocessor. Sun optimized the Java ISA for high code density (hence the byte-sized instructions) and for easy portability to any CPU architecture. The ISA also has several unusual instructions that support the object-oriented features of the Java programming language.

Because Sun's Java definition largely determined the processor's architecture, Javalon-1 is a rudimentary stack-based CISC machine. Octera also kept the microarchitecture simple. There is no instruction pipeline in the usual sense, unless you consider the fetch and execution units to be a two-stage pipe. For the sake of determinism, there are no conventional instruction or data caches, because their behavior is unpredictable. Nor are there any programmer-visible registers, because Java bytecode instructions manipulate all their operands on the stack.

Although it's possible for programmers to directly access the stack with low-level bytecode instructions—Java bytecode assemblers are rarely used but available—the stack is completely transparent to high-level Java-language programmers. In effect, Javalon-1 is a black box from the Java programmer's point of view. Figure 1 shows what's inside the box.

The only registers in Javalon-1 are a small top-of-stack cache and a handful of datapath registers for internal bookkeeping. There is also a small instruction-fetch buffer. Different members of the Javalon family will have stack caches and fetch buffers of different sizes. Javalon-1 has a mere three-entry stack cache, 32 bits wide, and an eight-byte fetch buffer, enough for eight instructions. According to Octera's code profiling, those are the minimum sizes for a stack cache and fetch buffer that are large enough to significantly boost performance while scarcely affecting the size of the core; they add only about 1,000 gates. Octera found that a stack cache with fewer than three entries caused excessive memory thrashing, canceling any advantage from the cache.

For customers that want higher performance at the expense of, perhaps, a couple of thousand more gates, Octera will offer an eight-entry stack cache and a slightly larger instruction-fetch buffer, doubling the processor's performance. That may seem like a surprising improvement for such minor enhancements, but Octera says the larger cache and fetch buffer dramatically reduce memory contentions between the fetch and execution units. Nevertheless, the enhancements aren't a standard feature in Javalon-1, because Octera believes the processor's performance already exceeds requirements for the intended market. The deeper cache and fetch buffer are luxuries for those who want them.

Javalon-1's internal datapath registers keep track of the stack pointer, class-file index, local-variable index, and other Java housekeeping details. Java programmers will never see them. The fetch unit, shown in Figure 1, prefetches instructions into the fetch buffer and hands them off to the execution unit for decoding and execution. The most common instructions execute in a single clock cycle. If Octera

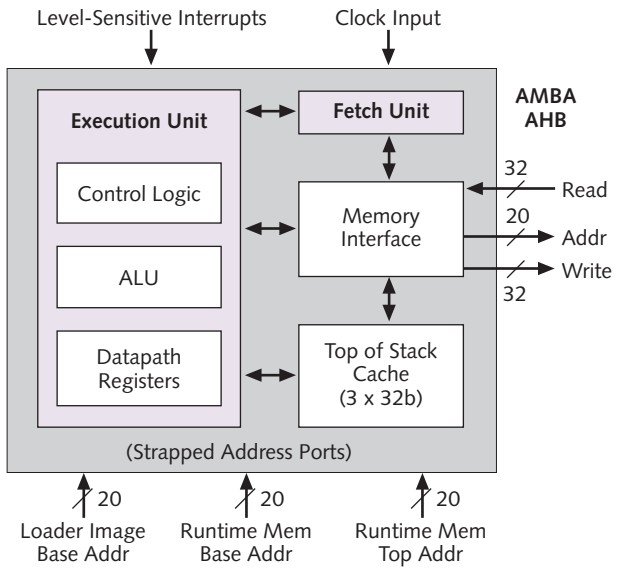

Figure 1. Javalon-1 block diagram. There are no programmer-visible register files or instructions other than what's available in the Java bytecode definition. Internal datapaths are 32 bits wide. Note the AMBA High-speed Bus (AHB) interface, which allows chip designers to use Javalon-1 with other microprocessor cores and peripheral IP.

Octera's Javalon-1 design team, from left: Laury Flora, David Castle, Dale Rigtrup, Bob Noble, Randy Rhodes, Lee Burke, Howard Keller, Jill Kiggens, Joe Mizerak, Gary Whitlock, Carlos Guerra, and Paul Nunnally.

perceives demand for a more powerful processor, a future Javalon might have superscalar pipelines and a fetch unit that predecodes instructions for the execution units. For now, Octera is keeping the core as small as possible.

Thanks to the core's fast datapaths and simple microarchitecture, Octera estimates that Javalon-1 could run at 300MHz (worst case) in a 0.13-micron CMOS process. However, the company believes anything close to 300MHz is overkill for the vast majority of deeply embedded applications, which probably use 8- or 16-bit processors today. Therefore, Octera expects most customers to run Javalon-1 at a more leisurely pace of 1MHz to 10MHz, which would drop power consumption to extremely low levels.

Octera's engineers have early implementations of the processor running at 50MHz in an FPGA and expect to reach 100MHz soon. That's fast enough for many embedded controllers. Indeed, a low-volume embedded system could forgo the high nonrecurring engineering costs of spinning a silicon chip and deploy the Javalon-1 in a PLD instead. With about 25,000 ASIC gates, the core fits into a 150,000-programmable-gate FPGA costing about $20.

Trashing the Garbage Collector

Because Javalon-1 natively executes Java bytecodes, it doesn't need a bytecode interpreter or JIT compiler. That alone can save a megabyte or more of memory, especially since JIT compilers create a duplicate memory image of each program they translate (the bytecode version and the recompiled native version). In fact, Javalon-1 doesn't need a JVM at all, because it either renders the functions of a JVM redundant or omits the features that would require a JVM.

For example, to eliminate Java's prime source of nondeterministic behavior, Javalon-1 dispenses with the JVM's garbage collector. Normally, Java relieves programmers of the burden of memory management by automatically allocating and deallocating memory while a program executes. The JVM does this by running a garbage-collection routine in a background thread to periodically free up memory no longer referenced by objects. Unfortunately, garbage collection is an unpredictable and fairly heavyweight task that can seriously impair a system's ability to respond to real-time events.

Octera's solution is simple: never deallocate memory. Any memory allocated for an object remains encumbered for as long as the program runs. This eliminates the need for a garbage collector, but it also shifts the burden of memory management back to the programmer. Programs written for Javalon-1 must create all their objects at startup and ensure there's enough memory to accommodate them at runtime.

This is sacrilege to Java purists, who will protest that Octera's solution negates one of the biggest advantages of Java and reintroduces a probable source of bugs that plagues C/C++. However, Javalon-1 programs should never suffer from memory leaks—a common memory-management problem in C programs—because they're required to initialize all objects at startup and never deallocate the memory. In effect, a Javalon-1 program "leaks" all its memory during initialization and never needs more, because it doesn't create any additional objects.

Contrary to popular belief, garbage collection isn't required by Sun's JVM specification, which Javalon-1 doesn't religiously follow anyway. The official specification states that the JVM "assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements." (Sun's JVM specification is available online at *http://java.sun.com/docs/books/vmspec/*.)

Nevertheless, Octera isn't deaf to the cries of Sun worshippers. The company says future members of the Javalon family may restore automatic memory management, at least as an option. There are simpler alternatives to a garbage collector, such as object-reference counters.

Different Ways of Going Native

Another Java feature that Javalon-1 makes redundant—in the usual sense, at least—is native method calls. To Java programmers, a native method call is the ability to run program code written in another language (such as C/C++ or assembly language) from a Java program. Normally, when a JVM encounters a native method call, it hands off execution to the system's host processor, which executes the native code and returns any results to the Java program. Native methods are generally the last resort for subroutines that simply won't run fast enough in 100%-pure Java. They also provide a gateway to legacy code so programmers don't have to rewrite a whole project in Java.

In a Javalon-1 system, however, there may not be another host processor, so Java bytecodes *are* the native machine language. In that sense, Javalon-1 doesn't support native methods compiled to executable code for other processors, because it can't run anything but Java bytecodes. This limitation sets Javalon-1 apart from some other embedded processors with bolt-on Java accelerators, such as the ARM1026EJ-S, ARM1136JF-S, and ARCtangent-A5. (See *MPR 10/21/02-02*, "MPF Hosts Premiere of ARM1136.")

Nevertheless, Javalon-1 does support native methods—just not in the way most Java programmers would think. Sun's JVM specification opens a little-known door for defining implementation-specific bytecode instructions that aren't part of the standard Java ISA. Indeed, the ISA reserves two opcodes for this purpose, with the mnemonics `impdep1` and `impdep2` (seen in Table 1). Although implementation-specific instructions are verboten in standard Java class files, programmers can use them in subroutines called as native methods. And because subroutines with implementation-specific Java bytecodes deviate from the standard Java ISA, technically speaking, they are native methods.

In that sense, then, Javalon-1 supports native methods. Octera has created about a half-dozen proprietary instructions that it prefers not to disclose publicly at this time. Octera has also created a bytecode assembler that recognizes those instructions as well as standard instructions. The proprietary instructions appear in native methods stored in the processor's supporting firmware. Among other things, they help execute some standard instructions that Javalon-1 doesn't implement in logic. (More on this later.)

Developers who port existing Java code to Javalon-1 will have to rewrite their C/C++ or assembly-language native methods in Java. One of the tools Octera supplies will scan Java programs for native method calls so they won't be overlooked.

No Downward Class Mobility

To conserve memory, Javalon-1 also doesn't use the Sun-standard Java class-file format. Class files are the product of a Java compiler that converts high-level Java source code into bytecodes. Java class files contain a great deal of symbolic information about variables, objects, and other structures, because, usually, the class files aren't directly executable: a Java interpreter or JIT compiler must translate them into the native machine language of the underlying hardware. But because Javalon-1 *does* execute bytecodes without further translation, it can dispense with some redundant information in the class files and save some application memory.

One downside of abandoning the standard class-file format is that Javalon-1 isn't suitable for embedded systems that must be compatible with the universe of Java software. For example, next-generation mobile phones and PDAs that can download and execute Java applets must be able to handle standard Java class files. Ditto for embedded Web browsers that need to run Java applets.

However, a nonstandard class-file format is an acceptable compromise for a deeply embedded system that doesn't need to dynamically load and run new classes. Indeed, a creative product manager could even pitch it as a security feature that prevents malicious hackers from altering the software of mission-critical systems.

Instead of using standard Java class files, Javalon-1 uses a slightly modified, proprietary version of the Java Executable File Format (JEFF). JEFF was created by the independent J Consortium and was adopted as an ISO standard (ISO/IEC DIS 20970) in 2001. JEFF compresses Java class files to save memory and improves upon the Java archive (JAR) format, an older compressed class-file format created by Sun. Among other things, JEFF allows Java classes to execute from ROM, a valuable feature for embedded systems. (For more information about JEFF, visit the J Consortium's Web site at *www.j-consortium.org/*.)

Octera, a member of the J Consortium, modified JEFF to make the class files even more compact and easier for Javalon-1 to decode. Dynamic class loading is sacrificed, but this also eliminates the need for a conventional class loader and verifier, two other runtime components that occupy memory in a JVM. Octera will provide a design-time utility that converts standard JEFF class files into the modified JEFF files that Javalon-1 requires. Another tool in the package is Octera's own JEFF generator, which converts standard class files into JEFF files, although other JEFF generators may also work.

A Few More Missing Pieces

Octera deleted a few more features from the standard Java platform that were deemed irrelevant for embedded controllers, such as support for a computer keyboard and mouse. If a more sophisticated embedded system (such as a point-of-sale terminal) requires those devices, Javalon-1 could function as a slave to another processor via the AMBA bus.

Two more-serious omissions are multithreading and floating-point math. Multithreading is a stellar feature of Java—built-in classes make it easy to implement—but Octera believes that thread management would use too much memory and impair the processor's real-time response. Likewise, Javalon-1 doesn't support floating-point math at all, either in hardware or software. Octera says a software floating-point

package could occupy as little as 1K of memory, but it would be slow and isn't written yet. Until then, the workaround is to use scaled integers—or a host processor.

In a more familiar compromise, Javalon-1 executes about 80 of the most-complex or seldom-used Java bytecode instructions in software instead of in logic. It's difficult to justify the extra gates required to implement some bytecode instructions in logic; every Java processor we've seen traps at least a few instructions for software execution. For instance, the instruction that creates a new multidimensional array (`multianewarray`) occurs infrequently and requires an indeterminate number of cycles to execute, depending on the data type and dimensions of the array it must construct. The most thoroughly hard-wired Java processor is aJile's aJ-100, which executes all but two bytecode instructions in hardware: `multianewarray` and `athrow`.

Deciding which instructions to implement in logic and which in software depended not only on how often the instruction occurs and its complexity but also on its effect on real-time performance. The reason that Javalon-1 requires as many as 100 clock cycles to service an interrupt—which is somewhat high, by hard real-time standards—is that it can stop executing a complex instruction in software, save the partial state, service the interrupt, restore the partial state, and resume execution where it left off. Javalon-1 can do this even while executing in software an instruction for an interrupt-service routine that is interrupted by a higher-priority interrupt.

Octera says future Javalon processors will slash the worst-case response time to five to ten clock cycles by implementing more instructions in logic. With Javalon-1, the most complex hardware instruction executes in only eight clock cycles (or perhaps a few more, depending on memory contention and the state of the stack cache). Therefore, when Javalon-1 is executing the most-common instructions likely to occur in embedded-controller code, its interrupt-response time is much better than 100 clocks.

Javalon-1 executes a few instructions in software that could significantly affect performance in other ways, depending on the application. The most important of these are integer multiplication and division operations. Octera has designed hardware implementations of those instructions, but they require about 5,000 gates, so Javalon-1 executes them in software instead. The company says a future, higher-performance member of the Javalon family will probably use the hardware multiply/divide implementations.

Thanks to all these economies, Javalon-1 requires only 5K of supporting firmware, exclusive of application memory. The firmware includes the software routines that trap and execute instructions not supported in logic, plus some miscellaneous native methods. All are written in Javalon-1's native machine language: Java bytecodes augmented by Octera's proprietary bytecodes.

In silicon, a Javalon-based chip should compare favorably with other 32-bit embedded processors, whether they can natively execute Java bytecodes or not. At 25,000 gates, Javalon-1 is about the same size as a small ARM or ARC core, especially after the addition of a Java accelerator, such as ARM's Jazelle or DCT's Bigfoot. (See *MPR 2/12/01-01*, "Java to Go: Part 1," and *MPR 1/28/02-04*, "DCT Marches Into Java Processors.") Javalon-1's firmware is small enough to fit inside on-chip flash memory or mask ROM, probably occupying about as much room as the instruction and data caches of a conventional embedded processor.

By Java standards, 5K of firmware is positively microscopic. Normally, a standard JVM requires hundreds of kilobytes or megabytes, especially if it includes such luxuries as a JIT compiler. Even Sun's stripped-down K virtual machine (KVM) for J2ME typically occupies 128K of memory and requires still more memory to operate. Of course, J2ME is a Sun-standard Java platform that has many features Javalon-1 lacks.

Is It Coffee or Postum?

Octera makes no claims that Javalon-1 provides a Sun-standard Java platform. Obviously, Octera's designers made several compromises in their quest to bring some flavor of Java to deeply embedded real-time controllers. Squeezing everything down to 25,000 gates and 5K of firmware simply isn't possible without discarding some baggage. The first question for developers is whether they need anything in that discarded baggage to reach their destination.

The list of what's missing appears lengthy: automatic memory management, native method calls (in the usual sense), universal class-file compatibility, dynamic class loading, keyboard/mouse support, multithreading, and floating-point math. Keep in mind, however, that those features are uncommon in embedded controllers of any flavor. They seem like sacrifices only to developers accustomed to using a richer Java platform on larger systems.

The next question is whether there's enough Java left in Javalon-1 to make it a better alternative than any other embedded processor. The answer is a qualified yes. Octera's diluted version of Java still has some advantages over programming in C and assembly language, including easier code portability to other embedded processors in the future, whether those processors are Java-native or not.

Still, some Java features will be sorely missed. Automatic memory management improves programmer productivity and eliminates a common source of bugs. The ability to invoke native methods written in C or assembly language preserves investments in legacy code. Standard class-file compatibility and dynamic class loading are imperative for Java-enabled Web browsers and applications that must download periodic software updates in the field. Multithreading is invaluable for managing multiple tasks. Floating-point math may be required for some controllers. If a project depends on any of those features, Javalon-1 isn't the best solution.

Finally, there's the question of whether Javalon-1 can satisfy the performance requirements of an application. Is it

fast enough to run the software at the specified power level, and will it respond quickly enough to real-time interrupts?

Benchmark data for Java processors is hard to come by; the often-quoted CaffeineMark is a poor substitute for a rigorous embedded test suite. EEMBC has a new suite of embedded Java benchmarks, but because Javalon-1 isn't 100% Java compatible, the benchmark code probably won't run without extensive modifications. For now, the best way to evaluate Javalon-1 is to extrapolate from its performance in a simulator or FPGA.

Our conclusion is that Javalon-1 will be exactly what some developers are looking for. Others will find Javalon-1's compromises too bitter to swallow. And many developers will remain skeptical that Java is appropriate for hard real-time applications at all. Selling the concept is a bigger challenge than selling the processor. Previous embedded Java processors have found a mixed welcome in the marketplace, with at least one company—Vulcan Machines—recently folding its tent. (See *MPR 7/1/02-03*, "Vulcan Moon Shines Again.")

On the other hand, ARM is enjoying some success with its Jazelle accelerator for the ARM10 and ARM11 cores, and ARC is tempting similar customers with DCT's Java extensions for ARCtangent-A5. What Octera has in common with ARM and ARC is that embedded Java processors are not central to its business. Octera is still primarily an independent design house; Javalon-1 isn't a bet-the-company proposition.

For embedded-system developers, more choices are always better than fewer choices. Octera's Javalon family is a different and creative new option.

Price & Availability

Octera plans to begin licensing the Javalon-1 processor core in 3Q03. It will be supplied in Verilog or VHDL with some special development tools, including a JEFF class-file generator and a JEFF-to-Javalon converter. Future Javalon-family cores will be announced later. Licensing fees have not yet been determined.