The World's First Java^{TM} Processor *

David A. Greve and Matthew M. Wilding

Advanced Technology Center
Rockwell Collins, Inc.

Rockwell-Collins' startling announcement of the world's first Java processor, the JEM1, perhaps should not have been so surprising. The JEM1 is merely the latest in a microprocessor family used in Rockwell products for more than 25 years.

In early 1996 a small group of Rockwell-Collins engineers gathered to review the newly acquired Java^{TM} Virtual Machine (JVM) specification. After a time, one engineer mumbled, "Aren't we already building one of these?" This offhand remark struck a chord with the others and led to a frenzy of activity. In July of 1997 devices arrived at the Rockwell-Collins facility in Cedar Rapids, Iowa and, following initial testing and validation, Rockwell announced to the world the development of the first silicon implementation of the JVM, the JEM1 microprocessor. (See "First Java-specific chip takes wing", EE Times 22 Sept 97.) The JEM1 is the first Java processor because it is the first to execute the JVM instruction set directly.

This announcement came as a surprise to many in the microprocessor industry because Rockwell-Collins is known mainly as a developer of avionics equipment. Nevertheless, Rockwell-Collins has a long history of innovation in the development of computer technology and particular expertise in the application of high-level languages to embedded applications. This history, spanning more than 25 years, ultimately brought together this unique group of engineers and lead to their surprising development.

History of AAMP

In 1969, Collins fielded a sophisticated area-navigation system written in AED, a programming language based on ALGOL and originally developed at MIT. This system was the first commercial avionics product developed using a high-level language. Although the future of such languages in embedded systems was uncertain at the time, it was nevertheless evident that high-level languages had the potential to provide substantial savings in both development and maintenance costs. What was needed to ensure their success was a significant advance in compiler and processor technology.

Collins engineers recognized that stack-oriented processor architectures provided inherent advantages in support of both high-level languages and implementation efficiency. In 1971, an architecture was developed specifically to support efficient compilation of block-structured, high-level languages such as AED. This architecture directly supported multitasking and automatically performed such functions as interrupt state saving and restoration, procedure and function parameter passage, procedure return linkage, and dynamic stack space allocation.

In 1972 this architecture became the foundation for the Collins Adaptive Processing System (CAPS). Collins became the first to apply stack machines to avionics and continues today as the only supplier to do so. Several SSI and MSI processor implementations instantiated the CAPS concepts during the 1970s. The same stack architecture was implemented as a single chip in 1981, making Collins the first to develop a 16-bit CMOS microprocessor of any kind, and leading to a feature article in the August 1982 IEEE MICRO magazine. The 16-bit CMOS design was dubbed the Advanced Architecture Microprocessor, or AAMP. The first single-chip AAMP implementation of the CAPS stack architecture was followed by AAMP designs that provided pipelining, enhanced instruction sets, integration of peripheral functionality, and utilization of improved process technology.

As new AAMP family members were developed, the hardware independent, virtual nature of the AAMP architecture allowed upgrades and improvements to

*Appears as "Stack-based Java a back-to-future step", EE Times, Jan 12 1998, pg. 92

the hardware implementations without changes in the instruction set. Additionally, the simplicity and elegance of the stack architecture supported relatively simple processor designs.

The AAMP family has been very successful inside Rockwell and is employed in a variety of systems, including avionics display systems, high-integration commercial GPS and wireless communications products, avionics collision avoidance systems, and flight control autopilot and autoland systems. The AAMP supports many high-level languages, but Ada and C are by far the most popular.

The AAMP owes much of its success to the unique features it provides to its users. Real-time programmers benefit from automatic state saving, context switching, and interrupt handler activation. System designers enjoy that the AAMP dissipates very little power, operates over the full military temperature range, and provides code densities twice to three times that of most commercial processors. Chip developers appreciate the small core size and verification engineers appreciate unique hardware features supporting software verification. Finally, program managers and many Collins customers appreciate that Collins controls the obsolescence and revision of the AAMP design as they plan systems with 20+ year lifecycles.

The AAMP was a breakthrough when originally developed and its unique capabilities continue to lead to many successful applications. In early 1996 the group of AAMP engineers pondering the JVM specification realized that they could capitalize on the success of the AAMP architecture while leveraging the considerable commercial development surrounding Java by modifying an AAMP to implement the JVM instruction set directly.

A JEM of an Idea

The popularity of Sun's JVM specification is, in a sense, a validation of the AAMP team's many years of effort. The use of a stack-based architecture providing high code density and portability is a mainstay of the AAMP architecture. The JVM leverages these concepts while simultaneously supporting a modern, object-oriented environment.

The JVM also represents an opportunity to serve better the AAMP team's internal customers. For years the AAMP architecture has been supported by a relatively small group of engineers who did everything from processor design to compiler and code generator development to test equipment and debugger design and build. The JVM's strong industry presence provides an opportunity to leverage an industry standard to obtain commercial support. It is now possible to obtain off-the-shelf Java compilers and debuggers that use standard interfaces. Further, the high profile enjoyed by the Java language makes the use of Java alluring to new recruits.

Developing a processor that executes JVM instructions has another important advantage. Other processor manufacturers such as Sun are developing JVM direct execution machines with different power/performance tradeoffs which can provide a path for JEM customers who need higher performance. This frees the JEM development team to concentrate on providing a low-power processor specifically designed to meet the requirements of embedded real-time applications.

JEM Development

Although Java was conceived as a language for use in embedded systems, it has several aspects that fuel debate over its usefulness in such systems. These debates typically involve direct memory access, garbage collection, memory requirements, object-oriented performance, and real-time capabilities. The JEM team understood that they needed to provide solutions to these problems for Java to be a success in the embedded market. In solving these problems, the JEM team benefited greatly from working with Intermetrics, a Massachusetts-based software company with significant Java experience. Intermetrics is primarily responsible for the development of the JEM runtime system and static linker as well as the development of a source-level debugger.

Although the JEM must extend the JVM instruction set to provide direct access to memory, it is undesirable to require native methods written in another language simply to utilize these instructions. The Intermetrics-developed JEM static linker solves this problem by directly replacing selected native method calls with JEM-specific opcodes. This technique allows an entire JEM system to be developed using Java and provides significant portability and language interoperability since only standard Java class files are required to program a JEM system.

Garbage collection is also an issue in the use of Java. Most real-time embedded systems cannot afford the overhead and non-determinism of garbage collection and, more importantly, most embedded systems do not require dynamic memory reclamation. The JEM team studied the Java language carefully and concluded that it was possible to write useful Java software that does not require garbage collection. As a result, the JEM directly implements a static memory allocation scheme useful in systems that do not require or cannot allow garbage collection while also providing the hooks necessary for implementing more sophisticated dynamic allocation and reclamation schemes.

Beyond the issue of garbage collection is the broader issue of Java memory requirements. The entire Java runtime image is far larger than typical embedded systems. In this area the JEM static linker proves useful as it provides optional pruning of the runtime system to minimize such memory resource requirements.

The JVM instruction set is inherently object-oriented and, because the JEM implements these instructions directly, the JEM accrues a significant advantage in efficient implementation of object-oriented languages. The JEM team also invested significant effort in the development of class file data structures that minimize space requirements and maximize computational efficiency. Because most embedded applications will be statically-linked, the data structures are specifically engineered to support dynamic class loading without penalizing staticly linked systems.

The automatic state saving and restoration features borrowed from the AAMP provide extremely fast context switching and interrupt handling. Support has also been added for periodic threads and deterministic scheduling, extending the usefulness of the JEM in real-time embedded applications.

The JEM team believes it has provided solutions to many of the challenges of using Java for embedded applications. Although many may still debate Java's future in embedded systems, the debates have a familiar ring to those involved in the initial introduction of high-level languages to embedded systems. It was once widely believed that the inefficiencies of compiler generated code would limit the applicability of high-level languages to embedded systems. However, compilers improved, as did processor performance, and the productivity and ease of use offered by high-level languages ultimately won over most embedded system programmers. The JEM team believes that the productivity gains and ease of use of the Java language will prove just as compelling as the attributes of the original high-level languages and embedded system developers will again embrace a new software paradigm.

JEM Applications

It is likely that the first application of the JEM processor will be in a product that is part of an avionics suite. Rockwell-Collins has no plans to market the JEM directly, although the processor has been developed with the knowledge that many markets could benefit from the technology brought together in this processor. Because the JEM is low power, it is suitable for battery-powered applications. Its small memory footprint and fast context switching enable it to support deeply embedded systems. The nature of the JVM makes it a good candidate for systems with safety and security requirements. Finally, the small size of the JEM1 core ($6.4mm^2$ in 0.5μ CMOS) makes it inexpensive to fabricate and attractive as an ASIC core.

The world has rediscovered with Java the advantages of a simple, stack-based virtual architecture, and a small group of Rockwell-Collins engineers who have developed processors of this kind for more than 25 years are now leading the way with their introduction of the world's first Java processor.