

Sun Reveals First Java Processor Core

PicoJava Core Executes Some Java Instructions Natively, Interprets Others



by Jim Turley

Adding some meat to the bone, Sun Microelectronics revealed the details of its first Java processor core. Unlike other CPU designs, PicoJava executes the Java programming language as its native instruction set, offering solid Java performance with a minimal memory footprint. Sun expects to see a new generation of chips based on the core accelerate Java-based applications before the end of 1997.

The PicoJava core will form the basis of several Java microprocessors, including Sun Microelectronics' own MicroJava-1 processor. The four public Java-hardware licensees (Mitsubishi, NEC, LG Semicon, and Samsung) all plan to start designing application-specific processors based on PicoJava early next year. This outlook may be overly aggressive, as Sun does not expect to deliver the synthesizable model to its partners before March.

With only simulated performance benchmarks available, PicoJava appears to speed past Pentium by as much as 8× when running Java applications.

Running Java in Hardware

PicoJava is fundamentally different from most microprocessors in that it is been optimized for a specific high-level language. It executes Java bytecodes (the binary form of Java source code) directly as its native instruction set. Sun expects PicoJava-based microprocessors to replace conventional CPUs in embedded systems that run Java applications exclusively. Because no Java interpreter or just-in-time compiler (JIT) is required, Java chips can reduce the application's overall memory requirements.

PicoJava executes all 227 Java bytecodes (plus a few more, described later) using three methods: directly, with microcode, or through emulation. The simplest Java instructions, such as logical and arithmetic operations, execute directly in 1–3 clock cycles. The more complex operations,

such as “quick” method invocations (procedure calls) and array accesses are also executed directly in hardware, though with the help of on-chip microcode. Such instructions can take 4–25 clocks to complete. The final class of instructions, which includes the normal method invocation, is trapped and emulated in software. From the user's point of view, it makes little difference how each Java bytecode is executed, apart from variations in speed.

In all, 176 bytecodes are executed directly in hardware, 28 with microcode, and 23 through emulation. Currently, Sun identifies the manner in which specific bytecodes are executed only to its hardware licensees.

Future PicoJava cores may divide the instruction set differently, implementing more bytecodes in hardware or moving others to microcode. Because all PicoJava licensees will receive the same synthesis model, the first generation of Java microprocessors will all have the same instruction set.

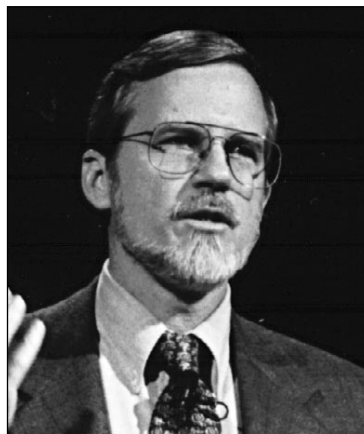
PicoJava extends beyond the defined Java bytecodes with a handful of system-level operations. These extended bytecodes are for low-level OS functions, such as reading from arbitrary memory addresses, clearing or flushing cache lines, performing non-cached loads and stores, and accessing the

core's control and status registers.

The core consists of a fairly normal four-stage pipeline, shown in Figure 1. The first stage fetches a four-byte group, which may contain as many as four complete Java instructions. The second stage decodes one or two bytecodes at once; if one of the bytecodes is a load, the unit may perform some instruction folding (described later).

The simplest bytecodes are executed in a single pass through the third stage; microcoded instructions require multiple clock cycles to execute, effectively lengthening the execute stage by a variable amount. No out-of-order execution is allowed. For complex operations, the nominal four-stage pipeline stalls until the current bytecode executes to completion. After the operation is finished, the result is written to the top of the operand stack.

Accesses to the stack take just one clock cycle, as when a conventional CPU accesses its register file. Loads and stores remain in the execute stage for a minimum of two clock cycles, assuming a cache hit. If the loaded data is required by the immediately subsequent instruction, PicoJava incurs an additional one-cycle load-use penalty.



Robert Garner of Sun describes the advantages of executing Java bytecodes natively on PicoJava.

MICHAEL MUSTACCHI

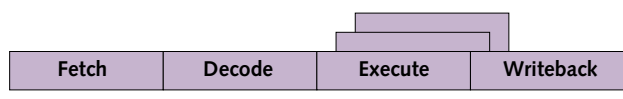


Figure 1. The PicoJava core has a nominal four-stage pipeline, although the execute stage is often extended to multiple clock cycles for Java bytecodes, such as method invocations, that do not execute in a single cycle.

On-Chip Stack Keeps Java Core Moving

From the programmer’s perspective, the Java virtual machine stack resides in memory. For performance, part of that stack is kept on-chip in PicoJava’s stack cache. The stack cache holds the top 64 words of the stack. Conceptually, pushing a 65th item onto the stack would cause the bottom entry to be flushed out to memory or—more likely—an on-chip data cache. Likewise, when the last item is popped, the stack cache would refill from memory (via the data cache, if present).

In practice, PicoJava applies some hysteresis to its fill and flush logic. When the stack cache is nearly full, PicoJava begins moving items from the bottom of the stack cache to memory. Likewise, when the stack cache is nearly empty, the chip starts loading from memory so that the stack cache won’t run dry. Programmers can set the stack cache’s “high water mark” through a five-bit field in a control register.

The number of words transferred between the stack cache and memory is not fixed. Once an impending overflow or underflow condition is detected, PicoJava continues to load or empty the stack cache as long as the program continues to push or pop operands. These automatic fill and flush operations are transparent to the user and independent of pipeline operation.

Since accesses to the stack cache take only one clock cycle, whereas accesses to PicoJava’s data cache take two cycles, it is still possible for a program to overrun or under-run the stack cache if it pushes or pops enough operands.

The basic PicoJava core consists of just the stack cache, integer ALU, and instruction buffer/decode logic. As Figure 2 shows, the instruction and data caches and the FPU are optional.

Opcode Distribution Stays Steady

Sun has simulated two Java applications running on PicoJava, a ray-tracing program and a Java compiler. The results

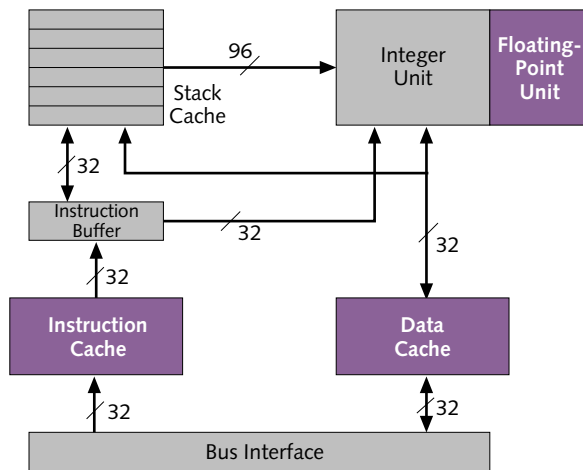


Figure 2. The basic PicoJava core consists of a 64-entry stack cache, integer unit, and instruction decoder. The FPU and caches (in purple) are optional modules.

are illuminating for the information they provide on instruction distribution for two (presumably) representative samples of nontrivial Java applications. Although the two benchmarks are very different in size and intent, they display very similar behavior in terms of bytecode distribution and average instruction size.

The Java compiler is Sun’s own JDK version 1.0.2, running natively on the simulated PicoJava system. JDK contains about 25,000 lines of Java source code, which compiled into 422K of memory. According to Sun, the average instruction length of JDK is 1.84 bytes per instruction (not counting lookup tables), which implies that the static program size is about 235,000 Java instructions. Dynamically, the simulated system executed more than 25 million Java bytecodes.

The ray-tracing application is much smaller: 3,500 lines of source, or about 21,000 Java bytecodes in 36K of memory. At 1.76 bytes/instruction on average, the ray tracer makes somewhat more use of smaller Java bytecodes. Although the program itself is smaller, its execution thread is much longer, with a run length of nearly 383 million Java instructions.

Figure 3 shows that the majority of both applications consists of single-byte bytecodes, a characteristic Sun is quick to tout. Both programs consist of more than 50% of these bytecodes. The next-highest distribution is for three-byte types, that is, byte codes with two extension bytes.

Distribution frequency, however, is only half of the equation. The figure also shows that the static size of each program is actually dominated by the three-byte bytecodes, with more than 50% of each program’s memory space given to three-byte operations. Next come the single-byte operations, which account for about 30% of program size.

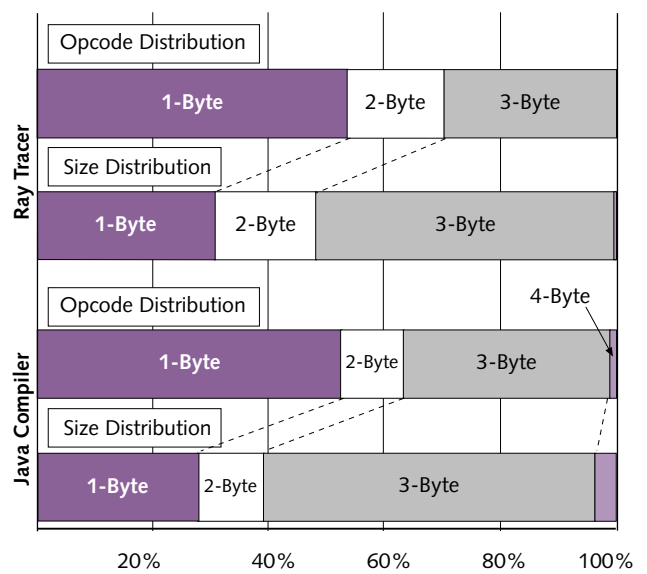


Figure 3. Bytecode distributions for JDK and ray-tracing application show similar characteristics, with more than half the instructions using the single-byte format. However, as a percentage of total code size, these two Java applications consist mainly of three-byte Java bytecodes. (Source: Sun Microelectronics)

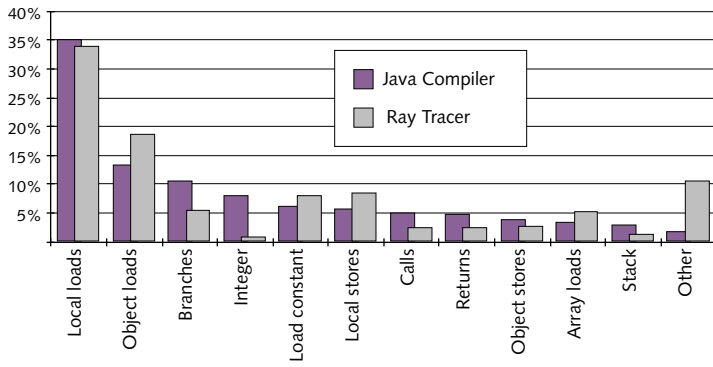


Figure 4. The instruction distribution for two Java applications shows roughly 66% of all operations are loads and stores; over 50% alone are loads. (Source: Sun Microelectronics)

Figure 4 charts instruction frequency for the two applications by instruction type. Both programs are heavily dependent on load operations, which rank first and second in order of frequency. Loads alone make up 48% of the compiler and 52% of the ray tracer; loads and stores combined account for 58% and 64% of instructions for the two programs.

The heavy reliance on loads and stores evidenced by these two samples should be fairly representative of other Java applications. Many loads are not actually memory references but stack manipulation. Stack entries must often be reordered to bring the required operands to the top. Even a perfectly efficient compiler must occasionally reorder stack elements; in real applications, it often dominates the code, as Sun’s two examples amply demonstrate.

Special Case Eliminates Some Stack Manipulation

In keeping with a stack-oriented philosophy, most Java instructions can operate only on the one or two items at the top of the stack. Typically, the top two items are replaced with their result, as when two numbers are added together. The operation pops two operands and pushes one back, leaving the stack one operand smaller.

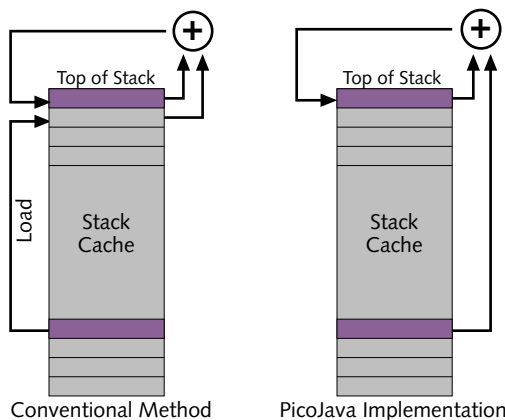


Figure 5. PicoJava transparently eliminates the preparatory load instruction before two-operand computation instructions, taking one of the operands directly from its stack location.

In practical terms, the two needed operands are rarely on the top of the stack. Thus, Java programmers often need to insert a load instruction before a two-operand instruction to copy one of the operands to the top of the stack.

PicoJava’s designers took advantage of this common construct and used it as an opportunity to implement simple instruction folding. When PicoJava detects the combination of a load instruction followed by a destructive two-operand computation, it effectively executes both instructions simultaneously.

As Figure 5 shows, the contents of the source register are routed directly to one input of the ALU instead of to the top of the stack, eliminating the load. In other hardware stack-machine implementations, such as Patriot Scientific’s ShBoom (see 100501.PDF), the ALU has a data path to only the top two stack elements. Because PicoJava’s stack cache is implemented as a circular queue, the design already contains complete datapath logic for all 64 elements. The happy side effect is that PicoJava can forward operands from any two arbitrary stack elements or, in this case, one arbitrary element and the top of the stack.

Sun’s simulations suggest a significant portion of a program’s load instructions can benefit from this technique; composite results are shown in Figure 6. Although each folded load saves a clock cycle, the load instructions themselves are still part of the bytecode stream, so this technique saves nothing in code size or transmission time.

Java Chips Could Outrun Others on Java Code

Table 1 presents Sun’s results for both PCs and the simulated PicoJava core running at 100 MHz. Because the simulated PicoJava system does not incur actual I/O, Sun added 0.8 seconds for the JDK and 0.4 seconds for the ray tracer for I/O overhead. These estimates were based on similar tasks running on a SparcStation system.

Sun’s results indicate that a PicoJava-based processor running at 100 MHz would execute these applications about three times faster than JIT-compiled code on a 166-MHz Pentium, or a whopping 18 times faster than on a 486DX-33.

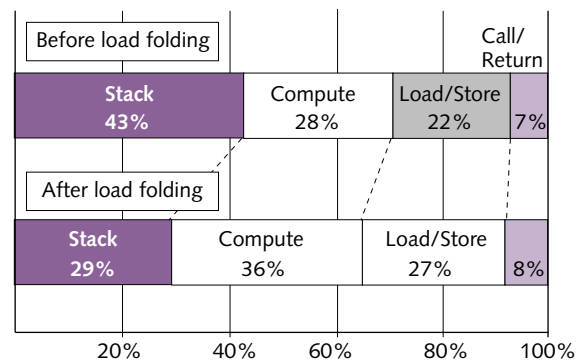


Figure 6. PicoJava’s load-folding technique reduces the number of stack-related operations by about one-third. (Source: Sun)

The small differences between JIT-compiled and interpreted code do not jibe well with results from other sources. Recent improvements in JIT compilers from Symantec, Borland, Sun, and others have widened the gap between interpreted Java code and JIT-compiled code. Their performance on a number of Java benchmarks differs by an order of magnitude (see [100705.PDF](#)). Sun's position is that this difference is much smaller when compiling or interpreting large applications rather than synthetic benchmarks.

Of course, Sun's tests measure different processors all running Java bytecode, which is equivalent to measuring a PowerPC running 68000 binaries. Compiling the Java source for different instruction sets should produce much more even results. Such Java compilers are available for a number of microprocessor architectures.

Borland reports a 2–3× expansion in code size when its JIT converts Java bytecode to x86 object code; the expansion factor for RISC targets is greater. This is one of the advantages of native execution that Sun pushes. Java chips need no interpreter or JIT, and they don't need to convert bytecodes. The size of PicoJava's emulation and trap library is only about 8K, far smaller than even a rudimentary interpreter. For a target system that will execute Java bytecodes (as opposed to code compiled from Java source), a native Java implementation should bring significant savings in memory, reducing the cost, power, and physical size of a Java-based device.

Complex Instruction Set Leads to Bulky Die Size

Although a PicoJava integer core has never been created, Sun conservatively estimates it will usurp 8 mm² in a 0.35-micron ASIC process; the FPU adds another 5.5 mm². For its benchmarks, Sun simulated a core, a 4K instruction cache, and an 8K data cache, bringing the total to 21 mm².

At 8 mm² for an integer core (or 13.5 mm² for the core with an FPU), PicoJava is far larger than contemporary 32-bit RISC or CISC cores in equivalent processes. Recent ARM and MIPS cores, for example, come in at under 3 mm², IBM's PowerPC 401 measures just 4.5 mm², and even the relatively complex ColdFire core uses less than 6 mm². In terms of CPU cores, PicoJava is about as bulky as they come.

One could argue, however, that PicoJava is more complex than any of those competitors, executing high-level constructs natively (more or less) rather than register-level machine code. By the time PicoJava is integrated in an ASIC, the few extra millimeters won't amount to much of a penalty.

Java Chips Make Sense for Java Applications

For designers debating the relative merits of Java processors versus general-purpose processors, the decision tree focuses on software: Will the application be written in Java? If so, will it always be converted into bytecode? If so, is that bytecode the exclusive software-delivery mechanism?

If the answer to any of these is no, native Java execution doesn't make sense. New code, whether it's written in Java or another language, can be compiled for nearly any processor

Price & Availability


The PicoJava-1 synthesizable model is scheduled to be delivered to licensees in March. Sun is expecting working PicoJava-based ASICs to be available before the end of 1997. For more information, contact Sun Microelectronics (Mountain View, Calif.) at 415.336.5714 or check the Web at www.sun.com/sparc/java.

architecture. Even Sun admits that Java chips can't keep up with conventional microprocessors on conventional code.

Java bytecodes can obviously be executed on a number of different microprocessors, although with some penalty in memory usage and, perhaps, speed. But these systems have the ability to run other operating systems and applications, vastly broadening their horizons. Although the performance of early JITs was abysmal, the competition among Java tool developers has forced them to improve dramatically. As the performance of JITs improves, any speed penalty for using a non-Java chip decreases.

If the system needs to run anything besides bytecodes, Java chips will fall flat. The definition of the Java virtual machine allows Java applets to run on non-Java processors; the reverse condition does not exist. If the system will download and execute nothing but Java bytecodes, then a native Java implementation makes sense. PicoJava-based chips should be able to turn bytecodes into useful work more efficiently than any other CPU design.

Java has been enthusiastically embraced by companies large and small because of its portability and reliability features. For systems that run only Java, a Java processor may be the obvious choice. But with chip details and pricing still months away, comparisons with other CPUs are premature.

The success of Java chips obviously rides on the success of Java itself—not just as a language, but as a distribution format. If, as today, there are no significant Java applications, there will be no demand for native Java processors. Sun and its licensees are betting that if they build the chips, the applications will come. 

| | | PicoJava 100 MHz | Pentium 166 MHz | | 486DX 33 MHz | |
|------------|-------|---------------------|--------------------|-----------|-----------------|-----------|
| Method | | Native | JIT | Interpret | JIT | Interpret |
| JDK 1.0.2 | Time | 1.0 + 0.8 | 5.6 | 12.3 | 32.5 | 82.8 |
| | Ratio | 1.0× | 3.1× | 6.8× | 18.0× | 46.0× |
| Ray Tracer | Time | 12.6 + 0.4 | 38.8 | 105.0 | 331.9 | 772.0 |
| | Ratio | 1.0× | 3.0× | 8.0× | 25.5× | 59.4× |

Table 1. Comparing two Java applications running on PCs and a simulated PicoJava processor indicates that PicoJava runs faster than either interpreted or compiled code. For the PCs, the interpreter is Sun's JDK 1.0.2 for Windows 95; the JIT compiler is Symantec's Cafe 1.5 for Windows 95. The 486 system has a 256K cache and 16M of RAM. The Pentium system contains 256K of cache and 32M of RAM. (Source: Sun Microelectronics)