Implementing the Java Virtual Machine

Java's Complex Instruction Set Can Be Built in Software or Hardware

by Brian Case

One of the unusual features of Java is that programs are compiled into binary code not for a particular microprocessor but for a "virtual" processor called the Java virtual machine (VM). The Java VM instruction set is currently implemented only by software emulators, but Sun Microelectronics has announced plans to implement a family of microprocessors that will execute VM instructions in hardware *(see* **1002MSB.PDF***)*. The complex instructions and stack architecture of the Java VM will make it challenging to implement efficiently in hardware, but there may be a vast market for such devices.

Java VM Based on Stack Architecture

The Java VM is a 32-bit stack (or zero-address) architecture. In a stack architecture, operands are fetched from memory and pushed onto the last-in-first-out (LIFO) expression-evaluation stack, which Java calls the value stack (VS). Instructions pop their operands from the stack and push their results onto the stack. Final results are popped from the stack and stored to memory.

The value stack is analogous to the register file of a RISC architecture: operations can reference only operands on the stack. In fact, Java microprocessors will probably use a register file to implement the value stack (although VM instructions cannot access the stack like a register file).

According to its specification, the Java VM has four processor registers: PC (program counter), VARS (points to the base of the current set of local variables), OPTOP (current top of the evaluation stack), and FRAME (points to the base of the current execution environment in memory).

Java is an object-oriented language, which, in the extreme, requires that the type of all objects (operands) be determined at run time. To improve performance for the common case of scalar operands, the Java VM directly supports the basic scalar types shown in Table 1. The VM has instructions to directly load, store, and operate on these built-in types. There are no unsigned integer types and no pointers. An object reference is the Java equivalent of a C-language pointer, but arithmetic on references is prohibited by the language.

The 64-bit *long* and *double* data types require two 32-bit words in memory and on the value stack. The order of the storage—least-significant or most-significant word first—is a choice made by the VM implementer. It is important only that each VM be internally consistent. Instruction operands, on the other hand, are always stored most-significant-byte first. (Note that the file format for Java VM programs is strictly defined to guarantee transportability to all machines, regardless of a particular machine's byte ordering.)

Variable-Length Instructions

The Java VM architecture currently defines 203 instructions, as Table 2 shows. Many are similar to those of a traditional microprocessor, while others are specifically for Java support. Java VM instructions, or bytecodes, consist of a one-byte opcode followed by zero or more operand bytes. Most instructions are either one, two, or three bytes long.

The most common instruction operand is a one- or two-byte index into the class-constant pool. The constant pool is the data structure of a Java VM binary program and is essentially a symbol table. Even the bytecodes for a program are stored in the constant pool.

To transfer a variable to or from the stack, an index into the constant pool is formed from an instruction's operand byte(s). The corresponding entry in the constant pool contains enough information to identify the name, type, and location of the variable. The Java VM uses the information from the constant pool to "resolve" the reference. Once resolved to an address, the actual value can be accessed. Method invocations (procedure calls) and field references (object-oriented variable references) also require resolution.

Run-time resolution is the key to part of Java's appeal: it provides dynamic linking to support independent, incremental compilation of Java classes *(see* **100402.PDF***)*.

Simple Instructions Operate on Stack

The Java stack architecture shares a trait with RISC architectures: memory is referenced only by loads and stores. The IINC instruction, which adds an 8-bit signed immediate directly to a local variable without using the stack, is the VM's only memory-to-memory instruction. Presumably, the VM designers included this instruction because incrementing a local variable occurs frequently in real programs.

Data Type	Length (bytes)	Description
byte	1	Signed 2s-complement integer
short	2	Signed 2s-complement integer
int	4	Signed 2s-complement integer
long	8	Signed 2s-complement integer
float	4	IEEE 754 single-precision floating-point
double	8	IEEE 754 double-precision floating-point
char	2	Unsigned Unicode character
object	4	Reference to Java object or array
returnAddress	4	Used with jsr, ret, jsr_w, ret_w

Table 1. Java data types. All arrays are managed as objects. Aside from object references and return addresses, there are no pointers.

Push Constant		
BIPUSH	2	1-byte signed integer
SIPUSH	3	2-byte signed integer
LDC1	2	Const.-pool (8-bit index)
LDC2	3	Const.-pool (16-bit)
LDC2W	3	Const.-pool long or DP
ACONST_NULL	1	NULL object reference
ICONST_M1	1	Integer constant –1
ICONST_n	1	Integer const. (n ∈ 0..5)
LCONST_0	1	Long int. constant 0
LCONST_1	1	Long int. constant 1
FCONST_n	1	FP constant (n ∈ 0..2)
DCONST_0	1	DP constant 0
DCONST_1	1	DP constant 1
Load/Store Local Variables		
xLOAD	2	Load local var. of type x
xLOAD_n	1	Load local var. n
xSTORE	2	Store local var. of type x
xSTORE_n	1	Store local var. n
x ∈ (I,L,F,D,A)		n ∈ (0,1,2,3)
IINC	3	Add signed 8-bit immed. to local variable
WIDE	2	Expand local variable index to 16 bits
Stack Management		
NOP	1	No operation
POP	1	Discard top word
POP2	1	Discard 2 top words
DUP	1	Copy top word
DUP2	1	Copy 2 top words
DUP_X1	1	Copy top, insert 2 down
DUP2_X1	1	Copy 2 top, insert 2 dn.
DUP_X2	1	Copy top, insert 3 down
DUP2_X2	1	Copy 2 top, insert 3 dn.
SWAP	1	Swap 2 top words
Arithmetic		
xADD	1	Add of type x
xSUB	1	Subtract of type x
xMUL	1	Multiply of type x
xDIV	1	Divide of type x
xREM	1	Remainder of type x
xNEG	1	Negate of type x
x ∈ (I,L,F,D)		

Shift and Logical		
ISHL	1	Integer left shift
ISHR	1	Integer arith. right shift
IUSHR	1	Integer logical right shift
LSHL	1	Long left shift
LSHR	1	Long arith. right shift
LUSHR	1	Long logical right shift
IAND	1	Integer AND
LAND	1	Long integer AND
IOR	1	Integer OR
LOR	1	Long integer OR
IXOR	1	Integer XOR
LXOR	1	Long integer XOR
Branch and Compare		
IFEQ	3	Branch if VS[0] == 0
IFNULL	3	Branch if VS[0] == NULL
IFLT	3	Branch if VS[0] < 0
IFLE	3	Branch if VS[0] <= 0
IFNE	3	Branch if VS[0] != 0
IF_NONNULL	3	Branch if VS[0] != NULL
IFGT	3	Branch if VS[0] > 0
IFGE	3	Branch if VS[0] >= 0
IF_ICMPEQ	3	Branch if VS[1] == VS[0]
IF_ICMPNE	3	Branch if VS[1] != VS[0]
IF_ICMPLT	3	Branch if VS[1] < VS[0]
IF_ICMPGT	3	Branch if VS[1] > VS[0]
IF_ICMPLE	3	Branch if VS[1] <= VS[0]
IF_ICMPGE	3	Branch if VS[1] >= VS[0]
LCMP	1	Long integer compare
FCMPL	1	FP compare (–1 on NAN)
FCMPG	1	FP compare (1 on NAN)
DCMPL	1	DP compare (–1 on NAN)
DCMPG	1	DP compare (1 on NAN)
IF_ACMPEQ	3	Branch object refs. ==
IF_ACMPNE	3	Branch object refs. !=
GOTO	3	Branch (16-bit offset)
GOTO_W	5	Branch (32-bit offset)
JSR	3	Branch subroutine (16-bit)
JSR_W	5	Branch subroutine (32-bit)
RET	2	Indirect branch (8-bit)
RET_W	3	Indirect branch (16-bit)
Exception Handling		
ATHROW	1	Force exception or error

Array Management		
NEW_ARRAY	2	Allocate array
ANEW_ARRAY	3	Allocate array of objects
MULTI_ANEW_ARRAY	4	Allocate multidim. array
ARRAY_LENGTH	1	Get length of array
xALOAD	1	Load/store value of
xASTORE	1	type x to/from array
x ∈ (I,L,F,D,A,B,C,S)		
Function Return		
IRETURN	1	Return integer
LRETURN	1	Return long integer
FRETURN	1	Return FP
DRETURN	1	Return DP
ARETURN	1	Return object reference
RETURN	1	Return (void)
BREAKPOINT	1	Invoke breakpoint handler
Switch-Table Branches		
TABLE_SWITCH	>16	Indexed jump table
LOOKUP_SWITCH	>16	Key-match jump table
Conversion Operations		
s2d	1	Type s to long type d
s, d ∈ (I,L,F,D) but s ≠ d		
INT2BYTE	1	Integer to signed byte
INT2CHAR	1	Integer to character
INT2SHORT	1	Integer to short
Monitors (Critical Sections)		
MONITOR _ENTER	1	Enter locked code
MONITOR_EXIT	1	Exit locked code
Object Field Manipulation		
PUTFIELD	3	Set field in object
GETFIELD	3	Fetch field from object
PUTSTATIC	3	Set static field in class
GETSTATIC	3	Fetch static field
Method Invocation		
INVOKE_VIRTUAL	3	Invoke method (run-time)
INVOKE_NONVIRTUAL	3	Invoke method (compile)
INVOKE_STATIC	3	Invoke class method
INVOKE_INTERFACE	5	Invoke interface method
Miscellaneous Object Handling		
NEW	3	Create object
CHECK_CAST	3	Check object type
INSTANCE_OF	3	Compare object type

Table 2. The Java virtual machine has a large instruction set. The table lists the mnemonic, the instruction length (in bytes), and a brief description of each instruction. Groups of mnemonics that differ only in the type character(s) are listed as a single mnemonic, with the set of valid type characters given below the grouped mnemonics (e.g., xLOAD stands for ILOAD, LLOAD, FLOAD, DLOAD, and ALOAD). VS[0] is the top entry in the value stack; VS[1] is the first entry below the top of the value stack.

The constant instructions push a constant value onto the value stack. The constant-pool pushes (LDC1, LDC2, and LDC2W) resolve a constant-pool item and push the resolved value onto the stack. Strictly speaking, the other constant instructions are redundant with the constant-pool pushes, but since no resolution is required, they are much faster. Two instructions (BIPUSH and SIPUSH) push immediate integer constants, and several one-byte instructions implement the common cases with both space and time savings.

Local variables are allocated as part of the activation record (run-time stack frame) for a called function. Because the number and types of these variables are known to the Java compiler, they can be allocated directly on the stack and require no complicated resolution process to determine their location in memory. (Variables that are not fully disclosed at compile time are called fields and do require full resolution before they can be accessed.)

The instructions for loading and storing local variables form the largest single group of VM instructions. For each static scalar type (*int, long, float, double,* and *object*), there are 10 instructions:

- A two-byte load with eight-bit local-variable index
- A two-byte store with eight-bit local-variable index
- Four one-byte loads for local-variable indexes 0–3
- Four one-byte stores for local-variable indexes 0–3

The WIDE instruction is a prefix—in the grand tradition of

the x86—that expands the local-variable index to 16 bits for the general loads and stores.

The array-management instructions are used for allocating arrays and for loading and storing array elements. Note that Java has instructions to allocate new arrays but none to deallocate them. In Java, dynamically allocated space is reclaimed by automatic garbage collection in the VM. The garbage collector determines when allocated space is no longer needed without explicit prompting from the program.

The arithmetic, shift, logical, and convert instructions are classic stack-architecture operations: one-byte opcodes that fetch operands from the top of the stack and push results back onto the stack. The *long* and *double* operations store the two 32-bit halves of the result in consecutive stack locations; the order is implementation dependent. For noncommutative operations (subtract, divide, remainder, shifts), the left operand is the deeper of the two on the stack.

Efficient Conditional-Branch Architecture

The Java VM has a fairly conventional conditional-branch architecture that implements compare-and-branch constructs efficiently in time and space. Most of the compare-and-branch instructions assume the value at the top of the value stack is an integer; IF_NULL and IF_NONNULL assume the top value is an object reference. The value is popped from the stack, tested against zero (or NULL) for the appropriate relationship, and a branch is taken if the relationship is satisfied. Each instruction provides a signed 16-bit offset.

The IF_ICMP instructions assume the two top stack elements are integers. These elements are popped and compared, and a branch is taken if the specified relationship is satisfied. The branch offset is a signed 16-bit value. The IF_ACMP instructions are the same except they assume the two top stack elements are references to objects.

Explicit compare instructions are provided for the other basic scalar types (*long, float,* and *double*). These instructions pop the two top values (four stack words for LCMP and DCMP), compare them, and push an integer result. The result of each comparison is either 1, 0, or –1. For LCMP, 1 is pushed if the deeper value is greater than the top value, 0 is pushed if they are equal, and –1 is pushed if the deeper value is less than the top value. The same result determination holds for the two FCMP and two DCMP instructions, but these also test for NaN (not a number) operands. The FCMPL and DCMPL instructions produce –1 when an operand is a NaN; FCMPG and DCMPG produce 1 when an operand is a NaN.

The GOTO and GOTO_W instructions are unconditional branches. The JSR and JSR_W instructions branch to a local subroutine without doing a method lookup; they push their return address on the value stack. The _W forms of these instructions specify a 32-bit instead of a 16-bit offset. The RET and RET_W instructions are simple indirect branches that get their branch value from a local variable.

Note that JSR stores its return address on the stack, but RET gets its return address from a local variable; according to the Java VM documentation, this asymmetry is intentional to ease support for Java's exception handling. The GOTO, JSR, and RET instructions are not used to implement method or procedure calls.

Switch-Table Branches Add Complexity

The switch instructions, TABLE_SWITCH and LOOKUP_SWITCH, can be very long, both in execution time and code size. The TABLE_SWITCH instruction specifies a jump table of 32-bit code addresses that can be directly indexed. The top value on the stack (which must be an integer) is popped, compared against the bounds of the table and, if in range, used to select one entry (just like an array reference). The VM then jumps to the address in that entry.

The LOOKUP_SWITCH instruction specifies a sequence of key-value pairs. The top of stack is compared against the key in successive key-value pairs until a match is found or all pairs have been searched. If a match is found, the VM jumps to the address in the value associated with the matching key. If no match is found, the VM jumps to the default address (specified at the beginning of the key-value list).

The compiler makes sure jump tables and key-value lists are aligned on 32-bit boundaries by inserting up to three pad bytes immediately after the opcode.

Stack Management Reduces Loads and Stores

The stack-management instructions can rearrange and duplicate operands on the value stack to, for example, reduce unnecessary loads when a value on the stack is used twice. With a standard register file, storing a value from a register to memory does not delete that value from the register file; it can be used again if needed. With a stack, storing a value to memory pops the value from the stack. If the value is needed again, it can be duplicated with the DUP instruction before storing it to memory.

Figure 1 shows an example. In this code fragment, the value in *k* is stored to memory by the first statement but then used immediately by the second statement. Without the DUP instruction, the stack code would contain a store followed by a load to reclaim the value for use in the second statement.

Code Fragment	Without 'dup'	With 'dup'
	.	.
	.	.
	.	.
.	iload_0	iload_0
.	iload_1	iload_1
.	add	add
k = i + j;	istore_2	dup
x = k + x;	iload_2	istore_2
.	iload_3	iload_3
.	add	add
.	istore_3	istore_3
	.	.
	.	.
	.	.

Figure 1. Use of the DUP instruction to eliminate a load.

With the DUP instruction, as shown in the figure, the load is eliminated, which saves time and code space if *k* cannot be referenced with one of the one-byte ILOAD instructions.

Note, however, that even with the DUP instruction, the stack code is less efficient than the code for a traditional, register-based instruction set. Traditional code would simply reuse the value of *k* from the register file; there would be no need to reload or duplicate it.

The overhead of these instructions in high-end implementations can be reduced by superscalar Java VM implementations that execute stack-management operations in parallel with surrounding instructions.

Method Invocation Is a Complex Procedure

The Java VM has four method invocation instructions (i.e., object-oriented procedure calls). These instructions specify a constant-pool entry with a two-byte index following the opcode and expect the arguments for the method call to be on the value stack.

The INVOKE_VIRTUAL instruction is the normal method call in Java programs. Before INVOKE_VIRTUAL is executed, an object reference and the arguments for the method call are pushed onto the value stack. The object reference specifies the object in which the method (procedure) will be found. The constant-pool item (specified by the index following the opcode) is resolved to a so-called method signature (which specifies how the method call is set up). The signature matches exactly one of the signatures in the object pointed to by the object reference on the value stack. Searching the object's method signature list to find the match results in an index; this index is then combined with the object's dynamic type (determined at run time) to find a pointer to the method block for the matched method signature. The method block specifies the type of the method, the number of arguments expected, and where to find the code for the method. Finally, after all the lookups, searching, and matching, the called method can begin executing.

The above explanation may be difficult to follow, but it illustrates the complexity of an object-oriented procedure call in the Java VM. Calling a method is a sophisticated operation, and Java microprocessors will need to either have extensive microcode support or trap to a software emulator. Perhaps some Java VM instructions will be too complex even for a Java microprocessor!

The remaining method invocation instructions are similar to INVOKE_VIRTUAL but handle other Java programming situations.

VM Supports Java Fields, Objects, and Threads

The field-manipulation instructions are used to reference static data (class variables) in an object. The data type and location within the object is determined at run time. Each field instruction specifies a two-byte constant-pool index following the opcode. The constant-pool entry is a reference to a class name and a field name, which are resolved to a field block pointer. The block pointer contains a field offset (from the beginning of the object) and a length (in bytes). The offset and length are used to access the field. The GET_FIELD and PUT_FIELD instructions are essentially load and store operations for global variables in an object, which is specified by an object reference on the value stack.

The Java VM has the concept of monitors, which allow an execution thread to gain exclusive access to an object. Other threads cannot access an object that has an active monitor. The MONITOR_ENTER and MONITOR_EXIT instructions expect an object reference on the top of the value stack. If MONITOR_ENTER is attempted on an object that is locked by another thread, the thread attempting to lock the object must wait until the other lock(s) are released. When MONITOR_EXIT releases the last lock on the object, a thread waiting to complete its MONITOR_ENTER is allowed to proceed.

LDC1_QUICK	ANEW_ARRAY_QUICK
LDC2_QUICK	MULTI_ANEW_ARRAY_QUICK
LDC2W_QUICK	INVOKE_VIRTUAL_QUICK
PUT_FIELD_QUICK	INVOKE_VIRTUAL_OBJECT_QUICK
PUT_FIELD2_QUICK	INVOKE_NONVIRTUAL_QUICK
GET_FIELD_QUICK	INVOKE_STATIC_QUICK
GET_FIELD2_QUICK	INVOKE_INTERFACE_QUICK
PUT_STATIC_QUICK	NEW_QUICK
PUT_STATIC2_QUICK	CHECK_CAST_QUICK
GET_STATIC_QUICK	INSTANCE_OF_QUICK
GET_STATIC2_QUICK	

Table 3. The Java VM "quick" instructions. The VM overwrites the normal versions of these opcodes with the quick versions, which run faster by eliminating the resolution of symbolic references.

"Quick" Instructions Optimize Performance

Sun's Java VM implementation currently uses an optimization strategy that involves self-modifying code. For a software interpreter, this is no big concern, since VM code is actually data from the viewpoint of the underlying microprocessor; self-modifying VM code is the same as any other volatile data, which processors and data caches already handle.

In software emulators, the VM uses hidden opcodes not exposed as part of the defined architecture. These opcodes are suffixed _QUICK in the Sun documentation. Table 3 lists the quick opcodes defined in the 8/22/95 VM specification.

The quick opcodes have the same semantics as the defined, documented opcodes they replace, but the quick versions assume that the constant-pool resolution process has already been completed successfully. Thus, where a quick opcode exists, the full semantics of the instruction are carried out on only the first execution. The second and subsequent executions of the instruction are faster.

The VM uses the quick opcodes by overwriting the normal version with the quick version. Thereafter, the quick version is automatically executed.

One problem with this strategy is that it consumes many opcodes, limiting the ability to expand the Java VM instruction set in the future. The problem will probably not

manifest itself soon, however, since only 203 of 256 possible opcodes are already defined, and the quick opcodes consume only 21 of the remaining 53.

Another problem is that self-modifying code in a hardware Java processor can be costly, because it may require flushing the structures that improve performance, including pipelines, prefetch buffers, decoded instruction caches, and reservation stations. On the other hand, flushing will occur only once for each quick opcode used in a program. For quick opcodes in loops, the flush cost is negligible if the loop iterates many times. For quick opcodes not in loops, the cost is probably also negligible, since programs spend most of their time in loops. It remains to be seen if Java microprocessors will use self-modifying code.

Implementing the Java VM in Hardware

The software implementation of the Java VM has proved itself to be powerful and portable, but its performance limits the application space. One estimate holds that interpreted Java applications run 10% as fast as native applications *(see* **1002MSB.PDF***)*. Our preliminary measurements show a range of from 3% to 10% the speed of x86 native code (produced by a Borland 32-bit compiler). It is reasonable to expect a Java microprocessor to improve this performance by a factor of five or more.

One of the most distinguishing characteristics of a Java microprocessor is its microcode. The simplest and most frequent instructions need no microcode, but the complex instructions will likely use microcode to sequence the actions of constant-table resolution, method searching, and looping through switch tables.

Most VM instructions, including the local load/store, constant push, compare/branch, and scalar computation instructions, can be implemented in much the same way they are implemented in current high-end processors. The powerful technique of register renaming can be applied to value-stack locations to break the bottlenecks caused by operand storage conflicts.

Superscalar techniques can be applied to Java VM stack code. It is possible to recognize a group of several instructions—e.g., load, load, add, store—and translate them into a single three-address internal operation. Furthermore, a high-performance implementation will cache the current stack frame (or a few frames) in processor registers; thus, if local variables are used, the above sequence could be implemented as a single register-to-register internal operation.

Other performance-enhancing techniques—instruction/data caches, branch prediction, reservation stations, and reorder buffers for speculative execution—should be applicable in a Java microprocessor. In short, while the Java VM's stack architecture and dynamically linked, object-oriented design present microprocessor designers with some unusual details, no fundamentally new problems are created.

One difficult problem for Java microprocessor designers will be finding inexpensive ways to achieve good performance for constant-pool resolution and method invocation. Resolution and invocation are inherently serial and require many clock cycles and memory references. In the current implementation, many of the quick opcodes still involve at least one level of indirection.

For a traditional microprocessor, the compiler and linker resolve all symbolic references and generate a simple addressing mode that requires only one memory reference to load a value from memory. The Java system uses dynamic linking, which requires the VM to resolve symbolic references at run time.

Keeping a hardware cache of the results of the most recent resolution and method invocation can improve performance, but this cache will not have a 100% hit rate and will make Java microprocessors more expensive than comparable traditional chips. In general, the major problem with the basic idea of high-performance Java microprocessors is that to achieve a given level of performance, Java chips will need more hardware. It may even be impossible for the fastest Java chip to compete with the fastest traditional microprocessors.

Remember that the Java VM performs automatic garbage collection. The garbage collector decides when to perform collection, and during collection a Java chip will not be executing application code. Sun expects its Java chips will use a software garbage collector. Application performance will be reduced—possibly even halted—while garbage collection is performed. It is difficult, but not impossible, to perform incremental garbage collection in small-memory, real-time embedded environments, limiting the suitability of Java chips to these environments.

Along with these problems, the Java VM gains something from its stack orientation and dynamic linking of symbolic references. Stack code is dense and improves portability by making a VM implementation simple. A stack-code interpreter maps well onto traditional architectures with few general registers, such as the x86.

Dynamic linking has a significant advantage for the distribution of program executables. A small change in one of the classes (code files) of a large program does not force recompilation of all the other classes. Only the compiled code for the modified class needs to be distributed. If the distribution medium is a network, in particular the Internet, this modularity can make program updates feasible where they would otherwise be infeasible.

The Java VM lacks the system-management instructions and resources (supervisor-mode bits, virtual-address translation, etc.) that are taken for granted in traditional 32-bit microprocessors. Before a Java microprocessor can honestly function as a standalone device, the lack of system-management capabilities must be addressed. In any modern general-purpose system, an operating-system kernel is required to support resource allocation, processes, and threads. A legitimate Java microprocessor must include features that are not part of the current VM specification, such

as an interrupt structure, a processor status word, and perhaps some memory protection and address translation.

Java Chips Limited to Java Applications

Compared with traditional microprocessor architectures, the Java VM is novel. Although other microprocessors have pioneered some of the Java VM's features, the combination of a stack architecture and full support for dynamic linking and object-oriented concepts has never been seen in a mainstream, high-volume microprocessor.

There are no fundamental barriers preventing the development of Java microprocessors with good performance or with very low cost. The problem is that inexpensive implementations will also likely have uncompetitive performance, while high-performance implementations will cost more than traditional processors with similar performance. Java chips will likely never compete with traditional microprocessors at the highest levels of performance due to the overhead of object manipulation. To match the performance of traditional microprocessors for traditional languages, the Java VM would have to be embellished with some conventional instructions.

Yet it may miss the point to compare Java microprocessors to traditional microprocessors in traditional applications. If Java deployment proceeds at a rate even close to current predictions, then Java microprocessors need to compete with traditional microprocessors only on Java's own turf: emulating the Java VM. Perhaps the only real question, then, is whether Java microprocessors offer compelling advantages over the best software emulators on traditional microprocessors. Since Java is so new, it is too early to tell, but if significant demand for standalone Java implementations develops rapidly, there will be tremendous economic incentive to develop high-performance software emulators. Perhaps traditional microprocessors will be embellished with a few Java-support instructions to greatly speed software emulation.

Thus, even on their home turf of Java execution, Java-based microprocessors may not succeed. These chips must overcome the lower costs associated with the economies of scale of current market leaders such as the x86. Techniques such as the JIT compiler may allow traditional microprocessors to approach the performance of a Java chip. These traditional CPUs would have the advantage of executing programming languages other than Java, while Java chips would be restricted to a single application.

As long as Java is successful, however, there will be several VM implementations, both hardware and software, on the market. As Java volumes grow, programmers will focus on improving the execution speed of the VM on traditional processors. In turn, CPU designers must consider how to improve their chips to efficiently execute Java code. Although Java originated as a virtual machine, it will have a future impact on real processor designs. M

Brian Case is a consultant for microprocessor design issues and can be reached at bcase@best.com.

