A new content-classification packet processor from PMC-Sierra illustrates the growing complexity of the network-processor market. ClassiPI is not a fully programmable processor, but it does perform moderately complex sequences of operations under the control of "rules" that amount to a stored program. PMC-Sierra doesn’t call ClassiPI a network processor, but other vendors use that term for chips that are less flexible than ClassiPI.

ClassiPI will generally be used in conjunction with fully programmable processors, both the conventional CPUs currently used for network processing—such as those offered by PMC-Sierra’s MIPS processor division, formerly Quantum Effect Devices (QED)—and the more highly optimized network processors available from other companies. PMC-Sierra itself is developing network processors and will use ClassiPI with these future products.

ClassiPI can also be used to augment ASIC-based designs not originally intended to support high-level packet processing. Because ClassiPI’s system interface is functionally equivalent to that of a standard synchronous SRAM (SSRAM), the task of integrating the chip into existing ASIC-based designs is greatly simplified. Any existing design that uses synchronous SRAM for packet buffering can be adapted to accept a ClassiPI chip, as Figure 1 shows. In such a system, ClassiPI will perform packet classification with only minimal supervision. The existing ASICs in the system will send unclassified packets to ClassiPI and will later retrieve the results of applying the routing rules that the control-plane processor has stored in the ClassiPI.

ClassiPI Is Much More Than a CAM

PMC-Sierra goes to some lengths to avoid another possible confusion in the classification of ClassiPI. Although a large...
Portion of the new chip is a content-addressable memory (CAM), PMC does not use that term in describing ClassiPI. Because the on-chip CAM is much more sophisticated than previous CAMs, PMC-Sierra calls it a “database.” This description is a fair representation of the function of the unit, which is shown in Figure 2.

Where a conventional CAM simply matches a search term to the corresponding line in the CAM array, ClassiPI’s database stores up to 16,384 “rules,” each of which consists of a 108-bit definition that can include complex logical comparisons. Portions of each packet are compared simultaneously with all the rules in the database. Comparisons can be made on the basis of equality, magnitude, and string matching, with optional case insensitivity. When multiple rules are combined into a single policy definition, packets can be identified on the basis of ranges, masked comparisons, and string regular expressions. Limited conditional branching functions are also provided. In effect, the rules stored in the chip are a program that produces an output based on the packet inputs. The major difference between these rules and the program in a conventional processor is that ClassiPI cannot be used to edit packets directly.

Multiple rules can also be combined to find matches based on more than 108 bits of each packet’s contents, which makes it possible for ClassiPI to look deeper into packets than a simple CAM could. Instead of performing routing based solely on the address in the packet header, ClassiPI can classify packets on the basis of data in the packet payload.

For example, a router in a server farm could look at the uniform resource locator (URL) in an incoming Web request to determine which server contains the requested Web page. Such an example represents an application-specific function at the highest level, layer 7, of the Open Systems Interconnect (OSI) networking model, but similar tasks exist in all but the lowest levels of the model. PMC-Sierra emphasizes the value of ClassiPI as an accelerator for OSI layers 4–7, since layers 1–3 are generally handled by even less expensive ASICs.

Each ClassiPI chip includes built-in cascading logic, which makes it possible to have up to eight chips working in parallel. In this way, up to 131,072 different rules can be applied to each packet. Each ClassiPI can return multiple matches; these may be prioritized so that only the most important result is returned, or all matching rules can be returned. Results are stored in a FIFO that is accessible through the SSRAM interface.

Statistics on the operation of the chip can be compiled using an optional off-chip SSRAM memory that has its own separate interface. Records of the number of bytes or packets processed, the number of times each rule was matched to a packet, and timestamp data can be accumulated in this off-chip memory and made available to the host processor.

ClassiPI does all this work very quickly. A single chip can apply all its 16,384 rules to every packet in a multi-Gigabit Ethernet flow. The maximum interface bandwidth of 6.4Gb/s is shared between arriving packets and returned results. PMC-Sierra says the chip can perform packet processing at OC-48 (2.4Gb/s) rates. Rules that involve multiple entries in the database and implementations that cascade multiple chips operate at full speed; in effect, these multiple operations are pipelined, which increases latency but does not diminish bandwidth.

The result of this sustained performance is that ClassiPI can contribute to order-of-magnitude improvements in the effective performance of networking hardware that
is performing high-level tasks. PMC-Sierra says that a conventional 800-mips processor alone can run firewall code on a 600Mb/s network segment or load balancing for a server farm at just 50Mb/s. Augmented by a single ClassiPI chip, however, throughput for these applications increases to 2Gb/s and 900Mb/s, improvements of more than 3× and 18×, respectively.

Speed increases as large as these are sure to get the attention of network-equipment designers. We believe that future generations of networking hardware will widely use ClassiPI and similar chips from other traditional vendors of content-addressable memories and related packet-processing accelerators.

© MICRODESIGN RESOURCES MARCH 5, 2001 MICROPROCESSOR REPORT

To subscribe to Microprocessor Report, phone 408.328.3900 or visit www.MDRonline.com
ARM’s Jazelle and Nazomi’s JSTAR. One thing these accelerators have in common is that they sit in the instruction decode path and translate Java byte codes, on the fly, into the native binary of the host CPU. In Part 2, we move out of the translate mode and into the execute mode as we put inSilicon’s new JVXtreme hardware accelerator through the grinder.

Not only does JVXtreme decode byte codes, it executes them as well. In that sense, JVXtreme is a discrete Java processor, except that it relies on a host CPU and is available only as an intellectual property core for integration onto an SoC.

In simplest terms, JVXtreme is a stack-based Java engine. It performs single-cycle execution of the basic Java codes (ALU operations, push constant operations, and variable loads and stores) and two- to three-cycle execution of the more complex byte codes (get/put field operations, array loads and stores, and branch operations). Like Jazelle and JSTAR, JVXtreme handles a subset of the Java byte codes, leaving more-complex codes for software execution on the host CPU. However, JVXtreme handles only 92 byte codes directly in hardware, far fewer than Jazelle and JSTAR (140 and 159, respectively). But the lack of certified industry-standard Java benchmarks or application profiling makes it difficult to determine whether this difference will be an issue (EEMBC is working to resolve the benchmarking situation). Like most decisions, inSilicon’s selection of which byte codes to implement was based on tradeoffs.

Byte Codes Percolate Through Pipeline

Obviously, the tradeoffs were based on a study of dynamic byte code frequencies (i.e., profiling statistics of typical applications) and the amount of silicon inSilicon could realistically throw at the problem (at approximately 35K gates, it is 3x the size of Jazelle). The fetch unit, in stage one of its pipeline, fetches byte codes from the memory subsystem and stores them in an internal instruction buffer. The depth of this buffer is optional, but the word width of buffer entries is defined by the width of the configurable data bus. Since a word in the buffer can contain more than one byte code, the fetch unit also performs the alignment before passing an individual byte code on to the decode unit. The fetch unit also performs static branch prediction, and a mispredicted branch incurs a four-cycle penalty associated with the flushing of the pipeline.

The instruction buffer compensates for the lack of an on-board cache, a choice made primarily to conserve transistors. The lack of a cache could affect performance, because every byte code access must go off chip. With respect to power, inSilicon uses extensive clock gating, an approach similar to most other chip designs. The interface logic is always active to detect commands from the host CPU, even when byte codes are not being executed.

The decode unit (DU), in stage two, decodes the byte codes from the instruction buffer. If the DU encounters an unsupported byte code, JVXtreme sends a branch vector to the CPU, which jumps to the function for the specific
software-emulated byte code. When JVXtreme is executing byte codes, the host CPU is idle, hence there are no latency issues associated with this jump (i.e., it’s not necessary to interrupt the CPU).

The DU also has an instruction-folding mechanism that looks for situations in which the accelerator could execute two byte codes in the same cycle. For example, an operation that performed $B=A+7$ would normally push the $A$ and 7 on top of the stack before doing the add. Folding performs the add first and pushes the result onto the stack.

Once byte codes are decoded, the resulting information is passed on to the execution and control unit (ECU). The ECU contains an ALU that performs the basic integer operations, without support for multiply or divide operations. JVXtreme has provisions to attach additional byte codes for customers on a custom basis, such as an optional floating-point multiplier or complete floating-point unit (FPU).

The ECU also supports Java multithreading: a single command, delivered via the coprocessor interface, loads the active portion of a task into on-chip resources. JVXtreme also maintains the frame pointers for the Java stack in on-chip hardware. This eliminates the involvement of the host’s operating system.

Unlike the Jazelle and JSTAR accelerators, JVXtreme contains a dedicated, 32-bit-wide, zero-wait-state Java hardware stack. The system designer would normally configure the depth of the stack during synthesis; however, inSilicon delivers JVXtreme with a 64-entry stack. This stack depth is a big contrast to that of Jazelle, which uses only four registers for its stack implementation.

(As a side note, there are several references on the study of the relationship between stack depth and the probability of overflow. In Chapter 6.4 of *Stack Computers—The New Wave* by Phillip J. Koopman, Jr. (1989, Carnegie Mellon University), the author summarizes the results of test program simulation by stating that stack overflow and underflow memory traffic tapers off at a steep exponential rate for all programs. At a stack buffer size of 24, even the Towers of Hanoi program generates a stack spill on less than 1% of instructions. As a practical matter, a stack size of 32 will eliminate stack buffer overflows for almost all programs).

JVXtreme supports stack operation with automatic overflow and underflow control. The control mechanism has a programmable watermark, set during core configuration, that potentially benefits overall system performance, because overflows and underflows steal CPU cycles.

**Please Pass the Beans**

Like any other Java accelerator, JVXtreme requires modifications to the JVM. Naturally, the biggest change is associated with the byte code interpreter, to allow the JVM to give JVXtreme control over byte code execution (Figure 1). When the host CPU encounters a Java program, the CPU uses the coprocessor interface to pass the corresponding memory address to JVXtreme. (The CPU also uses the coprocessor interface to send commands and other control information to JVXtreme’s register block). On the other hand, when JVXtreme encounters an unsupported byte code, it calculates a jump address, most probably by using an index into a callback table, and passes back to the host CPU.

JVXtreme’s coprocessor interface is generic, and inSilicon will provide a customized interface for each host CPU (it typically takes one month for this development). The company currently has implementations for ARM7 and ARM9. Alternatively, JVXtreme may be memory mapped into the host CPU’s address space (i.e., as a memory-mapped peripheral acting as a slave device). InSilicon has already developed a memory-map version for ARM’s AHB (Advanced High-speed Bus). The critical factor for each of these interfaces is the communication time between host CPU and accelerator. ARM’s coprocessor interface requires five clocks from the time the byte code is passed from the CPU to accelerator to the time JVXtreme begins executing the Java routine.

JVXtreme runs at 200MHz in a 0.18-micron process and consumes approximately 35K gates, depending on the host CPU and the depth of the stack. It is available from inSilicon as Verilog source code, and the company also provides a Verilog model and a test suite and hardware reference model for interfacing the coprocessor to the host CPU. For debugging, JVXtreme (as well as J VX) is accessible through the same in-circuit emulator used for the host CPU; all on-chip

---

**Figure 1.** The JVM must be modified to support the communication between the host CPU and Java accelerator. This handshake typically takes one clock in either direction.
resources are accessed through the coprocessor or memory interface.

You may already be familiar with inSilicon’s JVX Java accelerator, predecessor to JVXtreme, announced last June. The biggest difference is that JVX does not have a hardware-based interpreter; it supports only the execution of byte codes. In other words, the host CPU interprets the byte codes and converts them to JVX instructions; it therefore doesn’t allow the elimination of the interpreter-loop overhead within the JVM.

“Java to Go: Part 2” has focused on inSilicon’s coprocessor model for Java acceleration. The next article in this series will examine yet another distinctly different approach to Java acceleration. Specifically, we’ll look at Chicory Systems’ HotShot, which performs hardware-based, just-in-time compilation, with preprocessing for a variety of other types of media acceleration. The final article in this series will present an overall comparative analysis of the different approaches we have examined.