Cahners
MICROPROCESSOR REPORT
www.MPRonline.com
THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

JAZZ JOINS VLIW JUGGERNAUT

CMP and Java as an HDL Take System-on-Chip Design to a Parallel Universe

By Steven H. Leibson {3/27/00-03}

Now that microprocessor designers have bulked up their once-svelte RISC designs with all manner of transistor-laden constructs, such as superscalar architectures and speculative execution, the brave new world for clean microprocessor design seems to be moving to VLIW. In addition, several processor vendors are adopting chip multiprocessing (CMP) to further boost processing power. Into this modern microprocessor milieu steps 2½-year-old Improv Systems with its Jazz PSA (programmable system architecture) platform, a CMP design system based on a sort of multiprocessor metacore. Individual Jazz processors within the metacore are configurable VLIW processors with two-stage pipelines, four independent 32-bit data-memory buses, and a small, configurable amount of instruction memory. Jazz PSA generates synthesizable CMP designs and matching sets of compiled programs from your application description written in a specific Java style called a directed control dataflow network (DCDN).

A single configured Jazz processor is called a task engine. The Jazz PSA system architecture comprises a mix of heterogeneously configured task engines, on-chip data and instruction memory, I/O modules, and a global bus called the QBus for on-chip task and control communications among processors, as Figure 1 shows. The best superscalar RISC and VLIW processor architectures now peak at around 4 instructions per cycle. Improv claims that Jazz PSA achieves 8 to 12 instructions per cycle per task engine inside of some

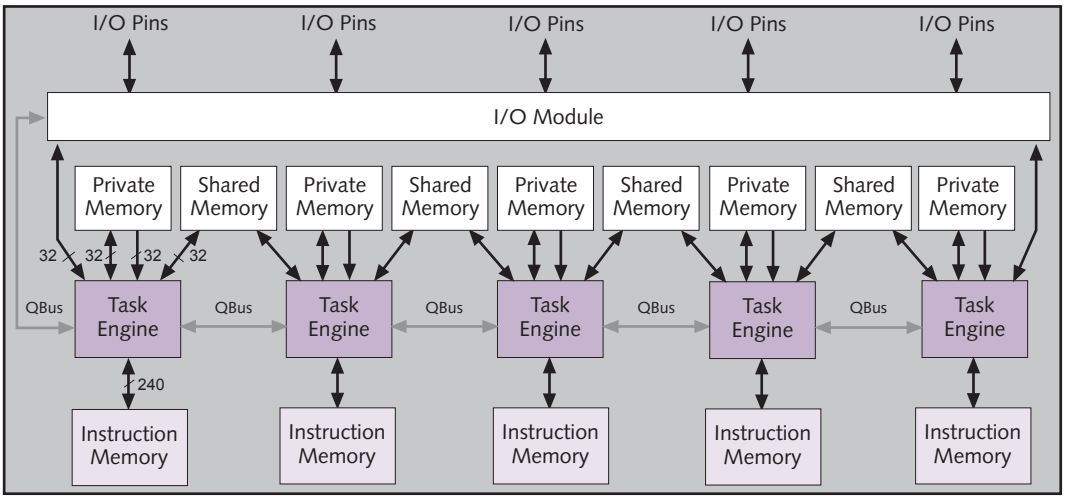

Figure 1. Jazz PSA lends itself to systolic processing applications with its mix of heterogeneously configured Jazz processors (task engines); on-chip data and instruction memory; I/O modules; and a global bus called the QBus for on-chip task and control communications among processors.

inner loops, a figure of merit based on task engines containing 13 execution units each. Sample applications Improv has implemented with Jazz PSA and standard task engine configurations include low-bit-rate MPEG-4 videoconferencing and a 15-channel telecommunications application with G.726 speech codecs and G.168 echo cancellation. Tailoring the task engines boosted the telecommunications application capacity to 30 channels.

Like superscalar processors, VLIW processors employ many function or computing units to execute multiple instructions per cycle. Unlike superscalar machines however, VLIW processors rely on smart compilers running on amply endowed hosts to fill the hungry maws of the execution units with continuous instruction streams. VLIW designers have abandoned the game of using on-chip resources to wring smaller and smaller gains in instruction-level parallelism (ILP) from RISC instruction streams and instead rely on compilers, with their superior abilities to perform global optimizations, to boost ILP (see *MPR 2/14/94-05*, "VLIW: The Wave of the Future?").

CMP Cores: The Same, But Different

At first blush, the Jazz PSA hardware description sounds somewhat similar to Cradle's UMS (universal microsystem) architecture introduced last October at Microprocessor Forum (see *MPR 10/6/99-05*, "Cradle Chip Does Anything") and to the recently announced DVine architecture from Silicon Magic (see *MPR 3/27/00-02*, "Silicon Magic: DVine-ly Inspired?"). All three designs spin complex processor arrays and memory blocks into CMP configurations. However, there are significant differences among these three architectures. The most obvious difference is topological. Cradle's UMS and Silicon Magic's DVine array multiple processors and memory controllers along one or two global buses. Shared buses and shared memory are both potential bottlenecks in these designs. Jazz PSA employs a flow-through CMP architecture that uses dual-ported shared memories for interprocessor communications. Consequently, there is no chance that tasks running on separate task engines will compete for shared memory or bus resources in a Jazz PSA design.

Another significant difference is Jazz PSA's considerably greater data-moving capacity. A five-engine implementation of Jazz PSA running at 100MHz has a peak aggregate data-moving capacity of 8GB/s, and the architecture isolates data movement from the VLIW instruction streams so all of that bandwidth is used to move data. Furthermore, Jazz PSA's peak data-movement capacity scales linearly with the number of task engines in a specific implementation, because each engine provides its own set of data pipes. Cradle's approach couples processor modules called quads through one 64-bit, 640MHz bus (5.12GB/s peak). DVine links multiple compute modules (CMs) to multiple memory interface units (MIUs) over two 128-bit, 166MHz buses (5.312GB/s peak, in aggregate). Both UMS and DVine must devote at least some of their available bus bandwidth to instruction traffic, because neither architecture segregates instruction and data streams, as does Jazz PSA, and peak data bandwidth is fixed for both the UMS and DVine architectures and doesn't scale with the number of processors used in a design.

There are other notable differences among Cradle's UMS, Silicon Magic's DVine, and Improv's Jazz PSA besides topology and raw data-transfer rate. Improv's Jazz PSA employs a heterogeneous array of variously configured VLIW task engines, each with its own 240-bit VLIW instruction memory. UMS quads contain homogeneous processor arrays, each comprising four 32-bit RISC processor elements and eight digital-signal engines. DVine's CMs are homogeneous engines consisting of a 32-bit RISC engine and a 16-wide vector processor.

In addition, system designers using Jazz PSA can tailor the computing resources within each task engine to match the computational needs of specific task(s) by adding execution units to a task engine when more performance is needed. A system designer can also remove unneeded execution units from a task engine to save silicon and power dissipation. Because all the processor modules in the UMS and DVine architectures employ the same design, each UMS quad or DVine CM may or may not be well suited to any particular portion of a given application, and

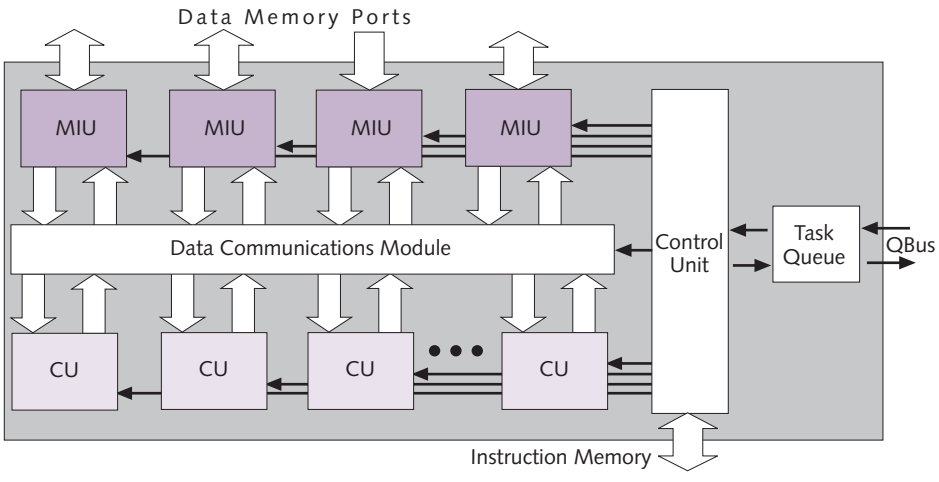

Figure 2. Each Jazz PSA task engine contains as many as 16 computation units (CUs), taken from a collection of blocks that include 32-bit ALUs, 32 x 32-bit MACs (with 64-bit accumulators), 64–bit shifters, 16-bit counters, and byte swappers. Task engines also contain four memory interface units (MIUs) and a control unit. Data flows between CUs and MIUs over a giant bus multiplexer called the data communications module (DCM).

any unused computational capability in these architectures remains on the chip, which costs money and burns power.

As Figure 2 shows, a Jazz PSA task engine contains as many as 16 computation units (CUs) taken from a collection of functional blocks that include 32-bit ALUs, 32 x 32-bit MACs (with 64-bit accumulators), 64-bit shifters, 16-bit counters, and byte swappers. Currently, all CU functional blocks are integer units, but Improv plans to add floating-point units within the next year. The specific mix of functional blocks used in a task engine depends on the specific tasks within an application that will run on that task engine. Depending on the number of CUs placed in a configured Jazz processor, each processor in a Jazz PSA implementation can execute between 0.5 and 1.6 GOPS (at 100MHz) and requires approximately 47K gates (a number that varies, depending on how many CUs the processor incorporates).

All CU operations execute in one processor cycle, and computation results are stored in local registers within individual CUs at the end of each cycle. The CUs derive their input data from a large array of intraprocessor bus multiplexers, collectively called the data communications module (DCM), as Figure 3 shows. Each CU's registered output feeds a separate 32-bit distribution bus within the DCM so that other CUs within the same task engine can use the results. (DCMs do not interconnect task engines.) The registered outputs of the task engine's four MIUs and the constant registers within the control unit also feed DCM distribution buses.

Each CU and MIU input port connects to the DCM distribution buses through a 16-channel multiplexer. In some task engines, there may be more than 16 distribution buses in the DCM, so Improv has developed some standard routing layouts, but a system designer with unique requirements can individually specify the sources attached to each input multiplexer.

Never the Same Datapath Twice

The control unit within the task engine supplies opcodes, constants (when needed), and control signals to each CU within that processor. The control unit also configures the DCM according to the requirements of the currently executing task. The multiplexer configuration within the DCM can change on a cycle-by-cycle basis. This scheme creates an extremely flexible datapath that reconfigures on the fly.

Jazz PSA connects individual task engines, using multiple shared-data memory blocks in an architecture well suited to systolic processing (an array-processing approach often used in digital-signal and image processing). Task engines employ shared memory blocks to pass processed data between processor pairs. These memory blocks incorporate a synchronous dual-port interface that eliminates arbitration between task engines. The shared memories support simultaneous reads, while simultaneous writes to shared memory are flagged as system errors during the initial functional development. Should a simultaneous write occur during program execution, the contents of the shared memory location will simply be undefined.

Each task engine contains four independent MIUs, which connect the processor to shared and private memory blocks and to the Jazz PSA's I/O module. Three of the MIUs in each task engine have bidirectional memory interfaces. The fourth MIU has only an input port; it cannot write to memory. Each MIU has a 32-bit data path and can address 64K of data memory. This small memory size may be less limiting than it seems, because these memories are used strictly for data, not for instructions. For example, in a video-coding application, task engines typically process individual 8 x 8 pixel blocks, with a resulting data-storage requirement of 256 bytes to 1K (depending on pixel depth). Add more memory to hold coding tables and the total memory requirement is still much less than the 64K limit.

MIU ports read from and write to memory in one processor clock cycle to maintain processor throughput. Consequently, the memories actually operate at twice the processor clock rate. Within each processor clock cycle, reads are initiated at midcycle, and the read results are loaded into a processor register at the end of the processor clock cycle. Writes occur at the end of a processor clock cycle. Although they are really 32-bit machines, MIUs also perform halfword and bytewide read and write operations. Partial-word write cycles employ indivisible read-modify-write operations and require two processor clock cycles. The read portion of the read-modify-write operation occurs during the cycle containing the write instruction; the associated write portion follows during the next processor clock cycle.

Each task engine has its own local instruction memory. Task engines have 12-bit instruction addresses, so a task engine can access a maximum of 4,096 VLIW instruction words. The relatively small instruction-address space is further diminished by its division into regular instruction memory and boot ROM. Regular instruction memory starts at address 0, while boot ROM starts at location 0xE00.

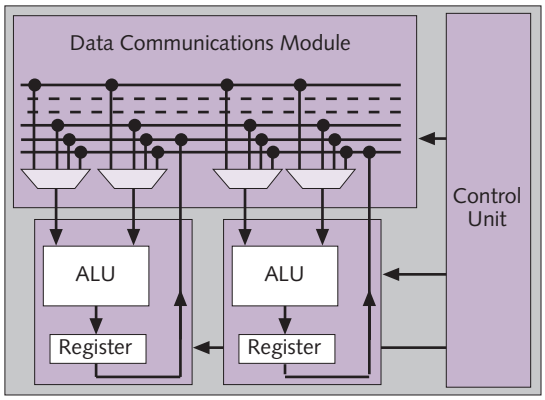

Figure 3. The CUs within each Jazz processor derive their input data from a huge array of 16-channel bus multiplexers, collectively called the data communications module (DCM), that perform the function of a full crossbar switch.

Smarter Than the Average Boot

Following a reset, each Jazz processor starts execution at the first location in boot ROM. The boot code for one or two of the task engines typically programs the on-chip I/O module to define an interface to an external ROM containing the entire program image for all task engines. Then the processor nearest the I/O module on the chip reads image blocks from the external ROM and downloads the blocks into the appropriate processor's on-chip instruction RAM. After all image blocks are loaded, each processor begins its initialization routine, starting at address 0.

The relatively small per-processor instruction memory space and reset-based program-loading scheme is a clear indicator of the way Improv believes its Jazz PSA will be used. The processor array is set up once after a reset and then operates with a fixed program for as long as the system is switched on. This approach is clearly oriented to streams-based processing, using relatively small blocks of code to direct the processing. Stream-oriented applications such as continuous filtering, coding/decoding, compression/decompression, and encryption/decryption are typical of processes that lend themselves to such an approach. By making instruction memory separate and local to each task engine and by using shared memories for passing data between processors, Improv's Jazz PSA permits massive amounts of simultaneous data movement on the chip.

The intended range of applications is further highlighted by the unusual tasking scheme developed for the Jazz PSA. The overall system behavior is reduced to a set of encapsulated tasks by Improv's development tools. Each task involves some combination of processing data and queuing other tasks for execution. The development tools assign tasks to individual processors in the array during compilation, and the assignment is static; task assignment doesn't change dynamically based on processor availability.

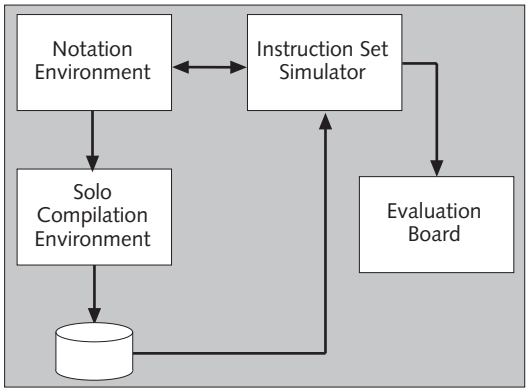

Figure 4. The major components of the Jazz PSA development tool set are the Notation Environment for application development; the Solo Compilation Environment for partitioning, resource allocation, task optimization, code generation, and performance analysis; a cycle-accurate instruction-set simulator; and a PCI-based evaluation board.

Tasks communicate with each other by queuing other tasks. Each task engine contains a task queue, filled by the QBus, and executes its list of tasks in FIFO order. Tasks always run to completion (terminated by executing an end-of-task instruction). Tasks cannot be suspended, and there are no interrupts. To achieve the equivalent of task suspension, work must be split into two tasks—one that terminates at the intended suspension point and one that is queued when the work needs to resume. In other words, within the Jazz PSA universe, tasks are scheduled only when they are needed, and they always run to completion. Similarly, there are no wait loops in Improv's universe, although there is support for periodic tasks; the Jazz I/O module contains timers that can be programmed to queue tasks on specific processors, either periodically or after a time delay.

Automated Help for Too Many Choices

By now, it should be apparent that the number of degrees of freedom and the inherent programming complexity of the Jazz PSA platform is stunningly huge. Configuring a complete version of the Jazz PSA requires decomposing of the overall application into smaller tasks; determining the optimum number of task engines; tailoring the execution resources and DCM routing within each task engine; arranging the interprocessor and local-data memory topology; and setting the size of each data and instruction memory block. There's also the complex job of partitioning the system software and distributing the resulting tasks over a collection of dissimilar VLIW processors.

Together, the need to configure and program such a complex machine might doom a new CMP architecture to obscurity, because the learning curve would simply be too steep for any but the most adventurous to climb. However, software developers don't write code for individual task engines in Improv's multiprocessing universe. Instead, Improv supplies a system-level development tool set that converts a high-level system definition into a set of task engine programs for a selected Jazz PSA configuration.

Figure 4 illustrates the major components of the Jazz PSA development tool set: the Notation Environment for application definition and development; the Solo Compilation Environment for applicationwide partitioning, resource allocation, task optimization, code generation, and performance analysis of the resulting hardware/software system; a cycle-accurate instruction-set simulator for detailed timing verification and fine-grained optimization; and a PCI-based evaluation board that incorporates a test chip for "near real-time" debugging. This entire tool set runs on Windows NT workstations, but the Notation Environment can run on any Java-capable computer.

Decaf Java—All the Syntax, No JVM

Improv selected Java, a somewhat unlikely system-definition language, for its Notation Environment, the embedded application development system that's part of the Jazz PSA

platform. So far, Java has garnered much notoriety but not much traction as an embedded development language, because current Java implementations execute slowly compared with programs written in compiled languages. Furthermore, Java's intrinsic garbage collector makes most existing JVM (Java virtual machine) implementations nondeterministic and therefore useless for hard-real-time systems. Although credible efforts are under way to make Java an effective real-time programming language, none of Java's liabilities as an embedded development language come into play with Improv's Jazz PSA. Improv circumvented Java's real-time problems by using only the language's syntax and discarding its execution models. Instead, Improv's tools compile the Java application description into a number of programs distributed to the various task engines within a Jazz PSA implementation.

Java may appear to be a trendy choice for a development environment, but the company selected Java for several valid technical reasons. First, the encapsulation of Java's object-oriented methods is a good match for the parallelism and task encapsulation inherent in applications like signal and image processing. Second, system developers can create and test virtual prototypes of CMP systems in Java, using relatively inexpensive PC-based Java development tools. Conversely, HDL design tools based on VHDL and Verilog, the two languages more traditionally used to define chip-level systems, are far more costly.

The third reason Improv selected Java is the primary target audience for its product: system architects and code developers currently defining and programming systems in C (not chip designers currently using HDLs to develop ASICs). Selecting Java as a system-definition language thus allows software application developers to define large portions of an ASIC, create test benches, and debug designs without using conventional HDLs. The company believes it's far easier to move existing C programmers to a system-description language based on Java than it is to have them learn VHDL or Verilog. Improv selected Java over C or C++ because the company believes that Java has better support for concurrency.

Yet another reason Improv selected Java as an HDL is that Java contains no pointer arithmetic and doesn't make any assumptions about specific target hardware. With an underlying silicon structure as fluid as the Jazz PSA, allowing system designers to hard-code assumptions about the underlying hardware guarantees problems. One final reason to pick Java is that future graduates of computer-oriented curricula are highly likely to know Java syntax, thanks to its popularity as a Web programming language.

Changing the Rules of Design

System developers describing systems based on the Jazz PSA don't use Java directly as a programming language. Instead, system designers use Java's syntax to describe system-level behavior by creating what Improv calls a directed control dataflow network or DCDN. The key elements of a DCDN description are not algorithms and data structures, as with conventional software programming. Instead, the ingredients of a DCDN are tasks, data managers, data arcs, control arcs, and components, as shown in Figure 5. Tasks are executable sequences of behavioral statements (i.e., algorithms); data managers are implementation-independent mechanisms for sharing data among tasks; data arcs describe data dependencies between tasks and data managers; and control arcs describe control dependencies between tasks.

Components encapsulate substructure with tasks—they essentially allow a system designer to group related tasks together as one entity. Each component can include one initialization task and one behavioral task. A component may also instantiate subcomponents (groups of subtasks). Each component has an explicit interface that includes data ports, control ports, and parameters that configure the component's behavior. Components can create data managers to manage communications between the component's initialization and behavioral tasks and among subcomponents.

This approach to system-level design therefore closely resembles the rigor of encapsulation used by object-oriented programming. Consequently, there is an assumption built into the DCDN design approach: that there will always be enough processing power available to execute the tasks as designed. This assumption must be validated later in the design process, using cycle-accurate simulation.

Improv has developed a Java class library for implementing DCDN models. The class library includes VirtualIP objects (the main building blocks of an application); data manager objects (which serve as one end point for data arcs); PSA data port objects (which serve as the other end-point for data arcs); PSA control port objects (which serve as the end-point for control arcs); Virtual ASSP objects (which

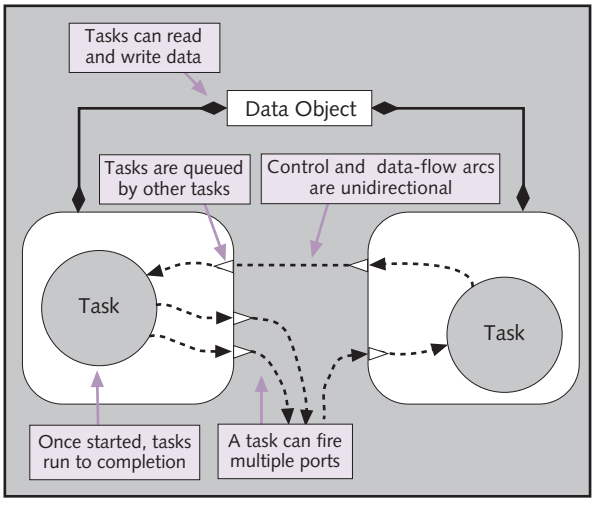

Figure 5. A directed control dataflow network (DCDN) decomposes an application into tasks, data objects, data-access arcs, and control arcs.

VLIW, Java, and CMP: Heard This Before?

If the words "Java," "VLIW," and "chip multiprocessing" remind you of Sun's MAJC architecture (see *MPR, 10/25/99-04* "Sun Makes MAJC With Mirrors"), your wetware associative memory is functioning well. The MAJC-5200 device that Sun discussed at Microprocessor Forum 1999 incorporates two processors, each with four execution pipes ganged as one VLIW machine. Three of the pipelines in each processor execute most of the MAJC instruction set (except for certain instructions such as branches, divides, and reciprocal square roots). The fourth pipe in each of the two processors executes 25% of the MAJC instruction set (including the instructions that the other pipelines cannot execute). If you count the pipelines separately, MAJC could be considered an array of eight heterogeneous processors, but it really consists of two identical VLIW processors.

There really is not much similarity between the MAJC-5200 and Improv's Jazz PSA beyond the buzzwords. First and foremost, MAJC is a conventionally programmed processor. Sun designed it to be a fast engine for running multithreaded Java or C code. Furthermore, MAJC's processors communicate either through one shared on-chip data cache or through mailboxes and other conventional mechanisms in external memory. Finally, MAJC's VLIW processors are not configurable (except by Sun).

So the words that Improv and Sun are singing may sound the same, but the songs are different.

map an application onto a target Jazz PSA configuration); and PSA test bench objects. These components, coupled with Java's existing class libraries, constitute the set of elements available to develop systems based on the Jazz PSA.

Automated System Construction: Going Solo

After creating and debugging a design with the Notation Environment, Improv's Solo Compilation Environment reduces the design to a set of software blocks that run on the user-defined Jazz PSA configuration. Solo performs an initial analysis of the Java-based DCDN application description, breaks the application into individual tasks; schedules tasks and assigns them to specific task engines; compiles the tasks into VLIW instruction streams for the various task engines in the target implementation; and then analyzes the performance of the resulting hardware/software system. The results of this second analysis provide information for tuning the configuration of each task engine. Improv has developed six task engines and expects that these standard engines will suffice for roughly 70% of all applications. For the remaining 30% of applications that need to extract more performance from every clock cycle, system designers can define their own task engines.

Solo is really the key to realizing the potential performance inherent in the multiple levels of parallelism in this architecture. Without this tool, the entire Jazz PSA would be too complex to be used by mere mortals with real design deadlines. So the success of Improv's approach seems to hinge on the effectiveness, reliability, and usability of the Solo Environment. Many of today's VLIW compilers are based on the pioneering work done a decade ago by the now-defunct Multiflow. Improv says that its Solo Compilation Environment is not based on Multiflow's compiler and claims that its schedule-driven task analysis, allocation algorithms, and resource-directed scheduling techniques are vastly superior to Multiflow's original trace scheduling, compilation, and optimization algorithms.

Beyond these basic tools, Improv also plans to offer a growing number of prewritten modules for the most common media-processing tasks such as multichannel voice-over-IP (VoIP), videoconferencing, and embedded Internet protocols. The company calls these tested application modules VirtualIP.

Finally Getting to Silicon

Once designed and debugged, Improv's tools produce a Verilog design ready to be used as is or incorporated into a larger ASIC. The complete implementation kit also includes RTL models, synthesis and simulation scripts, and verification suites to support configured implementations of the Jazz PSA. Although Improv expects the Jazz PSA platform to be used as an ASIC core generator, the company realizes that nothing beats real hardware for proof of concept. Accordingly, Improv has fabricated one-, two-, five-, and six-processor versions of its Jazz PSA architecture in silicon, using fabrication processes ranging from 0.25 to 0.18 micron, with clock rates ranging from 25 to 100MHz. These chips are unlikely to be used in production products, because they are not optimized for a specific application. However, the chips do provide systems developers with silicon for high-speed application debugging and prototype systems.

Improv offers three licensing models for the Jazz PSA platform: multiuse, per-use, and single-use licenses. The multiuse license allows unlimited use of cores generated by the tools for a single fee, or through an annual subscription plus a per-chip royalty. The per-use license sets a negotiated fee for the use of each core generated by the tools. The user also pays a per-chip royalty with the per-use license. Improv believes that the price crossover between the multiuse license and the per-use license is four new designs per year. The single-use license is for companies that want to use Improv's architecture just once. This license requires payment of a one-time fee plus per-chip royalties. These licensing arrangements are not unlike those of other IP vendors, such as Tensilica.

Revolutionaries like Improv Systems can change the world, but there is always a price for revolution. In this case,

it's the substantial change in the way system design is done. System developers must learn how to describe complex multiprocessing systems in Java-based DCDN models, not yet a common design approach. Such a radically new design concept must overcome the inertia that retards the adoption of any new processor architecture, let alone an entirely new system-design method. There's also hesitancy by mainstream system designers to adopt the complex and somewhat obscure regimen of any sort of multiprocessing because of the resulting exponential increase in system complexity.

Consequently, Improv Systems has a monumental education job to tackle if it wants to succeed broadly with the Jazz PSA. Designers must have enough faith in Improv's tool chain to turn over critical jobs like system-level task partitioning and resource allocation to new and relatively unproven automatic tools. Usually, this is a high-risk decision for system OEMs, so you can be sure those companies will make such a decision only after careful consideration of the costs and benefits. Improv is therefore currently targeting ASIC and semiconductor vendors for initial licenses and has recently signed Philips as a licensee.

Price & Availability

As with most IP products, licenses for Improv's Jazz PSA are negotiated individually. For more information about Improv and the Jazz PSA, go to *www.improvsys.com*.

The allure of Improv's approach for companies already involved in ASIC development is compelling where the application requirements demand multiprocessing. Development of multiprocessing systems is extremely difficult, and "push-button" creation of such systems is nearly inconceivable with conventional system-design methods. If the company can establish that Java is a viable system-description language (likely); that its Notation and Solo Environment tools are capable of performing the tasks for which they're sold; and that switching to a DCDN system-design method delivers big productivity gains, Improv will have a real winner on its hands.