A Component-Based Design Environment for ESL Design

Patrick Schaumont
Virginia Tech

Ingrid Verbauwhede
Katholieke Universiteit Leuven

Editor's note:
This article focuses on two key properties that the authors see as critical to ESL design: abstraction and reuse. The authors present an ESL design flow using the Gezel language. Using several very different design examples, they show how this design flow supports their case for abstraction and reuse.
—Carl Pixley, Synopsys

Recently, there has been an increasingly greater variety of target architecture options for digital electronics design. Whereas the driving applications for these architectures are often governed by standards and thus tend to be regularized, there is still a lot of design freedom in the target architectures themselves. There is a wide range of programmable-processor architectures,[1,2] and with any given application, designers must balance performance, power consumption, time to market, and silicon cost.[3] The obvious question is how to choose the most appropriate target architecture for a given application.

In this article, we present Gezel, a component-based, electronic system-level (ESL) design environment for heterogeneous designs. Gezel consists of a simple but extendable hardware description language (HDL) and an extensible simulation-and-refinement kernel. Our approach is to create a system by designing, integrating, and programming a set of programmable components. These components can be processor models or hardware simulation kernels. Using Gezel, designers can clearly distinguish between component design, platform integration, and platform programming, thus separating the roles of component builder, platform builder, and platform user.

Embedded applications have driven the development of this ESL design environment. To demonstrate the broad scope of our component-based approach, we discuss three applications that use our environment; all are from the field of embedded security.

ESL design has many faces

A common definition for ESL design is the collection of design techniques for selecting and refining an architecture. But ESL design has many aspects and forms. Even within a single application domain, system-level design can show wide variations that are difficult to capture with universal design languages and architectures. Therefore, you can also think of ESL design as the ability to successfully assemble a system out of its constituent parts, regardless of their heterogeneity or nature. Consider the following three examples. All of them closely relate to design for secure embedded systems, but they also require very different design configurations. Thus, these examples show the need for a more general approach, which we achieve using Gezel.

Example 1: Public-key cryptography on 8-bit microcontrollers

Sensor networks and radio-frequency identification tags are examples of the next generation of distributed wireless and portable applications requiring embedded privacy and authentication. Public-key systems are preferable because they allow a more scalable, flexible key distribution compared to secret-key cryptosystems. Unfortunately, public-key systems are computationally intensive and hence consume more power. Recent proposals suggest replacing the RSA (Rivest-Shamir-Adleman) system with more economical solutions such as elliptic-curve cryptosystems (ECCs) or hyper-elliptic-curve cryptosystems (HECCs). ECCs and HECCs provide security levels equivalent to RSA but with shorter word lengths (a 1,024-bit RSA key is equivalent to a 160-bit ECC key and an 83-bit HECC key), at the expense of highly complex arithmetic. Figure 1 shows the hierarchy and mapping of such a system. On top is the HECC

0740-7475/06/$20.00 © 2006 IEEE Copublished by the IEEE CS and the IEEE CASS

point multiplication operation, which consists of a sequence of basic elliptic-curve point operations. Each of these basic elliptic-curve operations consists of a sequence of more elementary operations in the underlying Galois field. For HECC, this field is 83 bits. If the system were an ECC, this field would be 160 bits.

We implemented this design as an 8051 microcontroller, extended with a hardware acceleration unit. The 8-bit microcontroller interfaces are quite narrow compared to HECC word lengths. Therefore, when building a hardware acceleration unit, it is crucial to consider overall system performance. Because of the hierarchy in the calculations, there are multiple ways to accelerate the HECC operations—in contrast to secret-key algorithms, which have fewer hierarchy layers and thus offer fewer implementation choices. As a stand-alone optimized C implementation, an HECC point multiplication takes 192 seconds to calculate. A small hardware accelerator, requiring only 480 extra FPGA lookup unit tables (LUTs) and 100 bytes of RAM, improves this time by a factor of 80, to only 2.5 seconds. Figure 1 indicates the resulting split between hardware and software, which is not yet optimal for an 8051.

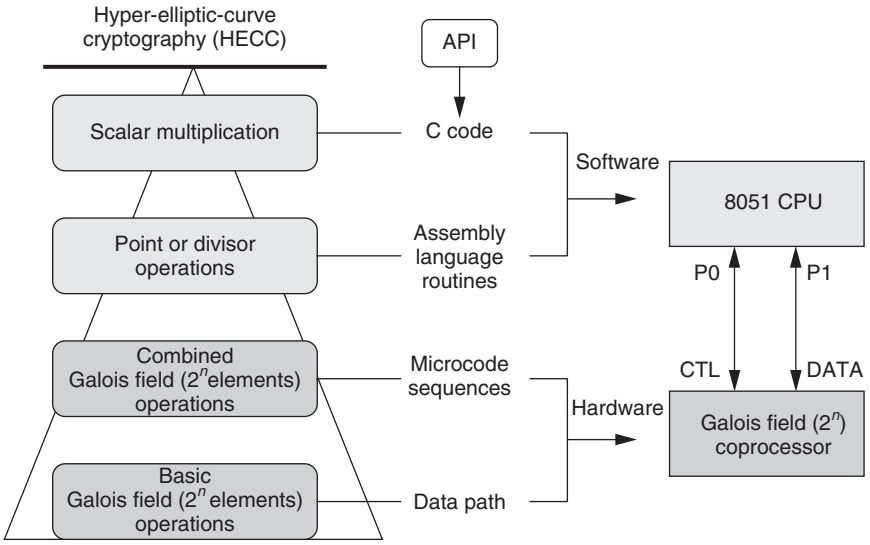

Figure 1. Public-key cryptography on an 8-bit microcontroller.

Hardware acceleration makes HECC public key possible on small, embedded microcontrollers. But the optimal implementation depends on the selection of the mathematical algorithms and the system-level architecture. Only a platform-based design approach makes this design space exploration possible and discloses opportunities for global improvement.

Example 2: Concurrent codesign for secure partitioning

The design of secure embedded systems leads to design cases requiring tight interaction between hardware and software—even down to the single-statement level. Figure 2 shows a fingerprint authentication design, the ThumbPod-2 system, which is resistant to side-channel attacks; we implemented and tested this design in silicon.[4] The protocol, shown in Figure 2a, accepts an input fingerprint and compares it to a prestored, secret template. The matching algorithm must treat this template as a secret, and the ThumbPod-2 system stores it in a secure circuit style that is resistant to side-channel attacks. However, because the matching algorithm manipulates the template, part of the algorithm's circuit must also migrate to a secure circuit style. Because this secure circuit style consumes twice the area of normal circuits, mapping the complete matching protocol to it would be inefficient. We therefore separated the protocol into an insecure software partition and a secure hardware partition, and we ended up with the implementation in Figure 2b. The software reads the input fingerprint and feeds the data to the oracle inside the secure partition. The oracle compares each input minutia with the template minutia, returning only a global-matching result: reject or accept. It is impossible for an attacker with full access to the untrusted software to determine how the oracle has obtained this decision.

The design and verification of the secure protocol requires continuous covalidation between hardware and software. We evaluated various attack scenarios that attempt to extract the secret template from the secure hardware partition, assuming that the attacker can arbitrarily choose the software program at the insecure partition. This led to an iterative refinement of the oracle interface and the driving software, which we designed completely within the Gezel environment.

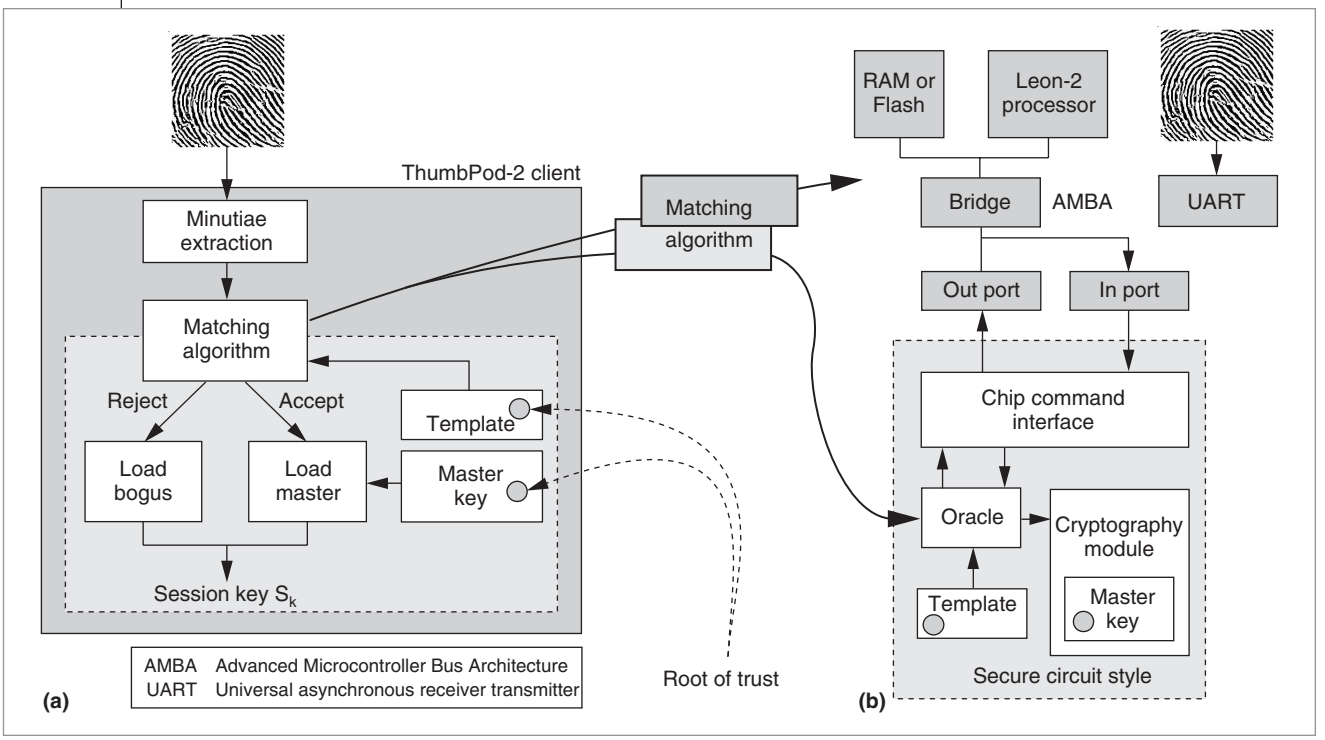

Figure 2. Partitioning for security in the ThumbPod-2 system: protocol for session key generation (a), and implementation (b).

Example 3: Accelerated embedded virtual machines

For a third application, shown in Figure 3, we had to provide hardware acceleration of a cryptographic library for an embedded virtual machine.[5] We used a Java embedded virtual machine, the Kilobyte Virtual Machine (KVM), extended with native methods that allow hardware access directly from a Java application. We integrated an advanced encryption standard (AES) coprocessor into the Java virtual machine's host processor, and we triggered execution of the coprocessor using a native method. The virtual machine handles all data management and synchronization. As Figure 3b shows, hardware acceleration can improve performance by two orders of magnitude. Moreover, data movement from Java, to and from the coprocessor, has two orders of magnitude of overhead compared to actual hardware execution. A combined optimization of the Java-native API, the coprocessor, and the coprocessor interface is necessary to avoid design errors and, more importantly, security holes in the final system.

All three examples are practical design problems from the field of embedded security. There is no unified design platform or unified design language that could solve all of them. However, it's still possible to generalize their underlying design principles by using a component-based approach.

Component-based ESL design

Each programmable architecture comes with a specific set of design techniques. ESL design, therefore, is no tightly knit set of techniques, tools, and data models. Unlike RTL design, which logic synthesis enabled, ESL design doesn't offer a standard design flow. In fact, ESL design might never be unified in a single design flow, given the architectural scope, the complexities in capturing all facets of an application, and the daunting task of developing tools for these different facets. Still, all ESL technologies share two fundamental objectives: facilitating design reuse and supporting design abstraction. These two objectives have guided every major technology step that has turned transistors into gates, and gates into systems.

Reuse and abstraction for ESL design, however, are unique and different from other technology transitions. In ESL design, reuse relates not only to architectures but also to design environments. For example, when a designer builds a SoC architecture around a microprocessor, the microprocessor's compiler and the instruction-set simulator (ISS) are as critical to the design's success as the actual microprocessor implementation.

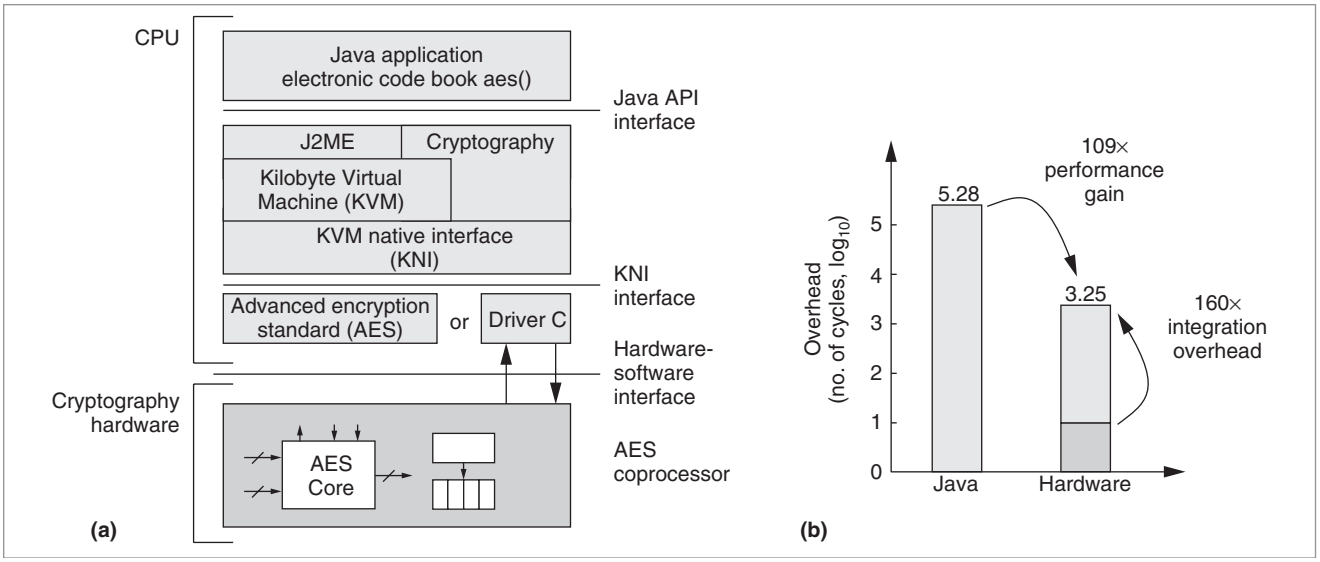

Figure 3. Accelerated embedded virtual machine: general structure (a) and performance improvements and associated overhead (b).

The compiler and the simulator are reused design environments, and the microprocessor is a reused design artifact. As another example, consider SystemC. You can view SystemC as a reusable design environment for RTL hardware components. As a C++ library, it can link to any environment that needs RTL hardware design capability; thus, the SystemC library itself is a reusable component.

Abstraction in ESL design concerns not only the masking of implementation details but also platform programming mechanisms. Finding successful system-level abstractions is extremely difficult, because abstractions tend to restrict the scope of coverable target architectures. For example, C is a successful programming abstraction for a single-core system, but it becomes cumbersome in multicore systems. Despite the multitude of system-level design languages, none has so far been able to unify the architecture design space in a single programming abstraction.

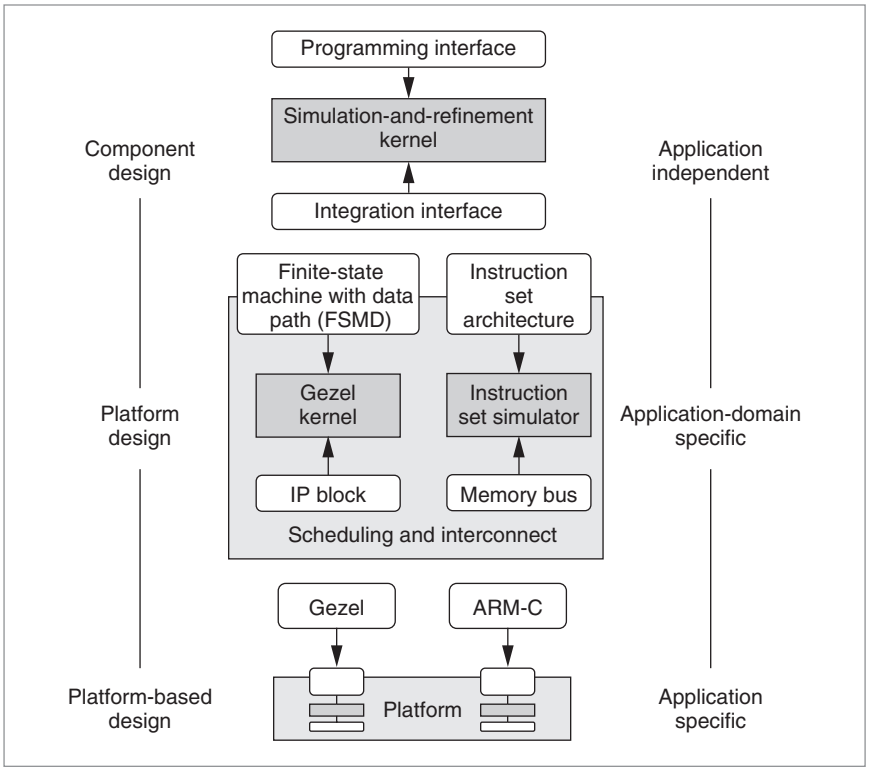

Figure 4. Three phases for ESL design automation: component, platform, and platform based.

These two elements of ESL design—its reuse of design environments and design artifacts, and the component-specific nature of its programming abstractions—guided us toward a component-based approach in system design. In ESL design, we define a component as a single programmable element included in a platform. For example, a microprocessor, reconfigurable hardware, a software virtual machine, and the SystemC simulation kernel are all programmable components.

As Figure 4 shows, a component-based model for ESL

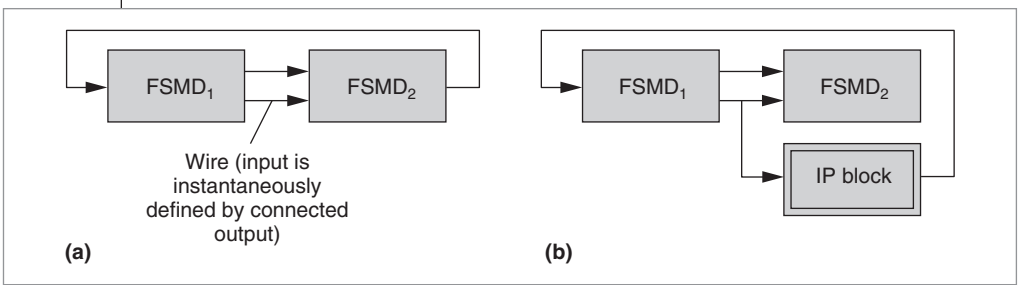

Figure 5. Finite-state machine with data path (FSMD) network: pure (a) and extended (b).

design requires a design flow with three phases of design: component, platform, and platform based. These phases correspond to the creation, integration, and use of programmable components. Several different engineers might work in each design phase, each with his own perspective on an application. These engineers generally fall into one of three categories: design automation, hardware design, or software design. Figure 4 offers the perspective of the design automation engineer.

In component design, a design automation engineer develops a design environment for a single programmable component. The engineers can do this independent of the application. Two interfaces—integration and programming—characterize a programmable component. Through the integration interface, a component connects to another (possibly heterogeneous) component. Between these two is a simulation-and-refinement kernel. Component design can be very elaborate, including, for instance, the development of an ISS and a C compiler for a new processor.

In platform design, a design engineer or design automation engineer selects various programmable components and combines them into a single platform by interconnecting their integration interfaces. Platform design requires the creation of a platform system scheduler to coordinate the individual components' activities. This phase also requires the creation of communication channels between components. The notion of a platform as an articulation point between application and architecture is a well-known concept.[6,7]

In platform-based design, a design engineer develops an application by writing application programs for each programmable component in the platform. The platform simulator lets the designer instantiate a particular application and tweak overall system performance. For heterogeneous components, it's important to bring the individual components' programming semantics sufficiently close together so that a designer can easily migrate between them.

Designers have used component-based design approaches, typically in software development, to address problems requiring high architectural flexibility. For example, Cesario et al. presents a component-based approach for multiprocessor SoC (MPSOC) design,[8] based on four types of components: software tasks, processor cores, IP cores, and interconnects.

Designing and integrating FSMD components with Gezel

The Gezel design environment (http://rijndael.ece.vt.edu/gezel2) supports the modeling and design of hardware components. By integrating the Gezel kernel with other simulators (such as ISSs), we obtain a platform simulator. The three examples we discussed all rely on custom hardware design, each with a different platform. We've combined Gezel with other programmable components, such as 32- and 8-bit cores. We've also combined it with other types of programming environments, including the SystemC simulation kernel and Java. For the parts of the design described in the Gezel language, the Gezel design environment automatically creates VHDL, enabling technology mapping into FPGA or standard cells.

Platform-based design using Gezel

The Gezel language captures hardware using a cycle-based description paradigm based on the finite-state machine with data path (FSMD) model. Widely used for RTL hardware design, this model has been popularized through SpecCharts and SpecC.[9] The FSMD model expresses a single hardware module as a combination of a data path and its controller. You can combine several different FSMDs into a network, as Figure 5a shows. A pure FSMD network is only of limited value for a platform simulator, because such a network supports only communication between FSMDs. Such a network doesn't have the means to communicate with any part of a platform that is not captured as an FSMD.

To employ FSMDs as platform components, Gezel supports extended FSMD networks, as Figure 5b shows. Such an extended FSMD network also includes a second type of module called an IP block. An IP block has an interface similar to that of an FSMD, but the IP block

is implemented outside the Gezel language. A similar concept of capturing heterogeneity also exists in Ptolemy.[10] Technically, an IP block is implemented as a shared library in C++ and thus can include arbitrary programming constructs within the boundaries of a cycle-based interface. To the Gezel programmer, the IP block looks like a simulation primitive. The platform designer defines the IP block's behavior. In a component-based design model, these IP blocks implement communication channels, which connect Gezel to a wide range of other components, such as ISSs, virtual machines, and system simulation engines.

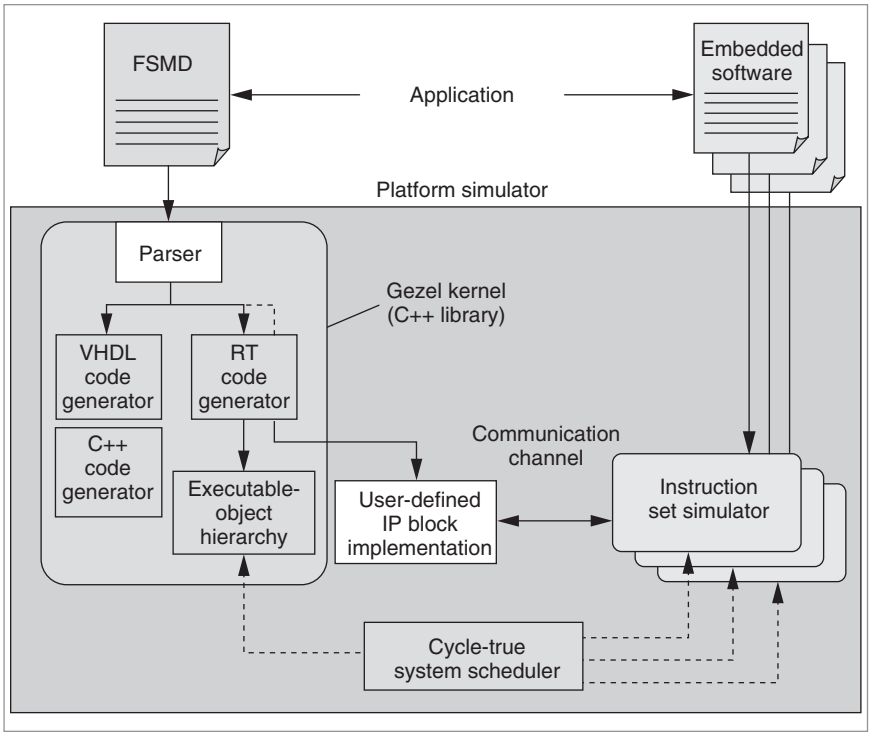

Figure 6. Gezel platform simulator.

Platform design using Gezel

Figure 6 illustrates a platform simulator that uses the Gezel kernel and several ISSs. Each component simulator exists as an individual (C++) library, linked together in a system simulation. For this platform simulator, we use IP blocks to implement the cosimulation interfaces between the Gezel model and the ISS. In addition, a system scheduler calls all the included component simulators. We implement the platform simulator in C++.

The extended FSMD network in Gezel, combined with the component-based design model, offers essential advantages over a traditional HDL- or SystemC-based approach. VHDL has no means to natively support a simulation setup like the one in Figure 6, because it lacks the equivalent of an IP block construct. Consequently, an HDL-based design flow usually implements such a simulation setup at the HDL level. This needlessly increases simulation detail and penalizes simulation performance.

It's also possible to implement such a simulation setup in SystemC. But the platform and the application are no longer distinguishable, because SystemC captures everything in C++. This complicates the synthesis of the application onto the final platform. In other words, SystemC does not distinguish between the platform and platform-based design phases.

Table 1 lists several platform components that we've used with Gezel to create platform simulators. They include 8- and 32-bit ISSs, Java (through its native interface), and SystemC. We coupled each of these simulators to the Gezel FSMD model using IP blocks. There are two categories of IP blocks, corresponding to two different design scenarios: IP blocks that model a processor's bus or a dedicated communication port to implement a coprocessor design scenario like the one in Figure 7a. Other IP blocks capture a complete component.

Designers can also use the Gezel IP block construct to explore multiprocessor architectures, such as the PicoBlaze microcontrollers shown in Figure 7b. In the multiprocessor design scenario, the Gezel model captures the complete platform, clearly improving flexibility. In addition, this model allows dynamically selecting the number and types of cores. The Gezel language captures synchronous, single-clock hardware designs. The platform simulators in Table 1, however, can accommodate multiple clock frequencies to the individual processors included within the simulation.

Many of the environments in Table 1 are open source, which greatly eases the construction of platform simulators. In commercial environments, open source might still be an unattainable goal, but there are still significant benefits from using an open interface. Several of our cosimulators (including TSIM and SH-ISS) use commercial, closed-source components, built on the basis of an open interface.

Table 1. Platform simulators using Gezel.

Component	Simulation engine*	Cross-compiler or assembler	IP block interface: Core	IP block interface: Port or bus
8-bit cores				
Atmel AVR	Avrora	GNU avr-gcc		•
PicoBlaze	kpicosim	KCPSM3 assembler	•	•
8051	Dalton ISS	SDCC, Keil CC	•	•
32-bit cores				
ARM	Simit-ARM	GNU arm-linux-gcc	•	•
Leon2-Sparc	TSIM	GNU sparc-rtems-gcc		•
SH3-mobile	SH-ISS	GNU sh-elf-gcc		•
Simulation engines				
Java	JVM 1.4	javac		•
SystemC	SystemC 2.0.1	GNU g++		•

* Information on simulation engines is available as follows:

Avrora: http://compilers.cs.ucla.edu/avrora (open source);

kpicosim: http://www.xs4all.nl/~marksix (open source);

Dalton ISS (Dalton 8051): http://www.cs.ucr.edu/~dalton/i8051 (open source);

Simit-ARM: http://sourceforge.net/projects/simit-arm (open source);

TSIM (TSIM 1.2; cross compiler, sparc-rtems-gcc 2.95.2): http://www.gaisler.com;

SH-ISS (Renesas SH3DSP simulator and debugger, v3.0; cross compiler: sh-elf-gcc 3.3): http://www.kpitgnutools.com

Systematic reuse with a component-based approach

We can also implement IP management with Gezel. IP transfer is notoriously difficult because reuse interfaces are hard to define. Microprocessor buses have traditionally been the reuse interface of choice. New industry efforts such as the Open Core Protocol IP (OCP-IP, http://www.ocpip.org) and the Spirit consortium (http://www.spiritconsortium.com) have focused on generically packaging IP components rather than using standard buses. Spirit's approach is to provide a metadata model that encapsulates existing IP components (expressed in VHDL or SystemC, for example). The metadata provides additional language-neutral information on the IP interface. However, a component-based design flow with Gezel does not need this encapsulation, because the language directly models the reuse interfaces. Indeed, these reuse interfaces correspond to the set of IP blocks that connect the Gezel models to other platform components.

Consider the case in which multiple parties participate in the platform-based design phase. For example, for the simulator of Figure 6, assume that an IP developer creates hardware components in Gezel, and a system integrator creates the system (embedded) software. In such a case, the IP developer expects a reasonable level of IP protection before releasing the actual implementation, whereas the system integrator wants access to the hardware components as detailed and as soon as possible. Gezel can support this scenario, as Figure 8 shows. We define two phases in the IP transfer. In IP creation and evaluation, the IP developer provides a cycle-based simulation model of the hardware IP as a black box to the system integrator; this model provides a nonsynthesizable simulation view of the IP. When the system

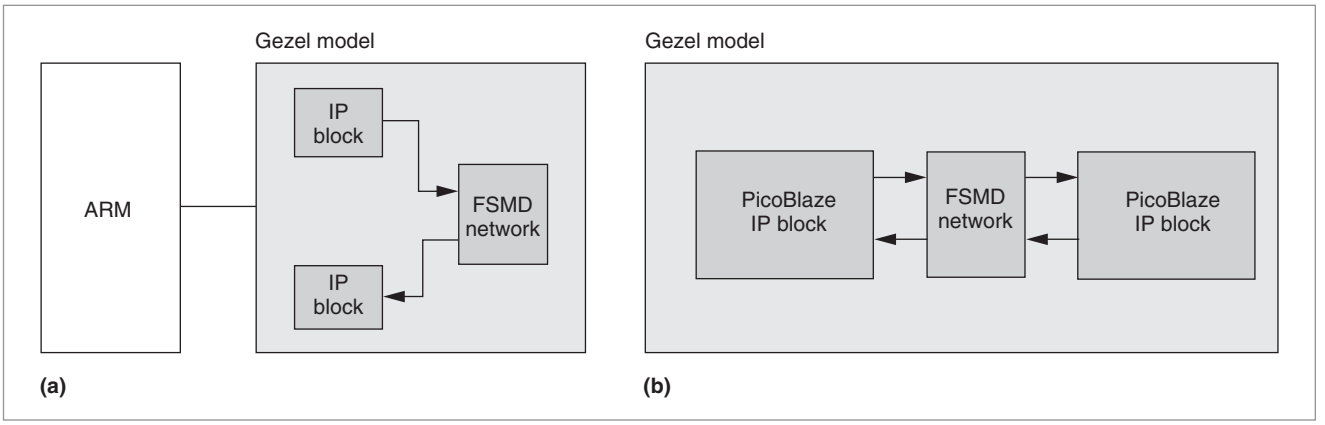

Figure 7. Application of different IP block categories: coprocessor (a) and multiprocessor (b) design scenarios.

integrator decides to acquire the hardware IP, the second phase of the IP transfer begins. Now the IP developer provides a synthesizable version of the hardware IP in VHDL.

The component-based approach of Gezel is well-suited for this IP design flow. We model black boxes as IP blocks. The IP block simulation views are in binary format as shared libraries, and thus of little value for this implementation. We wrote two code generators for FSMD networks in Gezel. The first converts FSMDs into equivalent IP block simulation views. The second converts FSMD into synthesizable VHDL code. The IP developer can use them together to implement the design flow of Figure 8.

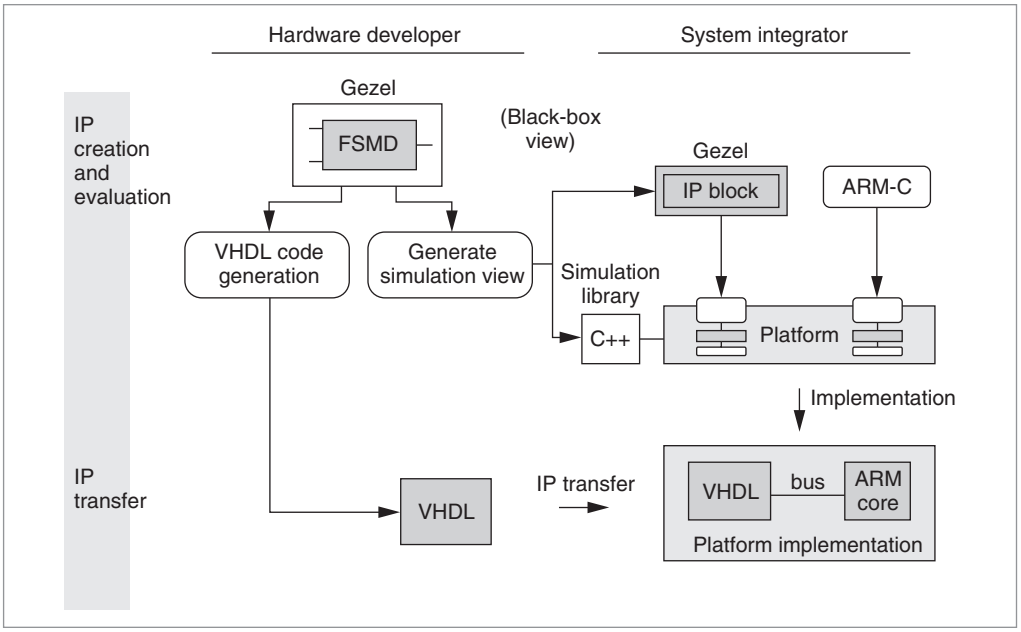

Figure 8. IP reuse in the platform-based design phase.

Table 2 shows several examples of IP modules designed in Gezel. They range from simple components, such as an Internet packet check-sum evaluation module (CHKSUM) to complex IP modules, such as an AES module and a high-speed Gaussian-noise generator for bit-error-rate measurements (BOXMUL). For each module, Table 2 lists the line counts of the original Gezel design and the amount of generated code in C++ and VHDL. We also mapped the VHDL code onto an FPGA, and Table 2 gives the area and speed of the results. We expect the numbers shown to be close to those of manually written VHDL. For example, a comparable AES design by Usselman on Xilinx Spartan3 technology lists a LUT count of 3,497.

Table 2. IP model complexity. (NCLOC: noncommented source line of code)

	Model line count (NCLOC)			Area (no.	
Design	**Gezel**	**C++ (IP blocks)**	**VHDL**	**of LUTs)***	**Speed (ns)****
CHKSUM	149	1,564	907	131	9.19
EUCLID	69	710	62	557	560.00
JPEG	526	8,091	719	5,514	14.62
AES	292	2,653	1,807	3,332	8.29
BOXMUL	763	6,105	6,282	4,225	20.30

* Target platform was Xilinx Virtex4, speed grade 12

** *Speed* is the clock period we recorded after place and route.

Design examples revisited

Now, we briefly discuss how we used our component-based approach to support the three design examples presented earlier.

Public-key cryptography

The platform simulator for the HECC application consisted of two components: the Gezel kernel and the 8051 ISS (http://www.cs.ucr.edu/~dalton/i8051/). Using IP block models, we designed communication links between the 8051 ISS and the coprocessor. We developed the driver software running on the 8051 using the Keil tool suite. The platform simulator maps the HECC mathematical formulas into a combination of C, assembly language, and hardware. After obtaining a suitable partitioning, we converted the hardware coprocessor into VHDL. We then combined this coprocessor with a synthesizable view of the 8051 processor and mapped it into an FPGA.

Security partitioning for an embedded fingerprint authentication design

This platform contains the Leon2 ISS and the Gezel kernel. We constructed it in a process similar to that of

constructing the public-key cryptography platform. We developed software using the GNU tool suite. In a later design phase, we used the VHDL code generator to convert the Gezel design into VHDL, eventually leading to a tested and fully functional chip.[4] This design, however, requires fitting the hardware coprocessor onto a nonstandard synthesis design flow based on logic for resisting side-channel attacks. So that chip designers could verify their custom synthesis flows, we extended the platform simulator to record trace stimuli for individual hardware modules. We can also provide this capability using the IP block approach. It is important to separate design flow issues, such as the stimuli recording facility, from actual design issues. The design flow in Figure 4 also supports this concept by distinguishing between the platform builder and the platform user. Gezel lets users write new IP blocks in C++ according to a standard template, and more advanced Gezel users can develop them as library plug-ins.

Acceleration of embedded virtual machines

For the third design, we integrated three components: a port of the Java-embedded virtual machine, the SH3-DSP ISS, and the Gezel kernel. We developed software in Java, C, and assembly language. In addition, this design required a considerable number of cryptographic support libraries. This kind of design demonstrates the importance of varying the design abstraction level within a single platform. The entire cryptographic application in Java can take millions of clock cycles, and the hardware coprocessor is active for a fraction of the time. On the one hand, we need increased simulation efficiency (and decreased simulation detail) for much of the design, but on the other hand, at a few select places we must observe every bit that toggles in every gate. A component-based design approach can cope with this heterogeneity.

HETEROGENEOUS SYSTEM architectures will continue to dominate in applications that require dedicated, high-performance, and energy-efficient processing. The challenge at the electronic system level will be to design these architectures in increasingly shorter design cycles. New tools will have to quickly create not only derivative platforms but also entirely new platforms. We are exploring novel mechanisms in Gezel to further accelerate platform construction, and we are presently working on such a platform designer for FPGA technology.

We'd also like to stress that ESL design requires not only new tools but also a change in design culture. Designers of heterogeneous architectures will inevitably get in touch with new design cultures and practices, not only from those novel ESL tools but also from their colleague designers. ■

Acknowledgments

We thank the reviewers for their constructive feedback. We also thank the many students who have experimented with Gezel and whose designs we've mentioned in this article. This research has been made possible with the support of STMicroelectronics, Atmel, the National Science Foundation, University of California Microelectronics and Computer Research Opportunities (UC Micro), SRC, and FWO (Fonds voor Wetenschappelijk Onderzoek).

■ References

1. C. Rowen and S. Leibson, *Engineering the Complex SoC: Flexible Design with Configurable Processors*, Prentice Hall, 2004.
2. T.J. Todman et al., "Reconfigurable Computing: Architectures and Design Methods," *Proc. IEE*, vol. 152, no. 2, Mar. 2005, pp. 193-207.
3. D. Talla et al., "Anatomy of a Portable Digital Mediaprocessor," *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 32-39.
4. K. Tiri et al., "A Side-Channel Leakage Free Coprocessor IC in 0.18um CMOS for Embedded AES-Based Cryptographic and Biometric Processing," *Proc. 42nd Design Automation Conf.* (DAC 05), ACM Press, 2005, pp. 222-227.
5. Y. Matsuoka et al., "Java Cryptography on KVM and Its Performance and Security Optimization Using HW/SW Co-design Techniques," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems* (CASES 04), ACM Press, 2004, pp. 303-311.
6. T. Claassen, "System on a Chip: Changing IC Design Today and in the Future," *IEEE Micro*, vol. 21, no. 3, May-June 2003, pp. 20-26.
7. A. Sangiovanni-Vincentelli, "Defining Platform-Based Design," *EE Times*, Feb. 2002, http://www.eetimes.com/news/design/showArticle.jhtml?articleID=16504380.
8. W.O. Cesario et al., "Multiprocessor SoC Platforms: A Component-Based Design Approach," *IEEE Design & Test*, vol. 19, no. 6, Nov.-Dec. 2002, pp. 52-63.
9. D. Gajski et al., *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.
10. E. Lee, "Overview of the Ptolemy Project," tech. memo UCB/ERL M03/25, Dept. of Electrical Eng. and Computer Science, Univ. of California, Berkeley, 2003.

Patrick Schaumont is an assistant professor in the Electrical and Computer Engineering Department at Virginia Tech. His research interests include design methods and architectures for embedded systems, with an emphasis on demonstrating new methodologies in practical applications. Schaumont has an MS in computer science from Ghent University, Belgium, and a PhD in electrical engineering from the University of California, Los Angeles. He is a senior member of the IEEE.

Ingrid Verbauwhede is an associate professor at the University of California, Los Angeles, and an associate professor at Katholieke Universiteit Leuven, in Belgium. Her research interests include circuits, processor architectures, and design methodologies for real-time, embedded systems in application domains such as security, cryptography, DSP, and wireless. Verbauwhede has an electrical engineering degree and a PhD in applied sciences, both from Katholieke Universiteit Leuven. She is a senior member of the IEEE.

■ Direct questions or comments about this article to Patrick Schaumont, 302 Whittemore Hall (0111), Virginia Tech, VA 24061; schaum@vt.edu.