Using a Soft Core in a SoC Design: Experiences with picoJava

Sujit Dey

Debashis Panigrahi

Li Chen
University of California, San Diego

Clark N. Taylor

Krishna Sekar
University of California, San Diego

Pablo Sanchez
University of Cantabria

Through our experience in synthesis, validation, test, and intregration of the picoJava processor core in a system-on-chip (SoC) design we point out the challenges faced and issues to address in efficient reuse of a soft core.

■ Major technologic and economic factors are driving a revolution in computer and communication system design. Networking applications like network processors, routers on chips and home gateways, portable products like wireless telephones and internet appliances, multimedia products like set-top boxes, and embedded controllers in automobiles, printers, and hard disk drives have generated a strong demand for hardware-software systems. Simultaneously, deep submicron technologies enable placement of complex systems on single chips. Giga-scale systems with millions of gates necessitate efficient design reuse to shorten the design productivity gap using predesigned intellectual property (IP) cores. This is leading to the emergence of IP core-based hardware-software system-on-chips (SoC) as a new paradigm in electronic system design.

Three types of IP cores exist, providing different optimization levels and flexibility of reuse.[1] A hard core, consisting of hard layouts, is the most optimized, but offers little flexibility in terms of changing the hardware features of the core itself. Most general-purpose processor and digital signal processing (DSP) processor cores available and used today, like the cores from ARM, LSI Logic, Motorola, and IBM, are hard cores. On the other hand, a soft core is a functional description of an IP, and the soft IP specification can be both simulated and synthesized. A soft IP allows flexibility in retargeting the IP specification to better fit the core user's needs. For example, a soft processor core allows the core user to reconfigure the features of the processor, such as its instruction set, caches, communication mechanisms, and interrupt mechanisms to make the processor core more suitable for a particular SoC application. However, as opposed to a hard core user, a soft core user (the SoC integrator) must synthesize, optimize, validate, and develop tests for the soft core before integrating it in the SoC being designed.

Firm cores try to balance the optimization and fast reuse potential of hard cores and the flexibility and reconfigurability of soft cores. Firm cores are typically delivered after certain
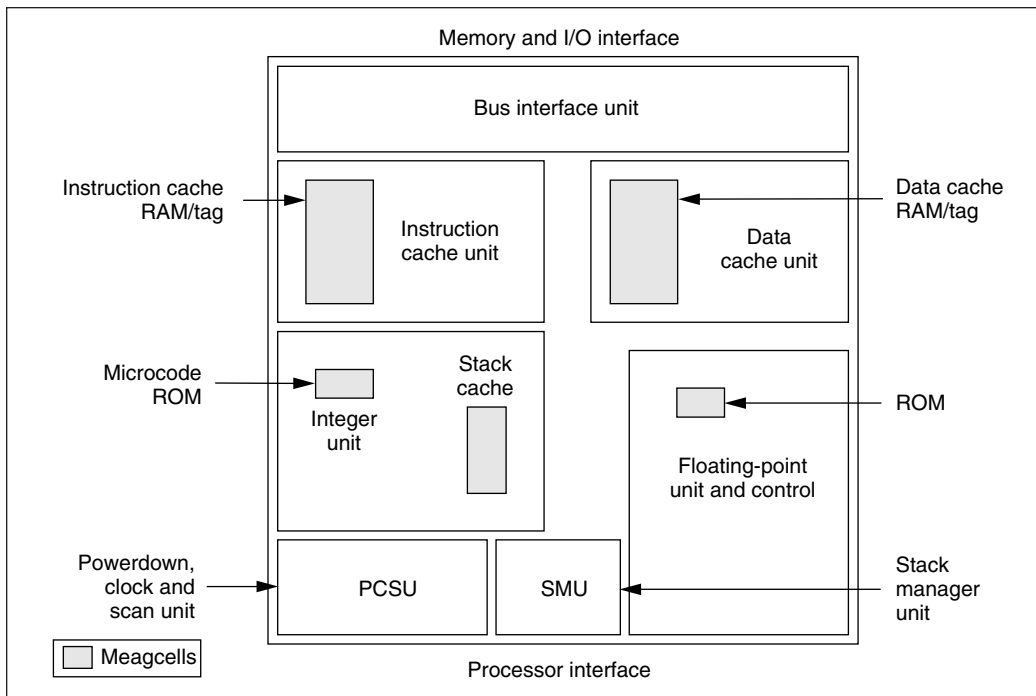
Figure 1. Block diagram of the picoJava processor core.

stages of the design process. For example, the firm core provider delivers a gate-level implementation mapped to a particular technology library, after going through the process of synthesis, but without performing physical design.

Here we present our experience with using the picoJava processor core, a commercial soft core from Sun Microsystems released under a community licensing program.[4] Starting from the register transfer level (RTL) and the instruction set architecture description of the picoJava processor core, we went through the steps that a user needs to take in a soft-core-based SoC design. Through our experiences in synthesis, validation, test, and integration of the picoJava core, we point out some of the challenges faced and issues to address in the efficient hand-off and use of a soft core.

The picoJava Core

PicoJava, a microprocessor core developed by Sun Microsystems, directly executes Java bytecode instructions.[2] The picoJava processor is a stack-based 32-bit microprocessor with 300 instructions and 6 pipeline stages. The core provides the flexibility of configuring the instruction and data cache sizes as well as the option

of including or excluding the floating-point unit (FPU). The core consists of seven major functional units, shown in Figure 1:

- *Integer unit* (IU): decodes the Java Virtual Machine instructions, which consist of an opcode (1 or 2 bytes) followed by zero or more operands. The major blocks of this unit are a 32-bit arithmetic logic unit (ALU) and shifter, multiply/divide unit, micro-code ROM, programmer-visible registers, and 64-entry stack cache (32-bit-wide array with read and write ports).
- *FPU*: executes the supported floating-point operations. The core user can include or exclude it from the picoJava processor core by appropriate configuration.
- *Instruction cache unit* and *data cache unit*: responsible for caching instructions and data, respectively. The instruction cache is a direct-mapped cache with line size of 16 bytes, whereas the data cache is a two-way set-associative cache with 16-byte cache lines. Both caches are configurable among 0-Kbyte, 1-Kbyte, 2-Kbyte, 4-Kbyte, 8-Kbyte, and 16-Kbyte types.
- *Stack manager unit*: stores and provides the

necessary data operands to the IU. It also handles the overflow and underflow conditions of the stack cache.

- *Powerdown*, *clock,* and *scan unit*: integrates clock generation, low-power management and design-for-test structure control. The processor core also supports various powerdown modes.
- *Bus interface unit*: implements the picoJava interface to the external components like external memory and I/O devices.

Sun's picoJava distribution package contains several support manuals that describe the picoJava core architecture, the instructions supported in the core, and the verification strategy.[3] The verification support includes an instruction-accurate simulator (22,454 lines of C code), Verilog simulation environment (6,968 lines of Verilog code), runtime library (34,642 lines of Java assembly code), and a set of functional tests (11,348 lines of Java code). The functional tests are small Java programs that cover most of the synthesizable model. For example, the block coverage of these tests is 93%, the path coverage is 91%, and the expression coverage is 85%. The distribution package also includes a mechanism for verifying the synthesizable description of the core with the instruction-accurate simulator (IAS). This mechanism concurrently executes the IAS and the synthesizable description of the core and compares results at the end of simulating every instruction.

The picoJava processor core is distributed as synthesizable RTL description (46,376 lines of Verilog code) that includes 7 technology-dependent megacells. The distribution also provides Synopsys synthesis scripts (1,755 lines) and associated design constraints (1,619 lines).

Mapping a soft core to silicon

As explained earlier, a soft core provides its user the flexibility of configuring the components to better fit user's needs and retargeting to the chosen library. In return, the core user has to perform the complex tasks of synthesis, optimization, layout, testing, and validation of the soft core. Before signing off the soft core (RTL specification), a soft core provider per-

forms the above tasks using a design-tools flow targeted to a particular technology library (usually a generic library) to verify the design and provide performance/power characterizations. In this process, the provider can support some of the above tasks performed using the particular tools. For example, the picoJava core distribution provides command script and constraint files for synthesis using Synopsys Design Compiler.[4] However, it's not possible for the core provider to support the above tasks for all possible design flows and target technology libraries. As a result, core users with different tools and target libraries face problems in mapping the soft core to the physical level.

Additionally, a soft core may include some macroblocks or megacells, typically designed at the physical level. While the soft core provider supplies behavioral or functional models for the megacells used in the core (for simulating the core at high levels), the core user has to implement the technology-dependent megacells—a potentially challenging task.

A soft core user faces challenges due to changes in both the design tools flow and the target technology library. If a design tool is used for which the soft core distribution provides no support, the core user must generate the necessary command files as well as set up the environment for that tool in the soft core context. Changes in the target technology library present more severe challenges, as they affect the entire process of mapping the core to silicon. A new technology library may necessitate change in the script files and the design itself, and also a large number of iterations in design flow to meet the required performance.

Issues in synthesizing picoJava

Though synthesis is primarily performed by the core user, the synthesis process depends on the core provider and the technology provider (silicon foundry). The inter-relation among these parties, as shown in Figure 2 (next page), poses some implementation problems.

During the use of a soft core, the core user must synthesize the core description provided by the core provider down to gate level using the technology library provided by the tech-
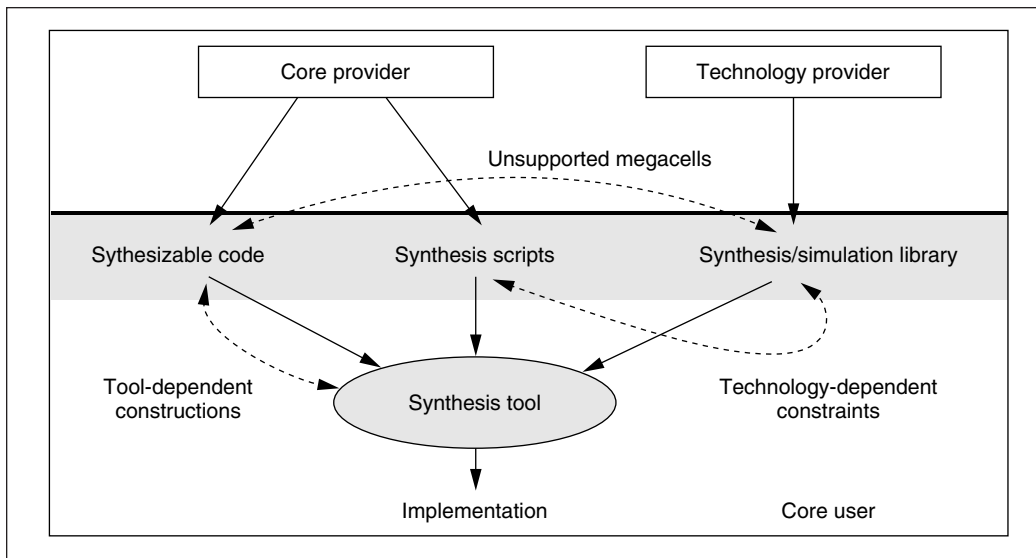
Figure 2. Soft core handoff and implementation: Dependency among core provider, technology provider, and core user.

nology provider. As core users, we faced the following challenges:

1. *Handling tool-dependent constructs in the core description*. The synthesis tool we used (Exemplar-Mentor Graphics' Leonardo) could not use the synthesis scripts provided in the distribution for a particular synthesis tool (Design Compiler). Thus we had to adapt the synthesis scripts and some tool-dependent hardware description language (HDL) constructs to synthesize the picoJava.

2. *Handling unsupported megacells*. The core description from the core provider contains references to several memory blocks (megacells). The core user should generate the megacells and change the constraints accordingly.

3. *Handling technology-dependent constraints*. The synthesis scripts contain timing-related constraints, which are functions of technology library parameters and design. It's very difficult for the core user to specify the constraints without any information about the design. Incorrect specification of these constraints can lead to incorrect synthesis and may require many iterations of the synthesis step.

From our experience, we can conclude that the handoff of a soft-core from a core provider to the core user is a complex process, involving overlaps of what the core provider supplies and the core user needs to do. For example, core users need to analyze the provided HDL code and adapt the synthesizable code and the synthesis scripts to their own synthesis tools and technology libraries, as explained before. The inner rectangle in Figure 2 shows the overlap between the core provider space and the core user space; the associated problems are shown by dotted lines. Hence, synthesizing a soft core is not an automatic synthesis process of a black-box module. In reality, the core user needs to modify the synthesizable code as well as synthesis scripts depending on the technology library (Figure 3). Additionally, the core user may need to go through multiple iterations of modifying RTL code, the timing constraints, and the scripts before finally obtaining a correct implementation.

Supporting megacells

Supporting megacells is one of the main challenges in mapping a soft core to a new technology. Because a megacell is technology dependent, the end user must be able to generate the necessary megacells in-house. This requires the user to have the tools and the
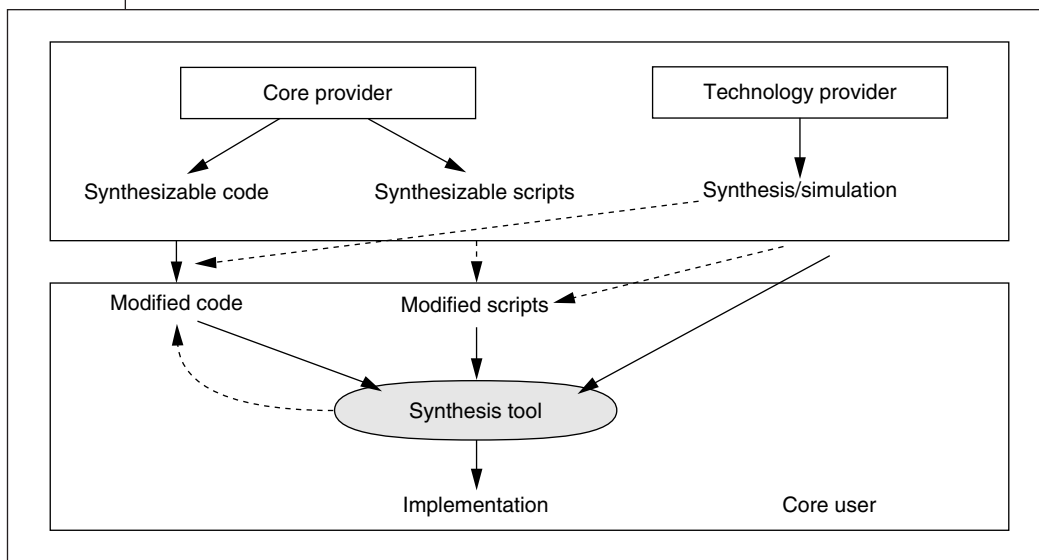
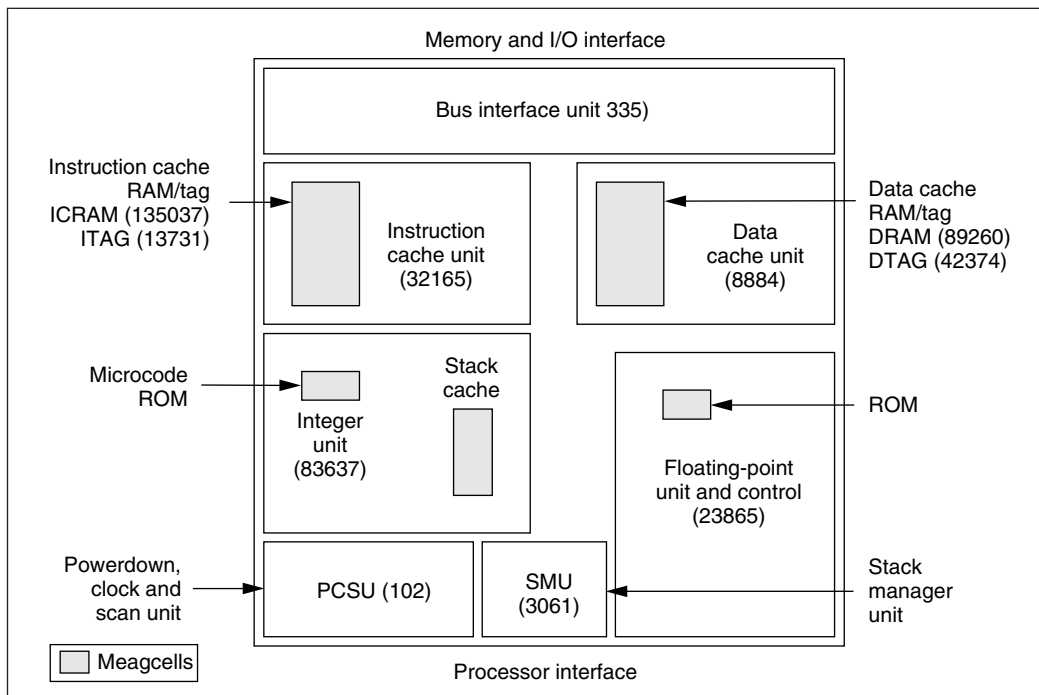**Figure 3. The actual implementation process for a soft core**

Figure 4. Block diagram of the synthesized picoJava-II processor core showing area of each unit (in terms of 2-input NAND equivalent gates).

us implement the stack cache with our megacell replacement (in this case, a register bank).

Once the user has completed the technology-specific design of a megacell, functional models of the megacell must also be generated for use at higher levels of the design. For example, in the case of the memory megacells required, once the transistor-level design is completed, two functional models must be created: a behavioral description of the memory using an HDL for use during RTL and gate level simulations, and a fault simulation model.

In addition to generating the megacells, the user must ensure c o m p a t i b i l i t y between the soft core specification and the generated megacells. This may require some modification of the soft core (such as modifying the timing of reads and writes to memories in relation to the address bus transitions). The user is also responsible for other tasks such as design-for-testability and test of the megacells.

Synthesis results

As a result of the synthesis process, the synthesized picoJava-II core contains 167 I/O ports and 6,801 flip-flops. Figure 4 shows the areas of

expertise necessary to generate the megacell. In our implementation of the picoJava soft core, we found this problematic—the stack cache is defined as a five-port memory, while the memory generator we used only supports single or dual port memories. Therefore, we had to modify the soft core specification to let
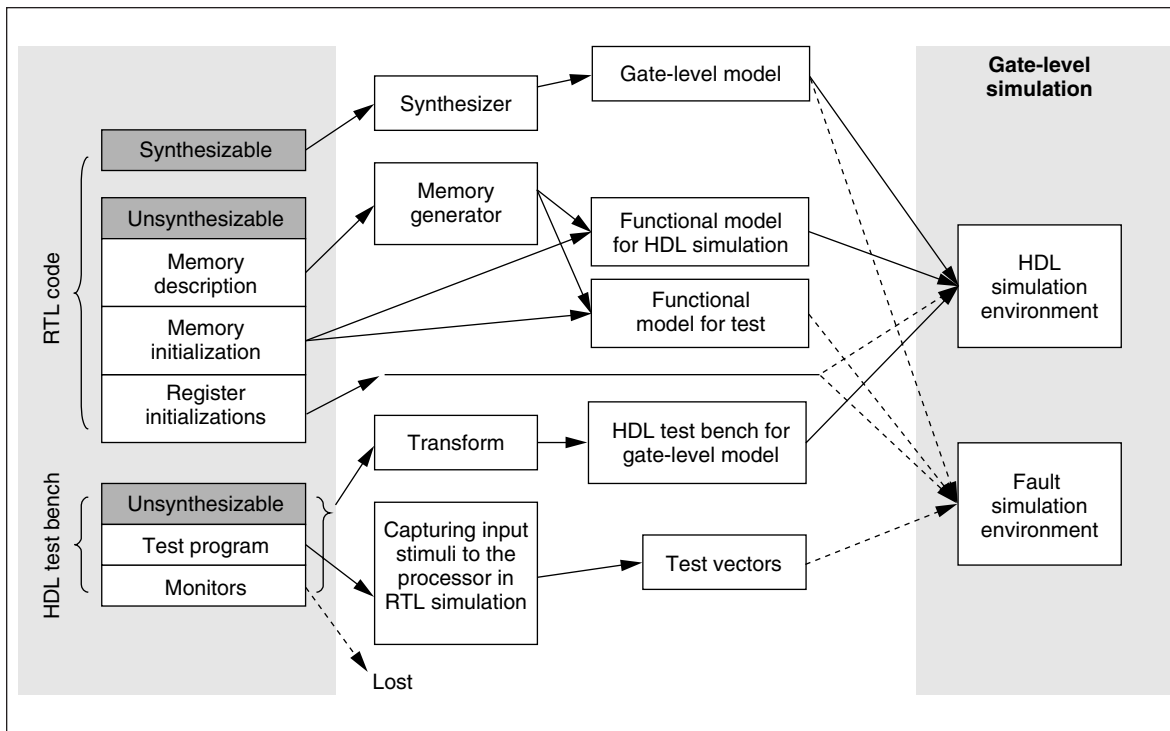
Figure 5. From RTL simulation to gate-level simulation: challenges and solutions.

each picoJava component in terms of the number of equivalent 2-input NAND gates. A memory generator obtained the areas for the data cache and the instruction cache. The total area is 127,887 equivalent NAND gates for the logic components and 313,989 equivalent NAND gates for the embedded memory components.

Validation of the RTL and gate-level implementations

Before the picoJava core can be integrated in an SoC design, the user must verify the core at the RT and gate levels. We describe our experience in validating the RTL model, as well as a methodology we have developed for validating the post-synthesis gate-level model.

Before synthesizing the picoJava core, the user must verify the core at the RT level, with the changes or configurations core users might make to satisfy their needs. Additionally, verifying the generated megacells (memory components) and associated interfaces (timing and protocol) with the picoJava core requires RTL simulation of the core with the newly generated megacells. The verification environment of picoJava provides a methodology for validating the RTL model, possibly modified by the core user, with the instruction-accurate model by concurrently running both simulation and checking the architectural state after every instruction.

We simulated the RTL implementation with ModelSim,[5] an HDL simulation tool from Mentor Graphics. Since the picoJava distribution did not have support for the ModelSim environment, we had to rewrite some programming language interface-related functions, along with some compilation and execution scripts. In addition, since different simulation tools handle events in different orders, the simulation behavior resulting from ModelSim did not match the specification. Therefore, we had to modify the test-bench environment of the picoJava core.

Verifying the correctness of the gate-level implementation of the picoJava processor core requires comparing the simulation results of the gate-level implementation with that of the RTL model using the picoJava validation test suite (a set of functional test programs).[3] In addition to being validated in the HDL simulation environment, the gate-level implementa-

tion had to be validated in the fault-simulation environment (used for developing manufacturing tests).

To ensure the success of the gate level simulations, it's crucial to convey all information used in the RTL simulation environment to the gate-level simulation environment (Figure 5). The information used by the RTL simulation environment includes the RTL code and the test bench. The test bench provides a means of executing an assembly program using the processor. In addition, it uses a large number of monitors for probing the processor's internal nodes, used for self-checking and debugging. The test bench is used for simulation only and cannot be synthesized.

Moreover, some sections of the RTL code are not meant to be synthesized. These include functional descriptions for the embedded memory blocks, and initialization for the memories and registers. Even though test benches and unsynthesized sections of the RTL code are not synthesized, they may contain information necessary to execute the test programs correctly. Therefore, for correct validation of the gate-level implementation of the core, the core user must ensure that all information, including the information in the unsynthesized code, is provided to the gate-level simulator.

We conveyed the unsynthesized information from the RTL simulation environment to the gate level simulation environment as follows:

1. *Memory descriptions and initializations.* Using a memory generator, we generated two functional models for each memory module. The two functional models can be used in the HDL simulation environment and the fault simulation environment, respectively. Compared to the original functional model, the new functional models are closer to their gate-level implementation, as they contain timing information. The initialization for the memory can be incorporated into these functional models in the form of initialization files.

2. *Register initialization.* For the registers, both gate-level simulators (HDL and fault) provide methods for their initialization.

3. *HDL test bench.* First, for the gate-level simulation in the HDL simulation environment, the HDL test bench for the RTL model can be transformed to a test bench for the gate-level model. Specifically, all monitors referencing to signals in the RTL model must be redirected to their counterparts in the gate-level model. Note that if the synthesizer uses a cross-boundary optimization, some RTL signals may no longer appear in the gate-level model. One possible solution is to preserve the design hierarchy during synthesis, which may lead to a suboptimal synthesis solution. Another solution is to remove the monitors referring to nonexisting signals, which could make debugging more difficult. Second, in the fault simulation environment, the monitoring information is lost, since the fault simulation environment does not provide any features equivalent to HDL monitors. Besides the monitors, the HDL test bench also contains information regarding the execution of a test program. We passed this information to the fault simulation environment in the form of test vectors by capturing the input stimuli to the processor during the RTL simulation.

After conveying all the RTL simulation information to the gate-level simulation environment, we validate the gate-level model by simulating test programs in the picoJava test suite. Because the test suite includes a large number of test programs, we developed an iterative method for simulating all test programs. First, a test program is chosen from the test suite. We then compare the gate-level simulation behavior with the RTL simulation behavior. If the simulation results are consistent, we conclude that the gate-level description behaves correctly under the selected test program. The validation process is then repeated for other test programs from the test suite. If a discrepancy occurs between the gate-level and the RTL simulation behaviors, the synthesis process has to be revisited.

The core user's first synthesis attempt does not always succeed. To pinpoint the source of the error, we used a top-down approach, as illustrat-
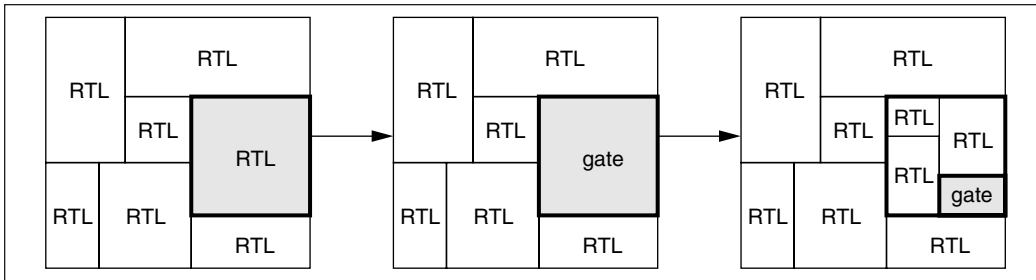
Figure 6. A top-down approach for debugging.

ed in Figure 6. Given that the original RTL model is working properly, we start from the RTL model and gradually replace its submodules by their gate-level implementations. We then perform HDL simulation on the entire processor. If the gate-level implementation of a submodule behaves incorrectly in this context, we may conclude that the error occurs within this submodule. We then repeat this process on the submodule until we finally isolate the error source.

To use this top-down approach, the core user needs to preserve the design hierarchy during synthesis. Once the hierarchical implementation has been verified, the core user can then move on to a flat implementation of the core, which may be more optimized due to cross-boundary optimization. The flat implementation will have to be verified again, since not all synthesis problems surface during the verification of the hierarchical implementation.

To summarize, according to our experience in validating the picoJava processor core, we faced three main challenges. First, the soft core distribution provides no clean boundary between the synthesizable and the unsynthesizable parts of the RTL simulation environment. It became the core user's responsibility to identify the unsynthesizable parts and transfer them to the gate-level simulation environment. Second, the core provider supplied no mechanism for transforming the RTL test bench to a test bench usable by the gate-level simulation environment. Third, the signals used as monitors in the RTL test bench may be absent from the gate-level implementation, making it difficult to debug the implementation. The core user's validation effort can be drastically reduced if the core provider addresses the above challenges and provides

mechanisms to aid in developing a correct and efficient gate-level simulation environment.

Testability of the picoJava processor core

An integral aspect of core-based SoC design is to ensure that each core is testable, since after the integration of the SoC, the entire SoC has to be tested for manufacturing defects.[6] Since a soft core is synthesized and implemented by the core user (system integrator), the testability of the soft core typically falls to the core user. In the case of picoJava, the core provider has included placeholders for design-for-test (DFT) structures in the RTL description, such as full-scan for the logic components and built-in self test (BIST) for the embedded memories. However, the core user must insert the actual DFT structures. In this section, we evaluate the testability of the logic components in picoJava using full scan and logic BIST (LBIST).

By converting all (6,801) flipflops in the picoJava core into scan flip-flops, full-scan results in an area overhead of 11.13% (with respect to the area of the logic components). This is an underestimation, as the area estimation does not include the routing overhead of scan chains. With 12,736 test vectors, the final fault coverage is 95.54%.

For a larger and more complex microprocessor, the tester time for full scan can become prohibitively high. Moreover, to apply at-speed testing with full scan requires high-speed external testers, which can become prohibitively expensive for today's high-speed ICs.

In recent years, self-testing methods have gained more popularity, enabling at-speed testing without high-speed testers. Among these methods, BIST[7] applies tests on-chip using
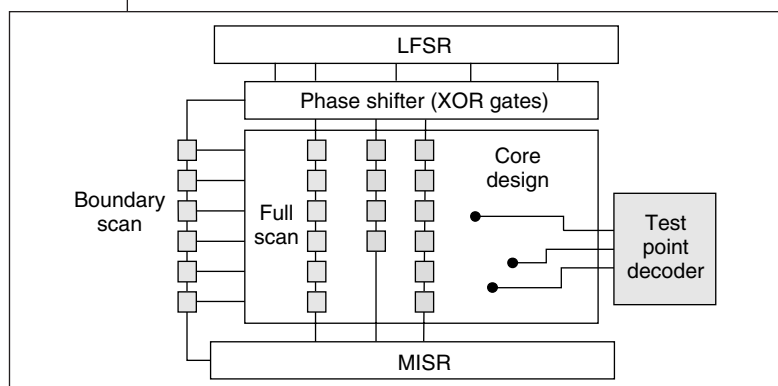
Figure 7. LBIST

embedded hardware test pattern generators and test response analyzers. To make picoJava self-testable, as well as to evaluate the effectiveness of current LBIST techniques for commercial processor cores, we applied a commercial LBIST tool on the picoJava processor core.

LBIST employs a STUMPS (self-test using MISR and parallel SRSG) -based BIST architecture,[8] as shown in Figure 7. Instead of relying on an external tester for applying tests to the scan chains, LBIST generates test vectors on-chip using a linear feedback shift register (LFSR). The outputs of the LFSR are connected to the scan chains through a phase shifter designed to reduce the linear correlation among scan chains. The outputs of the scan chains are compressed on-chip using a multiple input shift register (MISR). Test points may be inserted to improve fault coverage.

As LBIST relies on the application of random test patterns, the design-under-test often has to undergo extensive design changes to become random-pattern testable.[9] For picoJava, we had to make several design changes prior to application of LBIST. First, the embedded memories were bypassed with scan flip-flops in the test mode,[9] lest these become sources of undefined

values (X-generators), leading to the corruption of MISR signatures. In addition, a number of combinational loops, which did not exist in the functional mode, were formed when random test patterns were applied in the BIST mode. The signals in a combinational loop may toggle when the loop is activated, causing the generation of undefined values. The combinational loops can be broken with the help of control points. The breaking of the combinational loops, as well as insertion of the memory bypass circuits, had to be performed manually.

To study the effect of different parameters on the BIST results, we applied LBIST with different configurations on the picoJava processor core. Table 1 compares the results of different LBIST configurations with full scan. The values of area overhead and fault coverage reported are with respect to the logic part of the processor core.

In the first LBIST experiment (LBIST-1), we used an LFSR of size 24. LBIST chose a primitive polynomial to configure the LFSR. The number of random test patterns used was 32,767. The fault coverage was unacceptably low (58.81%), with an area overhead higher than that of full scan. The increase in the area overhead was caused by the insertion of the BIST circuitry, as well as the design modifications required for making picoJava BIST-ready.

To improve the testability of the design, in the second LBIST experiment (LBIST-2) we augmented the BIST circuitry in LBIST-1 by a large number of test points (100 control points and 100 observe points). This lead to a significant boost in the final fault coverage (an additional 23.72% compared to LBIST-1). The insertion of the test points lead to a slight increase in area.

We also investigated the effect of other BIST parameters on the resulting fault coverage,

Table 1. PicoJava test results: full scan and different LBIST configurations.

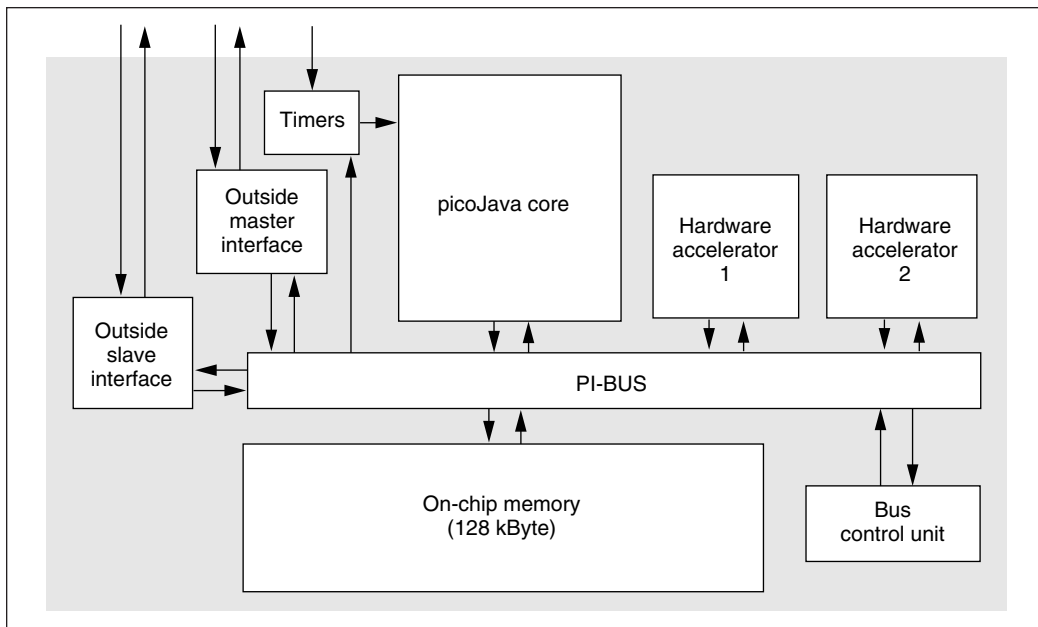|	LFSR Size	MISR Size	No. of Test Points Control	Observe	Area Overhead	No. of Test Patterns	Fault Coverage
Full Scan	—	—	—	—	11.13%	12,736	95.54%
LBIST-1	24	41	0	0	13.06%	32,767	58.81%
LBIST-2	24	41	100	100	13.29%	32,767	82.53%
LBIST-3	32	41	100	100	13.30%	32,767	82.93%
LBIST-4	24	41	100	100	13.30%	1,000,000	84.11%

Figure 8. PicoJava SoC architecture.

including the LFSR size and the number of random patterns used in test. In LBIST-3, we increased the LFSR size to 32-bit. In BIST-4, the number of random patterns used increased to 1,000,000. Neither change leads to a significant improvement in the final fault coverage.

These results show that LBIST by itself does not provide adequate fault coverage. However, with the aid of test points, LBIST can provide high fault coverage. Compared with the impact of test points, the size of the LFSR and the number of patterns used has much less impact on the fault coverage.

A Configurable Image Compression SoC

In this section we describe our experience in designing an SoC using the picoJava processor core.

SoC Architecture

The SoC architecture we developed, including the picoJava processor core, implements reconfigurable image compression for wireless multimedia communication. By using reconfigurable image compression, we can adapt the image compression algorithm and parameters being used to current channel conditions and quality-of-service requirements. This allows us

to implement the best image compression algorithm and parameters for current wireless conditions rather than a worst-case scenario.

Our SoC architecture for reconfigurable image compression appears in Figure 8. The architecture includes the picoJava processor core, two hardware accelerators, an on-chip memory, and an internal system bus. The hardware accelerators are designed to implement the most compute-intensive portions of image compression algorithms, while the picoJava processor implements the more control-intensive, parameterizable portions of the algorithms. The on-chip memory is 128 Kbytes of SRAM, while PI-Bus,[10] a soft core, is used as the system bus.

Integrating the picoJava Core in the SoC

Using a core in an SoC requires that the core be able to interface with the rest of the components on the chip. For our design, we chose to have a central system bus, which fits well with the picoJava bus interface model. However, using picoJava still has many effects on the design of the rest of the system.

The insertion of the picoJava core into our SoC affected our choice of communication architectures. First, the communication architecture must support transactions required by the picoJava core, which includes memory

reads and writes of different widths and burst lengths. Second, timing and control signals for all transactions coming from the picoJava core must match the specification of the communication architecture. In our implementation, we chose PI-Bus because it supports the same types of transactions required by the picoJava soft core. However, we had to implement an interface between picoJava and the bus to correctly match the PI-Bus specification.

In addition to implementing an interface between the picoJava core and the system bus, we also had to modify the other units in the SoC when inserting the picoJava core. Because picoJava assigns different memory address spaces for different purposes, each unit must be modified to make sure it resides in the proper memory space of picoJava. For example, the bus control unit and the hardware accelerators must be modified to ensure that the memory units of the hardware accelerators are in the uncached memory space of picoJava.

Sometimes additional components also have to be added to obtain full functionality of the picoJava processor. For example, in our design we had to implement hardware timers to allow for multitasking and other operating systems tasks that otherwise would not operate correctly on the picoJava processor core.

Hence, integration of the picoJava core in an SoC design requires modification of the entire design and not just the picoJava core. These modifications include modifying the bus interface unit between picoJava and the system bus, modifying the bus control unit and other hardware components to communicate with picoJava properly, and adding the other components necessary to use all the capabilities of picoJava.

The SoC design is validated using the RTL simulation environment of the picoJava core. However, we are currently developing a co-simulation environment to enable fast and efficient system-level validation.   ■

OUR EXPERIENCE relates to use of the picoJava processor core in developing an image compression SoC, but the design, test, and integration experiences are representative of the use of any soft core in any application-specific SoC. The design steps and challenges involve core synthesis and optimization to target the desired technology library, developing methodologies to validate the core implementation with the functional specification, making the core testable, and integrating the core in the application-specific SoC.   ■

Acknowledgments

■ References

1. R.K. Gupta and Y. Zorian, "Introducing Core-based System Design," *IEEE Design & Test of Computers*, Oct.-Dec. 1997, pp. 15-25.

2. "PicoJava Microprocessor Core," Sun Microsystems, http://www.sun.com/microelectronics/picoJava/.

3. *PicoJava-II Verification Guide*, Sun Microsystems, March 1999.

4. "Synopsys Design Compiler Family," http://www.synopsys.com/products/logic/design_compiler.html.

5. *ModelSim EE/PLUS Reference Manual*, Version 5.2, Model Technology, Portland, Ore., 1999.

6. Y. Zorian, E.J. Marinissen, and S. Dey, "Testing Embedded-Core-Based System Chips," *Computer*, June 1999, pp. 52-60.

7. V.D. Agrawal et al., "Built-in Self-Test for Digital Integrated Circuits," *AT&T Technical J.*, Mar. 1994, pp. 30.

8. P. H. Bardell ,W.J. McKenney, J.Savir, *Built-in test for VLSI: Pseudorandom Techniques*, John Wiley and Sons, New York, 1987.

9. G. Hetherington et al, "Logic BIST for Large Industrial Designs: Real Issues and Case Studies," Proc. of the Int. Test Conf. 1999 IEEE**,** Sept. 1999, pp. 358-367.

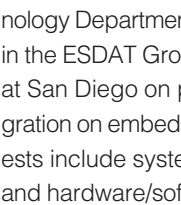10. "PI-Bus Toolkit," http://www.sussex.ac.uk/engg/research/vlsi-Jan97/projects/pibus/.

Sujit Dey is an associate professor in the Electrical and Computer Engineering Department at the University of California, San Diego. The goal of his research group at UCSD is enabling the use of nanometer technologies and software configurability to develop embedded solutions for mobile multimedia systems. He was a senior researcher at NEC C&C Research Laboratories in Princeton, NJ, before joining UCSD. Dey received his PhD degree in computer science from Duke University. He is a co-inventor of 10 items US patents. He received Best Paper awards at the Design Automation Conferences in 1994, 1999, and 2000, and the 11th VLSI Design Conference in 1998. He is affiliated with the DARPA/MARCO Gigascale Systems Research Center and the Center for Wireless Communications at UCSD.

Pablo Sanchez received his BS in 1987 and PhD in 1991 in electronics from the University of Cantabria, Spain, and is currently titular professor with the Electronics Technology Department there. During 1999, he worked in the ESDAT Group of the University of California at San Diego on projects related to IP Core integration on embedded systems. His research interests include system verification, core integration, and hardware/software co-design.

Debashis Panigrahi is a PhD student at the University of California, San Diego in electrical and computer engineering. He received his BTech Degree from the Indian Institute of Technology, Kharagpur, in computer science and engineering in July 1998. His main research interests include hardware/software co-design and co-simulation, low-power design, and mobile networks.

Li Chen is pursuing a PhD in electrical and computer engineering at the University of California at San Diego. Chen recieved her BS and MS degrees in electrical and computer engineering from Carnegie Mellon University in 1996 and 1998, respectively. Her research interests include the self-testing and the self-diagnosis of microprocessor cores.

Clark N. Taylor is currently a second-year PhD student and SRC Graduate Fellow at the University of California, San Diego in electrical and computer engineering. He received his MS in Electrical Engineering in August 1999, and his BS in Electrical and Computer Engineering in 1995, from Brigham Young University. His main research interests include test, low-power design, embedded systems design, and mobile multimedia algorithms and implementations.

Krishna Sekar is currently a PhD student at the University of California at San Diego in the Electrical and Computer Engineering Department. He received his BTech degree from the Indian Institute of Technology, Kharagpur, in computer science and engineering in July 1999. His main research interests include low-power VLSI testing, built-in self-test, and formal verification.

■ Address comments and questions to Sujit Dey, Department of Electrical and Computer Engineering, 9500 Gilman Drive, Mail Code 0407, University of California, San Diego, La Jolla, CA 92093-0407; dey@ece.ucsd.edu.