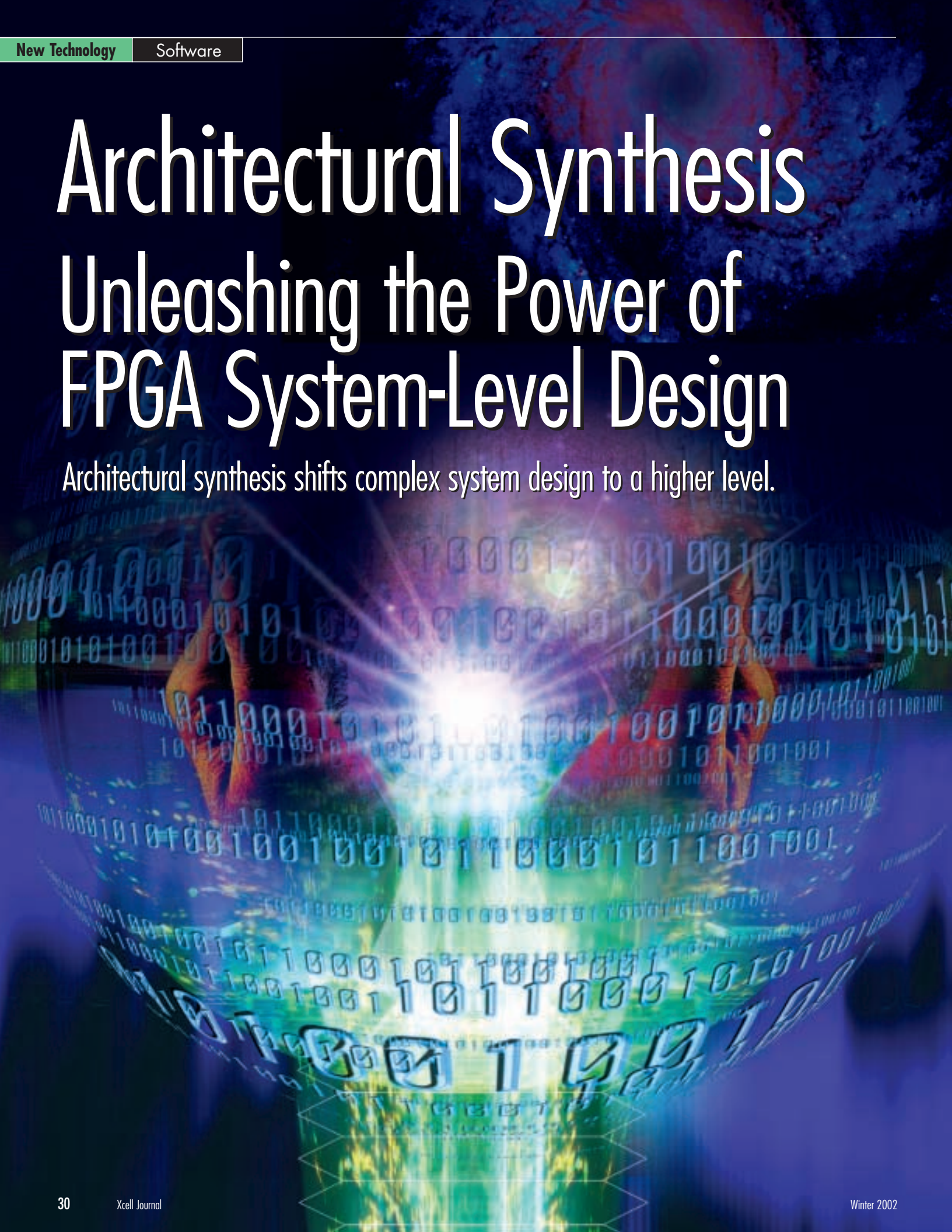
Architectural Synthesis

Unleashing the Power of FPGA System-Level Design

Architectural synthesis shifts complex system design to a higher level.



by Don Davis
Manager, High Level Language Tools
Xilinx, Inc
don.davis@xilinx.com

When you combine the capabilities of powerful Virtex-II™ Pro FPGAs with the wide range of hardware cores now available (from soft processors such as MicroBlaze™ to bus interfaces such as PCI), you've got all you need to develop complete systems on a single device. However, with all of this capability comes added design complexity. How do you take advantage of these vast resources and deal effectively with the added system complexity?

FPGAs have evolved beyond glue logic into fundamental system elements. To remain relevant, development methodologies must respond to this changing role by providing the appropriate abstraction levels and tools needed to manage this complexity. You need:

- High-level languages to support the capture of complex design functionality in an abstract manner
- Profiling and characterization to explore solution space tradeoffs
- Debugging and verification tools to ensure design integrity
- Compilers and optimizers to produce high quality implementations.

A New Approach

A compelling design methodology based on Architectural Synthesis (AS) offers a comprehensive strategy for managing all of these issues. AS streamlines design, verification, and implementation of complex systems by leveraging powerful development tools and advanced FPGA devices. AS enables you to define system functionality at a high level of abstraction (using a software-based design entry point) and then debug, synthesize, and verify a range of architecture implementations that meet the system specification. What makes AS so exciting is that it is based on a single system specification, so you can easily explore a variety of hardware implementations to achieve the optimal cost/performance point for your application – and even change the hardware/software partitioning – without having to modify the source specification.

The System Design Challenge

Effective system design in the era of platform FPGAs requires a holistic approach. No longer is FPGA design simply about mapping your algorithm to LUTs. In the first place, today's design complexity has grown to such an extent that you need higher-level methods of algorithm specification and design capture. What's more, with embedded processors of both the hard and soft varieties, your implementation options are vastly expanded: Should I implement this piece of functionality in the FPGA fabric or on the embedded processor? What is the impact on the system with respect to performance? How should the various processing elements communicate?

Complicating matters still further, the answer to any one of these questions can affect the answer to the others. A local optimization, for example, may not be a system optimization. You must be able to explore these interactions quickly and cover the entire solution space with minimal effort if you're to have any hope of achieving an optimal solution.

By contrast, the current process is a lot like throwing darts blindfolded. Your designer intuition will usually get you facing the dartboard, but hitting the bulls-eye is mostly a matter of luck. You pick a hardware/software partition based on your experience and some limited modeling or profiling, and hand it off to the rest of the design team. Barring catastrophic circumstances, the partition is fixed at that point; it is simply too difficult to go back and rework all the hardware implementations and interfaces because they're all products of manual translations of the specification.

Synopsys Provides the Multiple Levels of Abstraction Necessary for AS

"Higher levels of abstraction are crucial for exploring system specifications, for reaching hardware/software architecture closure early in the design cycle, and for decreasing implementation cycles," reports Joachim Kunkel, vice president of IP and systems marketing, Synopsys.

Synopsys' CoCentric System Studio application, for example, gives you the multiple levels of abstraction needed to accomplish a range of tasks: Untimed Functional for data exchange; Timed Functional for computational and communicational delay; and Transaction Level, the natural meeting point for hardware and software designers to achieve cycle-true platform performance analysis.

These abstraction levels offer verification speeds that are higher than those offered by RTL, by orders of magnitude – and yet they give you sufficient detail to do platform analysis and come to closure on the hardware/software architecture early. (Although high-level abstraction offers an effective way to deal with today's increasingly complex designs, and definitely offers design-cycle time benefits, it does not replace the pin-accurate models necessary for an automated path to hardware implementation.)

Because this methodology consists of design, debug, verification, and implementation in hardware and software, the company calls it "SystemC Design and Verification." But the CoCentric solution also gives you the option to use RTL, giving you complete design control, especially in cases where the required hardware architecture is very well understood.

Using the appropriate level of abstraction and automation for your analysis and implementation – and combining that with a unified hardware/software methodology and the unique dual programmability of Platform FPGAs – enables you to create differentiated products cost-effectively.

Your team may never know if your choice was a good one, or whether this was an optimal partition. You just have to make it work, so you spend lots and lots of time optimizing and tweaking code (hardware or software) in a struggle to make sure your design meets the system specification.

This is a major weakness in the flow. On the software side, the translation from the software specification (usually written in C) into the C implementation for the embedded processor is straightforward. But translating the software specification into hardware implementations (usually Verilog, VHDL, or RTL hardware description languages) is another matter entirely.

Typically, you or your team interpret the specification and tediously convert it into a hardware implementation that (you hope) will meet the system specifications. Here again, this approach gives you another chance at "blindfolded darts." The level of resource sharing, number of pipeline stages, and amount of loop unrolling are just a few examples of the many decisions that are difficult to change at the RTL – and which you have to make up front.

All of these decisions affect the performance and area of the final implementation, and all of these issues offer a range of options for a given algorithm. Your ability to explore the solution space is severely limited if you have to re-code the HDL by hand because you know that each re-code takes time, both in terms of the design itself and the subsequent verification of the new implementation.

Architectural Synthesis to the Rescue

AS comprises a suite of technologies designed to meet the challenges associated with system-level design – and help you realize its benefits. (See the sidebar stories for an overview of the different ways our partners are incorporating AS into their design flows.) AS offers an

Using Forte's Cynthesizer and AS to Improve Process and Outcome

Architecture modeling and synthesis allows groups to produce better designs faster. Combining the power of AS with C++ design, verification, and software development marks a significant step forward in the design process.

FORTE DESIGN SYSTEMS

With AS methodology and appropriate constraints and directives, you can create multiple RTL implementations from one C++ model in minutes, each implementation representing a unique tradeoff between performance, area, and power. Forte's Cynthesizer customers have found that designs created and verified in C++ typically yield a 20x to 30x reduction in lines of code, simulate faster by orders of magnitude, and reduce the design schedule by 50% or more.

Imagine your group is creating a cellular phone chipset. Among the design elements you'll want to consider are the tradeoffs between performance and die size, and between hardware and software implementations of a JPEG algorithm. To get an accurate hardware estimate, you'll first need to produce RTL code, and then apply RTL estimation tools or logic synthesis. Using traditional methods, that process would take your team several calendar months – 50 to 100 engineering months – to create one RTL implementation.

With AS, on the other hand, you can automatically create a range of RTL implementations from high-level C++. Armed with this data, you can trade off hardware and software implementations with confidence.

The exploration capability of AS, coupled with the sheer productivity gain in moving to behavioral C++, makes the AS/C++ combination the designer's power tool for the complex systems of tomorrow.

Mentor Graphics®

Savvy Design Teams Are Re-Evaluating Their Design Practices

Using programmable SOCs (system-on-chips) in combination with higher-level building blocks, design teams can now optimize an entire system's performance throughout the development process – eliminating the performance issues that cause costly delays. Mentor Graphics and Xilinx have teamed up to provide an advanced EDA (electronic design automation) and silicon solution, setting the stage for true platform-based design.

Multimillion-gate FPGAs with embedded processors and high-speed interfaces require architectural solutions tailored to specific design needs. Issues such as hardware/software partitioning and validation, board interconnect, and system-level verification can all lengthen your time to market. The key to efficient and effective design is to employ an integrated flow that brings together hardware, software, and board and layout engineers early in the design process.

Mentor's comprehensive, system-level FPGA design solution, including design creation and management, hardware/software co-verification, simulation, synthesis, and PCB (printed circuit board) analysis and layout, empowers the complete design team. All team members can take advantage of the advanced building blocks found in today's FPGAs and avoid costly delays. At Mentor Graphics we are committed to delivering complete and integrated solutions that support AS. Our goal is to help you eliminate performance issues and shorten your time to market.

impressive array of tools that address design, partitioning and optimization, debugging and verification, and reuse. Let's look at each one in turn.

Design

AS improves up-front design decisions because it's based on a high-level language specification. It's a lot easier to manage your design functionality when you don't have to worry about register and interface timing. AS works to capture the functionality and get it verified quickly, and then automatically compiles to an implementation that meets your system specifications. AS also makes it easier to trade off design constraints against performance goals. For example, if you run a compilation and the resulting implementation doesn't meet your performance criteria, you simply rerun the compiler and ask it to improve the performance by using more hardware resources – it's simple and painless.

Partitioning and Optimization

Improved partitioning and optimization are core benefits of an AS flow. AS enables you to define hardware/software partitions easily, push a button, and have the tools automatically generate the software-executable and hardware bitstream, as well as all of the routing required to enable the hardware and software components to communicate effectively. This includes the synthesis of buses and bus interfaces, as appropriate, as well as the software drivers necessary to support them.

The ability of AS to accomplish this automatically, with minimal user interaction, is key. The more you have to do by hand, the less you're going to iterate to find the optimal solution. The power of AS is that these automatic tools make it easy to explore the entire solution space quickly, enabling you to find the best solution for your application. To evaluate potential candidates in the solution space effectively, the tool must be able to profile the candidates with respect to such design considerations as throughput, memory usage, and FPGA area. You'll also appreciate being able to evaluate design bottlenecks. You can easily answer questions such as: "Is the system constrained because there is too much traffic over the system bus, or because the memory accesses are taking too long?"

On the hardware side of the partition, selection is even more complex. In other words, a typical software implementation most often involves a single performance point. If you want better performance, you need a faster processor. Code-tuning can provide improvements at the cost of memory, but overall the range of implementations is limited. An FPGA implementation, on the other hand, can involve a wide range of performance points based on varying the hardware architecture. More gates typically (but not always!) equals better performance. An example of a classic hardware tradeoff is adding pipeline stages to dramatically improve throughput at the expense of latency and area.

Here again, your ability to explore potential hardware architectures thoroughly is directly related to making the optimal design choices. AS enables you to do this automatically, simply by rerunning the compiler with different preferences. You don't have to rewrite specification code, which saves both design and verification time. Perhaps even more important, AS prevents you from introducing errors into this verified design.

Pipelining

It is in this area that high-level language-based methodologies truly shine. For example, changing the level of pipelining later in the traditional design flow is such a huge undertaking it's usually not even considered. If a design doesn't meet specifications, the entire design team typically spends significant time and energy trying to tweak the design to achieve the specification performance.

In an AS flow, you don't need to decide in the beginning about what level of pipelining is appropriate – you can make your decision at compile time. The resulting implementation is far more likely to converge on the optimal architecture for your system specifications.

Debug and Verification

The AS approach likewise expands your options for system debugging and verification. Traditional approaches to design verification rely exclusively on RTL simulation, which, while accurate, is unacceptably slow for large system designs. In addition, the interfaces to software code and ISS (Instruction Set Simulators) are clumsy and inelegant.

With AS, the system specification is automatically synthesized to implementations, so functional verification of the system specification is equivalent to functional verification of the implementation. By contrast, traditional methods of implementation involve a manual translation step, so verification of the system specification tells you little about the functional correctness of the ultimate implementation.

In addition, an AS debugging and verification toolset provides multiple layers of fidelity. The first level is the software paradigm. Here you can use traditional software debugging techniques, setting breakpoints and stepping to code, to chase down bugs. The next level creates a co-simulation environment, integrating simulation tools and ISS to delve into the details of how the hardware interacts with the software. Finally, you can even implement your design on a target FPGA and use tools, such as ChipScope™, to explore the details of the actual implementation running on real hardware.

Another key benefit of the AS approach is that it enables you to work at a level of abstraction appropriate for the level of design you're working on. For example, working at the clock cycle level, when your goal is simply to verify that the algorithm functions correctly, provides too much detail and will actually get in the way of understanding the algorithm's performance. Moreover, working at higher levels of abstraction is typically many times faster than working at the lower levels, enabling faster iteration time in the code-compile-debug cycle. Of course, for those times when you need to figure out how to remove "just one more" clock cycle to meet your throughput specification, working at the cycle-accurate simulation level, even though it is slower, will give you the detail you need.

Reuse

AS also facilitates efficient design reuse. A powerful tool in any designer's toolbox is the large set of IP you can use for standard system functions. Xilinx and its partners provide a wide range of IP products. These products plug directly into any design flow. However, designs that are functionally validated and implemented at higher levels of abstraction are more suitable for an AS flow because they can be reused in many future products – even where design constraints and performance goals are quite different. Because, with AS, a single source specification can provide multiple implementations simply by rerunning the compiler with different constraints, a single piece of IP can provide a variety of implementations addressing high throughput, small area, or some optimal combination of the two. Furthermore, the optimal combination can be determined expressly within the context of your specific design.

In addition, the AS flow facilitates development and verification of IP because the IP designer can work at the functional level. And because a single specification can be targeted to a wider range of implementations, IP developed using AS is likely to find broader application.

Conclusion

Architectural synthesis is both a powerful new tool in your quest for the optimal system design solution, and a mighty weapon in the fight against design complexity. AS requires a new way of thinking about systems – not as separate hardware and software domains – but as an integrated whole, the boundary of which is extremely fluid. FPGAs, with the flexibility of embedded processors and the ability to transmit data at Gigabit rates, provide the power to drive new classes of systems. Architectural synthesis provides a way to harness that power and develop your designs in record time.