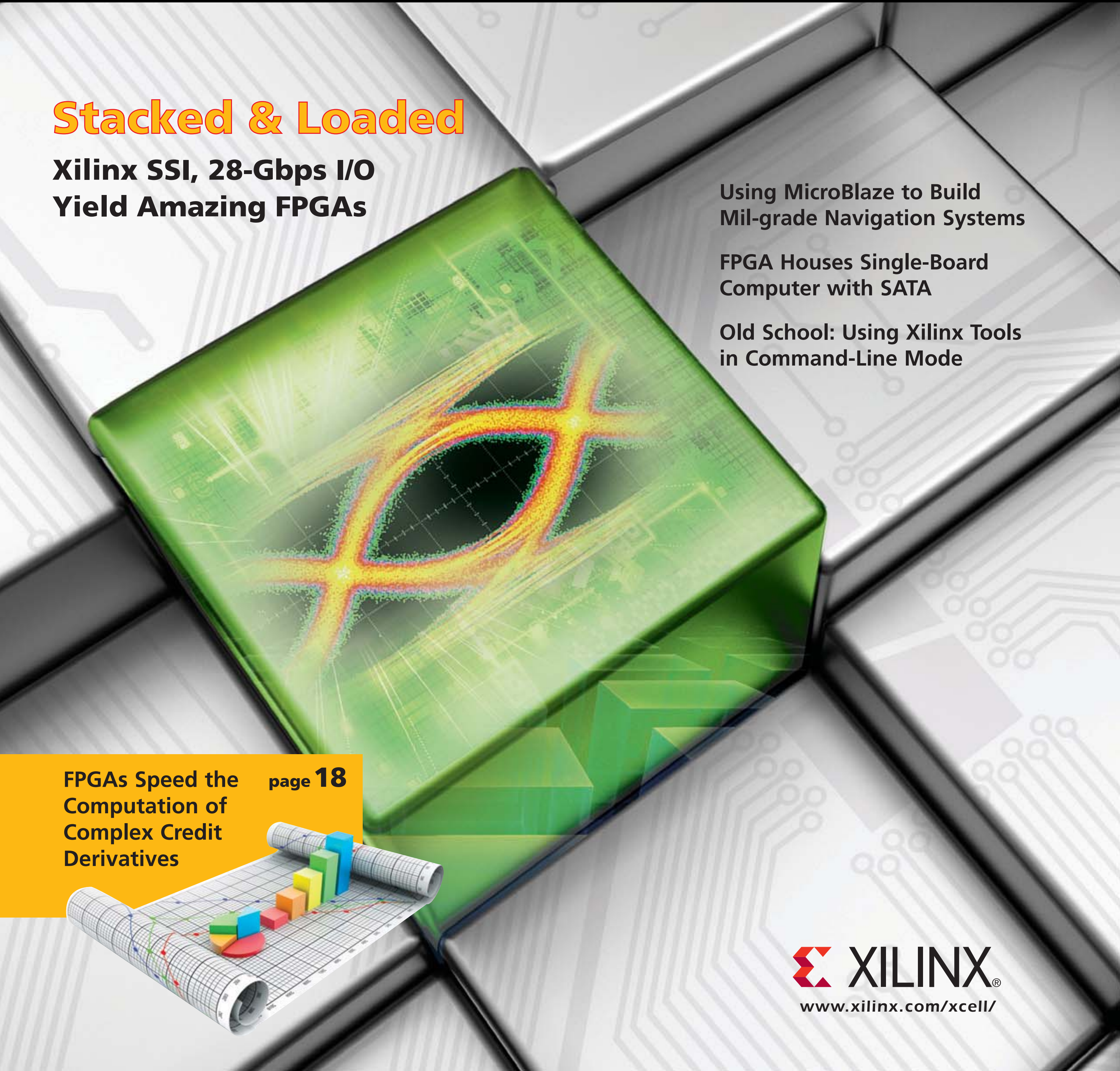
Xcell journal

ISSUE 74, FIRST QUARTER 2011

SOLUTIONS FOR A PROGRAMMABLE WORLD



Stacked & Loaded

Xilinx SSI, 28-Gbps I/O Yield Amazing FPGAs

Using MicroBlaze to Build Mil-grade Navigation Systems

FPGA Houses Single-Board Computer with SATA

Old School: Using Xilinx Tools in Command-Line Mode

FPGAs Speed the Computation of Complex Credit Derivatives

page 18

XILINX®

www.xilinx.com/xcell/

Quickly see inside your FPGA

Get the most out of your test equipment and shave time out of your next debug cycle with Agilent oscilloscopes and logic analyzers. Our innovative FPGA dynamic probe application allows you to quickly connect to multiple banks of internal signals for rapid validation with real time measurements. Reduce errors with automatic transfer of signal names from the .cdc file from your Xilinx Core Inserter.

Toggle among banks of internal signals for incremental real-time internal measurements without:

- Stopping the FPGA
- Changing the design
- Modifying design timing

Shown with: 9000 Series oscilloscope and 16900 logic analyzer with FPGA dynamic probes

Also works with all InfiniiVision and Infiniium MSO models.

See how you can save time by downloading our free application note.

www.agilent.com/find/fpga_app_note

u.s. 1-800-829-4444 canada 1-877-894-4414

Agilent Technologies

Xcell journal

PUBLISHER Mike Santarini
mike.santarini@xilinx.com
408-626-5981

EDITOR Jacqueline Damian

ART DIRECTOR Scott Blair

DESIGN/PRODUCTION Teie, Gelwicks & Associates
1-800-493-5551

ADVERTISING SALES Dan Teie
1-800-493-5551
xcelladsales@aol.com

INTERNATIONAL Melissa Zhang, Asia Pacific
melissa.zhang@xilinx.com

Christelle Moraga, Europe/
Middle East/Africa
christelle.moraga@xilinx.com

Miyuki Takegoshi, Japan
miyuki.takegoshi@xilinx.com

REPRINT ORDERS 1-800-493-5551

Xcell PUBLICATIONS
www.xilinx.com/xcell/

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124-3400
Phone: 408-559-7778
FAX: 408-879-4780
www.xilinx.com/xcell/

The articles, information, and other materials included in this issue are provided solely for the convenience of our readers. Xilinx makes no warranties, express, implied, statutory, or otherwise, and accepts no liability with respect to any such articles, information, or other materials or their use, and any use thereof is solely at the risk of the user. Any person or entity using such information in any way releases and waives any claim it might have against Xilinx for any loss, damage, or expense caused thereby.

Xilinx Alliance Program Streamlines IP-Based FPGA Design

In the last quarter, Xilinx has made several major announcements, including our 28-nanometer 7 series FPGAs with their new "More than Moore" stacked-die architecture, and even a Xilinx® Virtex®-7 FPGA family member that enables 400-Gbps bandwidth, as described in this issue's cover story. One announcement that you may have overlooked in this flurry of news came from the Xilinx Alliance Program group. Under the leadership of Dave Tokic, the team has completely revamped Xilinx's activities to strengthen the ecosystem of design and integration services, embedded software, design tools, development and evaluation boards, complementary silicon device technology and, especially, IP providers. This massive effort marks quite an accomplishment—one that should yield great benefits to Xilinx users as well as partners for years to come.

The two biggest issues that have plagued the IP industry since its emergence in the 1990s are how to know that the piece of IP you are using is of high quality, and which bus to use to connect all these different pieces of IP. Tokic's group at Xilinx has made huge inroads in addressing these problems. With the Extensible Processing Platform (EPP) architecture, Xilinx is currently creating a device that will marry an ARM® MPU core with FPGA fabric on the same device (see cover story, *Xcell Journal*, issue 71). We also worked with ARM to create the next version of the industry-standard AMBA® interconnect protocol, called AXI4, that is optimized for use in FPGAs. All of the EPP products as well as the 7 series devices will use the AXI4 interconnect. This companywide standardization on AXI4 allows customers and ecosystem providers that have already developed IP for AMBA to more easily migrate their IP into Xilinx FPGAs. It also makes it easier for Xilinx to build our own IP and better maintain our existing IP libraries moving forward.

Xilinx has also adopted IP industry standards such as IP-XACT, or IEEE 1685 (formerly known as SPIRIT), which establishes a standard structure for packaging, integrating and reusing IP within tool flows. We have settled on a common set of quality and implementation metrics (leveraging the efforts first undertaken by the VSIA) to create transparency for customers into IP quality and implementation details. Tokic's group also helped establish the industry's "SignOnce" program to facilitate customer licensing. Many IP vendors within the Alliance Program offer licensing via these simplified contracts. And with the release of Virtex-6 devices over a year ago, Xilinx standardized on the VITA FMC PCB standard, making it easier for IP vendors to offer development boards and daughtercards that easily plug into Targeted Design Platforms from Xilinx, Avnet and Tokyo Electron Devices.

What's more, Tokic's group took the guesswork out of choosing the right provider by establishing a deeper qualification process and tiering these suppliers based on criteria including business practices, technical and quality processes and track record. The program has three tiers: Alliance Program "Members" maintain credible customer satisfaction with their products and services and a track record of success. "Certified Members" have additionally met specific technical qualification on FPGA implementation expertise. At the top level, "Premier Partners" are audited to ensure they adhere to the highest standards.

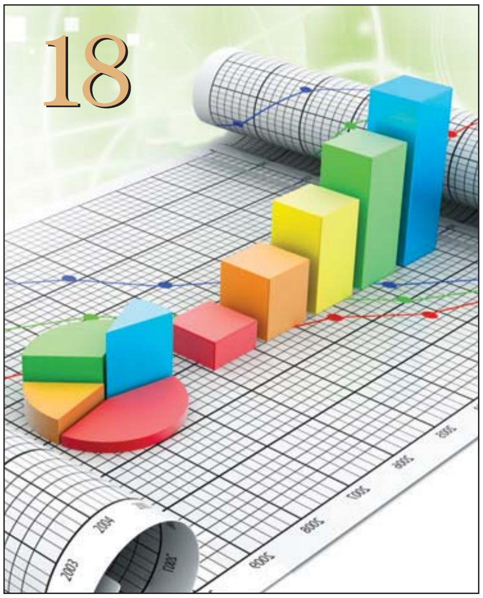
For a reporter who has spent many years covering the wars and dramas involved in establishing standards, it's awesome to see them finalized and being put to fabulous use. For more information, visit *http://www.xilinx.com/alliance/index.htm*.

Mike Santarini
Publisher

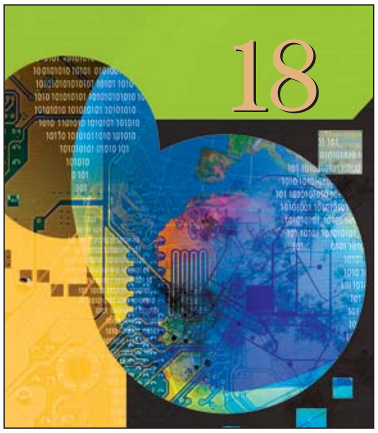
LINEAR
LTM4601V
µModule
XILINX
VIRTEX-6
XC6VLX240T
FF1156ABC0842
X6-RX

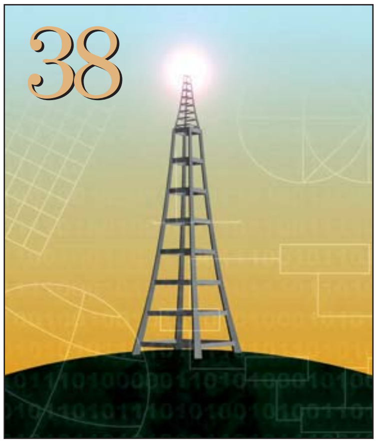
CONTENTS

VIEWPOINTS



XCELLENCE BY DESIGN APPLICATION FEATURES

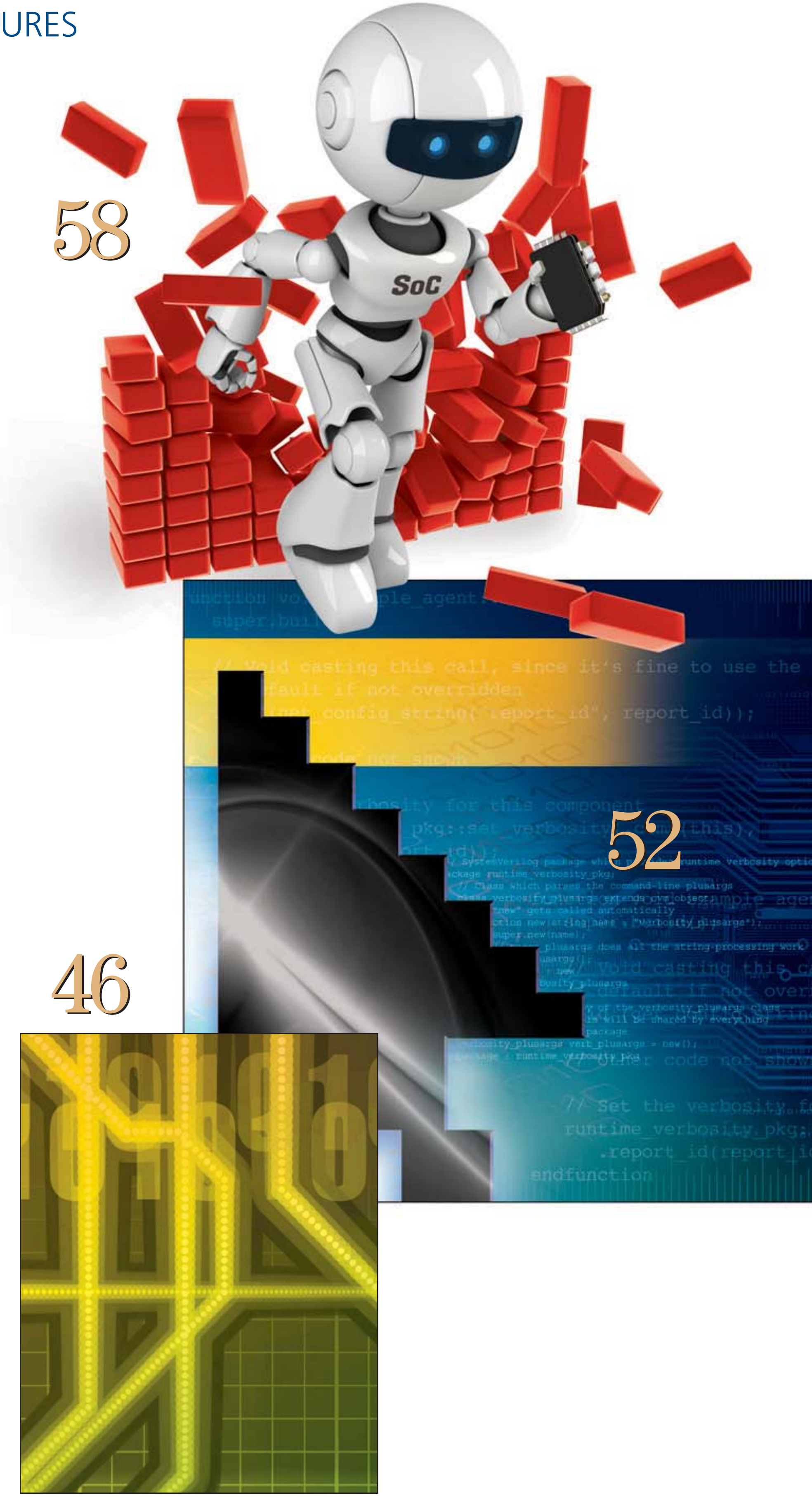




Cover Story

THE XILINX XPERIENCE FEATURES

XTRA READING



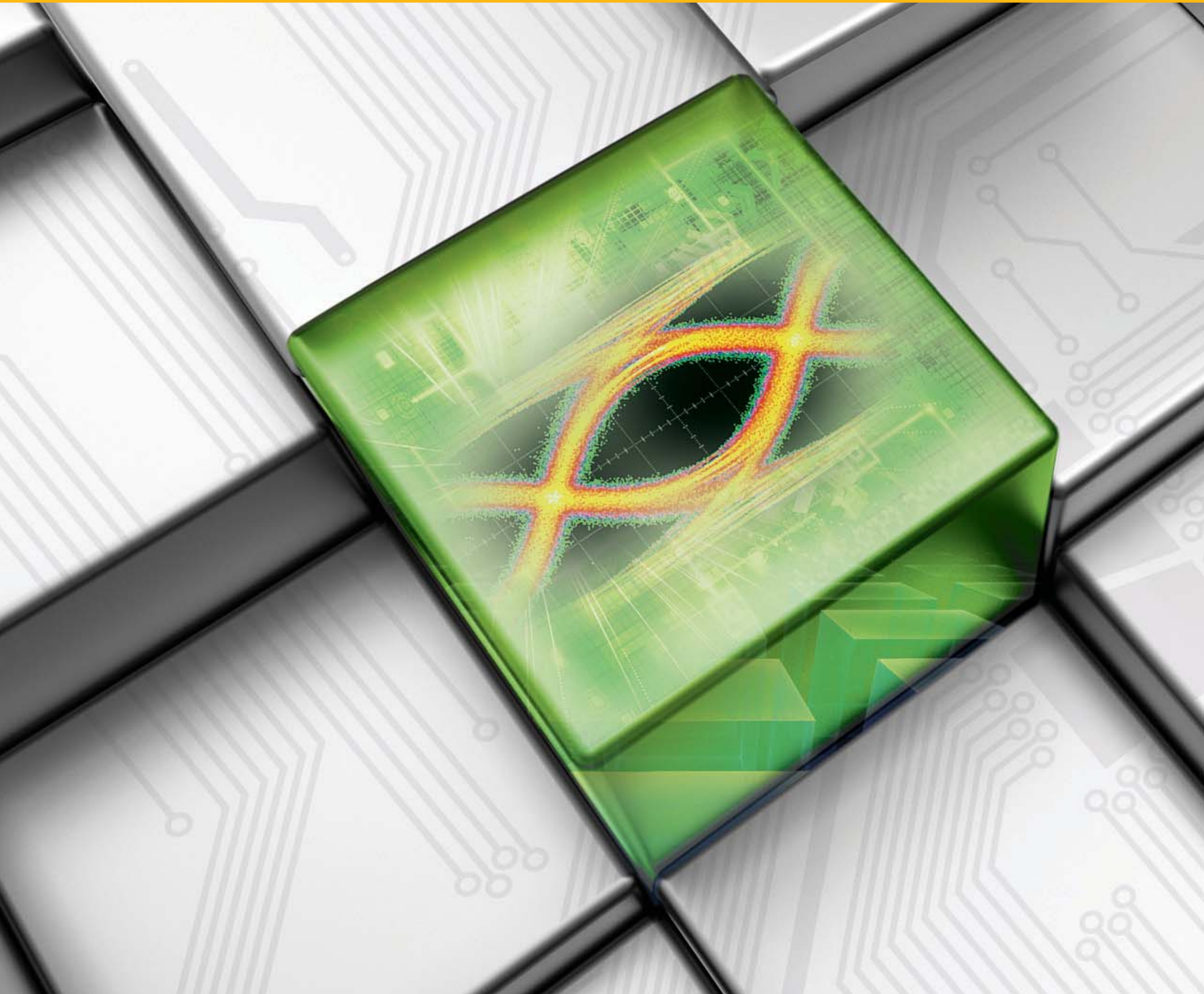
2010
APEX®
AWARDS FOR
PUBLICATION EXCELLENCE

Xcell Journal recently received 2010 APEX Awards of Excellence in the categories "Magazine & Journal Writing" and "Magazine & Journal Design and Layout."

DMA 2010
Quark
DIGITAL MAGAZINE AWARDS 2010
FINALIST
CELEBRATING THE BEST IN DIGITAL PUBLISHING

Stacked & Loaded: Xilinx SSI, 28-Gbps I/O Yield Amazing FPGAs

by Mike Santarini
Publisher, *Xcell Journal*
Xilinx, Inc.
mike.santarini@xilinx.com



‘More than Moore’ stacked silicon interconnect technology and 28-Gbps transceivers lead new era of FPGA-driven innovations.

Xilinx recently added to its lineup two innovations that will further expand the application possibilities and market reach of FPGAs. In late October, Xilinx® announced it is adding stacked silicon interconnect (SSI) FPGAs to the high end of its forthcoming 28-nanometer Virtex®-7 series (see *Xcell Journal*, Issue 72). The new, innovative architecture connects several dice on a single silicon interposer, allowing Xilinx to field Virtex-7 FPGAs that pack as many as 2 million logic cells—twice the logic capacity of any other announced 28-nm FPGA—enabling next-generation capabilities in the current generation of process technology.

Then, in late November, Xilinx tipped the Virtex-7 HT line of devices. Leveraging this SSI technology to combine FPGA and high-speed transceiver dice on a single IC, the Virtex-7 HT devices are a giant technological leap forward for customers in the communications sector and for the growing number of applications requiring high-speed I/O. These new FPGAs carry many 28-Gbit/second transceivers along with dozens of 13.1-Gbps transceivers in the same device, facilitating the development of 100-Gbps communications equipment today and 400-Gbps communications line cards well in advance of established standards for equipment running at that speed.

MORE THAN MOORE

Ever since Intel co-founder Gordon Moore published his seminal article “Cramming More Components onto Integrated Circuits” in the April 19, 1965 issue of *Electronics* magazine, the semiconductor industry has doubled the transistor counts of new devices every 22 months, in lockstep with the introduction of every new silicon process. Like other companies in the semiconductor business, Xilinx has learned over the years that to lead the market, it must keep pace with Moore’s Law and create silicon on each new generation of process technology—or better yet, be the first company to do so.

Now, at a time when the complexity, cost and thus risk of designing on the latest process geometries are becoming prohibitive for a greater number of companies, Xilinx has devised a unique way to more than double the capacity of its next-generation devices, the Virtex-7 FPGAs. By introducing one of the semiconductor industry’s first stacked-die architectures, Xilinx will field a line of the world’s largest FPGAs. The biggest of these, the 28-nm Virtex-7 XC7V2000T, offers 2 million logic cells and 305,400 logic slices, along with 46,512 kbits of block RAM, 2,160 DSP slices and 36 GTX 10.3125-Gbps transceivers. The Virtex-7 family includes multiple SSI FPGAs as well as monolithic FPGA configurations. Virtex-7 is the high end of the 7 series, which also includes the new low-cost, low-power Artix™ FPGAs and the midrange Kintex™ FPGAs—all implemented on a unified Application Specific Modular Block Architecture (ASMBL) architecture.

The new SSI technology is more than just a windfall for customers itching to use the biggest FPGAs the industry can muster. The successful deployment of stacked dice in a mainstream logic chip marks a huge semiconductor engineering accomplishment. Xilinx is delivering a stacked silicon chip at a time when most companies are just evaluat-

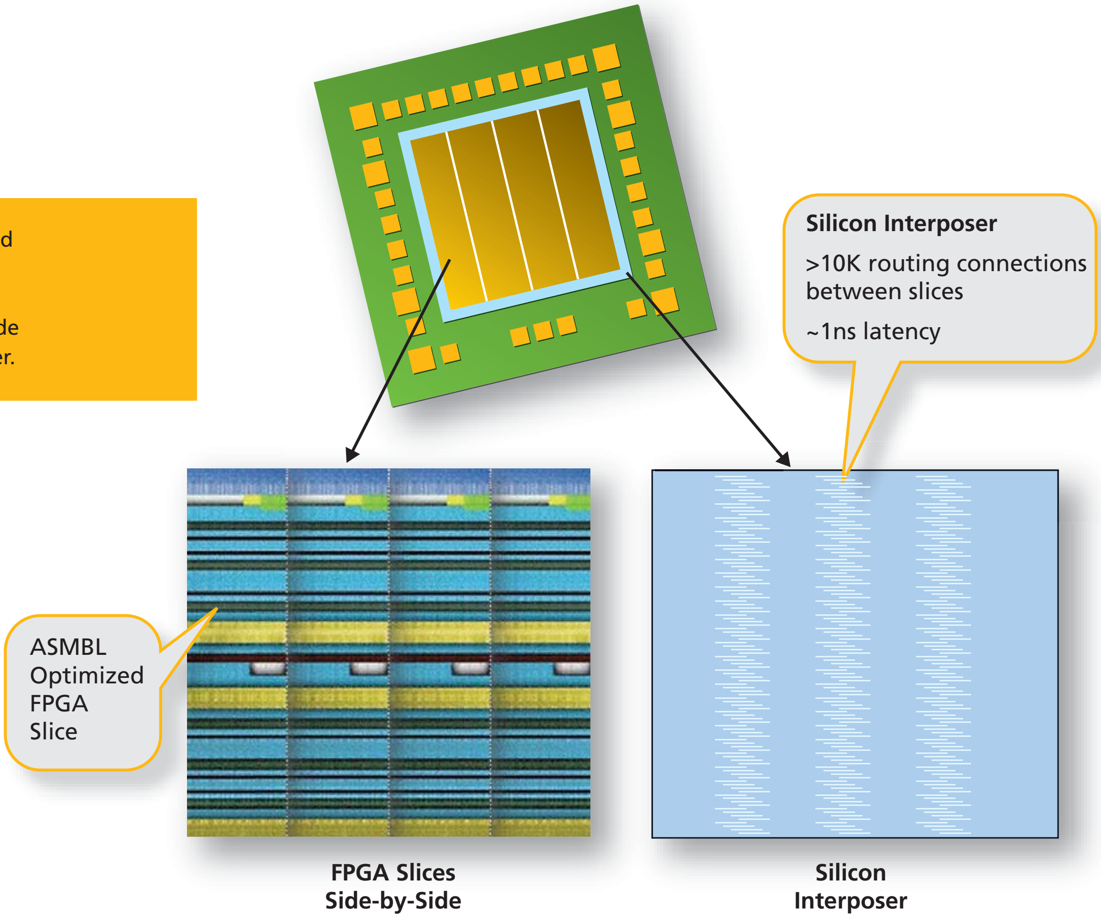


Figure 1 – The stacked silicon architecture places several dice (aka slices) side-by-side on a silicon interposer.

ing stacked-die architectures in hopes of reaping capacity, integration, PCB real-estate and even yield benefits. Most of these companies are looking to stacked-die technology to simply keep up with Moore's Law—Xilinx is leveraging it today as a way to exceed it and as a way to mix and match complementary types of dice on a single IC footprint to offer vast leaps forward in system performance, bill-of-materials (BOM) savings and power efficiency.

THE STACKED SILICON ARCHITECTURE

"This new stacked silicon interconnect technology allows Xilinx to essentially offer next-generation density in the current generation of process technology," said Liam Madden, corporate vice president of FPGA development and silicon technology at Xilinx. "As die size gets larger, the yield goes down exponentially, so building large dice is quite difficult and very costly. The new architecture allows us to build a number of smaller dice and then use a silicon interposer to connect those smaller dice lying side-by-side on top of the interposer so they appear to be, and function as, one integrated die" (Figure 1).

Each of the dice is interconnected via layers in the silicon interposer in much in the same way that discrete components are interconnected on the many layers of a printed-circuit board (Figure 2). The die and silicon interposer layer connect by means of multiple microbumps. The architecture also uses through-silicon vias (TSVs) that run through the passive silicon interposer to facilitate direct communication between regions of each die on the device and resources off-chip (Figure 3). Data flows between the adjacent FPGA die across more than 10,000 routing connections.

Madden said using a passive silicon interposer rather than going with a system-in-package or multichip-module configuration has huge advantages. "We use regular silicon interconnect or metallization to connect up the dice on the device," said Madden. "We can get many more connections within the silicon than you can with a system-in-package. But the biggest advantage of this approach is power savings. Because we are using chip interconnect to connect the dice, it is much more economical in power than connecting dice through big traces, through packages or through circuit boards."

In fact, the SSI technology provides more than 100 times the die-to-die connectivity bandwidth per watt, at one-fifth the latency, without consuming any high-speed serial or parallel I/O resources.

Madden also notes that the microbumps are not directly connected to the package. Rather, they are interconnected to the passive interposer, which in turn is linked to the adjacent die. This setup offers great advantages by shielding the microbumps from electrostatic discharge. By positioning dice next to each other and interfaced to the ball-grid array, the device avoids the thermal flux, signal integrity and design tool flow issues that would have accompanied a purely vertical die-stacking approach.

As with the monolithic 7 series devices, Xilinx implemented the SSI members of the Virtex-7 family in TSMC's 28-nm HPL (high-performance, low-power) process technology, which Xilinx and TSMC developed to create FPGAs with the right mix of power efficiency and performance (see cover story sidebar, *Xcell Journal*, Issue 72).

NO NEW TOOLS REQUIRED

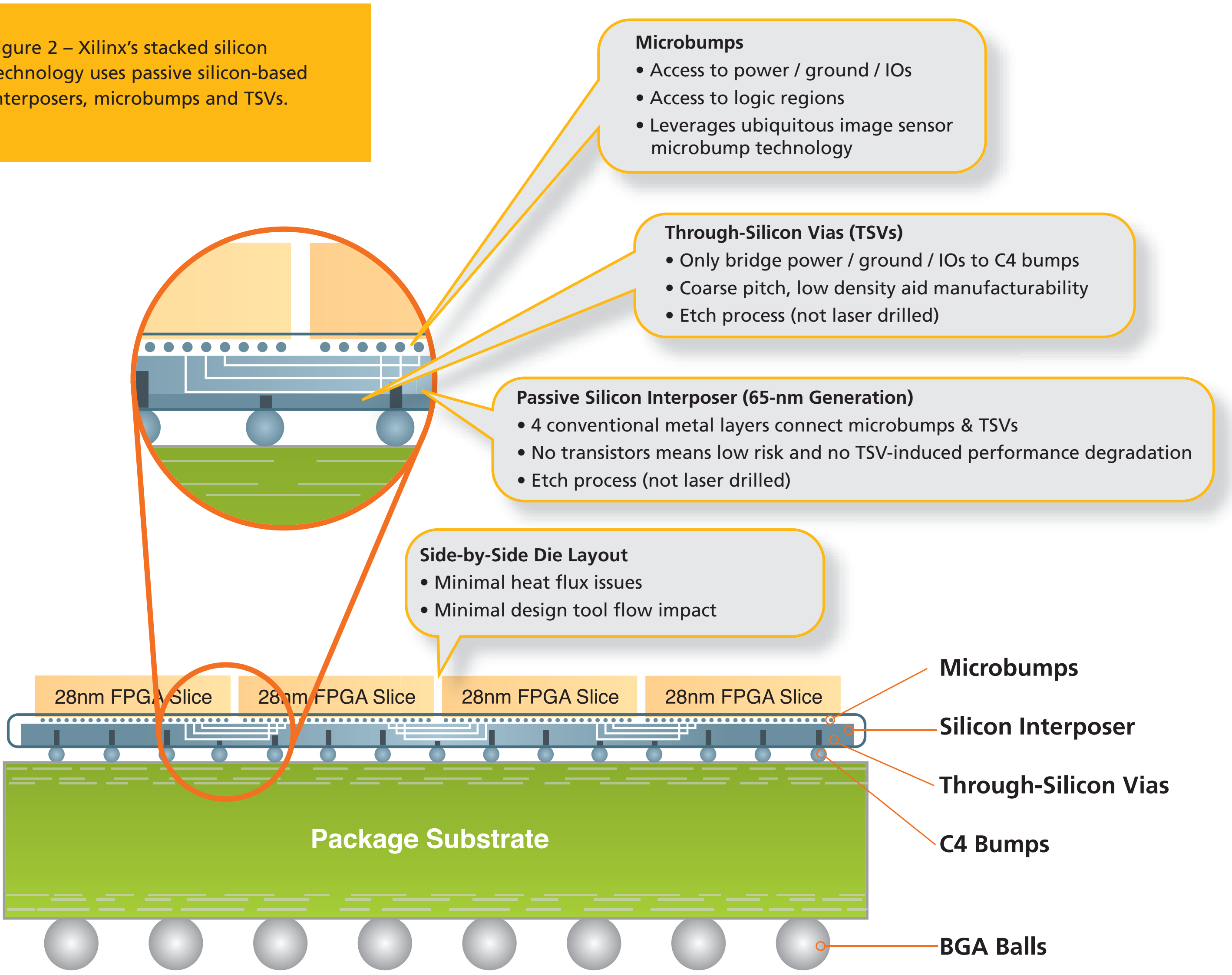
While the SSI technology offers some radical leaps forward in terms of capacity, Madden said it will not force a radical change in customer design methodologies. "One of the beautiful aspects of this architecture is that we were able to establish the edges of each slice [individual die in the device] along natural partitions where we would have traditionally run long wires had these structures been in our monolithic FPGA architecture," said Madden. "This meant that we didn't have to do anything radical in the tools to support the devices." As a result, "customers don't have to make any major adjustments to their design methods or flows," he said.

At the same time, Madden said that customers will benefit from adding floor-planning tools to their flows because they now have so many logic cells to use.

A SUPPLY CHAIN FIRST

While the design is in and of itself quite innovative, one of the biggest challenges of fielding such a device was in putting together the supply chain to manufacture, assemble, test and distribute it. To create the end product, each of the individual dice must first be tested extensively at the wafer level, binned and sorted, and

Figure 2 – Xilinx's stacked silicon technology uses passive silicon-based interposers, microbumps and TSVs.



Microbumps
- Access to power / ground / IOs
- Access to logic regions
- Leverages ubiquitous image sensor microbump technology

Through-Silicon Vias (TSVs)
- Only bridge power / ground / IOs to C4 bumps
- Coarse pitch, low density aid manufacturability
- Etch process (not laser drilled)

Passive Silicon Interposer (65-nm Generation)
- 4 conventional metal layers connect microbumps & TSVs
- No transistors means low risk and no TSV-induced performance degradation
- Etch process (not laser drilled)

Side-by-Side Die Layout
- Minimal heat flux issues
- Minimal design tool flow impact

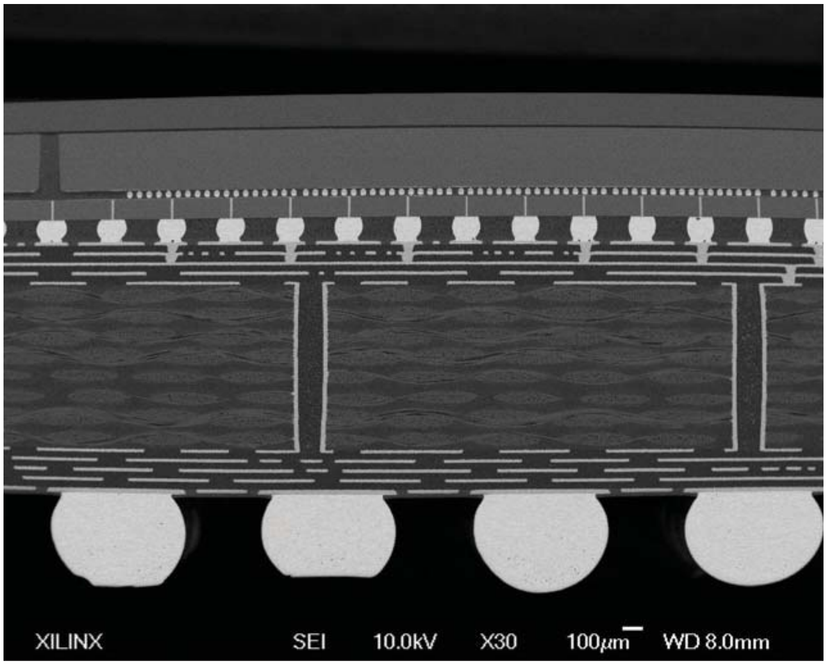


Figure 3 –Actual cross-section of the 28-nm Virtex-7 device. TSVs can be seen connecting the microbumps (dotted line, top) through the silicon interposer.

then attached to the interposer. The combined structure then needs to be packaged and given a final test to ensure connectivity before the end product ships to customers.

Madden's group worked with TSMC and other partners to build this supply chain. "This is another first in the industry, as no other company has put in place a supply chain like this across a foundry and OSAT [outsourced semiconductor assembly and test]," said Madden.

"Another beautiful aspect of this approach is that we can use essentially the same test approach that we use in our current devices," he went on."Our current test technology allows us to produce known-good dice, and that is a big advantage for us because in general, one of the biggest barriers of doing stacked-die technology is how do you test at the wafer level."

Because the stacked silicon technology integrates multiple Xilinx FPGA dice on a single IC, it logically follows that the architecture would also lend itself to mixing and matching FPGA and other dice to create entirely new devices. And that's exactly what Xilinx did with its ultrafast Virtex-7 HT line, announced just weeks after the SSI technology rollout.

DRIVING COMMUNICATIONS TO 400 GBPS

The new Virtex-7 HT line of devices is targeted squarely at communications companies that are developing 100- to 400-Gbps equipment. The Virtex-7 HT combines on a single IC multiple 28-nm FPGA dice, bearing dozens of 13.1-Gbps transceivers, with ASIC dice that have 28-Gbps transceivers. The result is to endow the final device with a formidable mix of logic cells as well as cutting-edge transceiver performance and reliability.

The largest of the Virex-7 HT line includes sixteen GTZ 28-Gbps transceivers, seventy-two 13.1-Gbps transceivers plus logic and memory, offering transceiver performance and capacity far greater than competing devices (see Video 1, *http://www.youtube.com/user/XilinxInc#p/c/71A9E924ED61B8F9/1/eTHjt67ViK0*).

Video 1 – Dr. Howard Johnson introduces the 28-Gbps transceiver-laden Virtex-7 HT. *http://www.youtube.com/user/XilinxInc#p/c/71A9E924ED61B8F9/1/eTHjt67ViK0.*

Dr. Howard Johnson, Signal Integrity Expert
Author of "High Speed Digital Design: A Handbook of Black Magic"
XILINX.

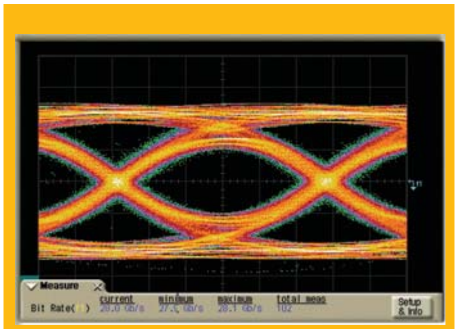


Figure 4a –Xilinx 28-Gbps transceiver displays an excellent eye opening and jitter performance (using PRBS31 data pattern).

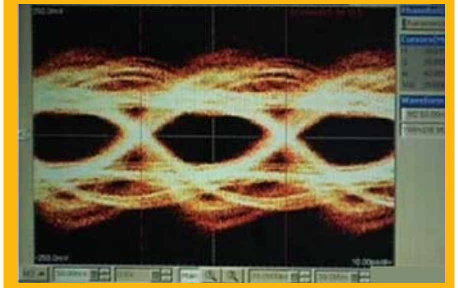


Figure 4b – This is a competing device's 28-Gbps signal using a much simpler PRBS7 pattern. The signal is extremely noisy with a significantly smaller eye opening. Eye size is shown close to relative scale.

"We leveraged stacked interconnect technology to offer Virtex-7 devices with a 28G capability," said Madden. "By offering the transceivers on a separate ASIC die, we can optimize our 28-Gbps transceiver performance and electrically isolate functions to offer an even higher degree of reliability for applications requiring cutting-edge transceiver performance and reliability."

With the need for bandwidth exploding, the communications sector is franticly racing to establish new networks. The wireless industry is scrambling to produce equipment supporting 40-Gbps data transfer today, while wired networking is approaching 100 Gbps. FPGAs have played a key role in just about every generation of networking equipment since their inception (see cover stories in *Xcell Journal*, Issues 65 and 67).

Communications equipment design teams traditionally have used FPGAs to receive signals sent to equipment in multiple protocols, translate those signals to common protocols that the equipment and network use, and then forward the data to the next destination. Traditionally companies have placed a processor in between FPGAs monitoring and translating incoming signals and those FPGAs forwarding signals to their destination. But as FPGAs advance and grow in capacity and functionality, a single FPGA can both send and receive, while also performing processing, to add greater intelligence and monitoring to the system. This lowers the BOM and, more important, reduces the power and cooling costs of networking equipment, which must run reliably 24 hours a day, seven days a week.

In a white paper titled "Industry's Highest-Bandwidth FPGA Enables World's First Single-FPGA Solution for 400G Communications Line Cards," Xilinx's Greg Lara outlines several communications equipment applications that can benefit from the Virtex-7 HT devices (see *http://www.xilinx.com/support/documentation/white_papers/wp385_V7_28G_for_400G_Comm_Line_Cards.pdf*).

To name a few, Virtex-7 HT FPGAs can find a home in 100-Gbps line cards supporting OTU-4 (Optical Transfer Unit) transponders. They can be used as well in muxponders or service aggregation routers, in lower-cost 120-Gbps packet-processing line cards for highly demanding data processing, in multiple 100G Ethernet ports and bridges, and in 400-Gbps Ethernet line cards. Other potential applications include base stations and remote radio heads with 19.6-Gbps Common Public Radio Interface requirements, and 100-Gbps and 400-Gbps test equipment.

JITTER AND EYE DIAGRAM

A key to playing in these markets is ensuring the FPGA transceiver signals are robust, reliable and resistant to jitter or interference and to fluctuations caused by power system noise. For example, the CEI-28G specification calls for 28-Gbps networking equipment to have extremely tight jitter budgets.

Signal integrity is an extremely crucial factor for 28-Gbps operation, said Panch Chandrasekaran, senior marketing manager of FPGA components at Xilinx. To meet the stringent CEI-28G jitter budgets, the transceivers in the new Xilinx FPGAs employ phase-locked loops (PLLs) based on an LC tank design and advanced equalization circuits to offset deterministic jitter.

"Noise isolation becomes a very important parameter at 28-Gbps signaling speeds," said Chandrasekaran. "Because the FPGA fabric and transceivers are on separate dice, the sensitive 28-Gbps analog circuitry is isolated from the digital FPGA circuits, providing superior isolation compared to monolithic implementations" (Figures 4a and 4b).

The FPGA design also includes features that minimize lane-to-lane skew, allowing the devices to support stringent optical standards such as the Scalable Serdes Framer Interface standard (SFI-S).

Further, the GTZ transceiver design eliminates the need for designers to employ external reference resistors, lowering the BOM costs and simplifing the board design. A built-in "eye scan" function automatically measures the height and width of the post-equalization data eye. Engineers can use this diagnostic tool to perform jitter budget analysis on an active channel and optimize transceiver parameters to get optimal signal integrity, all without the expense of specialized equipment.

ISE® Design Suite software tool support for 7 series FPGAs is available today. Virtex-7 FPGAs with massive logic capacity, thanks to the SSI technology, will be available this year. Samples of the first Virtex-7 HT devices are scheduled to be available in the first half of 2012. For more information on Virtex-7 FPGAs and SSI technology, visit *http://www.xilinx.com/technology/roadmap/7-series-fpgas.htm*.

MicroBlaze Hosts Mobile Multisensor Navigation System

Researchers used Xilinx's soft-core processor to develop an integrated navigation solution that works in places where GPS doesn't.

by Walid Farid Abdelfatah
Research Assistant
Navigation and Instrumentation
Research Group
Queen's University
w.abdelfatah@queensu.ca

Jacques Georgy
Algorithm Design Engineer
Trusted Positioning Inc.

Aboelmagd Noureldin
Cross-Appointment
Associate Professor in ECE Dept.
Queen's University /
Royal Military College of Canada

The global positioning system (GPS) is a satellite-based navigation system that is widely used for different navigation applications. In an open sky, GPS can provide an accurate navigation solution. However, in urban canyons, tunnels and indoor environments that block the satellite signals, GPS fails to provide continuous and reliable coverage.

In the search for a more accurate and low-cost positioning solution in GPS-denied areas, researchers are developing integrated navigation algorithms that utilize measurements from low-cost sensors such as accelerometers, gyroscopes, speedometers, barometers and others, and fuses them with the measurements from the GPS receiver. The fusion is accomplished using either Kalman filter (KF), particle filter or artificial-intelligence techniques.

Once a navigation algorithm is developed, verified and proven to be worthy, the ultimate goal is to put it on a low-cost, real-time embedded system. Such a system must acquire and synchronize the measurements from the different sensors, then apply the navigation algorithm, which will integrate these aligned measurements, yielding a real-time solution at a defined rate.

The transition from algorithm research to realization is a crucial step in assessing the practicality and effectiveness of a navigation algorithm and, consequently, the developed embedded system, either for a proof of concept or to be accepted as a consumer product. In the transition process, there is no unique methodology that designers can follow to create an embedded system. Depending on the platform of choice—such as microcontrollers, digital signal processors and field-programmable gate arrays—system designers use different methodologies to develop the final product.

MICROBLAZE FRONT AND CENTER

In research at Queen's University in Kingston, Ontario, we utilized a Xilinx® MicroBlaze® soft-core processor built on the ML402 evaluation kit, which features a Virtex®-4 SX35 FPGA. The applied navigation algorithm is a 2D GPS/reduced inertial sensor system (RISS) integration algorithm that incorporates measurements from a gyroscope and a vehicle's odometer or a robot's wheel encoders, along with those from a GPS receiver. In this way, the algorithm computes an integrated navigation solution that is more accurate than standalone GPS. For a vehicle moving in a 2D plane, it computes five navigation states consisting of two position parameters (latitude and longitude), two velocity parameters (East and North) and one orientation parameter (azimuth).

The 2D RISS/GPS integration algorithm involves two steps: RISS mechanization and RISS/GPS data fusion using Kalman filtering. RISS mechanization is the process of transforming the measurements of an RISS system acquired in the vehicle frame into position, velocity and heading in a generic frame. It is a recursive process based on the initial conditions, or the previous output and the new measurements. We fuse the posi-

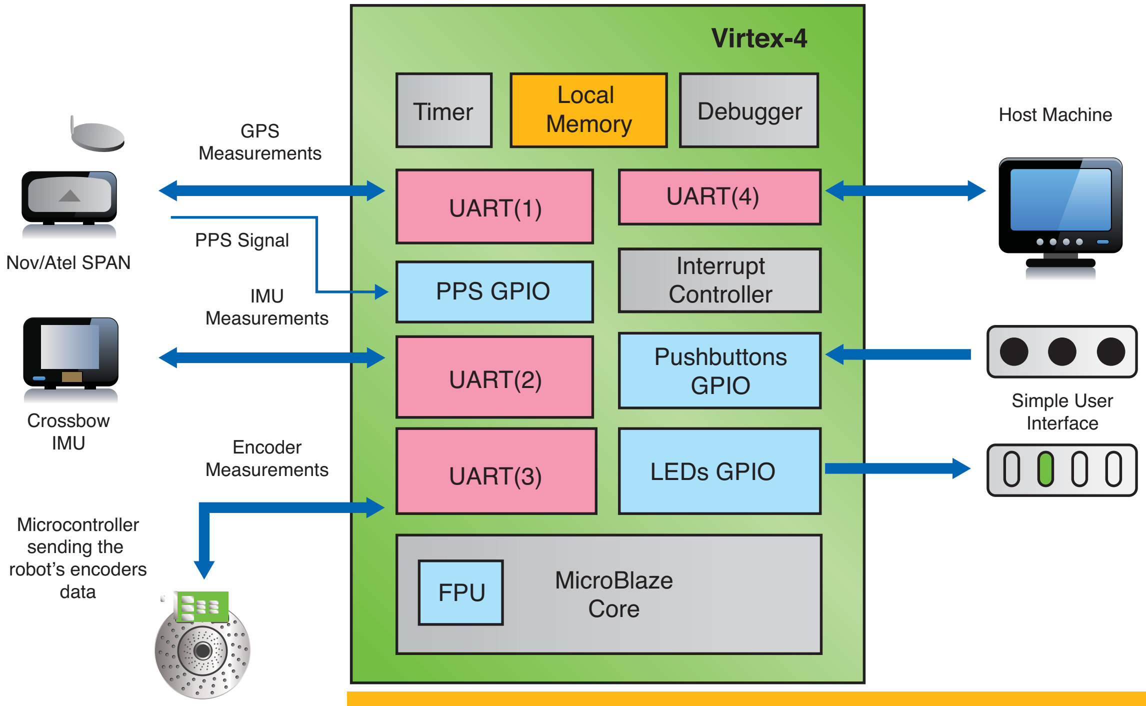


Figure 1 – MicroBlaze connections with the mobile robot sensors and the host machine

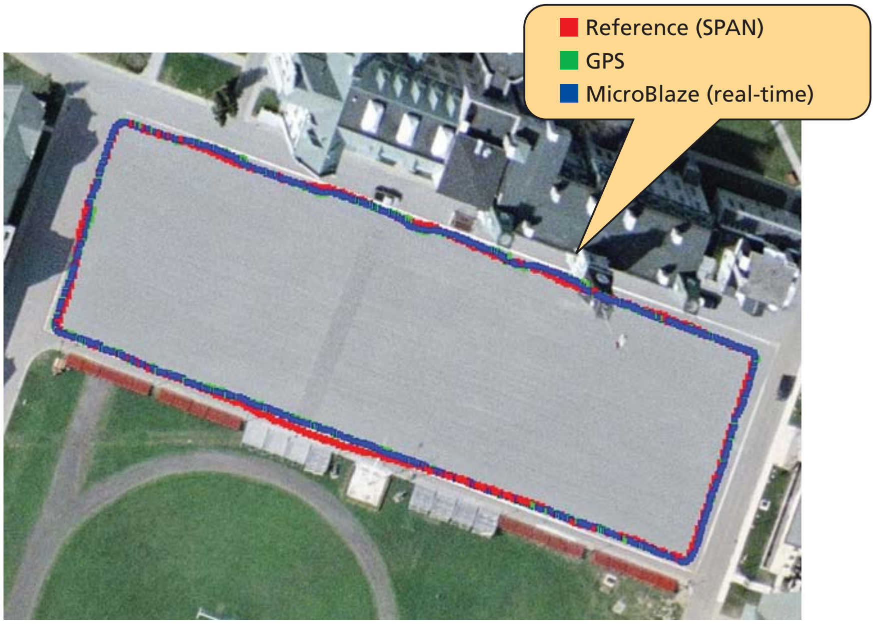


Figure 2 – Mobile robot trajectory, open-sky area

tion and velocity measurements from the GPS with the RISS-computed position and velocity using the conventional KF method in a closed-loop, loosely coupled fashion. [1, 2]

A soft-core processor such as MicroBlaze provides many advantages over off-the-shelf processors in developing a mobile multisensor navigation system. [3] They include customization, the ability to design a multiprocessor system and hardware acceleration.

With a soft-core device, designers of the FPGA-based embedded processor system have the total flexibility to add any custom combination of peripherals and controllers. You can also design a unique set of peripherals for specific applications, adding as many peripherals as needed to meet the system requirements.

Moreover, more-complex embedded systems can benefit from the existence of multiple processors that can execute the tasks in parallel. By using a soft-core processor and accompanying tools, it's an easy matter to create a multiprocessor-based system-on-chip. The only restriction to the number of processors that you can add is the availability of FPGA resources.

Finally, one of the most compelling reasons for choosing a soft-core processor is the ability to concurrently develop hardware and software, and have them coexist on a single chip. System designers need not worry if a segment of the algorithm is found to be a bottleneck; all you have to do to eliminate such a problem is design a custom coprocessor or a hardware circuit that utilizes the FPGA's parallelism.

INTERFACES AND ARITHMETIC

For the developed navigation system at Queen's University, we mainly needed a system capable of interfacing with three sensors and a personal computer via a total of four serial ports, each of which requires a universal asynchronous receiver transmitter (UART) component. We use the serial channel between the embedded system and a PC to upload the sensor's measurements and the navigation solution for analysis and debugging. In addition, the design also requires general-purpose input/output (GPIO) channels for user interfacing, a timer for code profiling and an interrupt controller for the event-based system.

We also needed a floating-point unit (FPU), which can execute the navigation algorithm once the measurements are available in the least time possible. Although the FPU used in the MicroBlaze core is single-precision, we developed our algorithm making use of double-precision arithmetic. Consequently, we used software libraries to emulate our double-precision operations on the single-precision FPU, resulting in a slightly denser code but a far better navigation solution.

We first tested the developed navigation system on a mobile robot to reveal system bugs and integration problems, and then on a standard automobile for further testing. We chose to start the process with iterative testing on the mobile robot due to the flexibility and ease of system debugging compared with a vehicle. We compared the solution with an off-line navigation solution computed on a PC and with the results from a high-end, high-cost, real-time NovAtel SPAN system. The SPAN system combines a GPS receiver and high-end tactical-grade inertial measurement unit (IMU), which acts as a reference. Figure 1 shows the MicroBlaze processor connections with the RISS/GPS system sensors on a mobile robot.

Figure 2 presents an open-sky trajectory which we acquired at the Royal Military College of Canada for 9.6 minutes; the solution of the developed real-time system is compared to the GPS navigation solution and the SPAN reference system. The difference between the NovAtel SPAN reference solution and our solution is shown in Figure 3. Both solutions are real-time, and the difference in performance is due to our use of the low-cost MEMS-based Crossbow IMU300CC-100 IMU in the developed system. The SPAN unit, by contrast, uses the high-cost, tactical-grade HG1700 IMU.

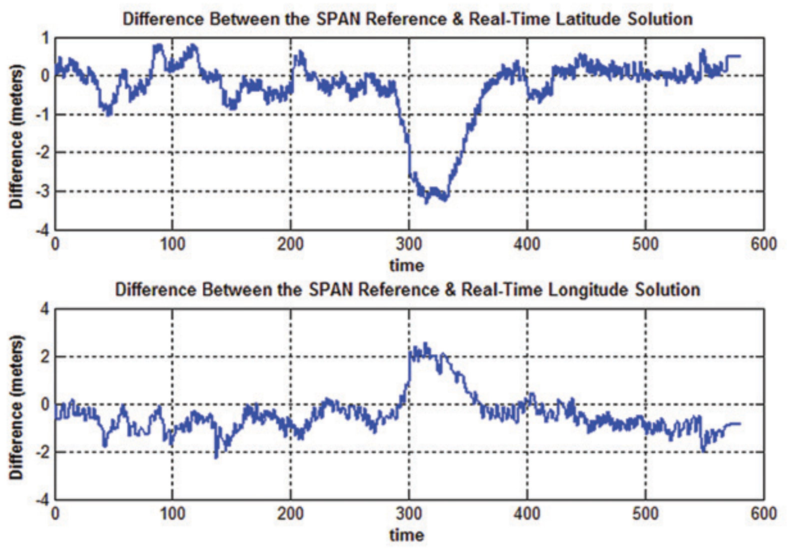


Figure 3 – Mobile robot trajectory: Difference between NovAtel SPAN reference and MicroBlaze real-time solutions

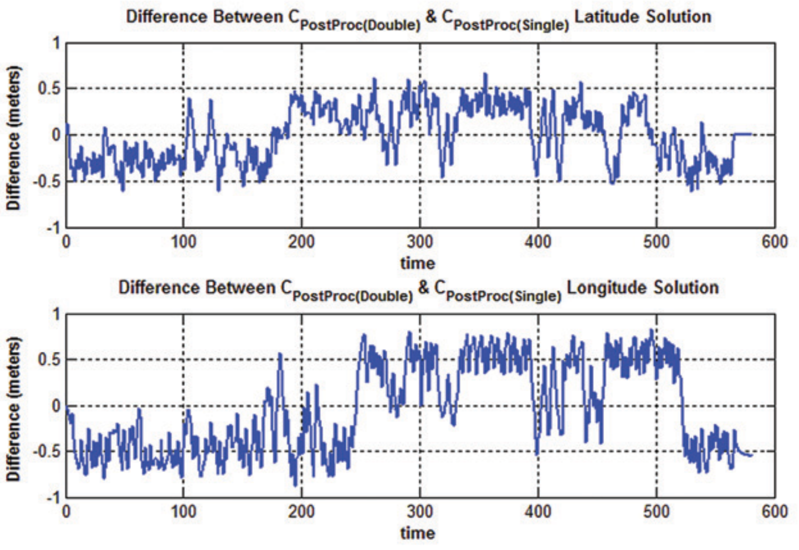


Figure 4 – Mobile robot trajectory: Difference between C double-precision and single-precision off-line solutions

Figure 4 presents the difference between the positioning solutions from the C double-precision off-line algorithm and the C single-precision off-line algorithm computations, where the difference is in the range of half a meter. The results show that the choice of emulating the double-precision operations on the MicroBlaze core is far more accurate than using single-precision arithmetic, which would have consumed less memory and less processing time.

MULTIPROCESSING AND HARDWARE ACCELERATION

Utilizing a soft-core processor such as the MicroBlaze in a mobile navigation system composed of various sensors provides the flexibility to communicate with any number of sensors that use various interfaces, without any concern with the availability of the peripherals beforehand, as would be true if we had used off-the-shelf processors. In terms of computation, designers can build the navigation algorithm using embedded software that utilizes the single-precision FPU available within the MicroBlaze, as is the case in this research. You can also partition a design between software and hardware by implementing the segments of the algorithm that are found to be a bottleneck as a custom coprocessor or a hardware circuit that speeds up the algorithm.

The future of this research can progress in two directions. The first is to use a double-precision floating-point unit directly with the MicroBlaze processor instead of double-precision emulation. The second is related to implementing more powerful and challenging navigation algorithms that utilize more sensors to be integrated via particle filtering or artificial-intelligence techniques, which can make use of the power of multiprocessing and hardware acceleration.

More information about the embedded navigation system design process and more results can be found in the master's thesis from which this work was derived. [4]

REFERENCES

[1] U. Iqbal, A. F. Okou and A. Noureldin, "An integrated reduced inertial sensor system — RISS / GPS for land vehicle," Position, Location and Navigation Symposium proceedings, 2008 IEEE/ION.

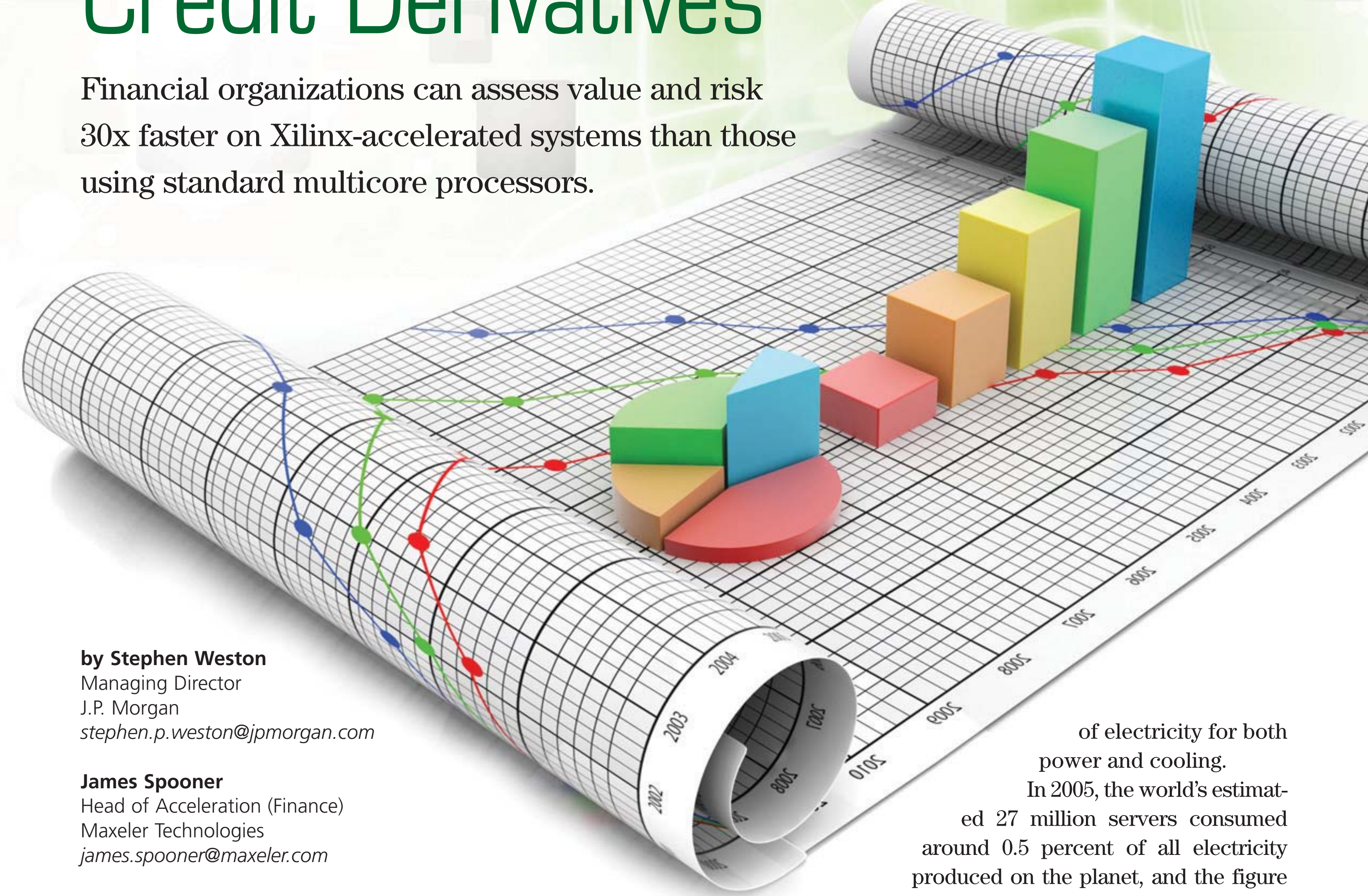
[2] U. Iqbal and A. Noureldin, Integrated Reduced Inertial Sensor System/GPS for Vehicle Navigation: Multisensor positioning system for land applications involving single-axis gyroscope augmented with a vehicle odometer and integrated with GPS, *VDM Verlag Dr. Müller, 2010.*

[3] B. H. Fletcher, "FPGA Embedded Processors—Revealing True System Performance," Embedded Systems Conference, 2005.

[4] Walid Farid Abdelfatah, "Real-Time Embedded System Design and Realization for Integrated Navigation Systems" 2010. http://hdl.handle.net/1974/6128

FPGAs Speed the Computation of Complex Credit Derivatives

Financial organizations can assess value and risk 30x faster on Xilinx-accelerated systems than those using standard multicore processors.



by Stephen Weston
Managing Director
J.P. Morgan
stephen.p.weston@jpmorgan.com

James Spooner
Head of Acceleration (Finance)
Maxeler Technologies
james.spooner@maxeler.com

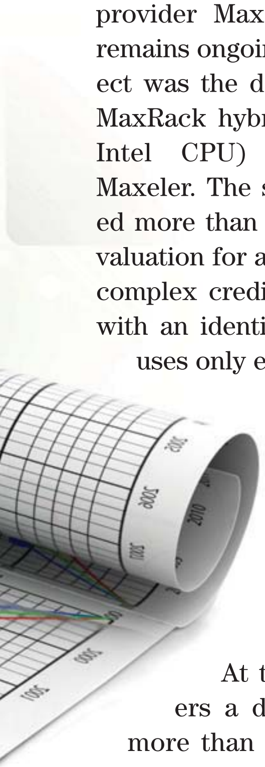
Jean-Tristan Marin
Vice President
J.P. Morgan
jean-tristan.marin@jpmorgan.com

Oliver Pell
Vice President of Engineering
Maxeler Technologies
oliver@maxeler.com

Oskar Mencer
CEO
Maxeler Technologies
mencer@maxeler.com

Innovation in the global credit derivatives markets rests on the development and intensive use of complex mathematical models. As portfolios of complicated credit instruments have expanded, the process of valuing them and managing the risk has grown to a point where financial organizations use thousands of CPU cores to calculate value and risk daily. This task in turn requires vast amounts of electricity for both power and cooling.

In 2005, the world's estimated 27 million servers consumed around 0.5 percent of all electricity produced on the planet, and the figure edges closer to 1 percent when the energy for associated cooling and auxiliary equipment (for example, backup power, power conditioning, power distribution, air handling, lighting and chillers) is included. Any savings from falling hardware costs are increasingly offset by rapidly rising power-related indirect costs. No wonder, then, that large financial institutions are searching for a way to add ever-greater computational power at a much lower operating cost.



Toward this end, late in 2008 the Applied Analytics group at J.P. Morgan in London launched a collaborative project with the acceleration-solutions provider Maxeler Technologies that remains ongoing. The core of the project was the design and delivery of a MaxRack hybrid (Xilinx® FPGA and Intel CPU) cluster solution by Maxeler. The system has demonstrated more than 31x acceleration of the valuation for a substantial portfolio of complex credit derivatives compared with an identically sized system that uses only eight-core Intel CPUs.

The project reduces operational expenditures more than thirtyfold by building a customized high-efficiency high-performance computing (HPC) system. At the same time, it delivers a disk-to-disk speedup of more than an order of magnitude over comparable systems, enabling the computation of an order-of-magnitude more scenarios, with direct impact on the credit derivatives business at J.P. Morgan. The Maxeler system clearly demonstrates the feasibility of using FPGA technology to significantly accelerate computations for the finance industry and, in particular, complex credit derivatives.

RELATED WORK

There is a considerable body of published work in the area of acceleration for applications in finance. A shared theme is adapting technology to accelerate the performance of computationally demanding valuation models. Our work shares this thrust and is distinguished in two ways.

The first feature is the computational method we used to calculate fair value. A common method is a single-factor Gaussian copula, a mathematical model used in finance, to model the underlying credits, along with Monte Carlo simulation to evaluate the expected value of the portfolio. Our model employs standard base correlation methodology, with a Gaussian copula for default correlation and a stochastic recovery process. We compute the expected (fair) value of a portfolio using a convolution approach. The combination of these methods presents a distinct computational, algorithmic and performance challenge.

In addition, our work encompasses the full population of live complex credit trades for a major investment bank, as opposed to relying on the usual approach of using synthetic data sets. To fully grasp the computing challenges, it's helpful to take a closer look at the financial products known as credit derivatives.

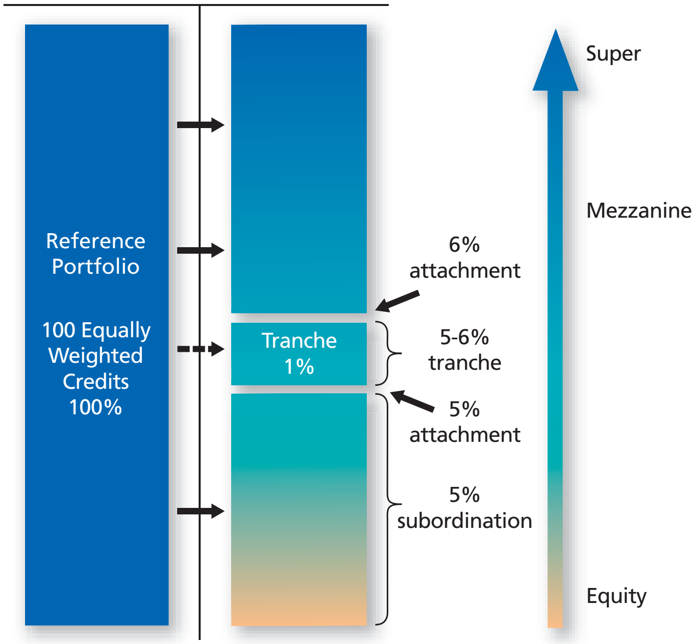
WHAT ARE CREDIT DERIVATIVES?

Companies and governments issue bonds which provide the ability to fund themselves for a predefined period, rather like taking a bank loan for a fixed term. During their life, bonds typically pay a return to the holder, which is referred to as a "coupon," and at maturity return the borrowed amount. The holder of the bond, therefore, faces a number of risks, the most obvious being that the value of the bond may fall or rise in response to changes in the financial markets. However, the risk we are concerned with here is that when the bond reaches maturity, the issuer may default and fail to pay the coupon, the full redemption value or both.

Credit derivatives began life by offering bond holders a way of insuring against loss on default. They have expanded to the point where they now offer considerably greater functionality across a range of assets beyond simple bonds, to both issuers and holders, by enabling the transfer of default risk in exchange for regular payments. The simplest example of such a contract is a credit default swap (CDS), where the default risk of a single issuer is exchanged for regular payments. The key difference between a bond and a CDS is that the CDS only involves payment of a redemption amount (minus any recovered amount) should default occur.

Building on CDS contracts, credit default swap indexes (CDSIs) allow the trading of risk using a portfolio of underlying assets. A collateralized default obligation (CDO) is an extension of a CDSI in which default losses

Figure 1 – Tranched credit exposure



Source: J.P. Morgan

The modeling of behavior for a pool of credits becomes complex, as it is important to model the likelihood of default among issuers. Corporate defaults, for example, tend to be significantly correlated—the failure of a single company tends to weaken the remaining firms.

on the portfolio of assets are divided into different layers or "tranches," according to the risk of default. A tranche allows an investor to buy or sell protection for losses in a certain part of the pool.

Figure 1 shows a tranche between 5 percent and 6 percent of the losses in a pool. If an investor sells protection on this tranche, he or she will not be exposed to any losses until 5 percent of the total pool has been wiped out. The lower (less senior) tranches have higher risk and those selling protection on these tranches will receive higher compensation than those investing in upper (more senior, hence less likely to default) tranches.

CDOs can also be defined on nonstandard portfolios. This variety of CDO, referred to as "bespoke," presents additional computational challenges, since the behavior of a given bespoke portfolio is not directly observable in the market.

The modeling of behavior for a pool of credits becomes complex, as it is important to model the likelihood of default among issuers. Corporate defaults, for example, tend to be significantly correlated because all firms are exposed to a common or correlated set of economic risk factors, so that the failure of a single company tends to weaken the remaining firms.

The market for credit derivatives has grown rapidly over the past 10 years, reaching a peak of $62 trillion in January 2008, in response to a wide range of influences on both demand and supply. CDSes account for approximately 57 percent of the notional outstanding, with the remainder accounted for by products with multiple underlying credits. "Notional" refers to the amount of money underlying a credit derivative contract; "notional outstanding" is the total amount of notional in current contracts in the market.

```
for  i in 0 ... markets - 1
   for j in 0 ... names - 1
      prob = cum_norm ((inv_norm (Q[j]) - sqrt(rho)*M[i])/sqrt (1 - rho);
      loss= calc_loss (prob,Q2[j],RR[j],RM[j])*notional[j];
      n = integer(loss);
      L = fractional(loss);
      for k in 0 ... bins - 1
         if j == 0
            dist[k] = k == 0 ? 1.0 : 0.0;

         dist[k] = dist[k]*(1- prob) +
                   dist[k - n]* prob * (1 - L) +
                   dist[k - n - 1]* prob*L;
         if j == credits - 1
            final_dist [k] += weight[ i ] * dist[k];

      end # for k
   end # for j
end # for i
```

Figure 2 – Pseudo code for the bespoke CDO tranche pricing

A CREDIT DERIVATIVES MODEL

The standard way of pricing CDO tranches where the underlying is a bespoke portfolio of reference credits is to use the "base correlation" approach, coupled with a convolution algorithm that sums conditionally independent loss-random variables, with the addition of a "coupon model" to price exotic coupon features. Since mid-2007 the standard approach in the credit derivatives markets has moved away from the original Gaussian copula model. Two alternative approaches to valuation now dominate: the random-factor loading (RFL) model and the stochastic recovery model.

Our project has adopted the latter approach and uses the following methodology as a basis for accelerating pricing: In the first step, we discretize and compute the loss distribution using convolution, given the conditional survival probabilities and losses from the copula model. Discretization is a method of chunking a continious space into a discrete space and separating the space into discrete "bins," allowing algorithms (like integration) to be done numerically. We then use the standard method of discretizing over the two closest bins with a weighting such that the expected loss is conserved. We compute the final loss distribution using a weighted sum over all of the market factors evaluated, using the copula model.

Pseudo code for this algorithm is shown in Figure 2. For brevity, we've removed the edge cases of the convolution and the detail of the copula and recovery model.

ACCELERATED CDO PRICING

Each day the credit hybrids business within J.P. Morgan needs to evaluate hundreds of thousands of credit derivative instruments. A large proportion of these are standard, single-name CDSes that need very little computational time. However, a substantial minority of the instruments are tranched credit derivatives that require the use of complex models like the one discussed above. These daily runs are so computationally intensive that, without the application of Maxeler's acceleration techniques, they could only be meaningfully carried out overnight. To complete the task, we used approximately 2,000 standard Intel cores. Even with such resources available, the calculation time is around four and a half hours and the total end-to-end runtime is close to seven hours, when batch preparation and results writeback are taken into consideration.

With this information, the acceleration stage of the project focused on designing a solution capable of dealing with only the complex bespoke tranche products, with a specific goal of exploring the HPC architecture design space in order to maximize the acceleration.

MAXELER'S ACCELERATION PROCESS

The key to maximizing the speed of the final design is a systematic and disciplined end-to-end acceleration methodology. Maxeler follows the four stages of acceleration shown in Figure 3, from the initial C++ design to the final implementation, which ensures we arrive at the optimal solution.

In the analysis stage, we conducted a detailed examination of the algorithms contained in the original C++ model code. Through extensive code and data profiling with the Maxeler Parton profiling tool suite, we were able to clearly understand and map the relationships between the computation and the input and output data. Part of this analysis involved acquiring a full understanding of how the core algorithm performs in practice, which allowed us to identify the major computational and data movements, as well as storage costs. Dynamic analysis using call graphs of the running software, combined with detailed analysis of data values and

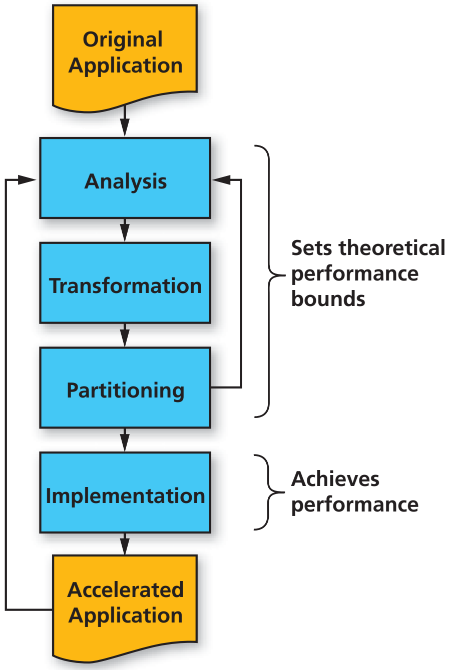


Figure 3 – Iterative Maxeler process for accelerating software

```
// Shared library and call overhead  1.2 %
for d in 0 ... dates -1
   // Curve Interpolation 19.7%
   for j in 0 ... names -1
         Q[j] = Interpolate(d, curve)
   end # for j
   for  i  in 0 ... markets -1
      for j in 0 ...  names  -1
       // Copula Section  22.0%
       prob =cum_norm  ((inv_norm (Q[j]) -sqrt(rho)*M[i])/sqrt (1 -rho);
       loss= calc_loss (prob,Q2[j],RR[j],RM[j])*notional[j];
       n = integer(loss);
       lower = fractional(loss);
       for k in 0 ... bins -1
          if j == 0
             dist[k] = k == 0 ? 1.0 : 0.0 ;
              // Convolution 51.4%
             dist[k] = dist[k]*(1-prob) +
                            dist[k-n]* prob *(1 -L) +
                            dist[k-n-1]* prob *L;
            // Reintegration, 5.1%
           if j == credits -1
              final_dist [k] += weight[i] * dist[k];
         end # for k
      end # for j
   end # for i
end # for d
// Code outside main loop 0.5%
```

Figure 4 – Profiled version of original pricing algorithm in pseudo-code form

runtime performance, were necessary steps to identifying bottlenecks in execution as well as memory utilization patterns.

Profiling the core 898 lines of original source code (see Figure 4) focused attention on the need to accelerate two main areas of the computation: calculation of the conditional-survival probabilities (Copula evaluation) and calculation of the probability distribution (convolution).

The next stage, transformation, extracted and modified loops and program control structure from the existing C++ code, leading to code and data layout transformations that in turn enabled the acceleration of the core algorithm. We removed data storage abstraction using object orientation, allowing data to pass efficiently to the accelerator, with low CPU overhead. Critical transformations included loop unrolling, reordering, tiling and operating on vectors of data rather than single objects.

PARTITIONING AND IMPLEMENTATION

The aim for the partitioning stage of the acceleration process was to create a contiguous block of operations. This block needed to be tractable to accelerate, and had to achieve the maximum possible runtime coverage, when balanced with the overall CPU performance and data input and output considerations.

The profiling process we identified during the analysis stage provided the necessary insight to make partitioning decisions.

The implementation of the partitioned design led to migrating the loops containing the Copula evaluation and convolution onto the FPGA accelerator. The remaining loops and associated computation stayed on the CPU. Within the FPGA design, we split the Copula and convoluter into separate code implementations that could execute and be parallelized independently of each other and could be sized according to the bounds of the loops described in the pseudo code.

In the past, some design teams have shied away from FPGA-based acceleration because of the complexity of the programming task. Maxeler's programming environment, called MaxCompiler, raises the level of abstraction of FPGA design to enable rapid development and modification of streaming applications, even when faced with frequent design updates and fixes.

Some design teams have shied away from FPGA-based acceleration because of the complexity of programming. MaxCompiler lets you implement the FPGA design in Java, without resorting to lower-level languages such as VHDL.

IMPLEMENTATION DETAILS

MaxCompiler builds FPGA designs in a simple, modular fashion. A design has one or more "kernels," which are highly parallel pipelined blocks for executing a specific computation. The "manager" dynamically oversees the dataflow I/O between these kernels. Separating computation and communication into kernels and manager enables a high degree of pipeline parallelism within the kernels. This parallelism is pivotal to achieving the performance our application enjoys. We achieved a second level of parallelism by replicating the compute pipeline many times within the kernel itself, further multiplying speedup.

The number of pipelines that can be mapped to the accelerator is limited only by the size of the FPGAs used in the MaxNode and available parallelization in the application. MaxCompiler lets you implement all of the FPGA design efficiently in Java, without resorting to lower-level languages such as VHDL.

For our accelerator, we used the J.P. Morgan 10-node MaxRack configured with MaxNode-1821 compute nodes. Figure 5 sketches the system architecture of a MaxNode-1821. Each node has eight Intel Xeon cores and two Xilinx FPGAs connected to the CPU via PCI Express®. A MaxRing high-speed interconnect is also available, providing a dedicated high-band-

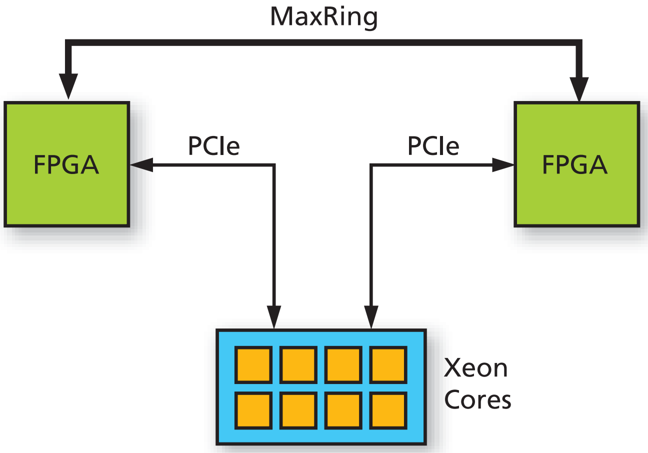


Figure 5 – MaxNode-1821 architecture diagram containing eight Intel Xeon cores and two Xilinx FPGAs

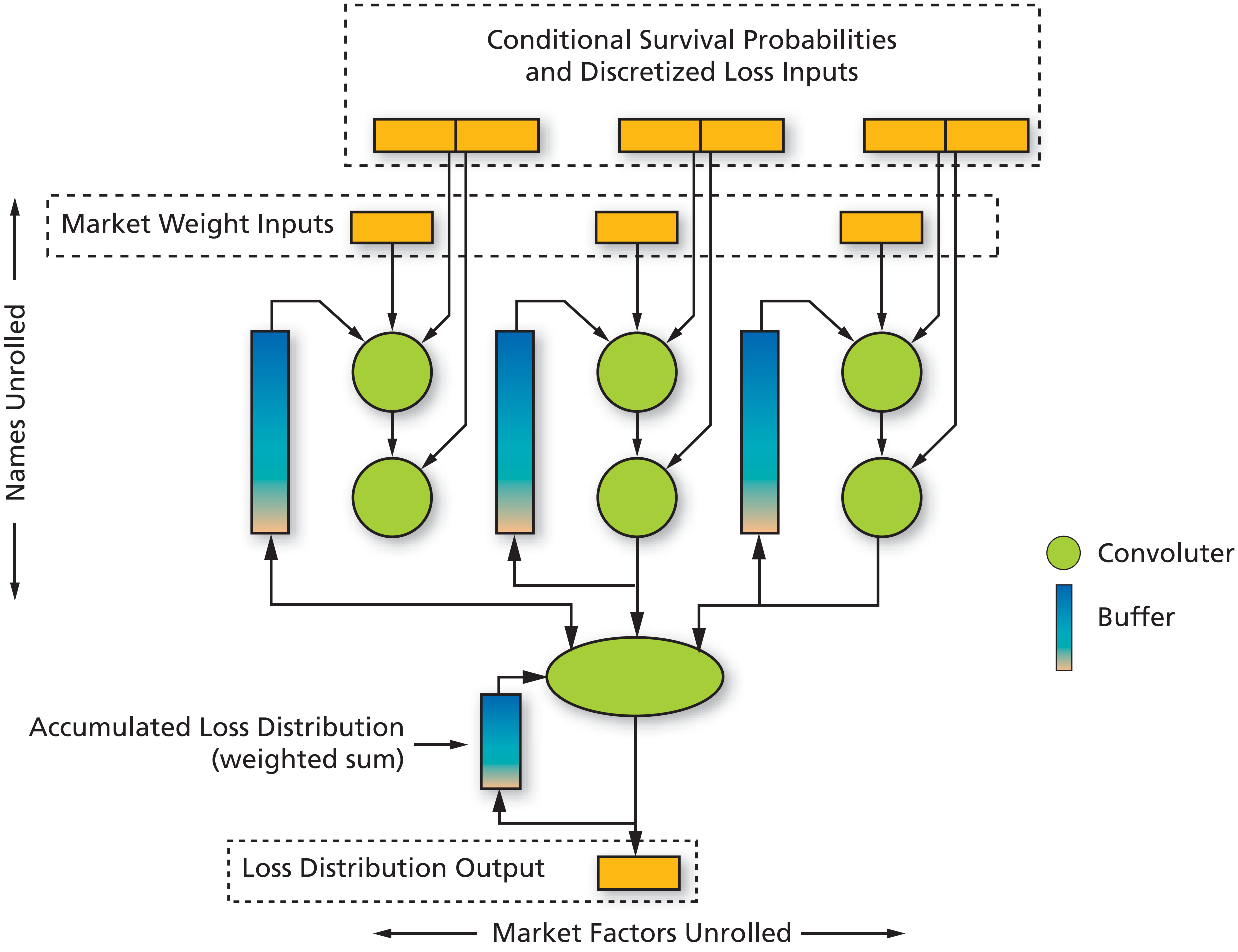

Conditional Survival Probabilities and Discretized Loss Inputs
Market Weight Inputs
Names Unrolled
Convoluter
Buffer
Accumulated Loss Distribution (weighted sum)
Loss Distribution Output
Market Factors Unrolled

Figure 6 – Convoluter architecture diagram

width communication channel directly between the FPGAs.

One of the main focus points during this stage was finding a design that balanced arithmetic optimizations, desired accuracy, power consumption and reliability. We decided to implement two variations of the design: a “full-precision” design for fair-value (FV) calculations and a “reduced-precision” version for scenario analysis. We designed the full-precision variant for an average relative accuracy of 10^{-8} and the reduced-precision for an average relative accuracy of 10^{-4}. These design points share identical implementations, varying only in the compile-time parameters of precision, parallelism and clock frequency.

We built the Copula and convolution kernels to implement one or more pipelines that effectively parallelize loops within the computation. Since the convolution uses each value the Copula model generates many times, the kernel and manager components were scalable to enable us to tailor the exact ratio of Copula or convolution resources to the workload for the design. Figure 6 shows the structure of the convolution kernel in the design.

When implemented in MaxCompiler, the code for the convolution in the inner loop resembles the original structure. The first difference is that there is an implied loop, as data streams through the design, rather than the design iterating over the data. Another important distinction is that the code now operates in terms of a data stream (rather than on an array), such that the code is now describing a streaming graph, with offsets forward and backward in the stream as necessary, to perform the convolution. Figure 7 shows the core of the code for the convoluter kernel.

IMPLEMENTATION EFFORT

In general, software programming in a high-level language such as C++ is much easier than interacting directly with FPGA hardware using a low-level hardware description language. Therefore, in addition to performance, development and support time are increasingly significant components of overall effort from the software-engineering perspective. For our project, we measured programming effort as one aspect in the examination of programming models. Because it is difficult to get accurate development-time statistics and to measure the quality of code, we use lines of code (LOC) as our metric to estimate programming effort.

Due to the lower-level nature of coding for the FPGA architecture when compared with standard C++, the original 898 lines of code generated 3,696 lines of code for the FPGA (or a growth factor of just over four).

Starting with the program that runs on a CPU and iterating in time, we transformed the program into the spatial domain running on the MaxNode, creating a structure on the FPGA that matched the data-flow structure of the program (at least the computationally intensive parts). Thus, we optimized the computer based on the program, rather than optimizing the program based on the computer. Obtaining the results of the computation then became a simple matter of streaming the data through the Maxeler MaxRack system.

In particular, the fact we can use fixed-point representation for many of the variables in our application is a big advantage, because FPGAs offer the opportunity to optimize programs on the bit level, allowing us to pick the optimal representation for the internal variables of an algorithm, choosing precision and range for different encodins such as floating-point, fixed-point and logarithmic numbers.

We used Maxeler Parton to measure the amount of real time taken for the computation calls in the original implementation and in the accelerated software. We divided the total software time running on a single core by eight to attain the upper bound on the multicore performance, and compared it to the real time it took to perform the equivalent computation using eight cores and two FPGAs with the accelerated software.

For power measurements, we used an Electrocorder AL-2VA from Acksen Ltd. With the averaging window set to one second, we recorded a 220-second window of current and voltage measurements while the software was in its core computation routine.

As a benchmark for demonstration and performance evaluation, we used a fixed population of 29,250 CDO tranch-

```
HWVar d = io.input("inputDist", _distType);

HWVar p = io.input("probNonzeroLoss", _probType);

HWVar L = io.input("lowerProportion", _propType);

HWVar n = io.input("discretisedLoss", _operType);

HWVar lower = stream.offset(-n-1,-maxBins,0,d);

HWVar upper = stream.offset(-n,-maxBins,0,d);

HWVar o = ((1-p)*d + L*p*lower + (1-L)*p*upper);

io.output("outputDist", _distType, o);
```

Figure 7 – MaxCompiler code for adding a name to a basket

```
// Shared library and call overhead  5%
for  d in 0 ...    dates  - 1
   // Curve Interpolation   54.5%
   for j in 0 ... names   - 1
         Q[j] = Interpolate(d, curve)
   end # for j
   for  i  in 0 ... markets  - 1
      for j in 0 ...   names  - 1
        // Copula Section  9%
        prob = cum_norm   (( inv_norm  (Q[j]) - sqrt  (rho)*M[i])/sqrt(1 - rho);
        loss=  calc_loss   (prob,Q2[j],RR[j],RM[j])*notional[j];
        // (FPGA Data preparation and post processing)   11.2%
        n = integer(loss);
        lower = fractional(loss);
        for k in 0 ... bins - 1
           if j == 0
              dist[k] = k == 0 ? 1.0 : 0.0 ;
              dist[k] = dist[k]*(1 - prob) +
                             dist[k  - n]* prob *(1 - L) +
                             dist[k  - n - 1]* prob *L;
            if j == credits      - 1
              final_dist    [k] += weight[ i ] * dist[k];
         end # for k
      end # for j
   end # for i
end # for d
// Other overhead (object construction, etc)  19.9%
```

Figure 8 – Profiled version of FPGA take on pricing

es. This population comprised real bespoke tranche trades of varying maturity, coupon, portfolio composition and attachment/detachment points.

The MaxNode-1821 delivered a 31x speedup over an eight-core (Intel Xeon E5430 2.66-GHz) server in full-precision mode and a 37x speedup at reduced precision, both nodes using multiple processes to price up to eight tranches in parallel.

The significant differences between the two hardware configurations include the addition of the MAX2-4412C card with two Xilinx FPGAs and 24 Gbytes of additional DDR DRAM. Figure 8 shows the CPU profile of the code running with FPGA acceleration.

As Table 1 shows, the power usage per node decreases by 6 percent in the hybrid FPGA solution, even with a 31x increase in computational performance. It follows that the speedup per watt when computing is actually greater than the speedup per cubic foot.

Table 2 shows a breakdown of CPU time of the Copula vs. convolution computations and their equivalent resource utilization on the FPGA. As we have exchanged time for space when moving to the FPGA design, this gives a representative indication of the relative speedup between the different pieces of computation.

THREE KEY BENEFITS

The credit hybrids trading group within J.P. Morgan is reaping substantial benefits from applying acceleration technology. The order-of-magnitude increase in available computation has led to three key benefits: computations run much faster; additional applications and algorithms, or those that were once impossible to resource, are now possible; and operational costs resulting from given computations drop dramatically.

The design features of the Maxeler hybrid CPU/FPGA computer mean that its low consumption of electricity, physically small footprint and low heat output make it an extremely attractive alternative to the traditional cluster of standard cores.

One of the key insights of the project has been the substantial benefits to be gained from changing the computer to fit the algorithm, not changing the algorithm to fit the computer (which would be the usual approach using standard CPU cores). We have found that executing complex calculations in customizable hardware with Maxeler infrastructure is much faster than executing them in software.

FUTURE WORK

The project has expanded to include the delivery of a 40-node Maxeler hybrid computer designed to provide portfolio-level risk analysis for the credit hybrids trading desk in near real time. For credit derivatives, a second (more general) CDO model is currently undergoing migration to run on the architecture.

In addition, we have also applied the approach to two interest-rate models. The first was a four-factor Monte Carlo model covering equities, interest rates, credit and foreign exchange. So far, we've achieved an acceleration speedup of 284x over a Xeon core. The second is a general tree-based model, for which acceleration results are projected to be of a similar order of magnitude. We are currently evaluating further, more wide-ranging applications of the acceleration approach, and are expecting similar gains across a wider range of computational challenges.

PLATFORM	IDLE	PROCESSING
Dual Xeon L5430 2.66 GHz Quad-Core 48-GB DDR DRAM	185W	255W
(as above) with MAX2-4412C Dual Xilinx SX240T FPGAs 24-GB DDR DRAM	210W	240W

Table 1 – Power usage for 1U compute nodes when idle and while processing

COMPUTATION	PERCENT TIME IN SOFTWARE	6-INPUT LOOKUP TABLES	FLIP-FLOPS	36-KBIT BLOCK RAMS	18X25-BIT MULTIPLIERS
Copula Kernel	22.0%	30.05%	35.12%	11.85%	15.79%
Convolution and Integration	56.1%	46.54%	52.84%	67.54%	84.21%

Table 2 – Computational breakdown vs. FPGA resource usage of total used

Look Ma, No Motherboard!

How one design team put a full single-board computer with SATA into a Xilinx FPGA.

by Lorenz Kolb
Member of the Technical Staff
Missing Link Electronics, Inc.
lorenz@missinglinkelectronics.com

Endric Schubert
Co-founder
Missing Link Electronics, Inc.
endric@missinglinkelectronics.com

Rudolf Usselmann
Founder
ASICS World Service Ltd.
rudi@asics.ws

Embedded systems for industrial, scientific and medical (ISM) applications must support a plethora of interfaces. That's why many design teams choose FPGA-based daughtercards that plug right into a PC's motherboard to add those special I/Os. Given the huge capacity of modern FPGAs, entire systems-on-chips can be implemented inside a Xilinx® device. These systems include hardware, operating system and software, and can provide almost the complete functionality of a PC, diminishing the need for a PC motherboard. The result is a more compact, less power-hungry, configurable single-board computer system.

Many of those ISM applications rely on fast, dependable mass storage to hold and store the results of data acquisition, for example. Solid-state drives have become the de facto standard in this application because of their high reliability and fast performance. These SSDs almost always connect via a Serial ATA (SATA) interface.

Let's examine the steps that we took to extend a single-board computer system, built around a Xilinx chip, with high-speed SATA connectivity to add SSD RAID functionality. For this task, the Xilinx Alliance Program ecosystem brought together ASICS World Service's (ASICS ws) expertise in high-quality IP cores and Missing Link Electronics' (MLE) expertise in programmable-systems design.

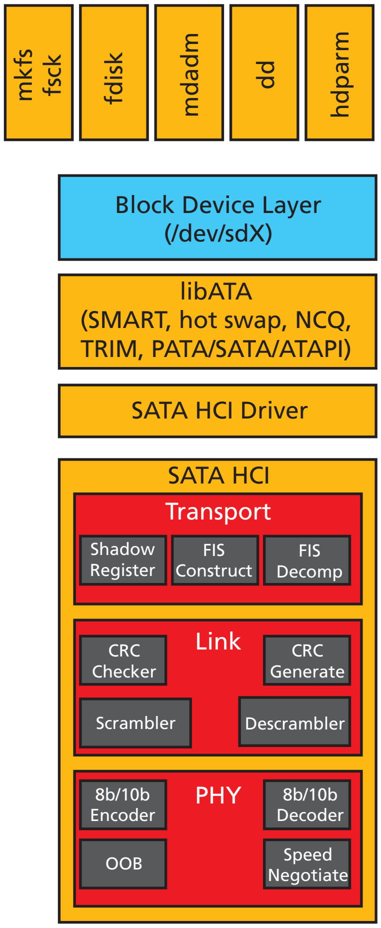


Figure 1 – Serial ATA function layers

But before delving into the details of the project, it's useful to take a deeper look at SATA itself. As shown in Figure 1, multiple layers are involved for full SATA host controller functionality. Therefore, when it comes to implementing a complete SATA solution for an FPGA-based programmable system, designers need much more than just a high-quality intellectual-property (IP) core. Some aspects of the design often get overlooked.

First, it makes sense to implement only the Physical (PHY), Link and some portions of the Transport Layer in FPGA hardware; that's why IP vendors provide these layers in the IP they sell. The SATA Host IP core from ASICS World Service utilizes the so-called MultiGigabit Transceivers, or MGT, [1] to implement the PHY layer—which comprises an out-of-band signaling block similar to the one described in Xilinx application note 870 [2]—completely within the FPGA. The higher levels of the Transport Layer, along with the Applications, Device and User Program layers, are better implemented in software and, thus, typically IP vendors do not provide these layers to customers. This, however, places the burden of creating the layers on the system design team and can add unanticipated cost to the design project.

The reason vendors do not include these layers in their IP is because each architecture is different and each will be used in a different manner. Therefore, to deliver a complete solution that ties together the IP core with the user programs, you must implement, test and integrate components such as scatter-gather DMA (SGDMA) engines, which consist of hardware and software.

In addition, communication at the Transport Layer is done via so-called

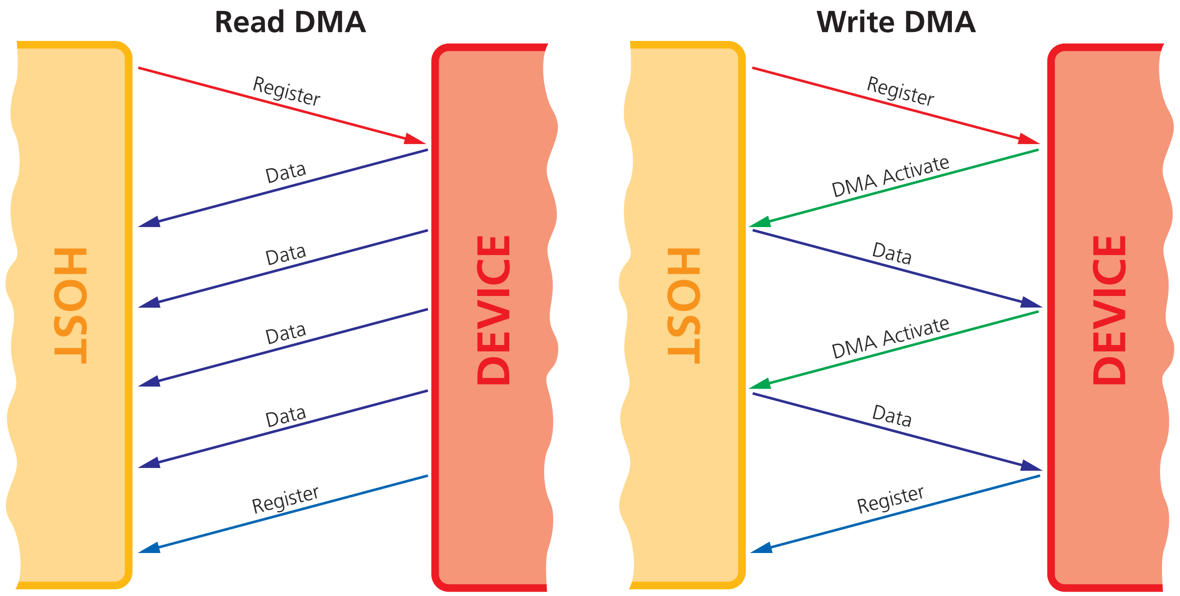


Figure 2 – FIS flow between host and device during a DMA operation

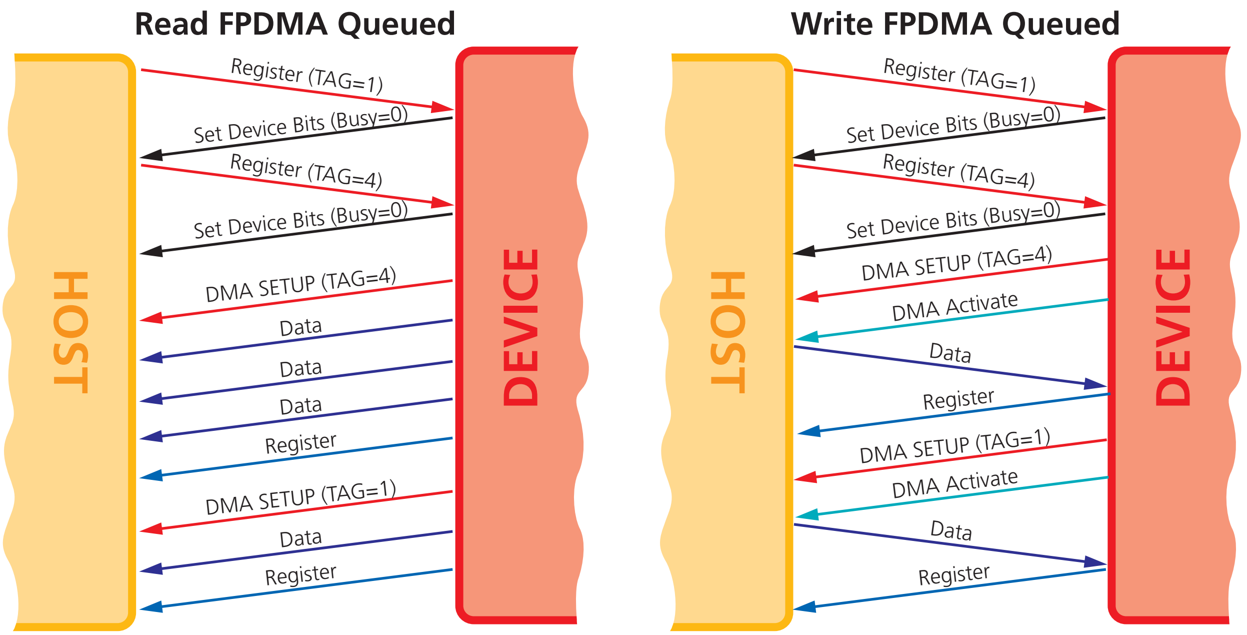


Figure 3 – FIS flow between host and device during first-party DMA queued operation

frame information structures (FIS). The SATA standard [3] defines the set of FIS types and it is instructive to look at the detailed FIS flow between host and device for read and write operations.

As illustrated in Figure 2, a host informs the device about a new operation via a Register FIS, which holds a standard ATA command. In case of a read DMA operation, the device sends one (or more) Data FIS as soon as it is ready. The device completes the transaction via a Register FIS, from device to host. This FIS can inform of either a successful or a failed operation.

Figure 2 also shows the FIS flow between host and device for a write DMA operation. Again, the host informs the device of the operation via a Register FIS. When the device is ready to receive data, it sends a DMA Activate FIS and the host will start transmitting a single Data FIS. When the device has processed this FIS and

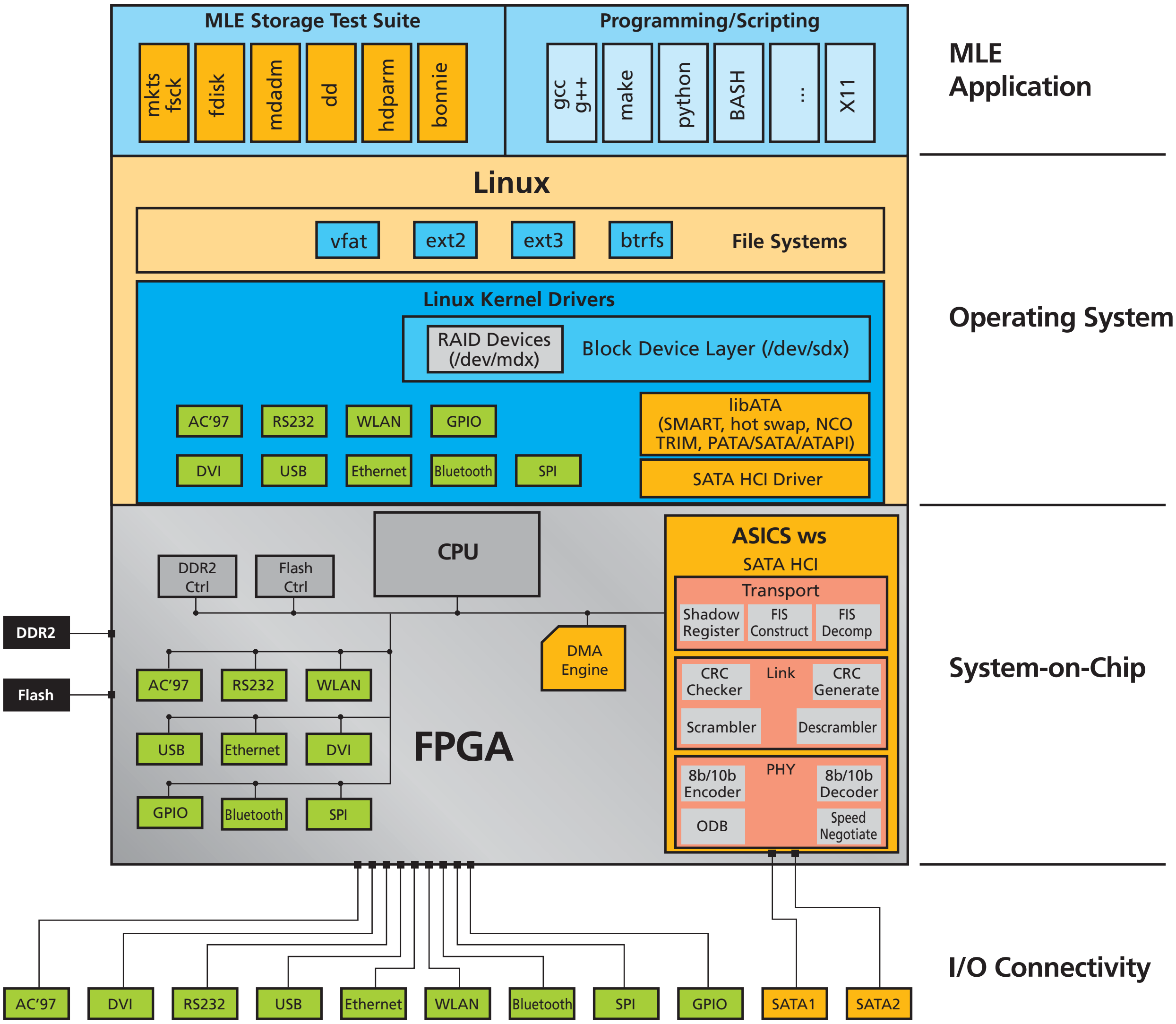


Figure 4 – Complete SATA solution

it still expects data, it again sends a DMA Activate FIS. The process is completed in the same way as the read DMA operation.

A new feature introduced with SATA and not found in parallel ATA is the so-called first-party DMA. This feature transfers some control over the DMA engine to the device. In this way the device can cache a list of commands and reorder them for optimized performance, a technique called native command queuing. New ATA commands are used for first-party DMA transfers. Because the device does not necessarily complete these commands instantaneously, but rather queues them, the FIS flow is a bit different for this mode of operation. The flow for a read first-party DMA queued command is shown on the left side of Figure 3.

Communication on the Application Layer, meanwhile, uses ATA commands. [4] While you can certainly implement a limited number of these commands as a finite state machine in FPGA hardware, a software implementation is much more efficient and flexible. Here, the open-source Linux kernel provides a known-good implementation that almost exactly follows the ATA standard and is proven in more than a billion devices shipped.

The Linux ATA library, libATA, copes with more than 100 different ATA commands to communicate with ATA devices. These commands include data transfers but also provide functionality for SMART (Self-Monitoring Analysis and Reporting Technology) and for security features such as secure erase and device locking.

The ability to utilize this code base, however, requires the extra work of implementing hardware-dependent software in the form of Linux device drivers as so-called Linux Kernel Modules. As Figure 4 shows, the Missing Link Electronics “Soft” Hardware Platform comes with a full

When integrating a SATA IP core into an FPGA-based system, there are many degrees of freedom. So, pushing the limits of the whole system requires knowledge of not just software or hardware, but both. Integration must proceed in tandem for software and hardware.

GNU/Linux software stack preinstalled, along with a tested and optimized device driver for the SATA host IP core from ASICS World Service.

When integrating a SATA IP core into an FPGA-based system, there are many degrees of freedom. So, pushing the limits of the whole system requires knowledge of not just software or hardware, but both. Integration must proceed in tandem for software and hardware.

Figure 5 shows examples of how to accomplish system integration of a SATA IP core. The most obvious way is to add the IP core as a slave to the bus (A) and let the CPU do the transfers between memory and the IP. To be sure, data will pass twice over the system bus, but if high data rates are not required, this easy-to-implement approach may be sufficient. In this case, however, you can use the CPU only for a small application layer, since most of the time it will be busy copying data.

The moment the CPU has to run a full operating system, the impact on performance will be really dramatic. In this case, you will have to consider reducing the CPU load by adding a dedicated copy engine, the Xilinx Central DMA (option B in the figure). This way, you are still transferring data twice over the bus, but the CPU does not spend all of its time copying data.

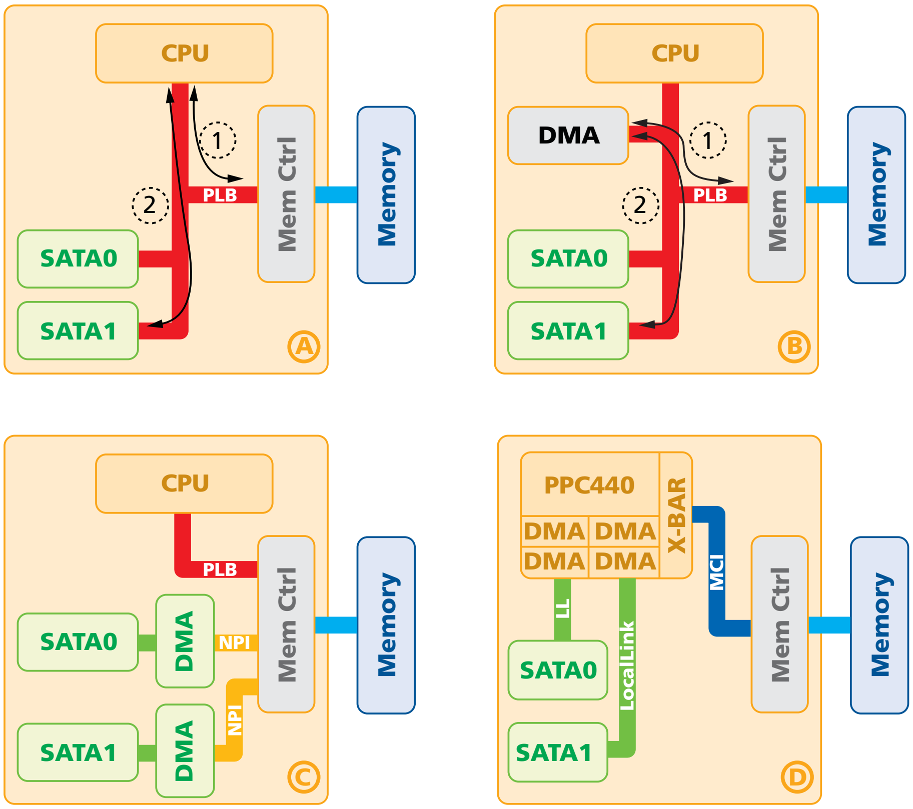


Figure 5 – Four architectural choices for integrating a SATA IP core

Still, the performance of a system with a full operating system is far away from a standalone application, and both are far from the theoretical performance limits. The third architecture option (C in the figure) changes this picture by reducing the load of the system bus and using simple dedicated copy engines via Xilinx's streaming NPI port and Multiport Memory Controller (MPMC). This boosts the performance of the standalone application up to the theoretical limit. However, the Linux performance of such a system is still limited.

From the standalone application, we know that the bottleneck is not within the interconnection. This time the bottleneck is the memory management in Linux. Linux handles memory in blocks of a page size. This page size is 4,096 bytes for typical systems. With a simple DMA engine and free memory scattered all over the RAM in 4,096-byte blocks, you may move only 4,096 bytes with each transfer. The final architectural option (D in the figure) tackles this problem.

For example, the PowerPC® PPC440 core included in the Virtex®-5 FXT FPGA has dedicated engines that are capable of SGDMA. This way, the DMA engine gets passed a pointer to a list of memory entries and scatters/gathers data to and from this list. This results in

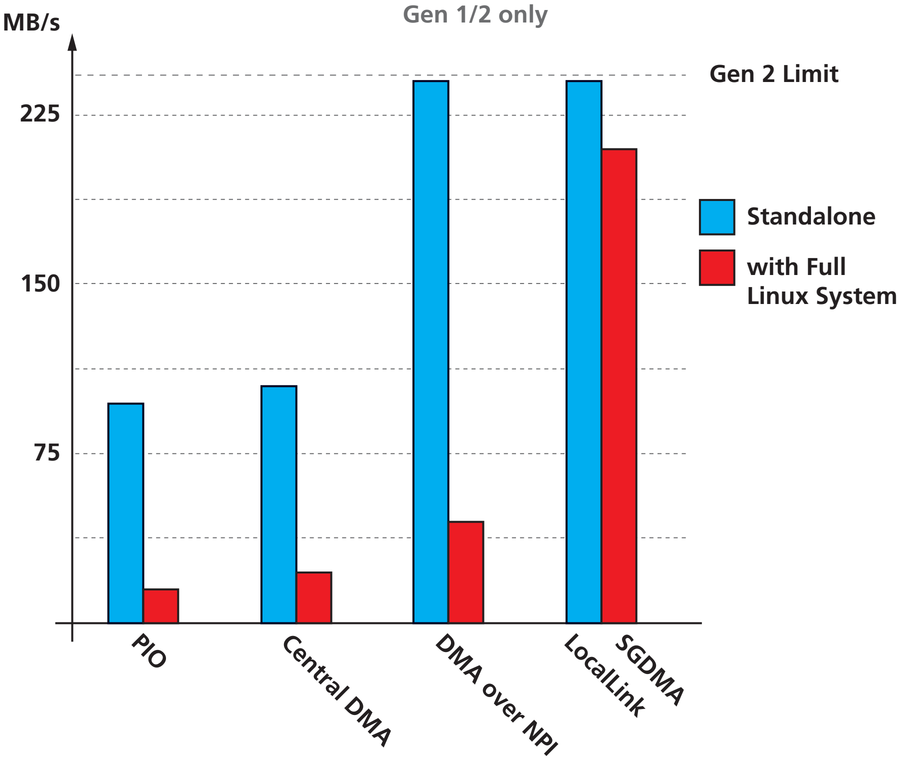


Figure 6 – Performance of complete SATA solution

larger transfer sizes and brings the system very close to the standalone performance. Figure 6 summarizes the performance results of these different architectural choices.

Today, the decision whether to make or buy a SATA host controller core is obvious: Very few design teams are capable of implementing a functioning SATA host controller for the cost of licensing one. At the same time, it is common for design teams to spend significant time and money in-house to integrate this core into a programmable system-on-chip, develop device drivers for this core and implement application software for operating (and testing) the IP.

The joint solution our team crafted would not have been possible without short turnaround times between two Xilinx Alliance Program Partners: ASICS World Service Ltd. and Missing Link Electronics, Inc. To learn more about our complete SATA solution, please visit the MLE Live Online Evaluation site at *http://www.missinglinkelectronics.com/LOE*. There, you will get more technical insight along with an invitation to test-drive a complete SATA system via the Internet.

References:

1. Xilinx, "Virtex-5 FPGA RocketIO™ GTX Transceiver User Guide," October 2009. http://www.xilinx.com/bvdocs/userguides/ug198.pdf.

2. Xilinx, Inc., "Serial ATA Physical Link Initialization with the GTP Transceiver of Virtex-5 LXT FPGAs," 2008 application note. http://www.xilinx.com/support/documentation/application_notes/xapp870.pdf.

3. Serial ATA International Organization, Serial ATA Revision 2.6, February 2007. http://www.sata-io.org/

4. International Committee for Information Technology Standards, AT Attachment 8 - ATA/ATAPI Command Set, September 2008. http://www.t13.org/

General-Purpose SoC Platform Handles Hardware/Software Co-Design

Built on a pair of Spartan FPGAs, the fourth-generation Gecko system serves industrial and research applications as well as educational projects.

By Theo Kluter, PhD
Lecturer
Bern University of Applied Sciences (BFH)
theo.kluter@bfh.ch

Marcel Jacomet, PhD
Professor
Bern University of Applied Sciences (BFH)
marcel.jacomet@bfh.ch

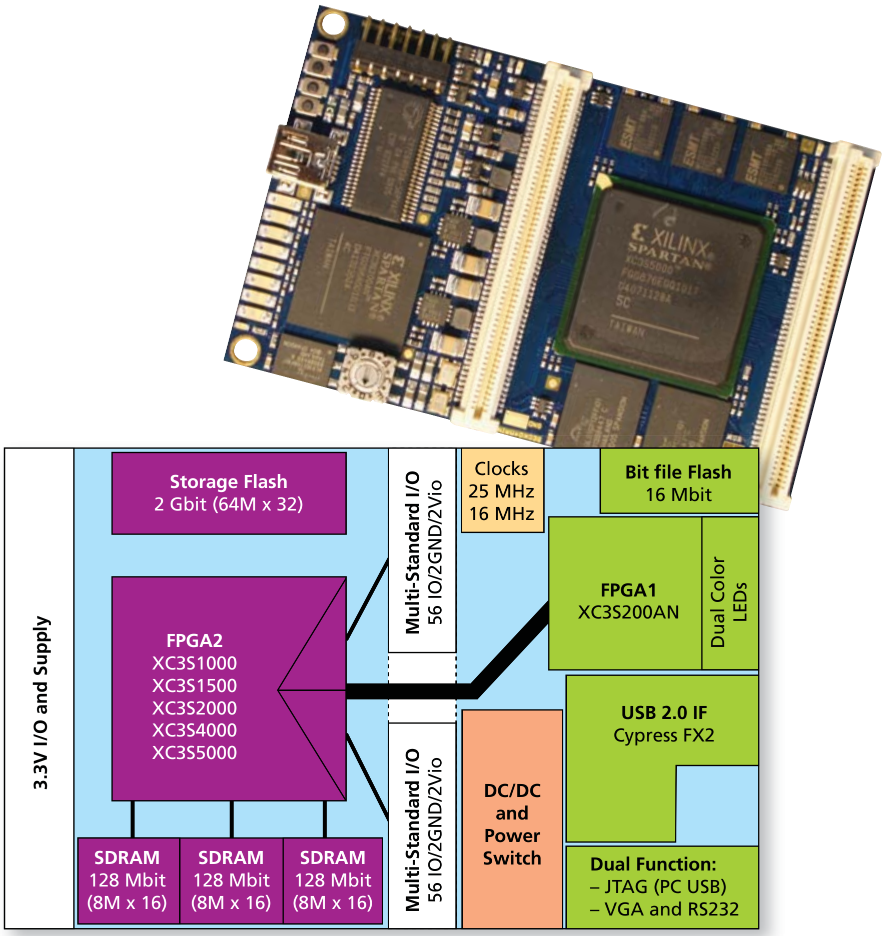


Figure 1 – The Gecko4main module is an eight-layer PCB with the components all on the top layer, reducing production cost.

The Gecko system is a general-purpose hardware/software co-design environment for real-time information processing in system-on-chip (SoC) solutions. The system supports the co-design of software, fast hardware and dedicated real-time signal-processing hardware. The latest member of the Gecko family, the Gecko4main module, is an experimental platform that offers the necessary computing power to speed intensive real-time algorithms and the necessary flexibility for control-intensive software tasks. Adding extension boards equipped with sensors, motors and a mechanical housing suits the Gecko4main module for all manner of industrial, research and educational projects.

Developed at the Bern University of Applied Sciences (BFH) in Biel/Bienne, Switzerland, the Gecko program got its start more than 10 years ago in an effort to find a better way to handle complex SoC design. The goal of the latest, fourth-generation Gecko project was to develop a new state-of-the-art version of the general-purpose system-on-chip hardware/software co-design development boards, using the credit-card-size form factor (54 x 85 mm). With Gecko4, we have realized a cost-effective teaching platform while still preserving all the requirements for industrial-grade projects.

The Gecko4main module contains a large, central FPGA—a 5 million-gate Xilinx® Spartan®-3—and a smaller, secondary Spartan for board configuration and interfacing purposes. This powerful, general-purpose SoC platform for hardware/software co-designs allows you to integrate a MicroBlaze™ 32-bit RISC processor with closely coupled application-specific hardware blocks for high-speed hardware algorithms within the single multimillion-gate FPGA chip on the Gecko4main module. To increase design flexibility, the board also is equipped with several large static and flash memory banks, along with RS232 and USB 2.0 interfaces.

In addition, you can stack a set of extension boards on top of the Gecko4main module. The current extension boards—which feature sensors, power supply and a mechanical housing with motors—can help in developing system-on-chip research and industrial projects, or a complete robot platform ready to use for student exercises in different subjects and fields.

FEATURES AND EXTENSION BOARDS

The heart of the Gecko4 system is the Gecko4main module, shown in Figure 1. The board can be divided into three distinct parts:

- Configuration and IP abstraction (shown on the right half of Figure 1)
- Board powering
- Processing and interfacing (shown on the left half of Figure 1)

One of the main goals of the Gecko4main module is the abstraction of the hardware from designers or student users, allowing them to concentrate on the topics of interest without having to deal too much with the hardware details. The onboard Spartan-3AN contains an internal flash memory, allowing autonomous operation. This FPGA performs the abstraction by providing memory-mapped FIFOs and interfaces to a second FPGA.

Users can accomplish the dynamic or autonomous configuration of that second Spartan in three ways: via a bit file that is stored in the bit-file flash for autonomous operation, by the PC through the USB 2.0 and the USBTMC protocol, or by connecting a Xilinx JTAG programmer to the dual-function connecter.

The main Spartan-3AN handles the first two options using a parallel-slave configuration protocol. The JTAG configuration method involves the direct intervention of the Xilinx design tools.

NEW TWISTS ON CHIP DESIGN

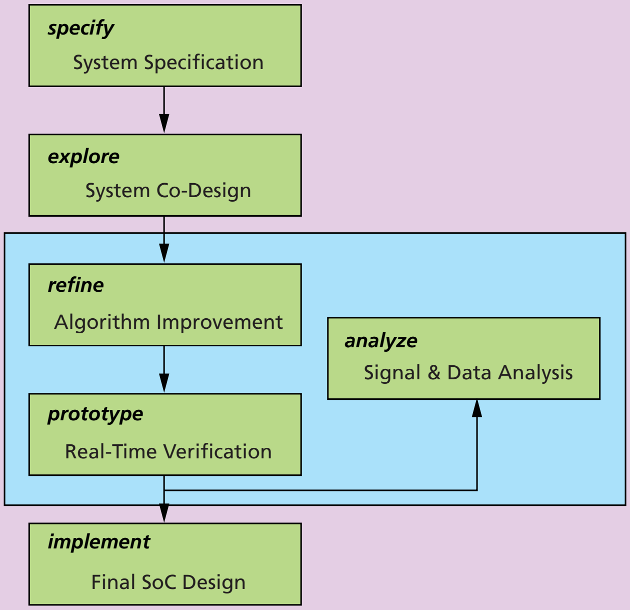
The inherently interdisciplinary character of modern systems has revealed the limits of the electronics industry's classical chip-design approach. The capture-and-simulate method of design, for example, has given way to the more efficient describe-and-synthesize approach.

More sophisticated hardware/software co-design methodologies have emerged in the past several years. Techniques such as the specify-explore-refine method give designers the opportunity of a high-level approach to system design, addressing lower-level system-implementation considerations separately in subsequent design steps. Speed-sensitive tasks are best implemented with dedicated hardware, whereas software programs running on microprocessor cores are the best choice for control-intensive tasks.

A certain amount of risk is always present when designing SoCs to solve new problems for which only limited knowledge and coarse models of the analog signals to be processed are available. The key problem in such situations is the lack of experience with the developed solution. This is especially true where real-time information processing in a closed control loop is needed. Pure-software approaches limit the risk of nonoptimal solutions, simply because it's possible to improve software algorithms in iterative design cycles. Monitoring the efficiency and quality of hardware algorithms is, however, no easy task. In addition, hardware algorithms cannot easily be improved on SoC solutions in the same iterative way as software approaches.

In control-oriented reactive applications, the system reacts continuously to the environment. Therefore, the monitoring of different tasks is crucial, especially if there are real-time constraints. As long as precise models of the information to be processed are not available, as often occurs in analog signal processing, a straightforward design approach is impossible or at least risky. The flexibility of iteratively improving analog/digital hardware-implemented algorithms is thus necessary in order to explore the architecture and to refine its implementation. Again, this involves monitoring capabilities of the SoC in its real-time environment.

Thus, an important problem in the classical ASIC/FPGA hardware/software co-design approach is decoupling of the design steps with the prototyping hardware in the design flow, which inhibits the necessary monitoring task in an early design phase. Adding an extra prototyping step and an analysis step opens the possibility to iteratively improve the SoC design. The refine-prototype-analysis design loop best matches the needs of SoC designs with real-time constraints (see figure). If precise models of the plant to be controlled are missing, thorough data analysis and subsequent refinement of the data-processing algorithms are needed for designing a quality SoC. — *Theo Kluter and Marcel Jacomet*



The specify-explore-refine design flow transforms into a specify-explore-refine-prototype-analyze design flow for SoC designs with real-time constraints.

While the latter option allows for on-chip debugging with, for example, ChipScope™, the former two provide dynamic reconfiguration of even a 5 million-gate-equivalent second FPGA in only a fraction of a second.

Many of our teaching and research targets require a fast communication link for features like hardware-in-the-loop, regression testing and semi-real-time sensing, among others. The Gecko4 system realizes this fast link by interfacing a USB 2.0 slave controller (the Cypress Fx2) with the USBTMC protocol. To both abstract the protocol and provide easy interfacing for users, the Spartan-3AN does protocol interpretation and places the payload into memory-mapped FIFOs.

Designers who do not need a large FPGA can use the Gecko4main module without mounting the left half of the PCB, reducing board costs significantly. In this setting, designers can use the Spartan-3AN directly to house simple systems—indeed, they can even implement a simple CPU-based system in this 200 million-gate-equivalent FPGA. For debugging and display purposes, the device provides eight dual-color LEDs, three buttons, an unbuffered RS232 connection and a text-display VGA connection at a resolution of 1,024 x 768 pixels (60 Hz). These interfaces are also available memory-mapped to the second FPGA.

BOARD POWERING

Due to the various usages of the Gecko4main module, powering must be very flexible. We provided this flexibility by means of the onboard generation of all required voltages. Furthermore, the module offers two possibilities of supplying power: the USB cable or an added power board. The attached power board is the preferred supply.

According to the USB standard, a device is allowed to use a maximum of 500 milliamps. The Gecko4main module requests this maximum current. If the design requires more than the available 2.5 watts, the board turns off automatically. The second way of supplying the unit is by attaching a power board. If these power boards are battery-powered, the Gecko4main module can function autonomously (as, for example, in a robot application).

If both a USB cable and a power board are connected (or disconnected), the Gecko4main module automatically switches seamlessly between the different supplies without a power glitch.

PROCESSING AND INTERFACING

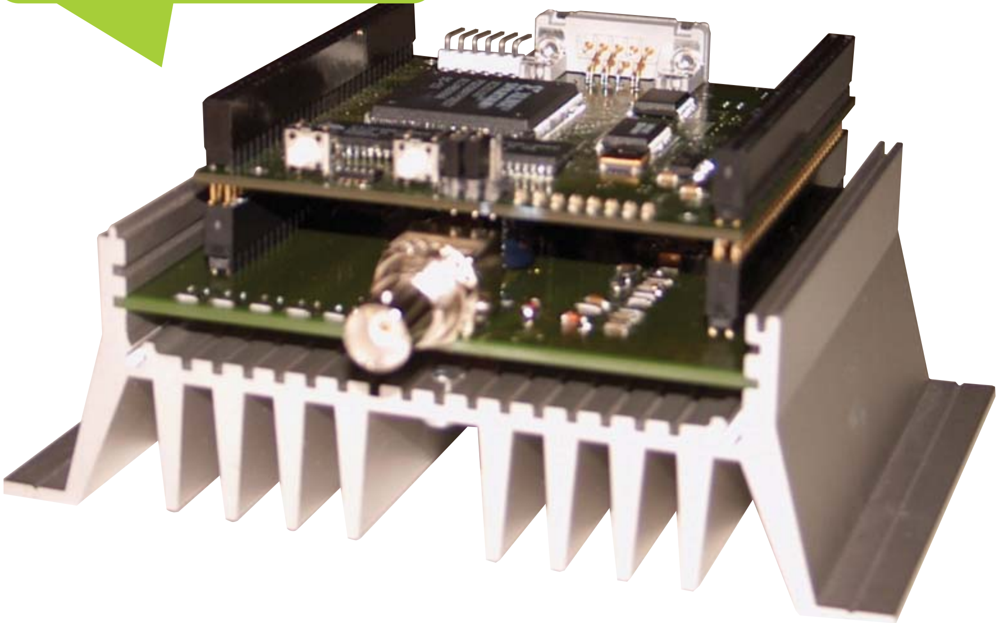
The processing heart of the Gecko4main module is the second FPGA, also a Spartan-3. The board can accommodate a Spartan-3 ranging from the "small" million-gate-equivalent XC3S1000 up to the high-end 5 million-gate-equivalent XC3S5000. The board shown in Figure 1 has an XC3S5000 mounted that was donated by the Xilinx University Program.

Except for the XC3S1000, all Spartan-3 devices have three independent banks of 16-Mbyte embedded SDRAM connected, along with 2 Gbits of NAND flash for dynamic and static storage. Furthermore, to interface to extension boards, the Gecko4main module provides two "stack-through" connectors. The one shown on the left in Figure 1 provides a standard 3.3-V-compliant bus including power supply pins and external clocking. This standard connector is backward-compatible with our Gecko3 system.

One of the advantages of the Xilinx FPGA family is its support for various I/O standards on each of the I/O banks of a given FPGA. To be able to provide flexibility in the I/O standard used on an extension board, the I/O connector shown in the middle of Figure 1 is divided into halves, each completely connected to one I/O bank of the FPGA. Furthermore, each half provides its own I/O supply voltage pin. If there is no attached extension board providing an I/O supply voltage, the I/O banks are automatically supplied by 3.3 V, placing them in the LVTTL standard.

For some applications, even the PCB trace lengths are important. To also provide the flexibility of implementing this kind of application, the Gecko4main module provides 38 out of the 54 available I/O lines that are length-compensated (+/- 0.1 mm).

Figure 2 – For an industrial project for a long-range RFID reader, a Gecko2 mainboard and an RF extension board inhabit a housing that also serves as a cooling element.



EXTENSIONS ARE AVAILABLE

The strength of the Gecko4main module lies in its reduction to the minimal required in terms of storage and peripherals for SoCs, and also in its stack-through principle. Both aspects allow for simple digital logic experiments (by simply attaching one board to the USB connector of a PC) or complex multiprocessor SoC emulation (by stacking several boards in combination with extension boards). The Bern University of Applied Sciences and several other research institutes have developed a variety of extension boards ranging from sensor boards up to complete mainboards. A full overview of our open-source Gecko system can be found at our Gecko Web page (*http://gecko.microlab.ch*) and on the OpenCores Web page (*http://www.opencores.org/project,gecko4*).

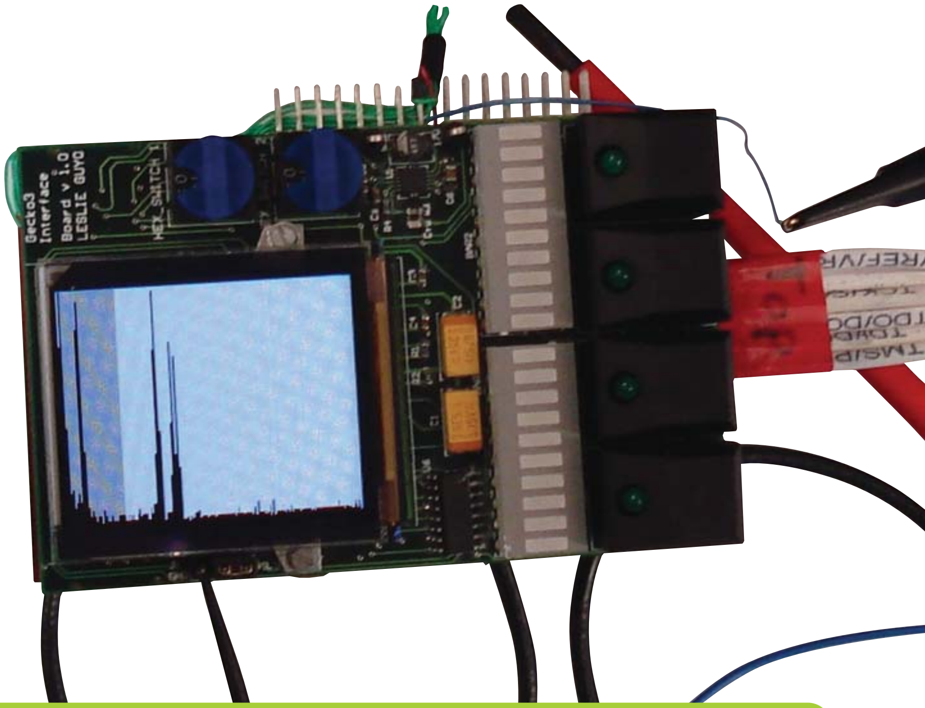


Figure 3 – An applied-research project is investigating high-speed hardware algorithms for optical-coherence tomography image processing. The image shows a depth scan (A-scan) illustrating various reflections of the sample as an intermediate result.

EDUCATION PLATFORM

The Gecko system is extensively used in the electronic and communication engineering curriculum at BFH. Different professors use the system in their laboratories, including the Institute of Human-Centered Engineering-microLab, in both the undergraduate and the master's curriculum. First-year exercises start with simple digital hardware designs using the Geckorobot platform, a small robot frame on which the Gecko3main can be attached. The frame provides two motors, some infrared sensors, two speed sensors and a distance sensor.

Robot control exercises in the signal-processing courses develop control algorithms for the two independent robot motors using the well-known MATLAB®/Simulink® tools for design entry, simulation and verification. A real-time environment allows the students to directly download their Simulink models into the Geckorobot platform and supports a real-time visualization of the control algorithm behavior of the running robot in the MATLAB/Simulink tools.

For this purpose, the students have access to the necessary Simulink S-functions and a MicroBlaze-based platform with the software and hardware IP interfacing the sensors and motors of the Geckorobot platform. Immediate feedback of the robot's behavior in real-world situations as well as in the MATLAB/Simulink design environment has a high learning value for the students. For more advanced student projects, we are planning drive-by-wire and cybernetic behavior of the robots in a robot cluster. Moreover, we are also developing a compressive-sampling Gecko extension board for audio signals through which students will learn—by listening—the impressive possibilities of this new theory as well as experience its limitations.

RESEARCH AND INDUSTRIAL PROJECTS

Outside the university, the Gecko platform is finding a home in applied research and industrial projects. In most cases researchers pair the Gecko4main module, for the computation-intensive tasks, together with application-specific extension boards. In an industrial project developing hardware algorithms for long-range radio frequency identification (RFID) readers, one group designed an RF power board as an extension board for the Gecko2main board (see Figure 2).

Currently, hardware algorithms for high-speed optical-coherence tomography (OCT) signal processing are under investigation. OCT is an optical signal-acquisition technology that makes it possible to capture high-resolution 3-D images in biological tissue using near-infrared light. In an ongoing applied-research project, a Gecko extension board with the optical interface is under development (see Figure 3). The Gecko4main module serves as a platform to develop high-speed OCT hardware algorithms.

TOOL CHAIN FOR GECKO

Our applied SoC design method for real-time information-processing problems is based on an optimal combination of approved design technologies that are widely used in the industry (see sidebar). The presented design flow (Figure 4) is exclusively based on Xilinx and some third-party tools, and on adding dedicated interfaces and IP. Control algorithms developed in the MATLAB/Simulink environment can be compiled, downloaded and implemented in the Gecko4 platform for the real-time analyses by a simple and fast pushbutton tool environment. The tool chain represents a step toward the upcoming specify-explore-refine design methodology by adding a fast-prototyping feature. Fast prototyping is a crucial feature for an iterative improvement of control or signal-processing algorithms with feedback loops where precise models are not available.

Experience over the last 10 years has shown that the Gecko approach provides a good platform for educational, research and industrial projects alike. We believe that the capability of producing system prototypes very early in an SoC design flow is a major key to success in today's industry. It helps to secure the system architecture work, develop usable prototypes that can be distributed to internal and external customers and, last but not least, ensure that designers can start functional verification at a very early design phase.

Rapid prototyping meets the requirements for hardware/software co-design as an early function integration. It enables system validation and various forms of design verification. Very important is the real-time algorithm verification of analog/digital hardware and software realizations. We firmly believe that the Gecko4 platform enables all of these aspects of the new breed of complex SoC designs.

Acknowledgements

We want to thank the Xilinx University Program (XUP) for its generous donation of the FPGAs used for the production of both the third- and fourth-generation Gecko mainboards. Furthermore, we want to thank the OpenCores community for hosting and certifying our project. Finally, we thank all the students and research assistants who helped build the Gecko platform, in particular Christoph Zimmermann, who realized and was responsible for the third generation of the Gecko mainboard.

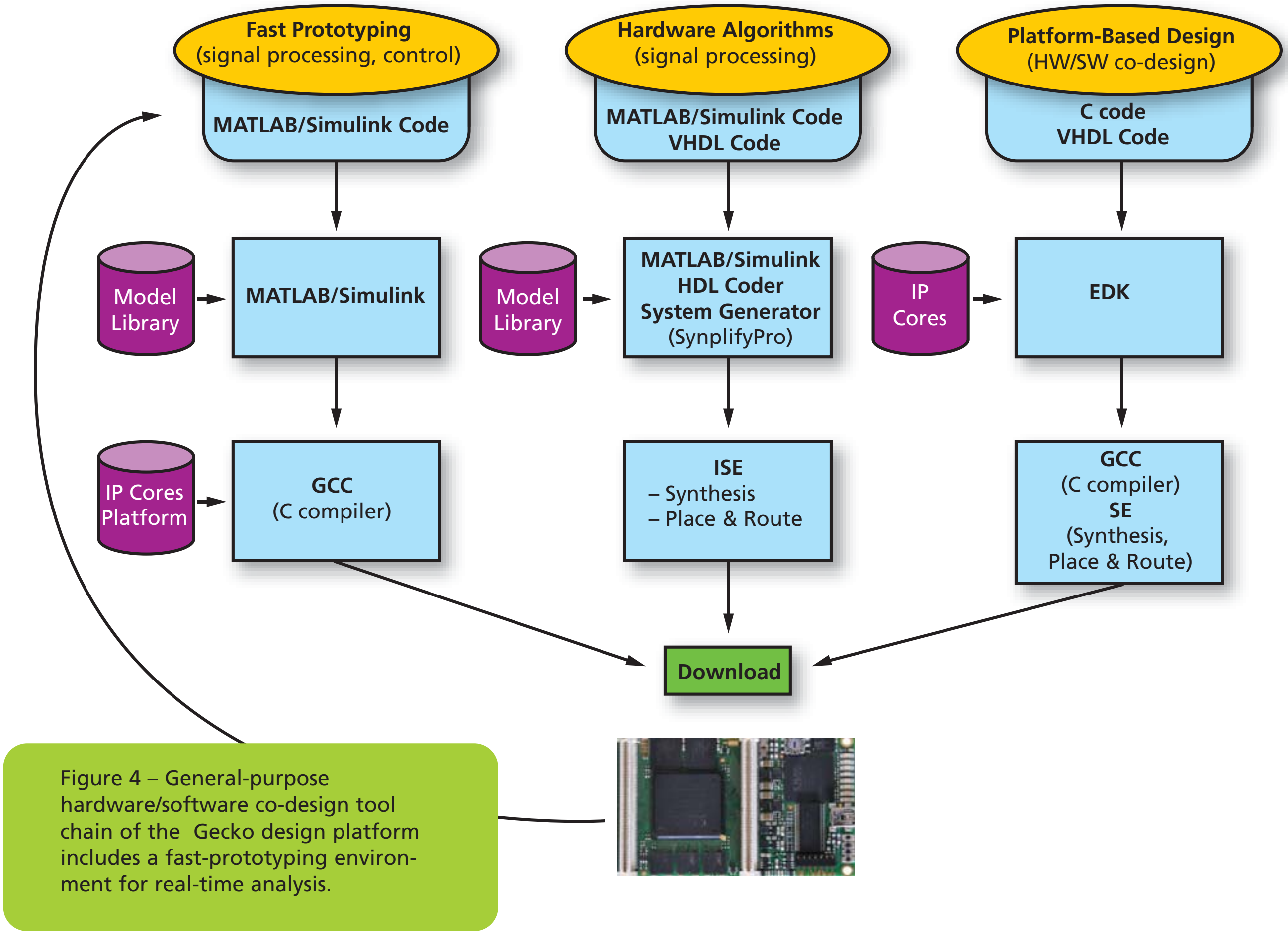


Figure 4 – General-purpose hardware/software co-design tool chain of the Gecko design platform includes a fast-prototyping environment for real-time analysis.

Wireless MIMO Sphere Detector Implemented in FPGA

High-level synthesis tools from AutoESL made it possible to build a complex receiver for broadband wireless systems in a Xilinx Virtex-5 device.

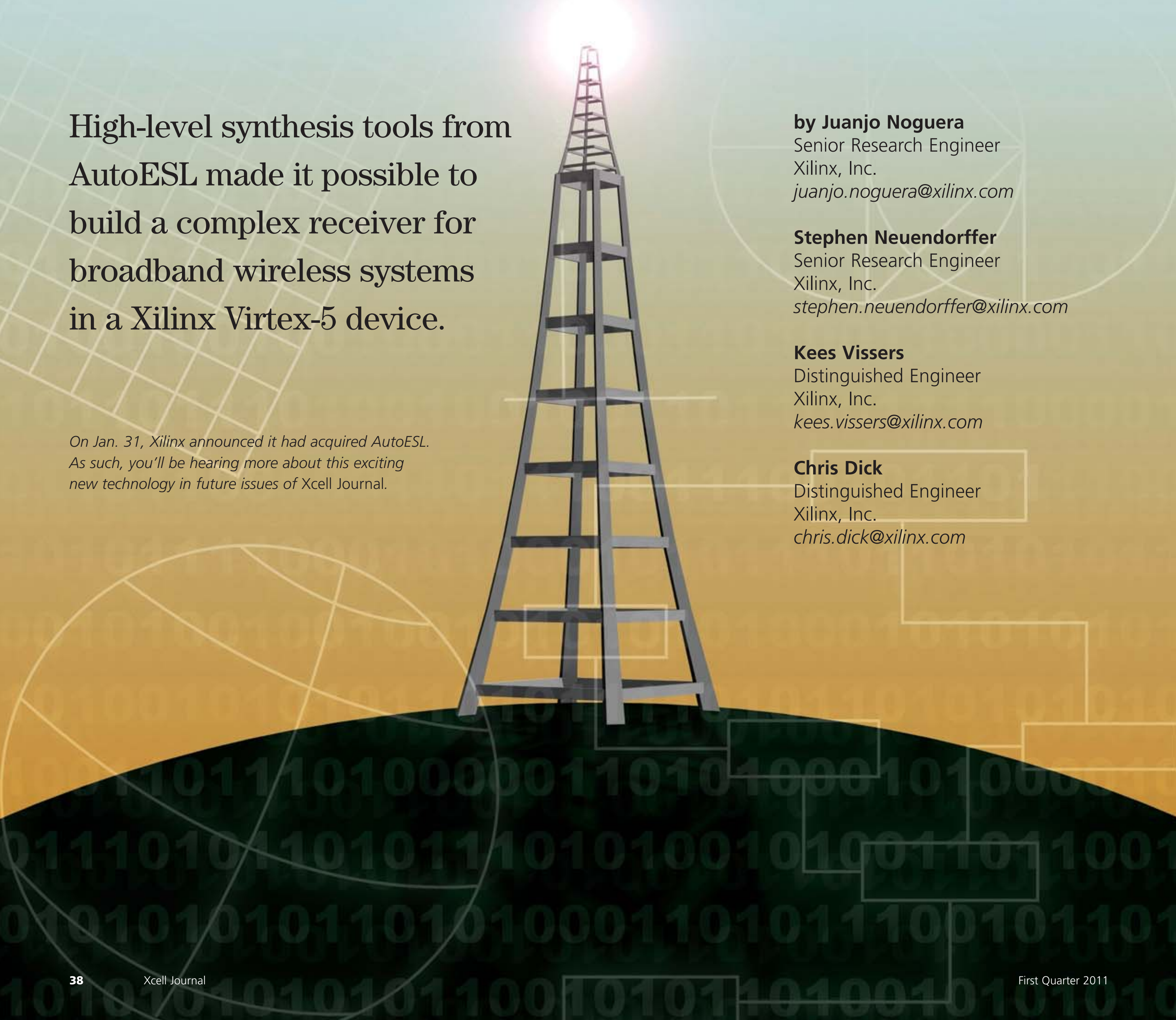
On Jan. 31, Xilinx announced it had acquired AutoESL. As such, you'll be hearing more about this exciting new technology in future issues of Xcell Journal.

by Juanjo Noguera
Senior Research Engineer
Xilinx, Inc.
juanjo.noguera@xilinx.com

Stephen Neuendorffer
Senior Research Engineer
Xilinx, Inc.
stephen.neuendorffer@xilinx.com

Kees Vissers
Distinguished Engineer
Xilinx, Inc.
kees.vissers@xilinx.com

Chris Dick
Distinguished Engineer
Xilinx, Inc.
chris.dick@xilinx.com



Spatial-division multiplexing MIMO processing significantly increases the spectral efficiency, and hence capacity, of a wireless communication system. For that reason, it is a core component of next-generation WiMAX and other OFDM-based wireless communications. This is a computationally intensive application that implements highly demanding signal-processing algorithms.

A specific example of spatial multiplexing in MIMO systems is sphere decoding, an efficient method of solving the MIMO detection problem while maintaining a bit-error rate (BER) performance comparable to the optimal maximum-likelihood detection algorithm. However, DSP processors don't have enough compute power to cope with the requirements of sphere decoding in real time.

Field-programmable gate arrays are an attractive target platform for the implementation of complex DSP-intensive algorithms like the sphere decoder. Modern FPGAs are high-performance parallel-computing platforms that provide the dedicated hardware needed, while retaining the flexibility of programmable DSP processors. There are several studies showing that FPGAs could achieve 100x higher performance and 30x better cost/performance than traditional DSP processors in a number of signal-processing applications. [1]

Despite this tremendous performance advantage, FPGAs are not generally used in wireless signal processing, largely because traditional DSP programmers believe they are hard to handle. Indeed, the key barrier to widespread adoption of FPGAs in wireless applications is the traditional hardware-centric design flow and tools. Currently, the use of FPGAs requires significant hardware design experience, including familiarity with hardware description languages like VHDL and Verilog.

Recently, new high-level synthesis tools [2] have become available as design aids for FPGAs. These design tools take a high-level algorithm description as input, and generate an RTL that can be used with standard FPGA implementation tools (for example, the Xilinx® ISE® design suite and Embedded Development Kit). The tools increase design productivity and reduce development time, while producing good quality of results. [3] We used such tools to design an FPGA implementation of a complex wireless algorithm—namely, a sphere detector for spatial-multiplexing MIMO in 802.16e systems. Specifically, we chose AutoESL's AutoPilot high-level synthesis tool to target a Xilinx Virtex®-5 running at 225 MHz.

SPHERE DECODING

Sphere detection, a part of the decoding process, is a prominent method of simplifying the detection complexity in spatial-multiplexing systems while maintaining BER performance comparable to that of optimum maximum-likelihood (ML) detection, a more complex algorithm.

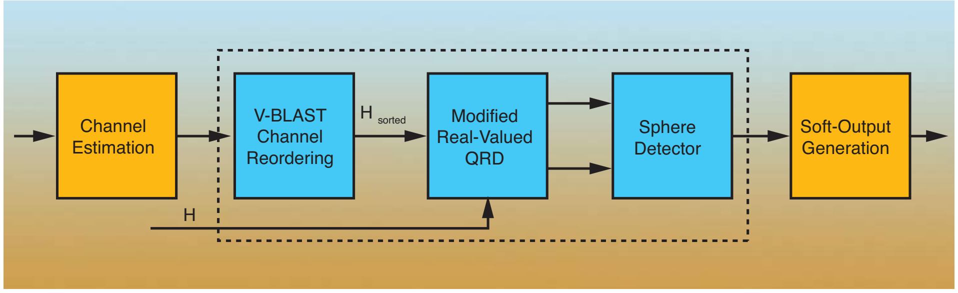


Figure 1 – Block diagram for sphere decoder

In the block diagram of the MIMO 802.16e wireless receiver shown in Figure 1, it is assumed that the channel matrix is perfectly known to the receiver, which can be accomplished by classical means of channel estimation. The implementation consists of a pipeline with three building blocks: channel reordering, QR decomposition and sphere detector (SD). In preparation for engaging a soft-input-soft-output channel decoder (for example, a turbo decoder), we produced soft outputs by computing the log-likelihood ratio of the detected bits. [4]

CHANNEL MATRIX REORDERING

The order in which the sphere detector processes the antennas has a profound impact on the BER performance. Channel reordering comes first, prior to sphere detection. By utilizing a channel matrix preprocessor that realizes a type of successive-interference cancellation similar in concept to that employed in BLAST (Bell Labs Layered Space Time) processing, the detector achieves close-to-ML performance.

The method implemented by the channel-reordering process determines the optimum detection order of columns of the complex channel matrix over several iterations. The algorithm selects the row with the maximum or minimum norm depending on the iteration count. The row with the minimum Euclidean norm represents the influence of the strongest antenna while the row with the maximum Euclidean norm represents the influence of the weakest antenna. This novel approach first processes the weakest stream. All subsequent iterations process the streams from highest to lowest power.

To meet the application's high-data-rate requirements, we realized the channel-ordering block using the pipelined architecture shown in Figure 2, which processes five channels in a time-division multiplexing (TDM) approach. This scheme provided more processing time between the matrix elements of the same channel while sustaining high data throughput. The calculation of the G matrix is the most demanding component in Figure 2. The heart of the process is matrix inversion, which we realized using QR decomposition (QRD). A common method for realizing QRD is based on Givens rotations. The proposed implementation performs the complex rotations in the diagonal and off-diagonal cells, which are the fundamental computation units in the systolic array we are using.

MODIFIED REAL-VALUED QRD

After obtaining the optimal ordering of the channel matrix columns, the next step is to apply QR decomposition on the real-valued matrix coefficients. The functional unit used for this QRD processing is similar to the QRD engine designed to compute the inverse matrix, but with some modifications. The input data in this case has real values, and the systolic array structure has correspondingly higher dimensions (i.e., 8x8 real-valued instead of 4x4 complex-valued).

To meet the desired timing constraints, the input data consumption rate had to be one input sample per clock cycle. This requirement introduced challenges around processing-latency problems that we couldn't address with a five-channel TDM structure. Therefore, we increased the number of channels in a TDM group to 15 to provide more time between the successive elements of the same channel matrix.

SPHERE DETECTOR DESIGN

You can view the iterative sphere detection algorithm as a tree traversal, with each level of the tree i corresponding to processing symbols from the ith antenna. The tree traversal can be performed using several different methods. The one we selected was a breadth-first search due to the attractive, hardware-friendly nature of the approach. At each level only the K nodes with the smallest partial-Euclidean distance (Ti) are chosen for expansion. This type of detector is called a K-best detector.

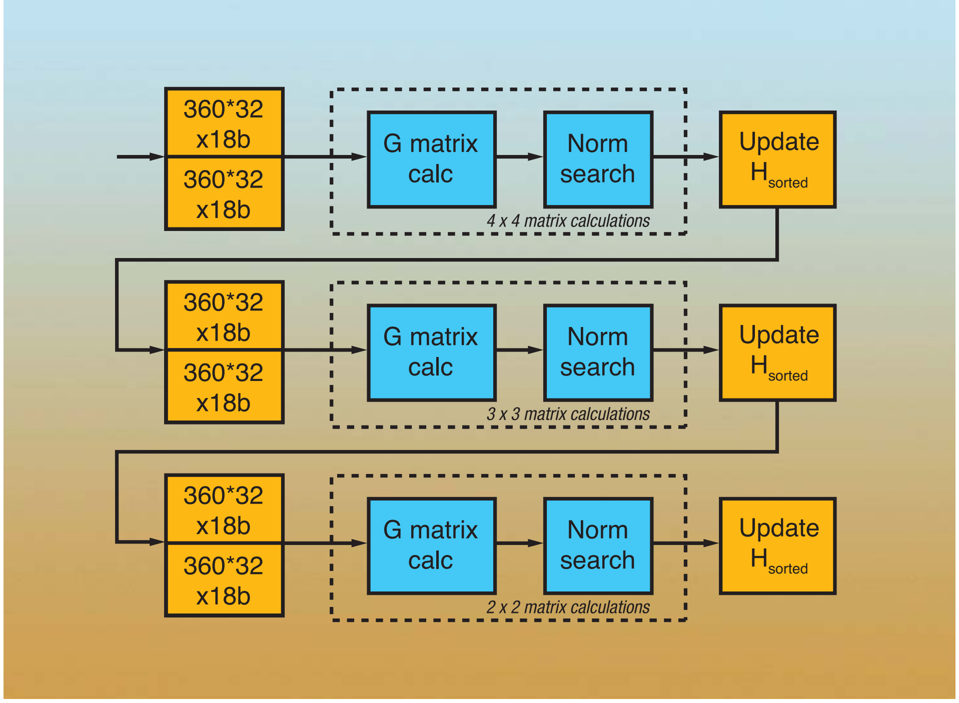


Figure 2 – Iterative channel matrix reordering algorithm

The norm computation is done in the partial Euclidean distance (PED) blocks of the sphere detector. Depending on the level of the tree, three different PED blocks are used. The root-node PED block calculates all possible PEDs (tree-level index is i = M = 8). The second-level PED block computes eight possible PEDs for each of the eight survivor paths generated in the previous level. This will give us 64 generated PEDs for the tree-level index i = 7. The third type of PED block is used for all other tree levels that compute the closest-node PED for all PEDs computed on the previous level. This will fix the number of branches on each level to K = 64, thus propagating to the last level i = 1 and producing 64 final PEDs along with their detected symbol sequences.

The pipeline architecture of the SD allows data processing on every clock cycle. Thus, only one PED block is necessary at every tree level. The total number of PED units is equal to the number of tree levels, which for 4x4 64-QAM modulation is eight. Figure 3 illustrates the block diagram of the SD.

FPGA PERFORMANCE IMPLEMENTATION TARGETS

The target FPGA device is a Xilinx Virtex-5, with a target clock frequency of 225 MHz. The channel matrix is estimated for every data subcarrier, which limits the available processing time for every channel matrix. For the selected clock frequency and a communication bandwidth of 5 MHz (corresponding to 360 data subcarriers in a WiMAX system), we calculated the available number of processing clock cycles per channel matrix interval as follows:

$$num\_cycles = \frac{(102.9\mu s/360)}{1/225MHz} \cong 64$$

As mentioned earlier, we designed the most computationally demanding configuration with 4x4 antennas and a

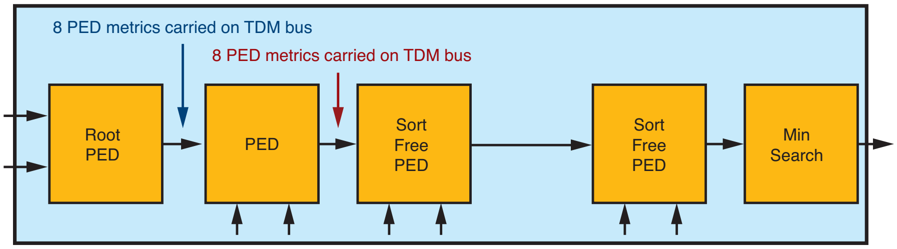

8 PED metrics carried on TDM bus
8 PED metrics carried on TDM bus
Root PED
PED
Sort Free PED
Sort Free PED
Min Search

Figure 3 – Sphere detector processing pipeline

64-QAM modulation scheme. The achievable raw data rate in this case is 83.965 Mbits per second.

HIGH-LEVEL SYNTHESIS FOR FPGAS

High-level synthesis tools take as their input a high-level description of the specific algorithm to implement and generate an RTL description for FPGA implementation, as shown in Figure 4. This RTL description can be integrated with a reference design, IP core or existing RTL code to create a complete FPGA implementation using traditional Xilinx ISE/EDK tools.

Modern high-level synthesis tools accept untimed C/C++ descriptions as input specifications. These tools give two interpretations to the same C/C++ code: sequential semantics for input/output behavior, and architecture specification based on C/C++ code and compiler directives. Based on the C/C++ code, compiler directives and target throughput requirements, these high-level synthesis tools generate high-performance pipelined architectures. Among other features, high-level synthesis tools enable automatic insertion of pipeline stages and resource sharing to reduce FPGA resource utilization. Essentially, high-level synthesis tools raise the level of abstraction for FPGA design, and make transparent the time-consuming and error-prone RTL design tasks.

We have focused on using C++ descriptions, with the goal of leveraging C++ template classes to represent arbitrary precision integer types and template functions to represent parameterized blocks in the architecture.

The overall design approach is seen in Figure 5, where the starting point is a reference C/C++ code that could have been derived from a MATLAB® functional description. As the figure shows, the first step in implementing an application on any hardware target is often to restructure the reference C/C++ code. By "restructuring," we mean rewriting the initial C/C++ code (which is typically coded for clarity and ease of conceptual understanding rather than for optimized performance) into a format more suitable for the target processing engine. For example, on a DSP processor it may be necessary to rearrange an application's code so that the algorithm makes efficient use of the cache memories. When targeting FPGAs, this restructuring might involve, for example, rewriting the code so it represents an architecture specification that can achieve the desired throughput, or rewriting the code to make efficient use of the specific FPGA features, like embedded DSP macros.

We achieved the functional verification of this implementation C/C++ code using traditional C/C++ compilers (for example, gcc) and reusing C/C++ level testbenches developed for the verification of the reference C/C++ code. The implementation C/C++ code is the main input to the high-level synthesis tools. However, there are additional inputs that significantly influence the generated hardware, its performance and the number of FPGA resources used. Two essential constraints are the target FPGA family and target clock frequency, both of which affect the number of pipeline stages in the generated architecture. Additionally, high-level synthesis tools accept compiler directives (e.g., pragmas inserted in the C/C++ code). The designer can apply different types of directives to different sections of the C/C++ code. For example, there are directives that are applied to loops (e.g., loop unrolling), and others to arrays (for example, to specify which FPGA resource must be used to the implementation of the array).

Based on all these inputs, the high-level synthesis tools generate an output architecture (RTL) and report its throughput. Depending on this throughput, the designer can then modify the directives, the implementation C/C++ code or both. If the generated architecture meets the required throughput, then the output RTL is used as the input to the FPGA implementation tools (ISE/EDK). The final achievable clock frequency and number of FPGA resources used are reported only after running logic synthesis and place-and-route. If the design does not meet timing or the FPGA resources are not the expected ones, the designer should modify the implementation C/C++ code or the compiler directives.

HIGH-LEVEL SYNTHESIS IMPLEMENTATION OF SD

We have implemented the three key building blocks of the WiMAX sphere decoder shown in Figure 1 using the AutoPilot 2010.07.ft tool from AutoESL. It is important to emphasize that the algorithm is exactly the algorithm described in a recent SDR Conference paper, [4] and hence has exactly the same BER. In this section we give specific examples of code rewriting and compiler directives that we used for this particular implementation.

The original reference C code, derived from a MATLAB functional description, contained approximately 2,000 lines of code, including synthesizable and verification C code. It contains only fixed-point arithmetic using C built-in data types. An FPGA-friendly implementation approximated all the required floating-point operations (for example, sqrt).

In addition to the reference C code describing the functions to synthesize in the FPGA, there is a complete C-level verification testbench. We generated the input test vectors as well as the golden output reference files from the MATLAB description. The original C/C++ reference code is bit-accurate with the MATLAB specification, and passes the entire regression suite consisting of multiple data sets.

This reference C/C++ code has gone through different types of code restructuring. As examples, Figure 5 shows three instances of code restructuring that we have implemented. We reused the C-level verification infrastructure to verify any change to the implementation C/C++ code. Moreover, we carried out all verification at the C level, not at the register-transfer level, avoiding time-consuming RTL simulations and hence, contributing to the reduction in the overall development time.

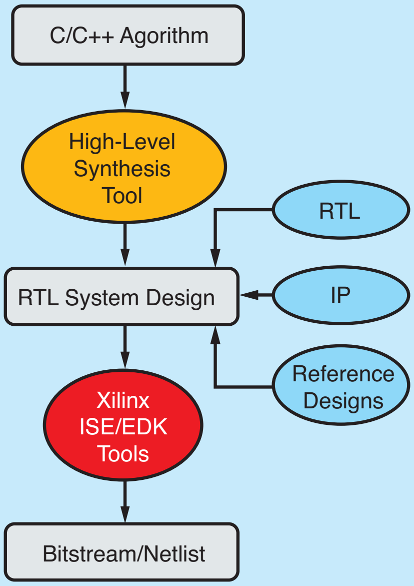


Figure 4 – High-level synthesis for FPGAs

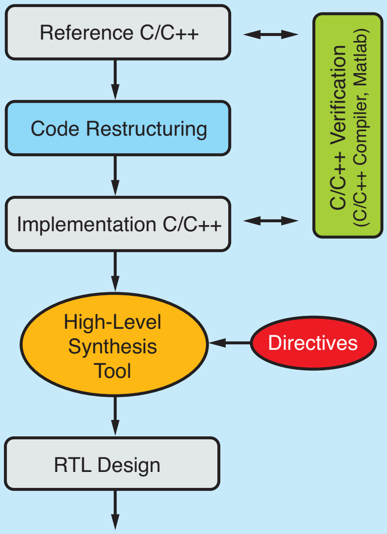


Figure 5 -- Iterative C/C++ refinement design approach

MACROARCHITECTURE SPECIFICATION

Probably the most important part of code refactoring is to rewrite the C/C++ code to describe a macroarchitecture that would efficiently implement a specific functionality. In other words, the designer is accountable for the macroarchitecture specification, while the high-level synthesis tools are in charge of the microarchitecture generation. This type of code restructuring has a major impact on the obtained throughput and quality of results.

In the case of our sphere decoder, there are several instances of this type of code restructuring. For example, to meet the high throughput of the channel-ordering block, the designer should describe in C/C++ the macroarchitecture shown in Figure 2. Such C/C++ code would consist of several function calls communicating using arrays. The high-level synthesis tools might automatically translate these arrays in ping-pong buffers to allow parallel execution of the several matrix calculation blocks in the pipeline.

Another example of code restructuring at this level would be to decide how many channels to employ in the TDM structure of a specific block (e.g., five channels in the channel matrix reordering block, or 15 channels in the modified real-valued QR decomposition block).

Figure 6 is one example of macroarchitecture specification. This snippet of C++ code describes the sphere detector block diagram shown in Figure 3. We can observe a pipeline of nine function calls, each one representing a block as shown in Figure 3. The communication between functions takes place through arrays, which are mapped to streaming inter-

```
1: void  sphere_detector_top   (…)  {
2:   #pragma  AP DATAFLOW
3:   // PED streams between pipeline blocks
4:   ap_int <18>     PED_7[RVD_MODULATION  ];
5:   #pragma  AP ARRAY_STREAM  variable=PED_7  depth=1 stream
6:   ap_int <4>      symb_7[RVD_MODULATION  ];
7:   #pragma  AP ARRAY_STREAM  variable=symb_7  depth=1 stream
8:   main_label :{
9:     RootPED (r_7 , y_7 , ..., symb_7);
10:    PED(r_6 , y_6 , ..., symb_7, symb_6 );
11:    SortFreePED <5,2>(r_5 , y_5 , ..., symb_6 , symb_5 );
12:    SortFreePED <4,3>(r_4 , y_4 , ..., symb_5 , symb_4 );
13:    SortFreePED <3,4>(r_3 , y_3 , ..., symb_4 , symb_3 );
14:    SortFreePED <2,5>(r_2 , y_2 , ..., symb_3 , symb_2 );
15:    SortFreePED <1,6>(r_1 , y_1 , ..., symb_2 , symb_1 );
16:    SortFreePED <0,7>(r_0 , y_0 , ..., symb_1 , symb_0 );
17:    // find minimum PED
18:    min_finder (PED_0 , symb_0 , min_PED , min_symb_list );
19:   }
20:}
```

Figure 6 – Sphere detector macroarchitecture description

```
1:   template< int X_DIMENSION,  int Y_DIMENSION,  int MM_II >
2:   void  matrix_multiply _(...)  {
3:   #pragma  AP ARRAY_PARTITION variable=  chunk_in_re  dim=1
4:   #pragma  AP ARRAY_PARTITION variable=  chunk_in_im  dim=1
5:     // matrix multiplication of a   A'*A matrix
6:    for  (index_a  = 0; index_a  < TDM_CHUNKS;  index_a ++) {
7:      for (index_b  = 0; index_b  < X_DIMENSION;  index_b ++) {
8:        for (index_c  = 0; index_c  < Y_DIMENSION;  index_c ++)  {
9:   #pragma  AP PIPELINE II = MM_II
10:                  <loop body >
11:    } } }
12: }
```

Figure 7 – Example of code parameterization

```
1:  template  <int Wa, int Wb, int Wc>
2:  ap_int <36> SUBMUL( ap_int <Wa> a,  ap_int <Wb> b,  ap_int <Wc> c) {
3:    #pragma  AP LATENCY max=2
4:    #pragma  AP INTERFACE  ap_none  port=return register
5:
6:    ap_int <36> c_36 = c; // sign extension
7:   return  c_36 -a*b;
8:  }
```

Figure 8 – FPGA optimization for DSP48 utilization

faces (not embedded BRAM memories in the FPGA) by using the appropriate directives (pragmas) in lines 5 and 7.

IMPORTANCE OF PARAMETERIZATION

Parameterization is another key example of code rewriting. We have extensively leveraged C++ template functions to represent parameterized modules in the architecture.

In the implementation of the sphere decoder, there are several cases of this type of code rewriting. A specific example would be the different matrix operations used in the channel-reordering block. The matrix calculations blocks (4x4, 3x3 and 2x2) shown in Figure 2 use different types of matrix operations, such as Matrix Inverse or Matrix Multiply. These blocks are coded as C++ template functions with the dimensions of the matrix as template parameters.

Figure 7 shows the C++ template function for Matrix Multiply. In addition to the matrix dimension, this template function has a third parameter, MM_II (Initiation Interval for Matrix Multiply), which is used to specify the number of clock cycles between two consecutive loop iterations. The directive (pragma) in line 9 is used to parameterize the required throughput for a specific instance. This is a really important feature, since it has a major impact on the generated microarchitecture—that is, the ability of the high-level synthesis tools to exploit resource sharing, and hence, to reduce the FPGA resources the implementation will use. For example, just by modifying this Initiation Interval parameter and using exactly the same C++ code, the high-level synthesis tools automatically achieve different levels of resource sharing in the implementation of the different Matrix Inverse (4x4, 3x3, 2x2) blocks.

FPGA OPTIMIZATIONS

FPGA optimization is the last example of code rewriting. The designer can rewrite the C/C++ code to more

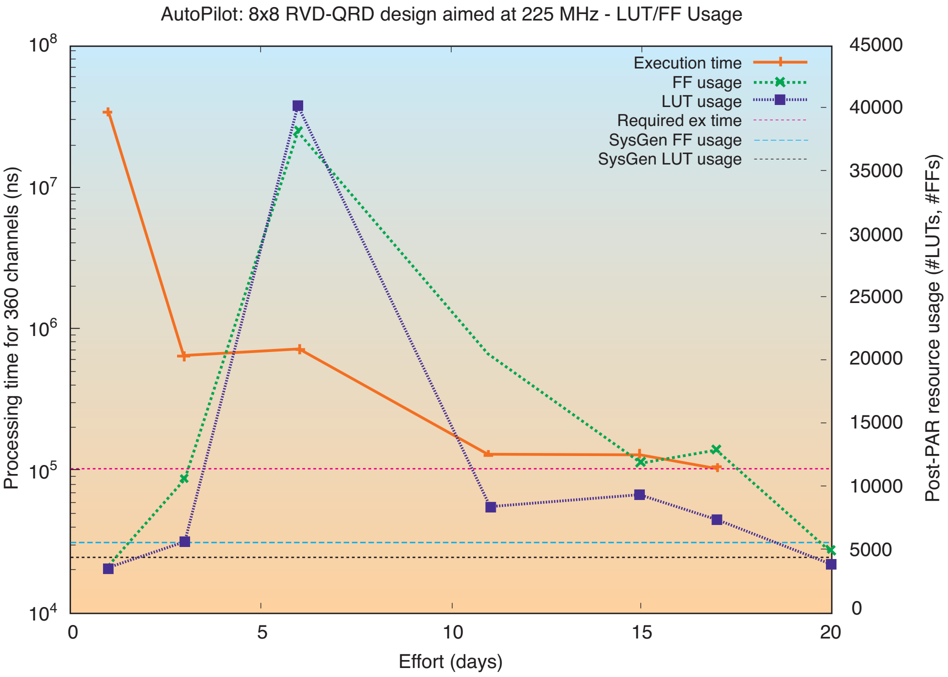


Figure 9 – Reduction of FPGA resources over development time

efficiently utilize specific FPGA resources, and hence, improve timing and reduce area. Two very specific examples of this type of optimization are bit-widths optimizations and efficient use of embedded DSP blocks (DSP48s). Efficient use of DSP48s improves timing and FPGA resource utilization.

We wrote our reference C/C++ code using built-in C/C++ data types (e.g., short, int), while the design uses 18-bit fixed-point data types to represent the matrix elements. We have leveraged C++ template classes to represent arbitrary precision fixed-point data types, hence reducing FPGA resources and minimizing the impact on timing.

Figure 8 is a C++ template function that implements a multiplication followed by a subtraction, where the width of the input operands is parameterized. You can map these two arithmetic operations into a single embedded DSP48 block. In Figure 8, we can also observe two directives that instruct the high-level synthesis tool to use a maximum of two cycles to schedule these operations and use a register for the output return value.

PRODUCTIVITY METRICS

In Figure 9 we plot how the size of the design (that is, the FPGA resources) generated using AutoESL's AutoPilot evolves over time, and compare it with a traditional SystemGenerator (RTL) implementation. With high-level synthesis tools we are able to implement many valid solutions that differ in size over time. Depending on the amount of code restructuring, the designer can trade off how fast to get a solution vs. the size of that solution. On the other hand, there is only one RTL solution, and it requires a long development time.

We have observed that it takes relatively little time to obtain several synthesis solutions that use significantly more FPGA resources (i.e., area) than the traditional RTL solution. On the other hand, the designer might decide to work at the tools' Expert level and generate many more solutions by implementing more advanced C/C++ code-restructuring techniques (such as FPGA-specific optimizations) to reduce FPGA resource utilization.

Finally, since we carried out all verification at the C/C++ level, we avoided the time-consuming RTL simulations. We found that doing the design verification at the C/C++ level significantly contributed to the reduction in the overall development time.

QUALITY OF RESULTS

In Figure 10, we compare final FPGA resource utilization and overall development time for the complete sphere decoder implemented using high-level synthesis tools and the reference System Generator implementation, which is basically a structural RTL design, explicitly instantiating FPGA primitives such as DSP48 blocks. The development time for AutoESL includes learning the tool, producing results, design-space exploration and detailed verification.

Metric	SysGen Expert	AutoESL Expert	% Diff
Development Time (weeks)	16.5	15	-9%
LUTs	27,870	29,060	+4%
Registers	42,035	31, 000	-26%
DSP48s	237	201	-15%
18K BRAM	138	99	-28%

Figure 10 – Quality-of-results metrics give AutoESL the edge.

To have accurate comparisons, we have reimplemented the reference RTL design using the latest Xilinx ISE 12.1 tools targeting a Virtex-5 FPGA. Likewise, we implemented the RTL generated by AutoESL's AutoPilot using ISE 12.1 targeting the same FPGA. Figure 10 shows that AutoESL's AutoPilot achieves significant savings in FPGA resources, mainly through resource sharing in the implementation of the matrix inverse blocks. We can also observe a significant reduction in the number of registers and a slightly higher utilization of lookup tables (LUTs). This result is partially due to the fact that delay lines are mapped onto SRL16s (i.e., LUTs) in the AutoESL implementation, while the SystemGenerator version implements them using registers. In other modules we traded off BRAMs for LUTRAM, resulting in lower BRAM usage in the channel preprocessor.

AutoESL's AutoPilot achieves significant abstractions from low-level FPGA implementation details (e.g., timing and pipeline design), while producing a quality of results highly competitive with those obtained using a traditional RTL design approach. C/C++ level verification contributes to the reduction in the overall development time by avoiding time-consuming RTL simulations. However, obtaining excellent results for complex and challenging designs requires good macroarchitecture definition and a solid knowledge of FPGA design tools, including the ability to understand and interpret FPGA tool reports.

REFERENCES

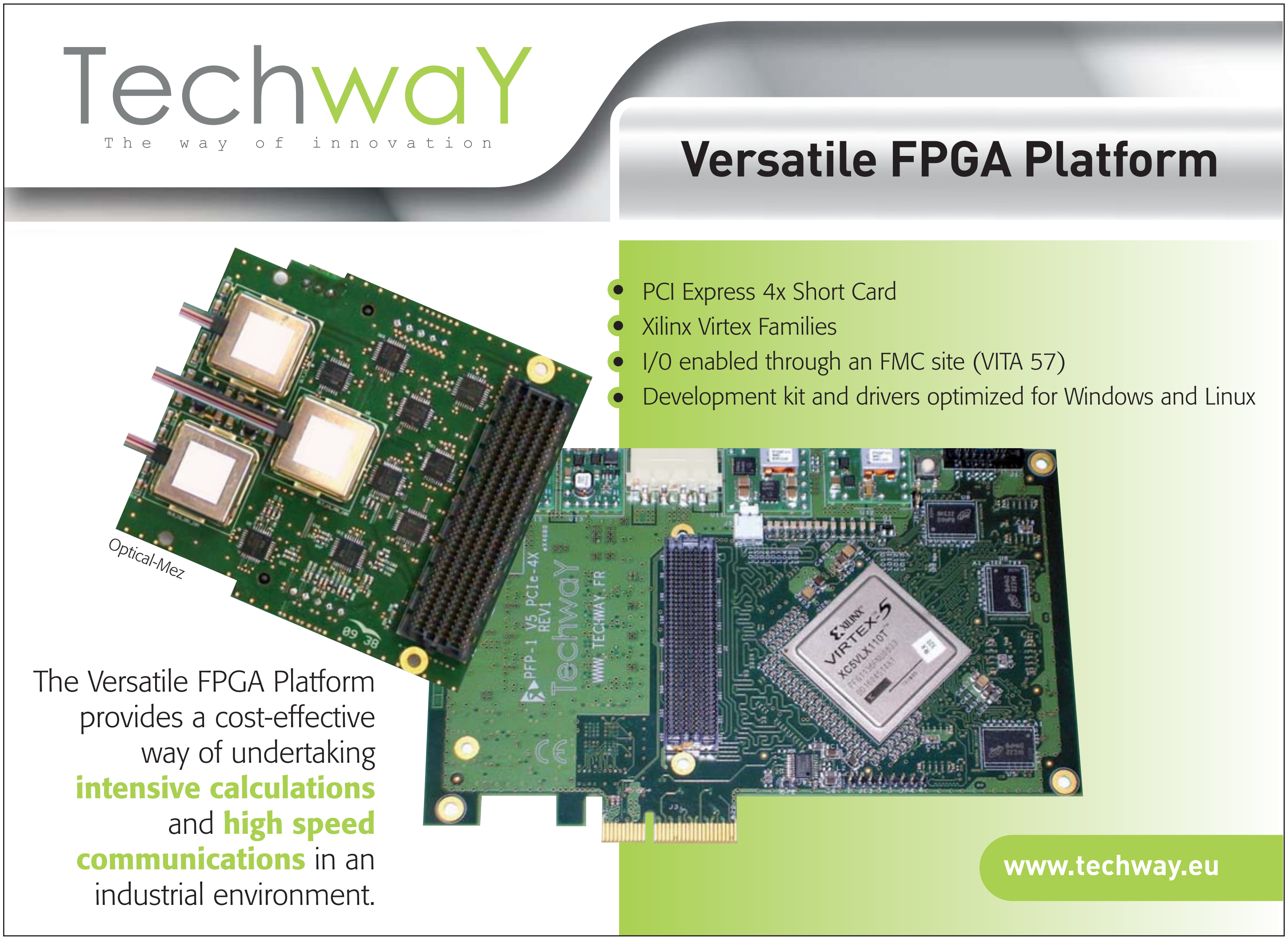
[1] *Berkeley Design Technology Inc., "FPGAs for DSP," white paper, 2007.*

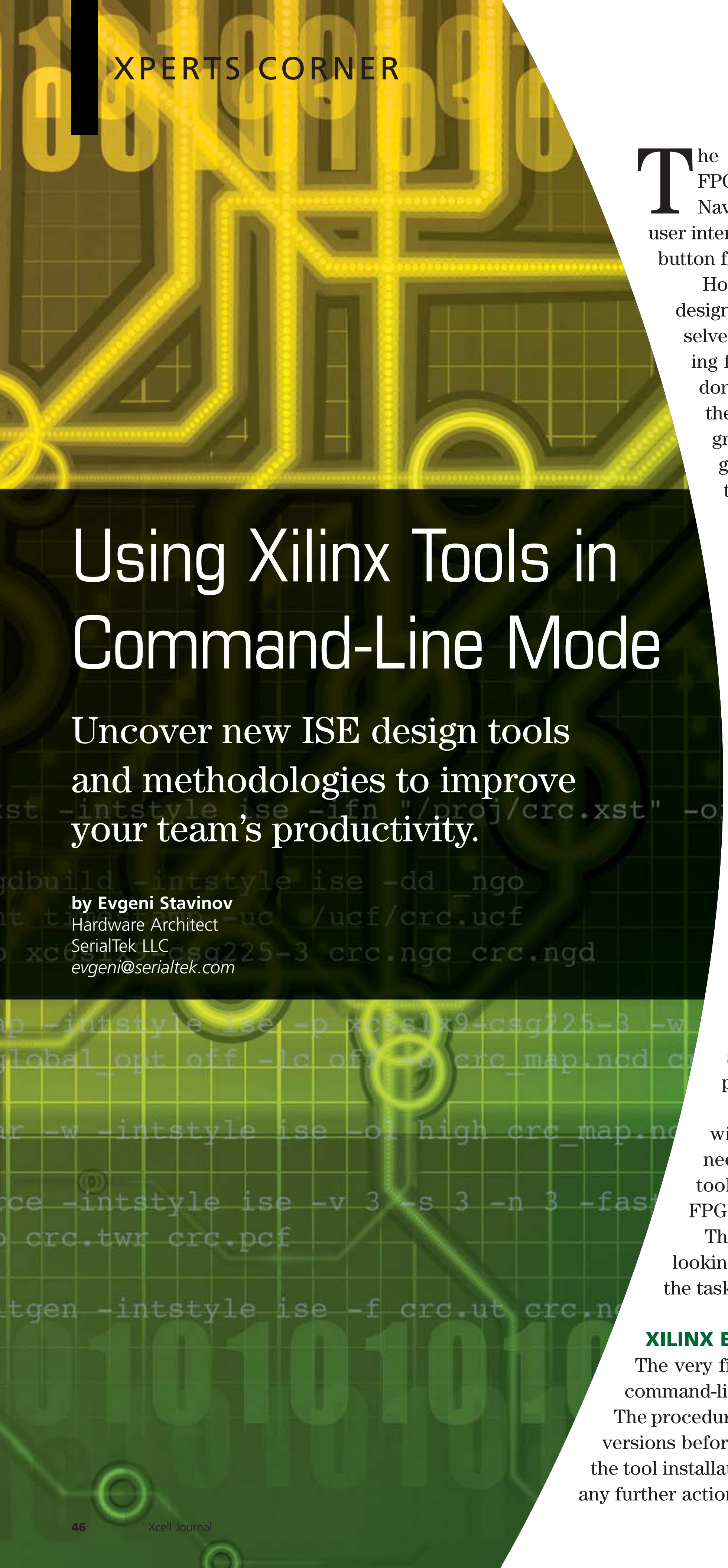
[2] *Grant Martin, Gary Smith, "High-Level Synthesis: Past, Present and Future,"* IEEE Design and Test of Computers, *July/August 2009.*

[3] *Berkeley Design Technology Inc., "High-Level Synthesis Tools for Xilinx FPGAs," white paper, 2010:* http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf.

[4] *Chris Dick et al., "FPGA Implementation of a Near-ML Sphere Detector for 802.16E Broadband Wireless Systems," SDR Conference'09, December 2009.*

[5] *K. Denolf, S. Neuendorffer, K. Vissers, "Using C-to-Gates to Program Streaming Image Processing Kernels Efficiently on FPGAs," FPL'09 conference, September 2009.*




st -intstyle ise -ifn "/proj/crc.xst" -o
gdbuild -intstyle ise -dd _ngo
nt -uc /ucf/crc.ucf
xc6slx9-csg225-3 crc.ngc crc.ngd
ap -intstyle ise -p xc6slx9-csg225-3 -w
global_opt off -lc off -o crc_map.ncd c
ar -w -intstyle ise -ol high crc_map.nc
ce -intstyle ise -v 3 -s 3 -n 3 -fast
crc.twr crc.pcf
tgen -intstyle ise -f crc.ut crc.nc

Using Xilinx Tools in Command-Line Mode

Uncover new ISE design tools and methodologies to improve your team's productivity.

by Evgeni Stavinov
Hardware Architect
SerialTek LLC
evgeni@serialtek.com

The majority of designers working with Xilinx® FPGAs use the primary design tools—ISE® Project Navigator and PlanAhead™ software—in graphical user interface mode. The GUI approach provides a push-button flow, which is convenient for small projects.

However, as FPGAs become larger, so do the designs built around them and the design teams themselves. In many cases GUI tools can become a limiting factor that hinders team productivity. GUI tools don't provide sufficient flexibility and control over the build process, and they don't allow easy integration with other tools. A good example is integration with popular build-management and continuous-integration solutions, such as TeamCity, Hudson CI and CruiseControl, which many design teams use ubiquitously for automated software builds.

Nor do GUI tools provide good support for a distributed computing environment. It can take several hours or even a day to build a large FPGA design. To improve the runtime, users do builds on dedicated servers, typically 64-bit multicore Linux machines that have a lot of memory but in many cases lack a GUI. Third-party job-scheduling solutions exist, such as Platform LSF, to provide flexible build scheduling, load balancing and fine-grained control.

A less obvious reason to avoid GUI tools is memory and CPU resource utilization. ISE Project Navigator and PlanAhead are memory-hungry applications—each GUI tool instance uses more than 100 Mbytes of RAM. On a typical workstation that has 2 Gbytes of memory, that's 5 percent—a substantial amount of a commodity that can be put to a better use.

Finally, many Xilinx users come to FPGAs with an ASIC design background. These engineers are accustomed to using command-line tool flows, and want to use similar flows in their FPGA designs.

These are some of the reasons why designers are looking to switch to command-line mode for some of the tasks in a design cycle.

XILINX ENVIRONMENT VARIABLES

The very first task a user encounters while working with command-line tools is setting up environment variables. The procedure varies depending on the Xilinx ISE version. In versions before 12.x, all environment variables are set during the tool installation. Users can run command-line tools without any further action.

That changed in ISE 12.x. Users now need to set all the environment variables every time before running the tools. The main reason for the change is to allow multiple ISE versions installed on the same machine to coexist by limiting the scope of the environmental variables to the local shell or command-line terminal, depending on the operating system.

There are two ways of setting up environment variables in ISE 12.x. The simpler method is to call a script settings32.{bat,sh,csh} or settings64.{bat,sh,csh}, depending on the platform. The default location on Windows machines is C:\Xilinx\12.1\ISE_DS\; on Linux, it's /opt/Xilinx/12.1/ISE_DS/.

Here is a Linux bash example of calling the script:

```
$ source /opt/Xilinx/12.1/ISE_DS/settings64.sh
```

The other option is to set the environment variables directly, as shown in the following chart:

Windows	`>set XILINX= C:\Xilinx\12.1\ISE_DS\ISE`
	`>set XILINX_DSP=%XILINX%`
	`>set PATH=%XILINX%\bin\nt;%XILINX%\lib\nt;%PATH%`

Linux bash	`$ export XILINX=/opt/Xilinx/12.1/ISE_DS/ISE`
	`$export XILINX_DSP=$XILINX`
	`$ export PATH=${XILINX}/bin/lin64:${XILINX}/sysgen/util:${PATH}`

Linux csh	`% setenv XILINX/opt/Xilinx/12.1/ISE_DS/ISE`
	`% setenv XILINX_DSP    $XILINX`
	`% setenv PATH  ${XILINX}/bin/lin64:${XILINX}/sysgen/util:${PATH}`

Xilinx command-line tools can run on both Windows and Linux operating systems, and can be invoked using a broad range of scripting languages, as Table 1 illustrates. Aside from these choices, other scripts known to run Xilinx tools are Ruby and Python.

MANY CHOICES IN SCRIPTING LANGUAGES

Perl	Perl is a popular scripting language used by a wide range of EDA and other tools. Xilinx ISE installation contains a customized Perl distribution. Users can start the Perl shell by running xilperl command: `$ xilperl –v # display Perl version`
TCL	Tool Command Language (TCL) is a de facto standard scripting language of ASIC and FPGA design tools. TCL is very different syntactically from other scripting languages, and many developers find it difficult to get used to. This might be one of the reasons TCL is less popular than Perl. TCL is good for what it's designed for—namely, writing tool command scripts. TCL is widely available, has excellent documentation and enjoys good community support. Xilinx ISE installation comes with a customized TCL distribution. To start the TCL shell use xtclsh command: `$ xtclsh –v # display TCL version`
Unix bash and csh	There are several Linux and Unix shell flavors. The most popular are bash and csh. Unlike other Unix/Linux shells, csh boasts a C-like scripting language. However, bash scripting language offers more features, such as shell functions, command-line editing, signal traps and process handling. Bash is a default shell in most modern Linux distributions, and is gradually replacing csh.
Windows batch and PowerShell	Windows users have two scripting-language choices: DOS command line and the more flexible PowerShell. An example of the XST invocation from DOS command line is: `>xst -intstyle ise -ifn "crc.xst" –ofn "crc.syr"`

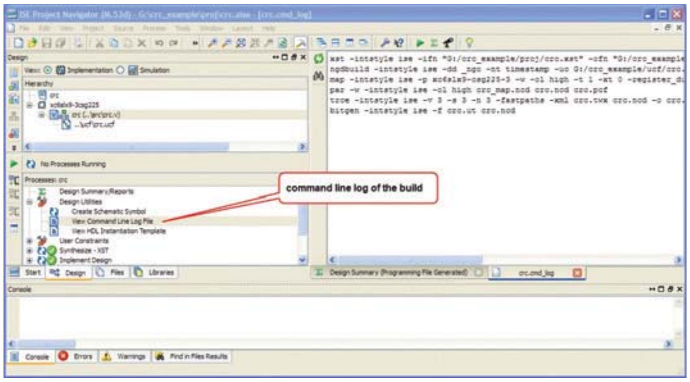
Table 1 – The most popular scripting languages are Perl, TCL, Unix/Linux and Windows batch and PowerShell.

BUILD FLOWS

Xilinx provides several options to build a design using command-line tools. Let's take a closer look at the four most popular ones: direct invocation, xflow, xtclsh and PlanAhead. We'll use a simple CRC generator project as an example.

The direct-invocation method calls tools in the following sequence: XST (or other synthesis tool), NGDBuild, map, place-and-route (PAR), TRACE (optional) and BitGen.

You can autogenerate the sequence script from the ISE Project Navigator by choosing the **Design -> View command line log file**, as shown in Figure 1.



```
xst -intstyle ise -ifn "G:/crc_example/proj/crc.xst" -ofn "G:/crc_example
ngdbuild -intstyle ise -dd _ngo -nt timestamp -uc G:/crc_example/ucf/crc.
map -intstyle ise -p xc6slx9-csg225-3 -w -ol high -t 1 -xt 0 -register_du
par -w -intstyle ise -ol high crc_map.ncd crc.ncd crc.pcf
trce -intstyle ise -v 3 -s 3 -n 3 -fastpaths -xml crc.twx crc.ncd -o crc.
bitgen -intstyle ise -f crc.ut crc.ncd
```

Figure 1 – Direct tool invocation sequence

The following script is an example of building a CRC generator project:

```
xst -intstyle ise -ifn "/proj/crc.xst" -ofn
"/proj/crc.syr"

ngdbuild -intstyle ise -dd _ngo
-nt timestamp -uc  /ucf/crc.ucf
-p xc6slx9-csg225-3 crc.ngc crc.ngd

map -intstyle ise -p xc6slx9-csg225-3 -w -ol high
-t 1 -xt 0
-global_opt off -lc off -o crc_map.ncd crc.ngd
crc.pcf

par -w -intstyle ise -ol high crc_map.ncd crc.ncd
crc.pcf

trce -intstyle ise -v 3 -s 3 -n 3 -fastpaths -xml
crc.twx crc.ncd
-o crc.twr crc.pcf

bitgen -intstyle ise -f crc.ut crc.ncd
```

EASIER TO USE

Xilinx's XFLOW utility provides another way to build a design. It's more integrated than direct invocation, doesn't require as much tool knowledge and is easier to use. For example, XFLOW does not require exit-code checking to determine a pass/fail condition.

The XFLOW utility accepts a script that describes build options. Depending on those options, XFLOW can run synthesis, implementation, BitGen or a combination of all three.

Synthesis only:

```
xflow -p xc6slx9-csg225-3 -synth synth.opt
../src/crc.v
```

Implementation:

```
xflow -p xc6slx9-csg225-3 -implement impl.opt
../crc.ngc
```

Implementation and BitGen:

```
xflow -p xc6slx9-csg225-3 -implement impl.opt -config
bitgen.opt ../crc.ngc
```

You can generate the XFLOW script (see Figure 2) manually or by using one of the templates located in the ISE installation at the following location: $XILINX\ISE_DS\ISE\xilinx\data\. The latest ISE software doesn't provide an option to autogenerate XFLOW scripts.

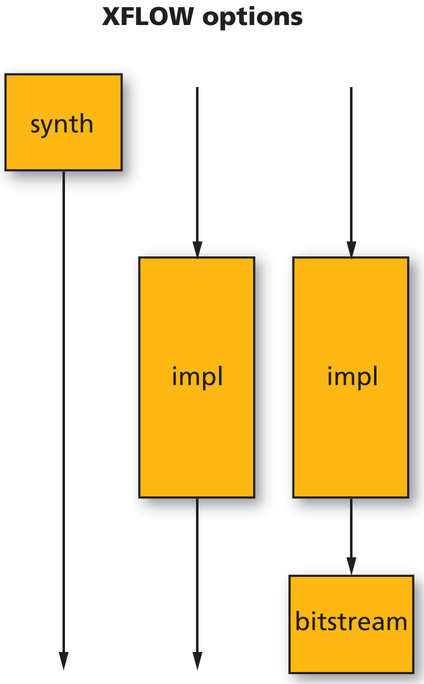


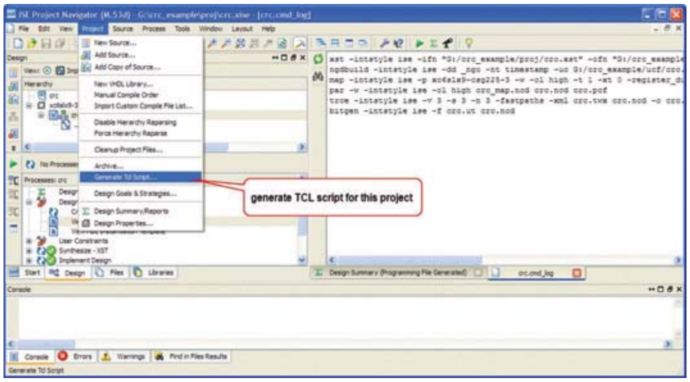
Figure 2 – The XFLOW utility is more integrated than direct invocation.

HOW TO USE XTCLSH

You can also build a design using TCL script invoked from the Xilinx xtclsh, as follows:

```
xtclsh crc.tcl rebuild_project
```

You can either write the TCL script, which passes as a parameter to xtclsh, manually or autogenerate it from the ISE ProjectNavigator by choosing **Project->Generate TCL Script** (see Figure 3).



```
xst -intstyle ise -ifn "G:/crc_example/proj/crc.xst" -ofn "G:/crc_example
ngdbuild -intstyle ise -dd _ngo -nt timestamp -uc G:/crc_example/ucf/crc.
map -intstyle ise -p xc6slx9-csg225-3 -w -ol high -t 1 -xt 0 -register_du
par -w -intstyle ise -ol high crc_map.ncd crc.ncd crc.pcf
trce -intstyle ise -v 3 -s 3 -n 3 -fastpaths -xml crc.twx crc.ncd -o crc.
bitgen -intstyle ise -f crc.ut crc.ncd
```

Figure 3 – TCL script generation

Xtclsh is the only build flow that accepts the original ISE project in .xise format as an input. All other flows require projects and file lists, such as .xst and .prj, derived from the original .xise project.

Each of the XST, NGDBuild, map, PAR, TRACE and BitGen tool options has its TCL-equivalent property. For example, the XST command-line equivalent of the Boolean "Add I/O Buffers" is –iobuf, while –fsm_style is the command-line version of "FSM Style" (list).

Each tool is invoked using the TCL "process run" command, as shown in Figure 4.

THE PLANAHEAD ADVANTAGE

An increasing number of Xilinx users are migrating from ISE Project Navigator and adopting PlanAhead as a main design tool. PlanAhead offers more build-related features, such as scheduling multiple concurrent builds, along with more flexible build options and project manipulation.

PlanAhead uses TCL as its main scripting language. TCL closely follows Synopsys' SDC semantics for tool-specific commands.

PlanAhead has two command-line modes: interactive shell and batch mode. To enter an interactive shell, type the following command:

```
PlanAhead –mode tcl
```

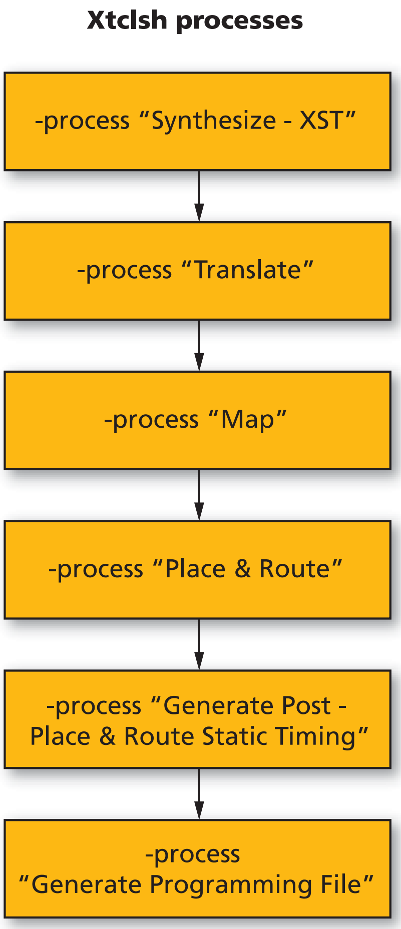

Xtclsh processes
-process "Synthesize - XST"
-process "Translate"
-process "Map"
-process "Place & Route"
-process "Generate Post - Place & Route Static Timing"
-process "Generate Programming File"

Figure 4 – Xtclsh is the only flow that accepts .xise input.

To run the entire TCL script in batch mode, source the script as shown below:

```
PlanAhead –mode tcl –source <script_name.tcl>
```

The PlanAhead build flow (or run) consists of three steps: synthesis, implementation and bitstream generation, as illustrated in Figure 5.

The PlanAhead software maintains its log of the operations in the *PlanAhead.jou* file. The file is in TCL format, and located in C:\Documents and Settings\<user name>\Application Data\HDI\ on Windows, and ~/.HDI/ on Linux machines.

Users can run the build in GUI mode first, and then copy some of the log commands in their command-line build scripts.

Below is a PlanAhead TCL script used to build an example CRC project.

```
create_project pa_proj {crc_example/pa_proj} -part
xc6slx9csg225-3
```

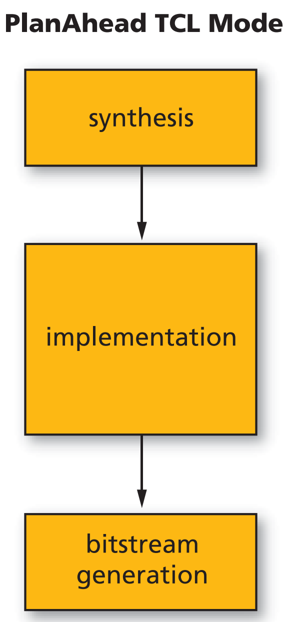


Figure 5 – PlanAhead offers many build-related features.

```
set_property design_mode RTL
  [get_property srcset
  [current_run]]

add_files -norecurse
  {crc_example/proj/.. /src/crc.v}

set_property library work
  [get_files -of_objects
  [get_property srcset [current_run]]
  {src/crc.v}]

add_files -fileset [get_property
  constrset [current_run]]
  -norecurse {ucf/crc.ucf}

set_property top crc [get_property
  srcset [current_run]]
  set_property verilog_2001 true
  [get_property srcset
  [current_run]]

launch_runs -runs synth_1 -jobs 1
  -scripts_only -dir
  {crc_example/pa_proj/pa_proj.runs}

launch_runs -runs synth_1 -jobs 1

launch_runs -runs impl_1 -jobs 1

set_property add_step Bitgen [get_runs impl_1]

launch_runs -runs impl_1 -jobs 1 -dir
  {crc_example/pa_proj/pa_proj.runs}
```

In addition to providing standard TCL scripting capabilities, PlanAhead offers other powerful features. It allows query and manipulation of the design database, settings and states, all from a TCL script. This simple script illustrates some of the advanced PlanAhead features that you can use in a command-line mode:

```
set my_port [get_ports rst]
report_property $my_port
get_property PULLUP $my_port
```

THE XILINX FPGA BUILD PROCESS

Before delving into the details of command-line flows, it's useful to briefly review the Xilinx FPGA build process (see figure), or the sequence of steps involved in building an FPGA design, from RTL to bitstream.

Xilinx provides two GUI tools with which to do a build: ISE Project Navigator and PlanAhead. There are several third-party tools that can do Xilinx FPGA builds as well, for example, Synopsys' Synplify product.

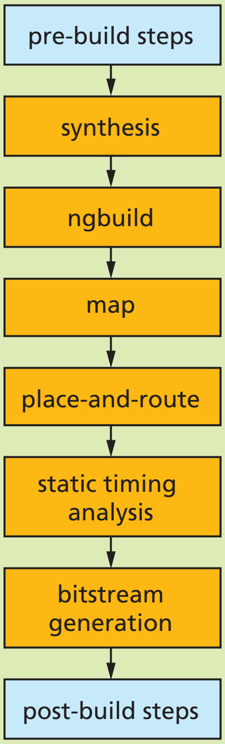
The exact build sequence will differ depending on the tool used. However, any Xilinx FPGA build will contain eight fundamental steps: pre-build, synthesis, NGDBuild, map, place-and-route, static timing analysis, BitGen and post-build.

The main pre-build tasks are getting the latest code from a source control repository, assembling project file lists, setting all tool environment variables, acquiring tool licenses, incrementing a build number and preprocessing the RTL.

During the synthesis stage, designers may use the Xilinx XST tool or a third-party offering. The major third-party synthesis tools for general-purpose FPGA design are Synopsys' Synplify and Mentor Graphics' Precision.

The NGDBuild phase involves netlist translation using the Xilinx NGDBuild tool, while the next step, map, involves mapping the netlist into FPGA-specific resources, such as slices, RAMs and I/Os, using the Xilinx MAP tool. Xilinx's PAR tool handles placement and routing, and the TRACE tool performs static timing analysis. Finally, the BitGen stage is the point in the design cycle at which FPGA bitstreams are generated.

Wrapping up the cycle, the post-build period involves tasks such as parsing the build reports, processing and archiving build results and sending e-mail notifications to users that own the build. – *Evgeni Stavinov*



The Xilinx FPGA build process involves a number of sequenced steps.

```
set_property PULLUP 1 $my_port
get_property PULLUP $my_port
```

The simple script adds a pull-up resistor to one of the design ports. Of course, you can do the same by changing UCF constraints and rebuilding the project, or by using the FPGA Editor. But PlanAhead can do it with just three lines of code.

Finally, many developers prefer using the *make* utility when doing FPGA builds. The two most important advantages of *make* are built-in checking of the return code and automatic dependency tracking. Any of the flows described above can work with *make*.

LESSER-KNOWN COMMAND-LINE TOOLS

The Xilinx ISE installation contains several lesser-known command-line tools that FPGA designers might find useful. ISE Project Navigator and PlanAhead invoke these tools behind the scenes as part of the build flow. The tools are either poorly documented or not documented at all, which limits their exposure to users.

Ten of these tools especially are worth investigating.

- **data2mem:** This utility, which is used to initialize the contents of a BRAM, doesn't require rerunning Xilinx implementation. It inserts the new BRAM data directly into a bitstream. Another data2mem usage is to split the initialization data of a large RAM consisting of several BRAM primitives into individual BRAMs.

- **fpga_edline:** A command-line version of the FPGA Editor, fpga_edline can be useful for applying changes to the post-PAR .ngc file from a script as part of the build process. Some of the use cases include adding ChipScope™ probes, minor routing changes, changing LUT or IOB properties.

- **mem_edit:** This is not a tool per se, but a script that opens Java applications. You can use it for simple memory content editing.

- **netgen:** Here is a tool you can use in many situations. It gets a Xilinx design file in .ncd format as an input, and produces a Xilinx-independent netlist for use in simulation, equivalence checking and static timing analysis, depending on the command-line options.

- **ngcbuild:** This is a tool that consolidates several design netlist files into one. It's mainly used for the convenience of working with a single design file rather than multiple designs and IP cores scattered around different directories.

- **obngc:** This is a utility that obfuscates .ngc files in order to hide confidential design features, and prevent design analysis and reverse-engineering. You can explore NGC files in FPGA Editor.

- **pin2ucf:** You can use this utility to generate pin-locking constraints from an NCD file in UCF format.

- **XDL:** An essential tool for power users, XDL has three fundamental modes: report device resource information, convert NCD to XDL and convert XDL to NCD. XDL is a text format that third-party tools can process. This is the only way to access post-place-and-route designs.

- **xreport:** This is a utility that lets you view build reports outside of the ISE Project Navigator software. It has table views of design summary, resource usage, hyperlinked warnings and errors, and message filtering.

- **xinfo:** A utility that reports system information, xinfo also details installed Xilinx software and IP cores, licenses, ISE preferences, relevant environment variables and other useful information.

The tools are installed in $XILINX/ISE/bin/{nt, nt64, lin, lin64} and $XILINX/common/bin/{nt, nt64, lin, lin64} directories. More inquisitive readers might want to further explore those directories to discover other interesting tools that can boost design productivity.

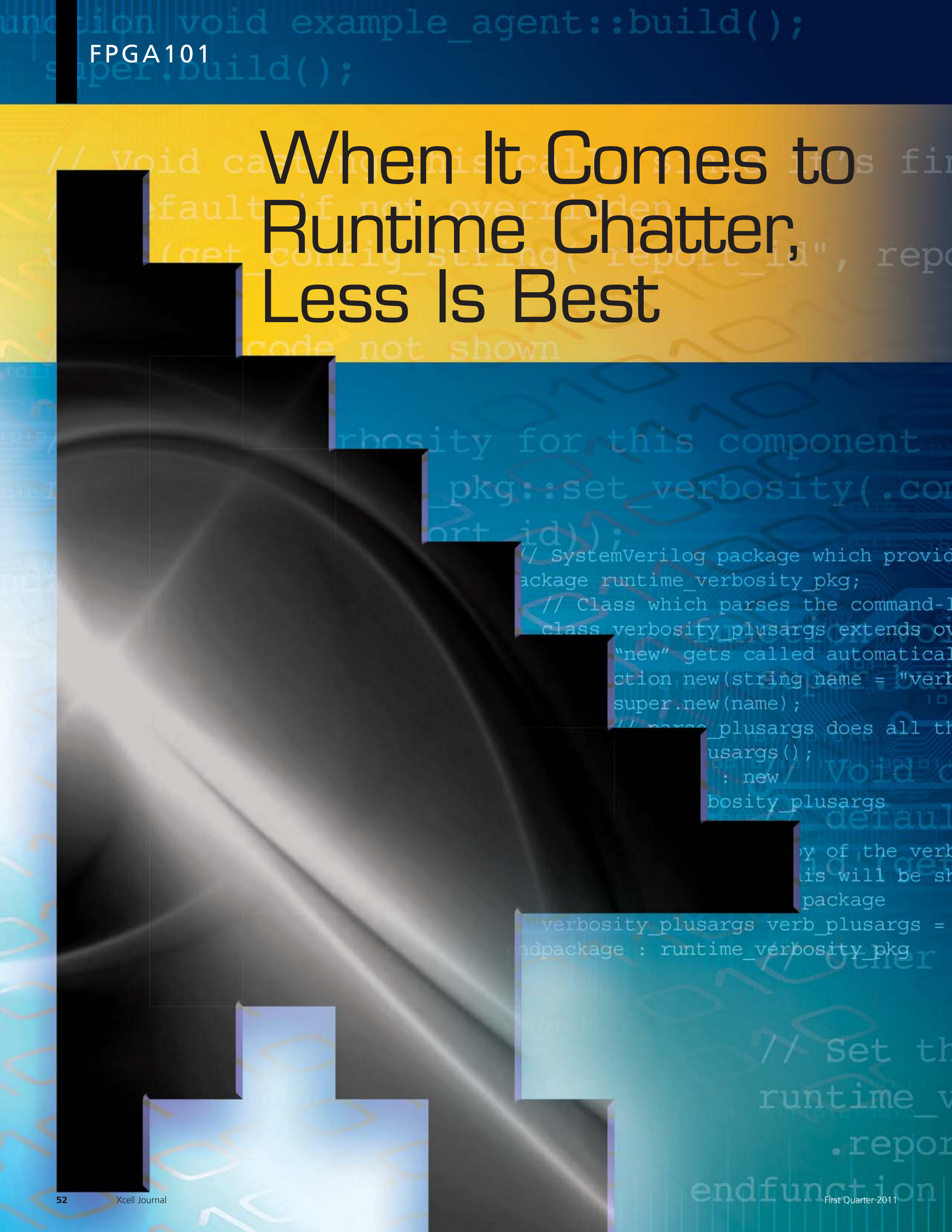
ASSESSING THE OPTIONS

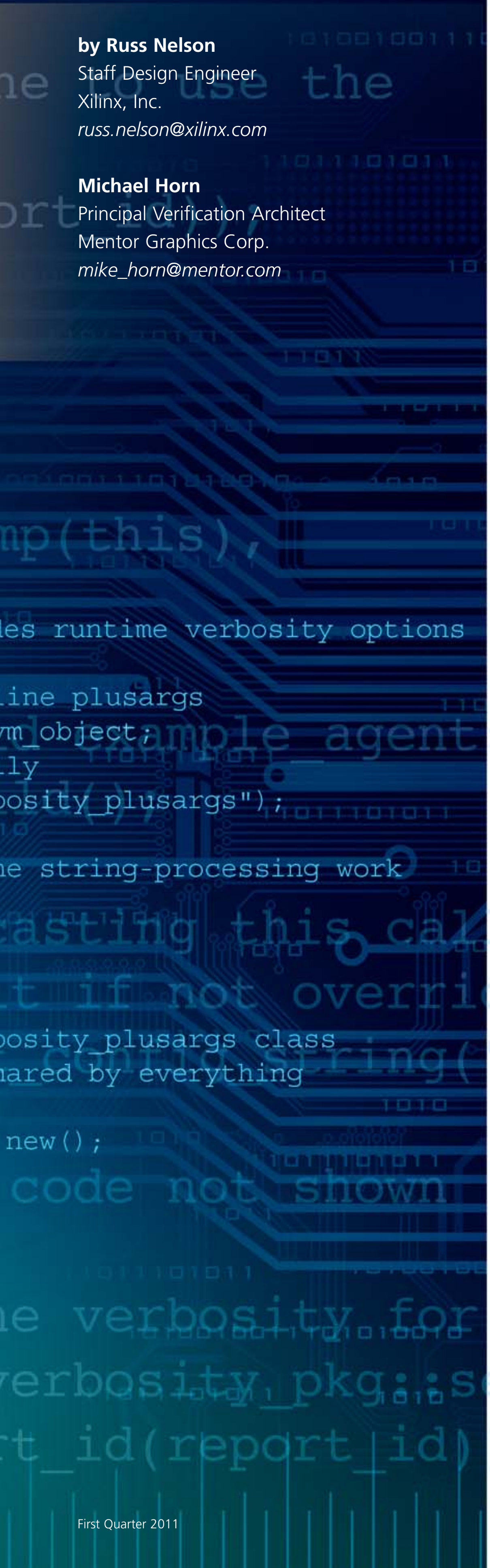
In this article we've discussed advantages of using Xilinx tools in command-line mode, explored several Xilinx build flows, examined different scripting languages and taken a closer look at lesser-known tools and utilities. The goal was not to identify the best build flow or script language, but rather to discuss the available options and parse the unique advantages and drawbacks of each to help Xilinx FPGA designers make an informed decision.

You can base your selection process on a number of criteria, such as existing project settings, design team experience and familiarity with the tools and scripts, ease of use, flexibility, expected level of customization and integration with other tools, among others. Users can find all scripts and example projects used in this article on the author's website: *http://outputlogic.com/xcell_using_xilinx_tools/.*

Evgeni Stavinov is a longtime Xilinx user with more than 10 years of diverse design experience. Before becoming a hardware architect at SerialTek LLC, he held different engineering positions at Xilinx, LeCroy and CATC. Evgeni is a creator of OutputLogic.com, an online productivity-tools portal.

When It Comes to Runtime Chatter, Less Is Best


unction void example_agent::build();
super.build();
code not shown
rbosity for this component
pkg::set_verbosity(.co
ort_id));
// SystemVerilog package which provid
ackage runtime_verbosity_pkg;
// Class which parses the command-l
class verbosity_plusargs extends ov
“new” gets called automatical
ction new(string name = "verb
super.new(name);
_plusargs does all th
usargs();
: new
bosity_plusargs
py of the verb
is will be sh
package
verbosity_plusargs verb_plusargs =
ndpackage : runtime_verbosity_pkg
// Set th
runtime_v
.repor
endfunction



by Russ Nelson
Staff Design Engineer
Xilinx, Inc.
russ.nelson@xilinx.com

Michael Horn
Principal Verification Architect
Mentor Graphics Corp.
mike_horn@mentor.com

Xilinx add-on package simplifies testbench debug by controlling verbosity on a component level.

The end goal of a testbench is to show that your design achieves its purpose without having any bugs. Getting to that point, however, takes work. You will have to find, diagnose and fix bugs, in both the testbench and the design. Anything that makes the debugging process easier and achieves closure on the testbench and design under test (DUT) is valuable—both in terms of time and money.

Controlling verbosity on a component level can save considerable time and effort when debugging failing tests by presenting only information relevant to the area where a failure is occurring. Controlling verbosity at runtime saves considerably more time by allowing engineers to focus on a problem area without having to edit and recompile the testbench code or sift through excessively verbose log files. While there is always the option of using "grep" or other command-line tools to post-process logs, having the right information in the log files to begin with saves effort.

Some testbench methodologies do a great job of controlling verbosity in a granular way, but they lack flexibility at runtime. This is the case with the Open Verification Methodology (OVM), the de facto industry testbench methodology standard.

The OVM provides a great framework for reporting debugging information. You can specify severity and verbosity levels for each reported message and implement different actions for various combinations of the two, specifying different verbosity thresholds and actions down to the ID. Moreover, during debug, OVM allows you to increase or decrease the verbosity threshold for the testbench as a whole. What OVM doesn't do is provide the ability to change verbosity for individual components at runtime.

To solve this runtime issue, we developed an add-on package at Xilinx, with additional direction from Mentor Graphics. When we employed it in a recent testbench environment, it saved us considerable time. Because of the advantages it delivers, we have adopted this package for several current designs at Xilinx. In fact, we almost always use runtime verbosity controls to zero in on problem areas. We have found that not having to recompile or sift through excessively verbose log files makes the designer's life a lot easier.

In this article, we will show the basic steps required to implement these verbosity controls. You can download the full methodology (including the standalone package, which you can easily add to existing testbenches) from *http://www.xilinx.com/publications/xcellonline/downloads/runtime-verbosity-package.zip*. Everything said here about OVM should apply equally to the Universal Verification Methodology, the next evolutionary step for testbench methodology. The initial UVM version is derived from the OVM with a few additions.

A FEW CHOICE WORDS MAKE THE POINT

Selectively increasing verbosity for a component or set of components is an easy and effective way of accelerating the time it takes to find bugs in a particular area of a verification testbench. Passing in directives via plusargs turns up the verbosity for components in the area of interest.

Take, for example, the following:

```
<sim_command> +OVM_VERBOSITY=OVM_LOW
+VERB_OVM_HIGH=TX_SCOREBOARD
```

In this command, OVM_VERBOSITY=OVM_LOW is the built-in OVM method for reducing the overall verbosity to a minimum, while VERB_OVM_HIGH=TX_SCOREBOARD is our command for boosting the TX scoreboard to a high level of verbosity.

Before adding any code to a testbench, it is important to define the methodology that you will use to identify each component. It is also helpful (but not necessary) to define what kind of information will be presented at each verbosity level—consistency makes the debugging process easier. (See the table on the final page for an example of how to plan the verbosity levels.) Once you have formulated a plan, there are two key testbench features required for enabling runtime verbosity controls. First, you must parse the command-line directives. Second, you must carry out the directives to change verbosity on a component-level basis.

> Before adding any code to a testbench, it is important to define the methodology that you will use to identify each component. It is also helpful (but not necessary) to define what kind of information will be presented at each verbosity level—consistency makes the debugging process easier.

The first step is to create a procedure for distinguishing one component from another; in other words, how you will identify the areas of interest. We recommend using a unique tag (we call it report_id) for each component, with a default value that you can override via the OVM configuration database. The following code snippet illustrates one way of implementing this functionality:

```
class example_agent extends ovm_agent;
   string report_id = "EXAMPLE_AGENT";
   // Other code not shown
endclass
```

```
function void example_agent::build();
   super.build();

   // Void casting this call, since
   // it's fine to use the
   // default if not overridden
   void'(get_config_string("report_id",
                           report_id));

   // Other code not shown
endfunction
```

If you will use a particular component in several locations within a testbench, it can be helpful to override its default tag (report_id) with a new tag in such a way that each component's tag is unique within the testbench. When using the tag to control verbosity, the level of control is only as precise as the tag—if two components share a tag, they cannot be controlled independently.

Another method for identifying a component is by its hierarchical name. The hierarchical name is an OVM built-in property of each component in a testbench. While there is nothing wrong with using the hierarchical name to identify a component, it takes a lot more typing to specify components of interest, as the names can be quite long. However, OVM requires that the names be unique, which can be an advantage. The example code provided with this article supports identifying components either by tag or by hierarchical name.

The next step is to determine the runtime changes to verbosity. The OVM takes care of default verbosity via the OVM_VERBOSITY plusarg (OVM_VERBOSITY=OVM_LOW, etc.). We recommend implementing a similar method, with a plusarg for each OVM-defined verbosity level.

We used VERB_OVM_NONE, VERB_OVM_LOW, VERB_OVM_MED, VERB_OVM_HIGH, VERB_OVM_FULL and VERB_OVM_DEBUG. These correspond to the verbosity enum built into OVM using similar names.

You can specify each verbosity level using a single report tag, a hierarchical name or a colon-separated list of tags or names. Additionally, for each tag, you can specify that the verbosity setting should be applied recursively to all the specified component's children. The included code uses the prefixes "^" (for a recursive setting) and "~" (for a hierarchical name). Here are a few examples:

- Low verbosity for most of the testbench, high verbosity for the scoreboard:
  ```
  <sim> +OVM_VERBOSITY=OVM_LOW
  +VERB_OVM_HIGH=SCOREBOARD
  ```

- Full verbosity for the slave and all its children:
  ```
  <sim> +VERB_OVM_FULL=^SLAVE
  ```

- Full verbosity for the TX agent and its children, except for the startup module:
  ```
  <sim> +VERB_OVM_FULL=^~ovm_test_top.tb.
  tx.tx_agent   +VERB_OVM_LOW=~ovm_test_
  top.tb.tx.tx_agent.startup
  ```

You must determine the passed-in arguments early in the simulation so that components can pick up the changes before they start reporting. The example code accomplishes this by using a singleton class instantiated in a package. By including the package somewhere in the testbench (we suggest the top-level module), the command-line processing gets called automatically at the beginning of the simulation. The following shows a simplified version of the package code, illustrating just the key concepts.

```
// SystemVerilog package which provides
// runtime verbosity options
package runtime_verbosity_pkg;
  // Class which parses the
  // command-line plusargs
  class verbosity_plusargs extends
                            ovm_object;
    // "new" gets called automatically
    function new(string name =
                   "verbosity_plusargs");
      super.new(name);
      // parse_plusargs does all the
      // string-processing work
      parse_plusargs();
    endfunction : new
  endclass : verbosity_plusargs

  // Instantiate one copy of the
  // verbosity_plusargs class in the
  // package - this will be shared by
  // everything which imports
  // this package
  verbosity_plusargs
                 verb_plusargs = new();
endpackage : runtime_verbosity_pkg
```

Finally, you must change the verbosity setting for each reporting item. It is important to apply the setting early in the item's life span, before reporting begins. OVM components operate in "phases," namely build, connect, end_of_elaboration, start_of_simulation, run, extract, check and report. UVM has an expanded phasing mechanism, but the need for applying the verbosity setting early still applies.

To ensure that you have tweaked the verbosity before reporting begins, you should make the adjustment in the build() task of the component. UVM adds the capability to customize the phases somewhat, but it still uses build() as the first phase. The example code provides a simple call to update the component's verbosity setting. Building on the example agent shown above, the build() function now looks like this:

```
function void example_agent::build();
   super.build();

   // Void casting this call, since
   // it's fine to use the default
   // if not overridden
   void'(get_config_string("report_id",
                           report_id));

   // Other code not shown

   // Set the verbosity for this component
   runtime_verbosity_pkg::set_verbosity(
      .comp(this),.report_id(report_id));
endfunction
```

For simplicity, we recommend adding this functionality to the build() function of each component in the testbench. However, if that is not feasible (for example, if it's too late to change the components or if you are using legacy or third-party verification components), you can override the default reporter, which is called by each component. The downside of overriding the reporter is that it won't increase the verbosity when using the OVM convenience macro `ovm_info, since the macro checks the component's verbosity level before calling the reporter (it still works for decreasing the verbosity). Overriding the default reporter in the example code is as simple as adding the following line somewhere in the testbench (we recommend adding it to the top-level module):

```
import runtime_verbosity_server_pkg::*;
```

You can combine the two methods if you need to support legacy or third-party components yet still want to use the macros in the code.

The methodology shown here also works with UVM, with small changes to a few function names. With a little more effort, you can also extend it to work with objects, such as transactions and sequences. We've included a small example testbench showing the method presented in this article in the methodology zip file at *http://www.xilinx.com/publications/xcellonline/downloads/verb-ovm-environment.zip*.

As the examples presented here show, with a little thought and planning, enabling runtime verbosity in a testbench is a very simple matter. This approach has proven itself in production testbenches and designs. Our experience shows that the effort required for this small addition has a disproportionately large payoff—debugging tests become much quicker and easier to do. This means the simulator is running simulations more often and simulation cycles are not wasted. This is crucial, since simulation cycles are not a renewable resource. If a simulator is not running at a particular moment, that time could be viewed as a wasted resource. Quicker debugging means more up-time for testbench development and shorter time-to-closure.

The end goal of writing a testbench is to get a design implemented and in the customer's hands. Any advantage that accelerates this goal is a welcome addition to the designer's bag of tricks. Adding finer-grained verbosity control at runtime, instead of requiring a loopback to compilation, saves valuable time and energy, speeding delivery of the end product and startup of your next project.

WHICH INFORMATION AT WHICH LEVEL?

It can be helpful to define what type of information is presented at each verbosity level, especially if a large design team is involved. Establish a list of all the types of information you expect to generate and create a table showing the verbosity for each message type. Our example table shows a few common types of messages.

VERBOSITY	MESSAGE TYPE	EXAMPLES
OVM_LOW	Major environment/DUT state changes	Beginning reset sequence; link established
OVM_MEDIUM	Other environment/DUT state changes	Interrupt asserted; arbitration timeout
OVM_HIGH	Sequences started and finished	Beginning read sequence; ending reset
OVM_HIGH	Scoreboard activity	Compare passed; check skipped
OVM_HIGH	Major driver and monitor events	Packet corrupted due to reset
OVM_HIGH	Reporting high-level configuration settings at startup	32-bit datapath; two masters and four slaves instantiated
OVM_FULL	Transactions driven and monitored	Full dump of transaction contents
OVM_FULL	Driver and monitor state changes	Arbitration request/grant; inserting <n> IDLE cycles
OVM_DEBUG	Objections raised and lowered	Raised objection; lowered objection
OVM_DEBUG	Detailed driver, monitor and scoreboard activity	Loop iterations, if-branches followed, etc.

Breaking Barriers with FPGA-Based Design-for-Prototyping

Automation and a new design methodology can improve software quality and supercharge SoC project schedules.

by Doug Amos
Business Development Manager
Synopsys Inc.
damos@synopsys.com

There has never been a better time to use FPGAs. The capability of the devices, the power of the tools and the wealth of FPGA-ready IP mean that we can tackle almost any application with confidence. However, there are still between 5,000 and 10,000 designs per year that require an ASIC, system-on-chip or ASSP solution—let's use the generic term SoC for all of them. Even in these situations, FPGAs still have a crucial role to play in the project's success as prototyping vehicles. There are many good reasons to opt for FPGA-based prototyping, especially if you can find ways for SoC teams and their FPGA-expert colleagues to combine their talents to speed up schedules and improve quality.

THREE 'LAWS' TO GET US STARTED

You may have noticed that technical disciplines often boil down to three laws. There's Sir Isaac Newton's famous trio of laws of motion, for example, or the three laws of thermodynamics and even Isaac Asimov's three laws of robotics. After nearly 30 years of working with FPGAs, I'd like to humbly add a new set of three "laws" that seem to govern or influence FPGA-based prototyping. They can be stated as follows:

1. SoCs can be larger than FPGAs.
2. SoCs can be faster than FPGAs.
3. SoC designs can be hard to port to FPGA.

OK, they are not going to win me a Nobel Prize and they are not really laws at all in the strict sense. Rather, they represent an attempt to describe the root causes of the challenges that face us when prototyping. Prototypers have spent a lot of development effort and engineering ingenuity on dealing with these challenges. Let's take a closer look at each of the three and in doing so, find a way around them by adopting a different approach to the SoC project, one that we at Synopsys call Design-for-Prototyping.

WORKING WITHIN THE THREE LAWS

Implementing an FPGA-based prototype of an SoC requires overcoming a variety of development challenges. Perhaps surprisingly, creating the FPGA hardware itself is not the most difficult problem. With the help of advanced FPGA devices, such as the XCV6LX760 from the Xilinx® Virtex®-6 family, it is possible to create boards and platforms that can host up to 100 million SoC gates. It's tempting to delve into the pros and cons of making vs. buying such boards but, leaving that subject for another day, let's focus instead on the particular expertise required to implement an SoC design in FPGA resources. Our three "laws" are a useful scaffolding upon which to build the argument for FPGA-based prototyping.

FIRST LAW: SOCS CAN BE LARGER THAN FPGAS

Even though today's largest FPGAs can handle millions of SoC gates, there will be SoC designs which are larger still. To prototype these hefty SoCs, we not only need to partition the SoC design across multiple FPGA devices but must then reconnect the signals among those devices. Typically, when you have to split the design between two or more FPGAs, the number of pins available to reconnect the FPGAs becomes the limiting factor, even when using the latest FPGAs sporting more than 1,000 I/O pins each. For example, partitioning designs with wide buses or datapaths can require a surprisingly large number of I/O pins.

Our goal must be to modify the SoC RTL as little as possible, but partitioning and reconnecting create artificial boundaries in the design. This presents complex challenges in optimal partitioning, pin allocation and board-trace assignment. Since the job involves working with thousands of pins at a time, automation will be a help, given that just one misplaced pin can cause our prototype not to function (not to mention, provide endless hours of fun debugging).

Full automation is difficult to achieve, however, because every SoC design is unique and the variety of designs is wide. It is difficult to devise an automated algorithm that will provide optimal results across so many scenarios. We usually find that it is best to start with some engineer-guided steps for critical sections of the design and then use automation to complete the parts where optimal results are not so crucial. Indeed, that is exactly the approach of many production FPGA designs, where some well-judged manual intervention can greatly improve results. For prototyping, it is the task of companies like Synopsys and Xilinx to provide tools that balance automation, optimization and intervention, with the understanding that the tools and methods needed for prototyping and for production designs are not necessarily the same.

Can we break the first law? Actually, no; there will probably always be SoC designs that are larger than the available FPGAs. However, with semi-automated partitioning, high-speed interconnect and some

designer intelligence, we can optimize results to minimize the effect of the partition boundaries on the overall prototype system speed.

Further, we can certainly mitigate the effect of the first law by using time-division multiplexing, allowing several design signals to pass through a single board trace between two FPGAs. This greatly eases the partitioning and reconnection tasks but may limit the prototype's overall performance, especially if high multiplexing ratios are needed. Happily, designers can now easily use the FPGA's fast differential I/O capability and source-synchronous signaling as the transport medium between the FPGAs to regain performance, even at multiplexing ratios as high as 128:2. You can achieve all of this in an automated way without any need for changing the source RTL for the SoC.

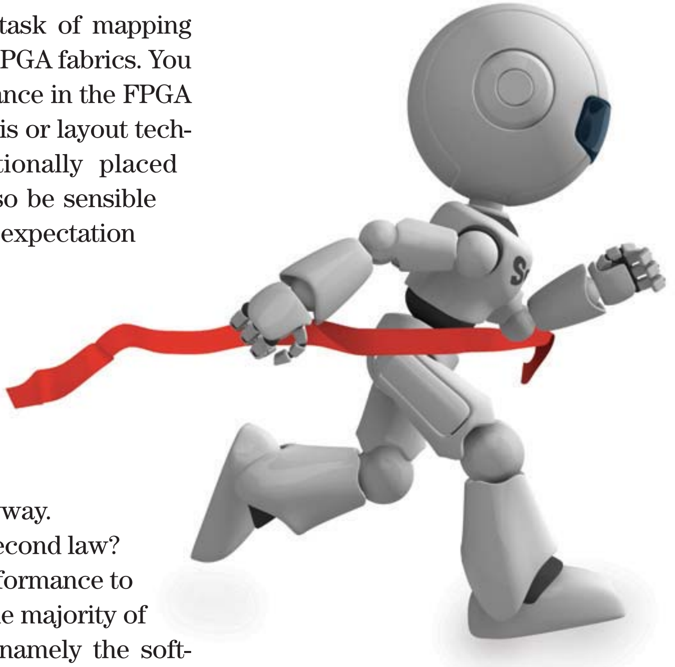
SECOND LAW: SOCS CAN BE FASTER THAN FPGAS

SoC designs can achieve a very high core speed for critical paths mapped into particular high-performance library elements that trade speed for area, power or cost. The unfortunate side effect is that critical paths in SoC designs may then be synthesized into more levels of logic than would otherwise have been the case. That leaves prototypers with the task of mapping these long paths into FPGA fabrics. You can optimize performance in the FPGA using physical synthesis or layout techniques such as relationally placed macros, but it may also be sensible to start with the right expectation of the maximum performance achievable for these critical paths. In many cases, it may not be necessary to run these paths at full speed anyway.

Can we break the second law? FPGAs deliver the performance to exceed the needs of the majority of prototype end users, namely the software engineers performing presilicon validation. These software engineers integrate embedded software into the target hardware and also validate its behavior in the context of complex multicore operating systems using real-time data and real-world environments.

To be most useful, the prototype has to run accurately and at a speed as close as possible to that of the final SoC silicon. However, we can also plan to run the prototype at, say, half or quarter speed, and let the software team adjust their timing parameters in order to allow for the lower speed and still run their tests. Even this reduced speed will be an order of magnitude faster than any comparable emulation system, or two or three orders of magnitude faster than an RTL simulator. The performance of FPGAs provides months of benefit before the first SoC silicon samples are finally available.

To be most useful, the prototype has to run accurately and at a speed as close as possible to that of the final SoC silicon. However, we can also plan to run the prototype at, say, half or quarter speed, and let the software team adjust their timing parameters in order to allow for the lower speed and still run their tests.



THIRD LAW: SOC DESIGNS CAN BE HARD TO PORT TO FPGA

A very common problem facing the team when porting an SoC design to an FPGA is that the authors of the original SoC RTL didn't plan ahead to meet the prototyper's needs. As a result, the design usually contains some SoC-specific elements—such as library-cell and RAM instantiations—that require special attention before they can be implemented in an FPGA tool flow. The two most dominant examples of such "FPGA-hostile" design elements are overly complex clock networks and nonportable IP.

Today's FPGA devices feature very sophisticated and capable clock-generation and distribution resources, such as the MMCM blocks in Virtex-6. However, SoC designs often feature very complex clock networks themselves, including clock gating, frequency scaling and a large number of clock domains. These SoC clock networks can overflow the resources of an FPGA, and the problem is compounded when the design is split across multiple FPGAs and requires board-level clock generation and distribution.

Regarding intellectual property (IP), it is hard to conceive of an SoC today that does not include many blocks of IP, from relatively simple functional blocks up to CPUs and related bus-connection subsystems, and standards-based peripheral IP such as PCI Express® or USB. The original SoC designers may have chosen IP to meet their own needs, without checking whether an FPGA-equivalent version exists. This means that

What if we could get the design in a much more FPGA-friendly version from the start? What if the whole SoC project team included FPGA experts from the outset, offering guidance on FPGA technology so that the design could be optimized for both SoC and FPGA implementation? Then we would have a Design-for-Prototyping methodology that would be a game changer for today's SoC projects.

the prototypers need to either remap the SoC version of the RTL or substitute an FPGA equivalent that is functionally as close as possible to the original IP. The good news is that many designers regularly use the FPGA's built-in hard IP and other FPGA-compatible IP available within CORE Generator™ from Xilinx or DesignWare from Synopsys as replacements for SoC IP with great success.

Can we break the third law? Yes, we can. First, we can use tools and ingenuity to work around the SoC-specific elements. For example, automated tools can convert gated clocks into system clocks and the appropriate clock-enables, which are more optimized for FPGA resources. We can also replace SoC RAM instantiations with their FPGA equivalents, or extract larger RAMs to an external memory resource on the prototype board.

HOW CAN WE MAKE FPGA-BASED PROTOTYPING EASIER?

Prototypers worldwide are successfully employing all the techniques cited above and many other methods today, but what if we could get the design in a much more FPGA-friendly version from the start? What if the whole SoC project team included FPGA experts from the outset, offering guidance on FPGA technology so that the design could be optimized for both SoC and FPGA implementation? Then we would have a Design-for-Prototyping methodology that would be a game changer for today's SoC projects, and this would help software engineers most of all.

DESIGN-FOR-PROTOTYPING IS COMING

Software development now dominates SoC project costs and schedules, and increasingly, the goal of the SoC silicon is to provide an optimal runtime environment for the software. The design priorities in SoC projects are changing to include methods for accelerating software development and validation. FPGA-based prototyping is a critical part of this change and as such, it must be integrated more completely into the SoC project flow, rather than being almost an afterthought in some worst-case examples.

Design teams must adopt different methods throughout the SoC flow in order to ease the creation of an FPGA-based prototype. Then they must provide this prototype to the software engineers as soon as possible to deliver maximum benefit before the first SoC silicon arrives. That means eliminating—or at the very least, clearly identifying and isolating—nonportable elements so that the prototyping team can make the necessary changes quickly and without error.

The happy coincidence is that guidelines in a Design-for-Prototyping methodology are also central to good SoC design practice and will make designs more portable and easier to reuse. For example, using wrappers to isolate technology-specific elements, such as memory and arithmetic blocks, so that they can be more easily replaced with FPGA equivalents will also make it easier to use formal equivalency-checking methods in the SoC flow itself. By using only SoC IP for which there is a proven and available FPGA equivalent, design teams can shave weeks off their prototype development cycle, leading to immeasurable benefits in final software quality and product scheduling.

SoC labs with an interest in Design-for-Prototyping must pay heed to two major initiatives, namely, procedural guidelines and design guidelines. Many examples of each exist; look for further literature from Xilinx and Synopsys very soon on this important subject area.

FPGA-based prototyping is of such crucial benefit to today's SoC and embedded-software projects that we should do all we can to ensure its success. The first order of business is to break or overcome the three "laws" that have governed many prototyping projects up until now, using automation and a Design-for-Prototyping methodology. If we can integrate the procedural and design guidelines of Design-for-Prototyping within the whole SoC project, then software quality and project schedules will improve—and so will the final product.

Application Notes

If you want to do a bit more reading about how our FPGAs lend themselves to a broad number of applications, we recommend these notes.

XAPP887: PRC/EPRC: DATA INTEGRITY AND SECURITY CONTROLLER FOR PARTIAL RECONFIGURATION

http://www.xilinx.com/support/documentation/application_notes/xapp887_PRC_EPRC.pdf

Partial reconfiguration of Xilinx® Virtex® FPGAs requires the use of partial bitstreams. These bitstreams reprogram regions of an operational FPGA with new functionality without disrupting the functionality or compromising the integrity of the remainder of the device. However, loading a faulty or corrupted partial bitstream might cause incorrect design behavior, not only in the region undergoing reprogramming but also in the entire device.

Reprogramming the device with a correct partial or full bitstream can be the simplest recovery method. However, for some systems, it is essential to validate the integrity of partial bitstreams before loading them into the device. This is because halting the design operation to reprogram the entire device is undesirable or delivering partial bitstreams is error prone (as in radio transmission, for example).

This application note by Amir Zeineddini and Jim Wesselkamper describes a data integrity controller for partial reconfiguration (PRC) that you can include in any partially reconfigurable FPGA design to process partial bitstreams for data integrity. The controller implements cyclic redundancy check (CRC) in FPGA logic for partial bitstreams before loading them through the internal configuration access port (ICAP). A special version of the controller for encrypted partial reconfiguration (EPRC) enables encrypted partial bitstreams to use the internal configuration logic for decryption.

This application note is applicable only to Virtex-5 and Virtex-6 devices.

XAPP884: AN ATTRIBUTE-PROGRAMMABLE PRBS GENERATOR AND CHECKER

http://www.xilinx.com/support/documentation/application_notes/xapp884_PRBS_GeneratorChecker.pdf

In serial interconnect technology, it is very common to use pseudorandom binary sequence (PRBS) patterns to test the robustness of links. Behind this choice is the fact that every PRBS pattern has a white spectrum in the frequency domain or, alternatively, a delta-shaped autocorrelation function in the time domain. This application note describes a PRBS generator/checker circuit where the generator polynomial, the parallelism level and the functionality (generator or checker) are programmable via attributes.

Along with “how to” information on circuit description, pinout and standard polynomials, authors Daniele Riccardi and Paolo Novellini offer a design example and then delve into the theoretical background on PRBS sequences, their generators and how to select them to obtain optimal spectral properties.

XAPP107: IMPLEMENTING TRIPLE-RATE SDI WITH SPARTAN-6 FPGA GTP TRANSCEIVERS

http://www.xilinx.com/support/documentation/application_notes/xapp1076_S6GTP_TripleRateSDI.pdf

Professional broadcast video equipment makes wide use of the triple-rate serial digital interface (SDI) supporting the SMPTE SD-SDI, HD-SDI and 3G-SDI standards. In broadcast studios and video production centers, SDI interfaces carry uncompressed digital video along with embedded ancillary data, such as multiple audio channels. Spartan®-6 FPGA GTP transceivers are well-suited for implementing triple-rate SDI receivers and transmitters.

In this document, author Reed Tidwell describes triple-rate SDI receiver and transmitter reference designs for the GTP transceivers in Spartan®-6 devices. The transceivers provide a high degree of performance and reliability in a low-cost device for this application. The Spartan-6 FPGA SDI transmitter supplies the basic operations necessary to support SD-SDI, HD-SDI, dual-link HD-SDI and 3G-SDI transmission. (It does not perform video mapping for 3G-SDI or dual-link HD-SDI. Video formats that require mapping must be mapped into SDI data streams prior to the triple-rate SDI TX module.)

The Spartan-6 FPGA GTP transceiver triple-rate SDI transmitter has a 20-bit GTP interface, supported on -3 and faster Spartan-6 devices. Only two reference clock frequencies are required to support all SDI modes.

XAPP495: IMPLEMENTING A TMDS VIDEO INTERFACE IN THE SPARTAN-6 FPGA

http://www.xilinx.com/support/documentation/application_notes/xapp495_S6TMDS_Video_Interface.pdf

Transition-minimized differential signaling (TMDS) is a standard used for transmitting video data over the Digital

Visual Interface (DVI) and High-Definition Multimedia Interface (HDMI). Both interfaces are popular in a wide range of market segments including consumer electronics, audio/video broadcasting, industrial surveillance and medical imaging. They are commonly seen in flat-panel TVs, PC monitors, Blu-ray disc players, video camcorders and medical displays. This application note by Bob Feng describes a set of reference designs able to transmit and receive DVI and HDMI data streams up to 1,080 Mbits/second using the Spartan-6 FPGA's native TMDS I/O interface.

The DVI and HDMI protocols use TMDS at the physical layer. The TMDS throughput is a function of the serial data rate of the video screen mode being transmitted. This in turn determines the FPGA speed grade that's needed to support this throughput. In recent generations of Spartan devices, Xilinx has offered embedded electrically compliant TMDS I/O, allowing implementation of DVI and HDMI interfaces inside the FPGA.

This application note successfully demonstrates a high-definition TMDS video connectivity solution featuring the processing of multiple HD channels in parallel. The design demonstrates the Spartan-6 FPGA in DVI or HDMI applications (including the 480p, 1080i and 720p video formats).

XAPP886: INTERFACING QDR II SRAM DEVICES WITH VIRTEX-6 FPGAS

http://www.xilinx.com/support/documentation/application_notes/xapp886_Interfacing_QDR_II_SRAM_Devices_with_V6_FPGAs.pdf

With an increasing need for lower latency and higher operating frequencies, memory interface IP is becoming more complex, and designers need to tailor it based on a number of factors such as latency, burst length, interface width and operating frequency. The Xilinx Memory Interface Generator (MIG) tool enables the creation of a large variety of memory interfaces for devices such as the Virtex-6 FPGA. However, in the Virtex-6, QDR II SRAM is not one of the options available by default. Instead, the focus has been on the QDR II+ technology using four-word burst-access mode.

This application note by Olivier Despaux presents a Verilog reference design that has been simulated, synthesized and verified on hardware using Virtex-6 FPGAs and QDR II SRAM two-word burst devices. The reference design is based on a MIG 3.4 QDR II+ four-word burst design. Despaux designed the code base to be adaptable so that designers can pass many parameters from the top-level RTL module to the levels lower down in the hierarchy. The control layers of the design operate mostly at half the rate of the memory interface, and the ISERDES and OSERDES are used to interface to the external memory components.

XAPP88: FAST CONFIGURATION OF PCI EXPRESS TECHNOLOGY THROUGH PARTIAL RECONFIGURATION

http://www.xilinx.com/support/documentation/application_notes/xapp883_Fast_Config_PCIe.pdf

The PCI Express® specification requires ports to be ready for link training at a minimum of 100 milliseconds after the power supply is stable. This becomes a difficult task due to the ever-increasing configuration memory size of each new generation of FPGA, such as the Xilinx Virtex-6 family. One innovative approach to addressing this challenge is leveraging the advances made in the area of FPGA partial reconfiguration to split the overall configuration of a PCIe® specification-based system in a large FPGA into two sequential steps: initial PCIe system link configuration and subsequent user application reconfiguration.

It is feasible to configure only the FPGA PCIe system block and associated logic during the first stage within the 100-ms window before the fundamental reset is released. Using the Virtex-6 FPGA's partial reconfiguration capability, the host can then reconfigure the FPGA to apply the user application via the now-active PCIe system link.

This methodology not only provides a solution for faster PCIe system configuration, it enhances user application security because only the host can access the bitstream, so the bitstream can be better encrypted. This approach also helps lower the system cost by reducing external configuration component costs and board space.

This application note by Simon Tam and Martin Kellermann describes the methodology for building a Fast PCIe Configuration (FPC) module using this two-step configuration approach. An accompanying reference design will help designers quick-launch a PlanAhead™ software partial reconfiguration project. The reference design implements an eight-lane PCIe technology first-generation link and targets the Virtex-6 FPGA ML605 Evaluation Board. The reference design serves as a guide for designers to develop their own solutions.

XAPP492: EXTENDING THE SPARTAN-6 FPGA CONNECTIVITY TARGETED REFERENCE DESIGN (PCIE-DMA-DDR3-GBE) TO SUPPORT THE AURORA 8B/10B SERIAL PROTOCOL

http://www.xilinx.com/support/documentation/application_notes/xapp492_S6_ConnectivityTRD_Aurora.pdf

Targeted reference designs provide Xilinx designers with turnkey platforms to create FPGA-based solutions in a wide variety of industries. This application note by Vasu Devunuri and Sunita Jain extends the Xilinx Spartan-6 FPGA PCIe-DMA-DDR3-GbE targeted reference design to support the Aurora 8B/10B serial link-layer protocol for high-speed data communications. Designers use Aurora 8B/10B to move data across point-to-point serial links. It provides a transparent interface to the physical serial links and supports both framing and streaming modes of operation. This application note uses Aurora in framing mode.

The PCIe-DMA base platform moves data between system memory and the FPGA. Thus transferred, the data can be consumed within the FPGA, forwarded to another FPGA or sent over the backplane using a serial connectivity protocol such as Aurora 8B/10B. Similarly, the Aurora protocol can bring data in to the FPGA from another FPGA or backplane, and send it to system memory for further processing or analysis. This enhancement retains the Ethernet operation as is and demonstrates PCIe-to-Ethernet and PCIe-to-Aurora bridging functionality.

The application note demonstrates the Spartan-6 FPGA Connectivity Targeted Reference Design in a platform solution. The example design uses this existing infrastructure as a base upon which to create a newer design with limited effort. The network path functioning as the network interface card remains the same. The memory path is modified to support packet FIFO over the Spartan-6 FPGA memory controller and to support integrating the Aurora 8B/10B LogiCORE® IP, which operates through the packet FIFO.

XAPP899: INTERFACING VIRTEX-6 FPGAS WITH 3.3V I/O STANDARDS

http://www.xilinx.com/support/documentation/application_notes/xapp899.pdf

All the devices in the Virtex-6 family are compatible with and support 3.3-volt I/O standards. In this application note, Austin Tavares describes methodologies for interfacing Virtex-6 devices to 3.3V systems. The document covers input, output and bidirectional buses, as well as signal integrity issues and design guidelines.

The Virtex-6 FPGA I/O is designed for both high performance and flexibility. Each I/O is homogenous, meaning every I/O has all features and all functions. This high-performance I/O allows the broadest flexibility to address a wide range of applications. Designers seeking to interface Virtex-6 FPGA I/O to 3.3V devices must take care to ensure device reliability and proper interface operation by following the appropriate design guidelines. They must meet the DC and AC input voltage specification when designing interfaces with Virtex-6 devices.

XAPP951: CONFIGURING XILINX FPGAS WITH SPI SERIAL FLASH

http://www.xilinx.com/support/documentation/application_notes/xapp951.pdf

The addition of the Serial Peripheral Interface (SPI) in newer Xilinx FPGA families allows designers to use multi-vendor small-footprint SPI PROM for configuration. Systems with an existing onboard SPI PROM can leverage the single memory source for storing configuration data in addition to the existing user data. In this application note, author Stephanie Tapp discusses the required connections to configure the FPGA from an SPI serial flash device and shows the configuration flow for the SPI mode, noting special precautions for configuring from an SPI serial flash. The note details the ISE® Design Suite iMPACT direct programming solution for SPI-formatted PROM file creation and programming during prototyping for select vendors.

SPI serial flash memories are popular because they can be easily accessed postconfiguration, offering random-access, nonvolatile data storage to the FPGA. Systems with SPI serial flash memory already onboard can also benefit from having the option to configure the FPGA from the same memory device. The SPI protocol does have a few variations among vendors. This application note discusses these variants.

XAPP881: VIRTEX-6 FPGA LVDS 4X ASYNCHRONOUS OVERSAMPLING AT 1.25 GB/S

http://www.xilinx.com/support/documentation/application_notes/xapp881_V6_4X_Asynch_OverSampling.pdf

The Virtex-6 FPGA's SelectIO™ technology can perform 4x asynchronous oversampling at 1.25 Gbits/second, accomplished using the ISERDESE1 primitive through the mixed-mode clock manager (MMCM) dedicated performance path. Located in the SelectIO logic block, the ISERDES1 primitive contains four phases of dedicated flip-flops used for sampling.

The MMCM is an advanced phase-locked loop that has the capability to provide a phase-shifted clock on a low-jitter performance path. The output of the ISERDESE1 is then sorted using a data recovery unit. The DRU used in this application note by Catalin Baetoniu and Brandon Day is based on a voter system that compares the outputs and selects the best data sample. The method consists of oversampling the data with a clock of similar frequency (±100 ppm), taking multiple samples of the data at different clock phases to get a sample of the data at the most ideal point.

XAPP493: IMPLEMENTING A DISPLAYPORT SOURCE POLICY MAKER USING A MICROBLAZE EMBEDDED PROCESSOR

http://www.xilinx.com/support/documentation/application_notes/xapp493_DisplayPort_SPM.pdf

The purpose of this application note and accompanying reference design is to help you implement the DisplayPort Source Policy Maker in a MicroBlaze™ processor. Although authors Tom Strader and Matt Ouellette target their design specifically for the Spartan-6 FPGA Consumer Video Kit, they have designed it to be architecture independent.

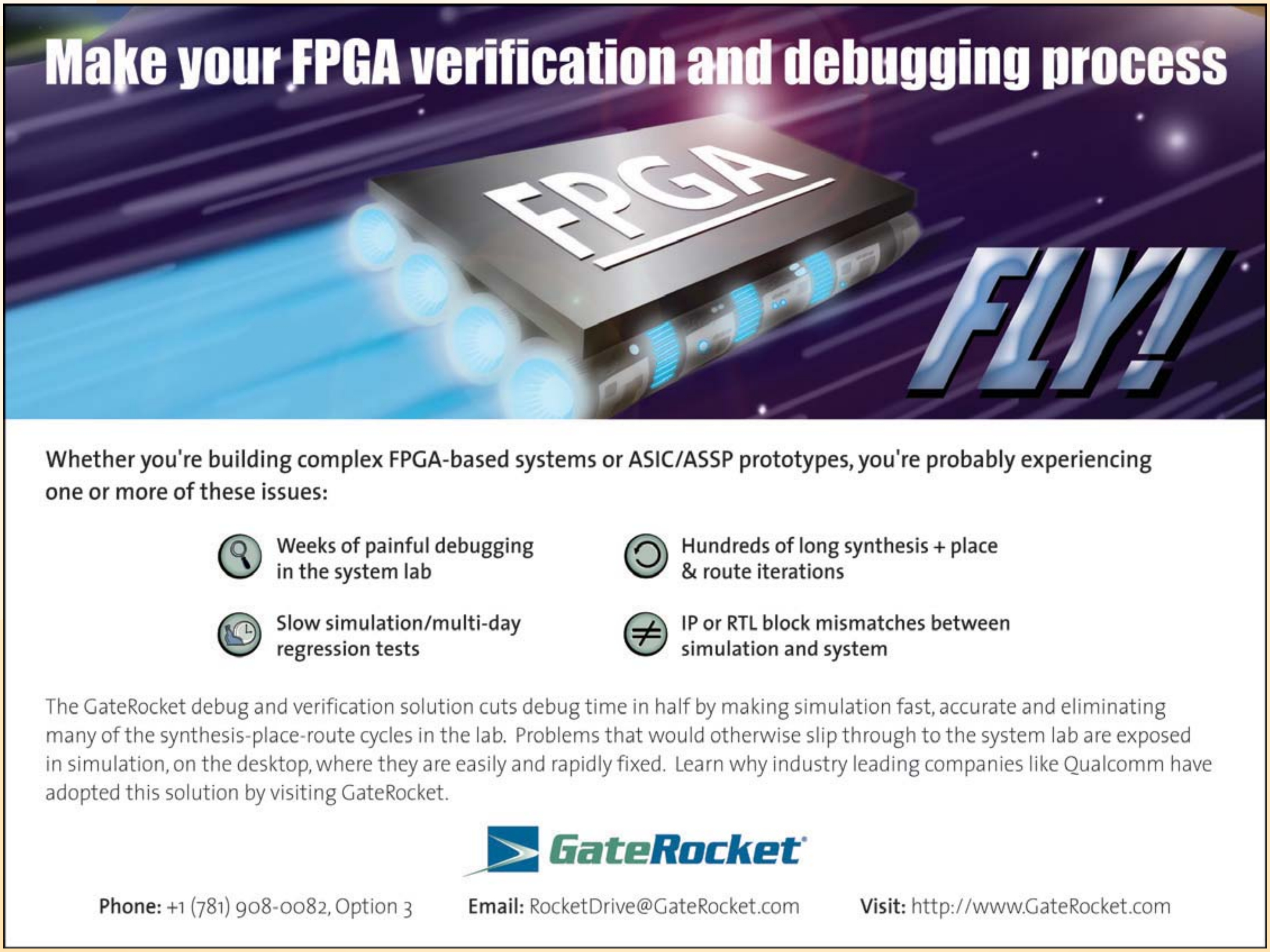
The DisplayPort Source Policy Maker communicates with both the transmitter (source) and the receiver (sink) to perform several tasks, such as initialization of GTP transceiver links, probing of registers and other features useful for bring-up and use of the core. This reference system implements the DisplayPort Source Policy Maker at the source side. The mechanism for communication to the sink side is over the auxiliary channel, as described in the DisplayPort standard specification (available from the Video Electronics Standards Association website).

The reference design encompasses the DisplayPort Source Policy Maker, a DisplayPort source core generated using the Xilinx CORE Generator™ tool and a video pattern generator to create video data for transmission over the DisplayPort link.

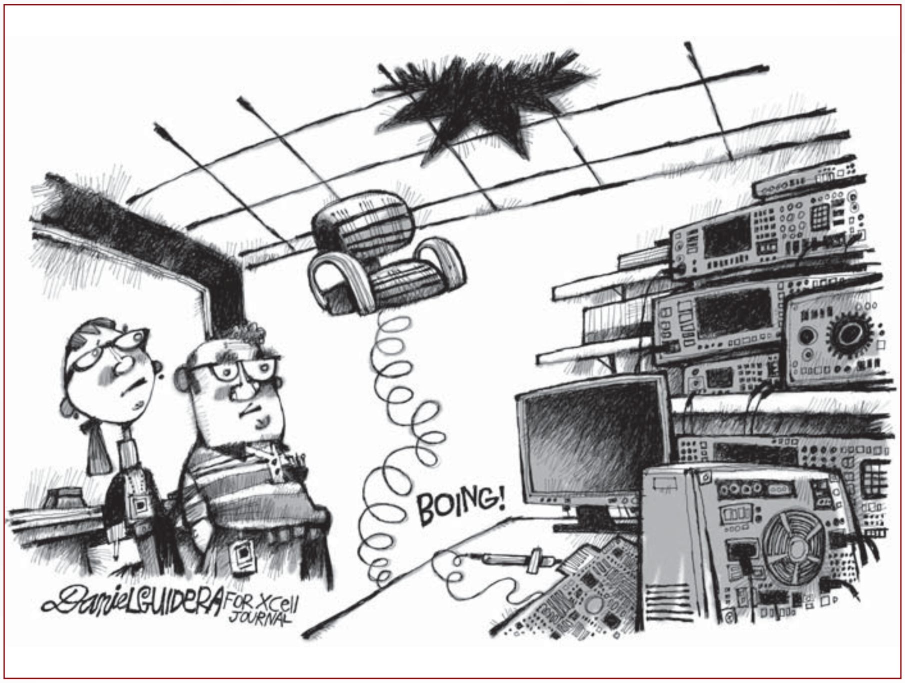
XAPP1071: CONNECTING VIRTEX-6 FPGAS TO ADCS WITH SERIAL LVDS INTERFACES AND DACS WITH PARALLEL LVDS INTERFACES

http://www.xilinx.com/support/documentation/application_notes/xapp1071_V6_ADC_DAC_LVDS.pdf

Virtex-6 FPGA interfaces use the dedicated deserializer (ISERDES) and serializer (OSERDES) functionalities to provide a flexible and versatile platform for building high-speed low-voltage differential signaling (LVDS) interfaces to all the latest data converter devices on the market. This application note by Marc Defossez describes how to utilize the ISERDES and OSERDES features in Virtex-6 FPGAs to interface with analog-to-digital converters (ADCs) that have serial LVDS outputs and with digital-to-analog converters (DACs) that have parallel LVDS inputs. The associated reference design illustrates a basic LVDS interface connecting a Virtex-6 FPGA to any ADCs or DACs with high-speed serial interfaces.



Xpress Yourself in Our Caption Contest



DANIEL GUIDERA

If there's a spring in your walk—or in your office chair—here's your opportunity to use it to good effect. We invite readers to bounce up to a verbal challenge and submit an engineering- or technology-related caption for this Daniel Guidera cartoon depicting something amiss in the lab. The image might inspire a caption like "The booby-trapped desk chair was Helen's way of breaking the glass ceiling in R&D."

Send your entries to *xcell@xilinx.com*. Include your name, job title, company affiliation and location, and indicate that you have read the contest rules at *www.xilinx.com/xcellcontest*. After due deliberation, we will print the submissions we like the best in the next issue of *Xcell Journal* and award the winner the new Xilinx® SP601 Evaluation Kit, our entry-level development environment for evaluating the Spartan®-6 family of FPGAs (approximate retail value, $295; see *http://www.xilinx.com/sp601*). Runners-up will gain notoriety, fame and a cool, Xilinx-branded gift from our SWAG closet.

The deadline for submitting entries is 5:00 pm Pacific Time (PT) on April 1st, 2011. So, get writing in time for April Fool's Day!

GLENN BABECKI,
principal system engineer at Comcast, won a Spartan-6 Evaluation Kit for this caption for the holiday turkey cartoon in the last issue of *Xcell Journal*.

CONFERENCE ROOM
Daniel GUIDERA FOR Xcell Journal

DANIEL GUIDERA

"When the boss said 'watch the slices in the fabric,' I didn't think he was talking about serving Thanksgiving turkey in the conference room!"

Congratulations as well to our two runners-up:

"The boss had a strange sense of humor. Last week he had commented that the new prototype was hot enough to cook a Thanksgiving dinner, but..."

– Robin Findley,
Findley Consulting, LLC

"The new wafer fab has been producing some real turkeys lately."

– Vadim Vaynerman,
Bottom Line Technologies Inc.

NO PURCHASE NECESSARY. You must be 18 or older and a resident of the fifty United States, the District of Columbia or Canada (excluding Quebec) to enter. Entries must be entirely original and must be received by 5:00 pm Pacific Time (PT) on April 1, 2011. Official rules available online at www.xilinx.com/xcellcontest. Sponsored by Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124.

XILINX
VIRTEX-6
XC6VLX760
SYNOPSYS
Predictable Success
HAPS-64
HAPS
GND
+12V
TOP