Xcell journal

ISSUE 79, SECOND QUARTER 2012

SOLUTIONS FOR A PROGRAMMABLE WORLD

Xilinx Unveils Vivado Design Suite for the Next Decade of 'All Programmable' Devices

Using Formal Verification for HW/SW Co-verification of an FPGA IP Core

How to Use the CORDIC Algorithm in Your FPGA Design

Smart, Fast Financial Trading Platforms Start with FPGAs

FPGAs Provide Flexible Platform for High School Robotics

page 28

XILINX®

www.xilinx.com/xcell/

Xilinx® Spartan®-6 FPGAs

Development Tools, Designed by Avnet

Xilinx® Spartan®-6 FPGA Motor Control Development Kit

AES-S6MC1-LX75T-G

Go beyond traditional MCUs to:

- Execute complex motor control algorithms
- Achieve higher levels of integration
- Implement custom safety features

$1,095

Xilinx® Spartan®-6 FPGA Industrial Video Processing Kit

AES-S6IVK-LX150T-G

Prototype & develop systems such as:

- High resolution video conferencing
- Video surveillance
- Machine vision

$2,195 ~~$2,695~~ (for a limited time only)

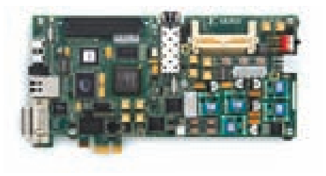
Xilinx® Spartan®-6 FPGA Industrial Ethernet Kit

AES-S6IEK-LX150T-G

Prototype & develop systems such as:

- Industrial networking
- Motor control
- Embedded control

$1,395 ~~$1895~~ (for a limited time only)



Xilinx® Spartan®-6 FPGA SP605 Evaluation Kit

EK-S6-SP605-G

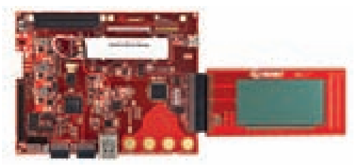
Designed by Xilinx, this kit enables implementation of features such as:

- High-speed serial transceivers
- PCI Express®
- DVI & DDR3

$450 ~~$495~~ (for a limited time only)

FOR A LIMITED TIME, AVNET IS OFFERING DISCOUNTS ON SEVERAL OF OUR MOST POPULAR TOOLS.

www.em.avnet.com/s6solutions

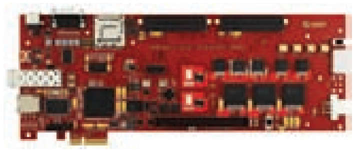


Xilinx® Spartan®-6 LX16 Evaluation Kit

AES-S6EV-LX16-G

- First-ever battery-powered Xilinx FPGA development board
- Achieve on-board FPGA configuration and power measurement with the included Cypress PSoC® 3

$225



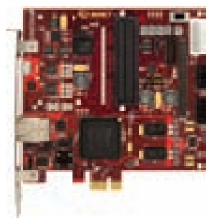
Xilinx® Spartan®-6 LX150T Development Kit

AES-S6DEV-LX150T-G

Prototype high-performance designs with ease:

- PCI Express x4 end-point
- SATA host connector
- Two general-purpose GTP ports
- Dual FMC LPC expansion slots

$995

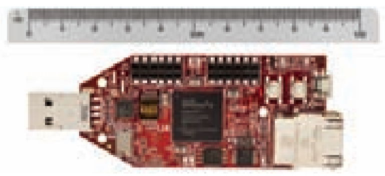


Xilinx® Spartan®-6 FPGA LX75T Development Kit

AES-S6PCIE-LX75T-G

- Optimize embedded PCIe® applications using a standard set of features in a compact PCIe form factor
- Dual banks of DDR3 memory
- Expand your design using a card edge-aligned FMC slot

$425



Xilinx® Spartan®-6 FPGA LX9 MicroBoard

AES-S6MB-LX9-G

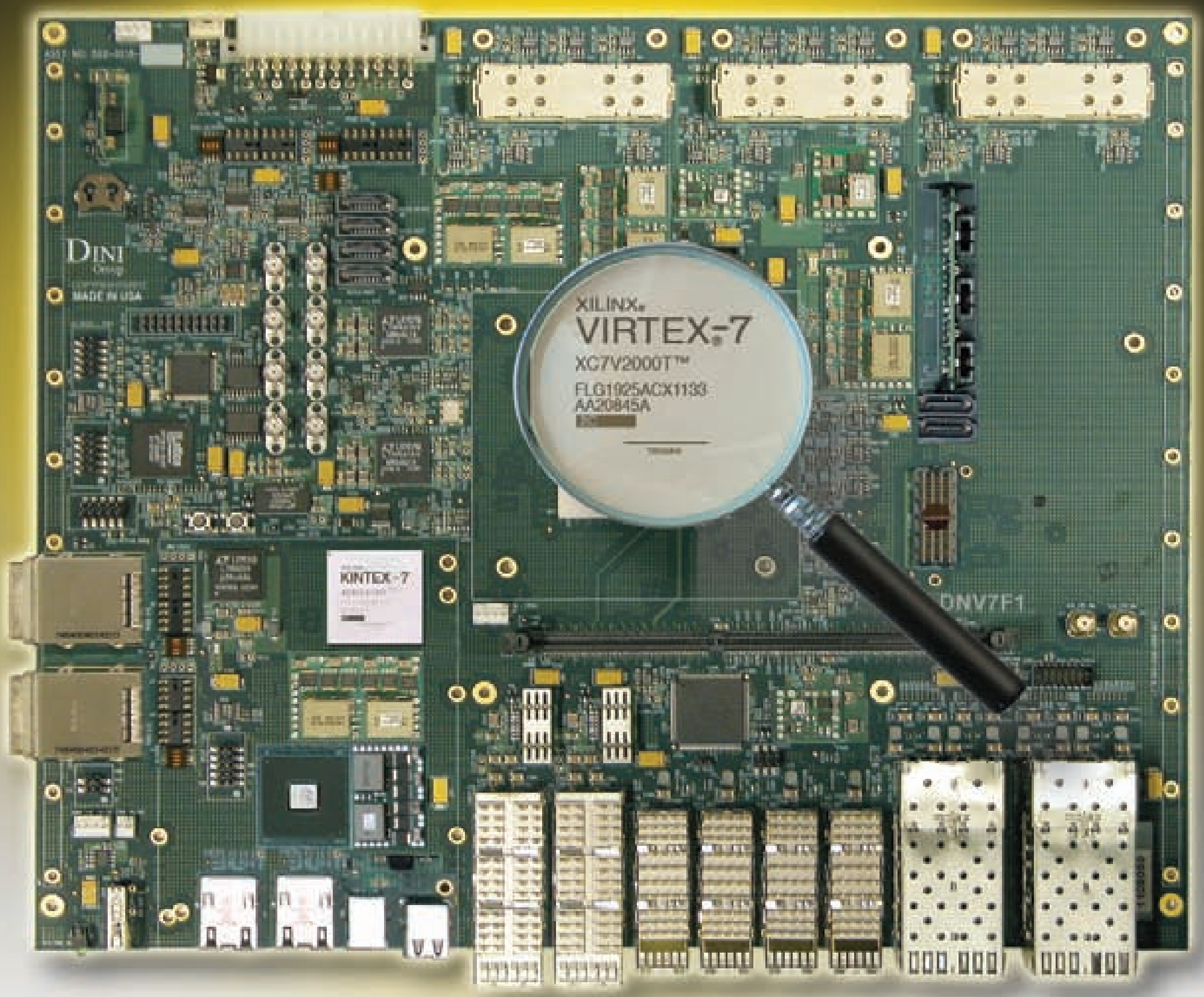
- Explore the MicroBlaze™ soft processor and Spartan®-6 FPGAs
- Leverage the included pre-built MicroBlaze "systems"
- Write & debug code using the included Software Development Kit (SDK)

$89

XILINX®

AVNET® electronics marketing

YOUR CHIP FITS HERE



15 Million ASIC Gates for Prototyping Your ASIC

New stacked silicon architecture from Xilinx makes your big design much easier to prototype. Partitioning woes are forgotten, and designs run at near final chip speed. The DINI Group DNV7F1 board puts this new technology in your hands with a board that gets you to market easier, faster and more confident of your design's functionality running at high speed. DINI Group engineers put the features you need most, right on the board:

- 10GbE
- USB 2
- PCIe, Gen 1, 2, and 3
- 240 pin UDIMM for DDR3

There is a Marvel Processor for any custom interfaces you might need and plenty of power and cooling for high speed logic emulation. Software and firmware developers will appreciate the productivity gains that come with this low cost, stand-alone development platform.

Prototyping just got a lot easier, call DINI today and get your chip up to speed.

DINI Group

www.dinigroup.com • 7469 Draper Avenue • La Jolla, CA 92037 • (858) 454-3419 • e-mail: sales@dinigroup.com

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/

Welcome to the Programmable Renaissance

Time flies. I recently celebrated my fourth anniversary here at Xilinx and have had a great time participating in the Herculean task of bringing to market two new generations of silicon—the 40-nanometer 6 series FPGAs and the 28-nm 7 series devices. I'm proud to be a part of what is most likely the single most inspirational and innovative moment in the history of programmable logic since Xilinx introduced the very first FPGA, the XC2064, in 1985.

Along with being the first to market with 28-nm silicon in 2011, Xilinx introduced two revolutionary technologies—the Zynq™-7000 Extensible Processing Platform and the Virtex®-7 2000T FPGA.

If you've been a faithful reader of *Xcell Journal* over the last couple of years, you are familiar with these two great devices. The Zynq-7000 EPP marries a dual ARM® Cortex™-A9 MPCore processor with programmable logic on the same device, and boots from the processor core rather than from programmable logic. It enables new vistas in system-level integration for traditional FPGA designers and opens up the world of programmable logic to a huge user base of software engineers. The possibilities are endless.

I'm not alone in my enthusiasm for this device: The editors and readers of *EE Times* and *EDN* recently voted the Zynq-7000 the Ultimate SoC of 2011 in the UBM Electronics ACE Awards competition (see *http://www.eetimes.com/electronics-news/4370156/Xilinx-Zynq-7000-receives-product-of-the-year-ACE-award*).

Not to be outdone, the Virtex-7 2000T, an ACE Awards finalist in the Ultimate Digital IC category, is in my opinion an equally if not even more technologically impressive accomplishment. It is the first commercially available FPGA with Xilinx's 3D stacked-silicon interconnect (SSI) technology, in which four 28-nm programmable logic dice (what we call slices) reside side-by-side on a passive silicon interposer. By stacking the dice, Xilinx was able to make the Virtex-7 2000T the world's single largest device in terms of transistor counts and by far the highest-capacity programmable logic device that has every existed.

The SSI technology not only allows customers to speed past Moore's Law but also opens up new integration possibilities in which Xilinx can integrate different types of dice on a single device, speeding up the pace of user innovation. For example, Xilinx has announced the Virtex-7 HT family of devices, enabled by SSI technology. Each member of this family will include transceiver slices alongside programmable logic slices. The Virtex-7 HT family will allow wired communications companies to create equipment to conform to new bandwidth standards for 100 Gbps and beyond. The biggest device in the family, the Virtex-7 H870T, will allow companies to create equipment that can run at up to 400 Gbps—developing equipment at the leading edge of advanced communications standards.

And now, to put the icing on the cake so to speak, Xilinx is launching its new Vivado™ Design Suite (cover story). Vivado, which the company started developing four years ago, not only blows away the runtimes of the ISE® Design Suite but is built from the ground up using open standards and modern EDA technologies, even high-level synthesis, that should dramatically speed up productivity for the 7 series devices and many generations of FPGAs to come.

I highly recommend you check out the new 7 series devices and the Vivado Design Suite. If you happen to be available for a trip to San Francisco in early June, Xilinx will be exhibiting at the Design Automation Conference (*www.dac.com*) from June 3 to 7 at Booth 730. You'll find me there, or at three of the Pavilion Panels I'm organizing on DAC's show floor (Booth 310): "Gary Smith on EDA: Trends and What's Hot at DAC," on Monday, June 4, 9:15-10:15 a.m.; "Town Hall: Dark Side of Moore's Law" on Wednesday, June 6, 9:15 to 10:15 a.m.; and "Hardware-Assisted Prototyping and Verification: Make vs. Buy?" on Wednesday, June 6, 4:30 to 5:15 p.m. I hope to see you there.

Mike Santarini
Publisher

Pass GO, Collect 3.6 Billion Samples per Second

COLLECT 3.6 GSPS AS YOU PASS
GO
Chance
INSTANT QUOTING & DATASHEETS AT WWW.INNOVATIVE-DSP.COM
XILINX VIRTEX-6 XC6VLX240T

X6 GSPS

Features

- Two 1.8 GSPS, 12-bit A/D Channels
- Single channel interleaved @ 3.6GHz
- +/-1V, AC-Coupled, 50 ohm, SMA Inputs
- Xilinx Virtex-6 SX315T/SX475T or LX240T
- 4 Banks of 1GB DRAM (4 GB total)
- Rugged levels available

Extremely Versatile

Complete chassis & embedded PC solutions available! Adapt to VPX, Cabled PCI-Express, PCI-Express, 64-bit PCI, and CompactPCI with our high-quality carriers.

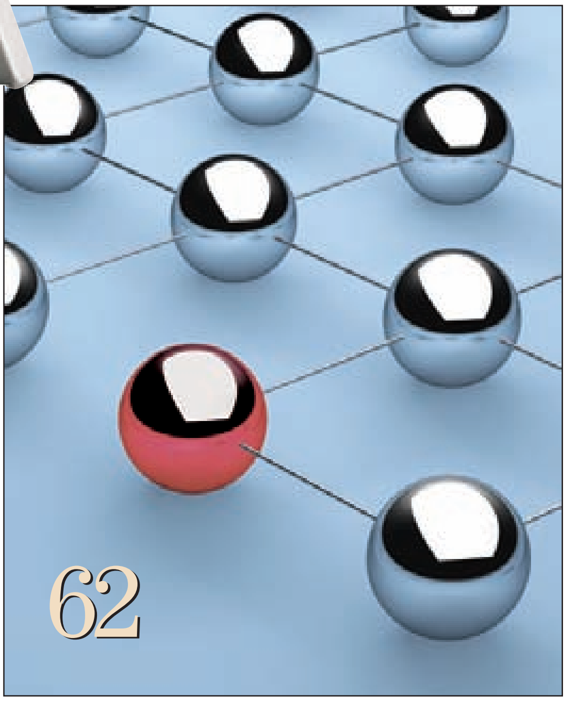
VIRTEX 6 · wireless ip cores

FrameWork Logic · RoHS

805.578.4260 phone • www.innovative-dsp.com

Innovative Integration
a subsidiary of ISI
... real time solutions!

THE XILINX XPERIENCE FEATURES



XTRA READING

2010 APEX® AWARDS FOR PUBLICATION EXCELLENCE
Excellence in Magazine & Journal Writing 2010, 2011

2011 APEX® AWARDS FOR PUBLICATION EXCELLENCE
Excellence in Magazine & Journal Design and Layout 2010, 2011

Xilinx Unveils Vivado Design Suite for the Next Decade of 'All Programmable' Devices

State-of-the-art EDA technologies and methods underlie a new tool suite that will radically improve design productivity and quality of results, allowing designers to create better systems faster and with fewer chips.

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

After four years of development and a year of beta testing, Xilinx is making its Vivado™ Design Suite available to customers via its early-access program, ahead of public access this summer. Vivado provides a highly integrated design environment with a completely new generation of system- to IC-level tools, all built on the backbone of a shared scalable data model and a common debug environment. It is also an open environment based on industry standards such as the AMBA® AXI4 interconnect, IP-XACT IP packaging metadata, the Tool Command Language (Tcl), Synopsys Design Constraints (SDC) and others that facilitate design flows tailored to the users' needs. Xilinx architected the Vivado Design Suite to enable the combination of all types of programmable technologies and to scale up to 100 million ASIC equivalent-gate designs.

"Over the last four years, Xilinx has pushed semiconductor innovation to new heights and unleashed the full system-level capabilities of programmable devices," said Steve Glaser, senior vice president of corporate strategy and marketing. "Over this time, Xilinx has evolved into a company that develops All Programmable Devices, extending programmability beyond programmable logic and I/O to software-programmable ARM subsystems, 3D ICs and analog mixed signal. We are enabling new levels of programmable system integration with devices such as the award-winning Zynq™-7000 Extensible Processing Platform, the 3D Virtex®-7 stacked-silicon interconnect (SSI) technology devices and the world's most advanced FPGAs. Now, with Vivado, we are offering a state-of-the-art tool suite that will accelerate the productivity of customers using these All Programmable Devices for the next decade."

Glaser said Xilinx developed All Programmable Devices to enable customers to achieve new levels of programmable systems integration, increased system performance, lower BOM cost and total system power reduction, and ultimately to accelerate design productivity so they can get their innovations to market quickly. To accomplish this, Xilinx needed to create a tool suite as innovative as its new silicon—a suite that would address nagging integration and implementation design-productivity bottlenecks.

"Customers face a number of integration bottlenecks, including integrating algorithmic C and register-transfer level (RTL) IP; mixing the DSP, embedded, connectivity and logic domains; verifying blocks and 'systems'; and reusing designs and IP," said Glaser. "They also face several implementation bottlenecks, including hierarchical chip planning and partitioning; multidomain and multidie physical optimization; multivariant 'design' vs. 'timing' closure; and late ECOs and the rippling effects of design changes. The new Vivado Design Suite addresses these bottlenecks and empowers users to take full advantage of the system integration capabilities of our All Programmable Devices."

In developing the Vivado Design Suite, Xilinx leveraged industry standards and employed state-of-the-art EDA technologies and techniques. The result is that all designers—from those who require a highly automated, push-button flow to those who are extremely hands-on—will be able to design even the largest Xilinx devices far faster and more effectively than before, while working in a state-of-the-art EDA environment that retains a familiar, intuitive look and feel.

The Vivado Design Suite gives customers a modern set of tools with full-system programmability features that far surpass the capabilities of the long-time flagship ISE® Design Suite. To help customers transition smoothly, Xilinx will continue to develop and support ISE indefinitely for those targeting 7 series and older Xilinx FPGA technologies. Going forward, the Vivado Design Suite will be the company's flagship design environment, supporting all 7 series and future devices from Xilinx.

Tom Feist, senior director of design methodology marketing at Xilinx, expects that when customers launch the Vivado Design Suite, the benefits over ISE will become immediately evident.

"The Vivado Design Suite improves user productivity by offering up to 4X runtime improvements over competing tools, while heavily leveraging industry standards such as SystemVerilog, SDC, C/C++/SystemC, ARM's AMBA® AXI version 4 interconnect and interactive Tcl scripting," said Feist. "Other highlights include comprehensive cross-probing of the Vivado's many reports and design views, state-of-the-art graphics-based IP integration and, last but not least, the first fully supported commercial deployment of high-level synthesis—C++ to HDL—by an FPGA vendor."

TOOLS FOR THE NEXT ERA OF PROGRAMMABLE DESIGN

Xilinx originally introduced its ISE Design Suite back in 1997. The suite featured a then very innovative timing-driven place-and-route engine that Xilinx had gained in its April 1995 acquisition of NeoCAD. Over a decade and a half, Xilinx added numerous new technologies—including multi-language synthesis and simulation, IP integration and a host of editing and test utilities—to the suite, striving to constantly improve its design tools on all fronts as FPGAs became capable of performing increasingly more complex functions. In creating the new Vivado Design Suite, Feist said that Xilinx drew upon all the lessons learned with ISE, appropriating its key technologies while also leveraging modern EDA algorithms, tools and techniques.

"The Vivado Design Suite will greatly improve design productivity for today's designs and will easily scale for the capacity and design-complexity challenges of 20-nanometer silicon and beyond," said Feist. "EDA technology has evolved greatly over the last 15 years. In building this tool from scratch, we were able to create a suite that employs the latest EDA technologies and standards and will scale nicely into the foreseeable future."

DETERMINISTIC DESIGN CLOSURE

At the heart of any FPGA vendor's integrated design suite is the physical-implementation flow—synthesis, floorplanning, placement, routing, power and timing analysis, optimization and ECO. With Vivado, Xilinx has built a state-of-the-art implementation flow to help customers quickly achieve design closure.

SCALABLE DATA MODEL ARCHITECTURE

To cut down on iterations and overall design time and to improve overall productivity, Xilinx built its implementation flow using a single, shared, scalable data model—a framework also found in today's most advanced ASIC design environments. "This shared scalable data model allows all the steps in the flow—synthesis, simulation, floorplanning, place and route, etc.—to operate on an in-memory data model that enables debug and analysis at every step in the process, so that users have visibility into key design metrics such as timing, power, resource utilization and routing congestion much earlier in the design processes," said Feist. "These estimates become progressively more accurate as the design progresses through the steps in the implementation processes."

Specifically, the unified data model allowed Xilinx to tightly link its new multidimensional, analytical place-and-route engine with the suite's RTL synthesis engine, new multiple-language simulation engines as well as individual tools such as the IP Integrator, Pin Editor, Floor Planner and Device Editor. Customers can use the tool suite's comprehensive cross-probing function to track and cross-probe a given problem from schematics, timing reports or logic cells to any other view and all the way back to HDL code.

"You now have analysis at every step of the design process and every step is connected," said Feist. "We also provide analysis for timing, power, noise and resource utilization at every stage of the flow after synthesis. So if I learn early that my timing or power is way off, I can do short iterations to address the issue proactively rather than run long iterations, perhaps several of them, after it's been placed and routed."

Feist said that tight integration afforded by the scalable data model enhanced the effectiveness of push-button flows for users who want maximum automation, relying on their tools to do the vast majority of the work. At the same time, he said, it also gives those users who require more-advanced controls better analysis and command of their every design move.

HIERARCHICAL CHIP PLANNING, FAST SYNTHESIS

Feist said that Vivado provides users with the ability to partition the design for processing by synthesis, implementation and verification, facilitating a divide-and-conquer team approach to big projects. A new design-preservation feature enables repeatable timing results and the ability to perform partial reconfiguration of the design.

Vivado also includes an entirely new synthesis engine that is designed to handle millions of logic cells. Key to the new synthesis engine is superior support for SystemVerilog. "Vivado's synthesis engine supports the synthesizable subset of the SystemVerilog language better than any other tool in the market," said Feist. It is three times faster than XST, the Xilinx Synthesis Technology in the ISE Design Suite, and supports a "quick" option that lets designers rapidly get a feeling for the area and size of the design, allowing them to debug issues 15 times faster than before with an RTL or gate-level schematic. With more and more ASIC designers moving to programmable platforms, Xilinx

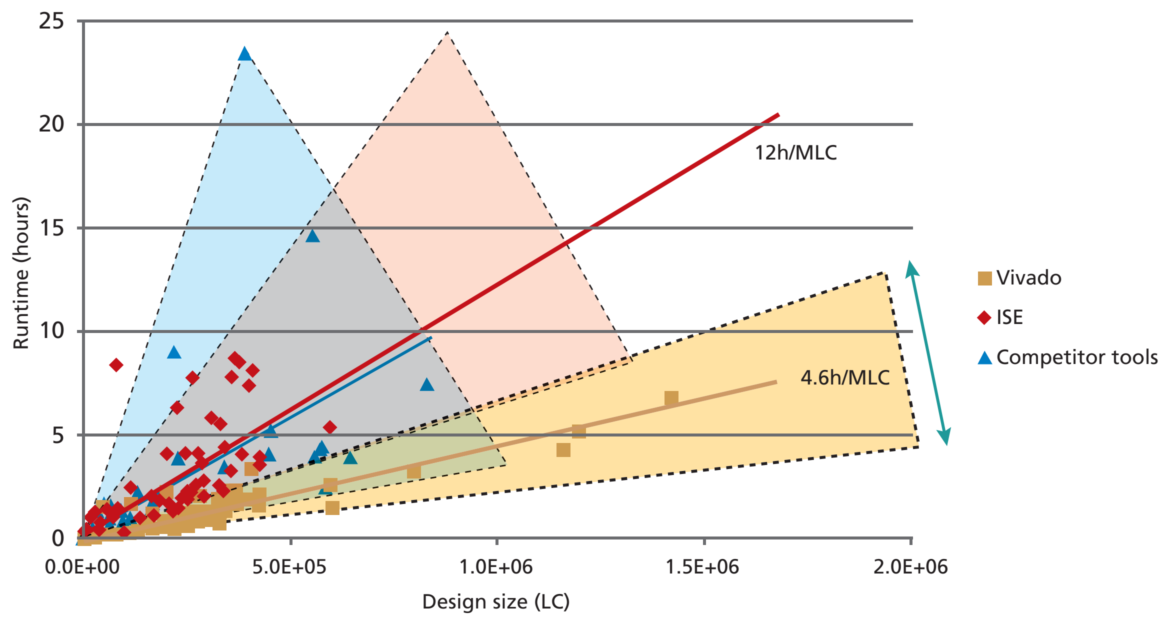
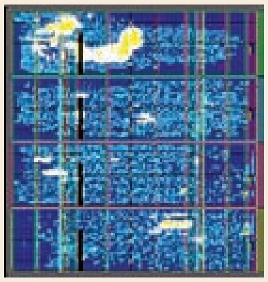
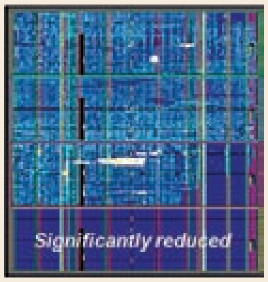


Figure 1 – The Vivado Design Suite implements large and small designs more quickly and with better-quality results than other FPGA tools.

	ISE	Vivado
P&R runtime	13 hrs.	**5 hrs.**
Memory usage	16 GB	**9 GB**
Wire length and congestion		

*Zynq emulation platform

Figure 2 – The Vivado Design Suite's multidimensional analytic algorithm optimizes layouts for best timing, congestion and wire length, not just best timing.

is also leveraging Synopsys Design Constraints throughout the Vivado flow. The use of standards opens up new levels of automation where customers can now access state-of-the-industry EDA tools for things like constraint generation, cross-domain clock checking, formal verification and even static timing analysis with tools like PrimeTime from Synopsys.

MULTIDIMENSIONAL ANALYTICAL PLACER

Feist explained that the older-generation FPGA vendor design suites use one-dimensional timing-driven place-and-route engines powered by simulated annealing algorithms that determine randomly where the tool should place logic cells. With these routers, users enter timing; then the simulated annealing algorithm pseudorandomly places features to get a "best as it can" match to timing requirements. "In those days it made sense, because designs were much smaller and logic cells were the main cause of delays," said Feist. "But today, with complex designs and advances in silicon processes, interconnect and design congestion contribute to the delay far more."

Place-and-route engines with simulated annealing algorithms do an adequate job for FPGAs below 1 million gates, "but they really start to underperform as designs grow," said Feist. "Not only do they struggle with congestion, but the results start to become increasingly more unpredictable as designs grow further beyond 1 million gates."

With an eye toward the multimillion-gate future, Xilinx developed a modern multidimensional analytic placement engine for the Vivado Design Suite that is on par with those found in million-dollar ASIC place-and-route tools. This engine analytically finds a solution that primarily minimizes three dimensions of a design: timing, congestion and wire length. "The Vivado Design Suite's algorithm globally optimizes for best timing, congestion and wire length simultaneously, taking into account the entire design instead of the local-move approach done with simulated annealing," said Feist. "As a result, the tool can place and route 10 million gates quickly, deterministically and with consistently strong quality of results" (see Figure 1). "Because it is solving for all three factors simultaneously, it means you run fewer iterations in your flow."

To illustrate this advantage, Xilinx ran the raw RTL for the Zynq-7000 EPP emulation platform, a very large and complex design, in both the ISE Design Suite and Vivado Design Suite in a push-button mode. Each tool was instructed to target Xilinx's largest FPGA device—the SSI-enabled Virtex-7 2000T FPGA. The Vivado Design Suite's place-and-route engine took five hours to place the 1.2 million logic cells, while the ISE Design Suite version 13.4 took 13 hours (Figure 2). The Vivado Design Suite also implemented the design with much less congestion (as seen in the gray and yellow portions of the design) and in a smaller area, reflecting the total wire-length reduction. In addition, the Vivado Design Suite implementation had better memory compilation efficiency, taking only 9 Gbytes to implement the design's required memory to ISE Design Suite's 16 Gbytes.

"Essentially what you're seeing is that the Vivado Design Suite met all constraints and only needed three-quarters of the device to implement the entire design," said Feist. "That means users could add even more logic functionality and on-chip memory to their designs [in the extra space] or, alternatively, even move to a smaller device."

POWER OPTIMIZATION AND ANALYSIS

Today, power is one of the most critical aspects of FPGA design. As such, the Vivado Design Suite focuses on advanced power-optimization techniques to provide greater power reductions for users' designs. "The technology uses advanced clock-gating techniques found in today's advanced ASIC tool suites and is capable of analyzing design logic and removing unnecessary switching activity by applying clock gating," said Feist. "Specifically, the new technology focuses on the switching-activity factor 'alpha.' It is able to achieve up to a 30 percent reduction in dynamic power." Feist said Xilinx introduced the technology in the ISE Design Suite last year but is carrying it forward and will continue to enhance it in Vivado.

In addition, with the new shared scalable data model, users can get power estimates at every stage of the design flow, enabling up-front analysis so that problem areas can be addressed early in the design flow, said Feist.

SIMPLIFYING ENGINEERING CHANGE ORDERS

Incremental flows make it possible to quickly process small design changes by simply reimplementing a small part of the design, making iterations faster after each change. They also enable performance preservation after each incremental change, thus reducing the need for multiple design iterations. Toward this end, the Vivado Design Suite includes a new extension to the popular ISE FPGA Editor tool called the Vivado Device Editor. Feist said that using the Vivado Device Editor on a placed-and-routed design, designers now have the power to make engineering change orders (ECOs)—to move instances, reroute nets, tap a register to a primary output for debug with a scope, change the parameters on a digital clock manager (DCM) or a lookup table (LUT)—late in the design cycle, without needing to go back through synthesis and implementation. No other FPGA design environment offers this level of flexibility, he said.

The tools will work for all levels of users, 'from folks who want an entirely pushbutton flow to folks who do analysis at each phase of the design.'

FLOW AUTOMATION, NOT FLOW DICTATION

In building the Vivado Design Suite, the Xilinx tool team's mantra was to automate—not dictate—the way people design. "Whether they start in C, C++, SystemC, VHDL, Verilog or SystemVerilog, MATLAB® or Simulink®—and whether they use our IP or third-party IP—we offer a way to automate all those flows and help customers be more productive," said Feist. "We also accounted for the broad range of skill sets and preferences of our users—from folks who want an entirely pushbutton flow to folks who do analysis at each phase of the design, and even for those who think GUIs are for wimps and want to do everything in command-line or batch mode via TCL. Users are able to suit the suite's features to their specific needs."

THE IP PACKAGER, INTEGRATOR AND CATALOG

Xilinx's tool architecture team placed top priority on giving the new suite specialized IP features to facilitate the creation, integration and archiving of intellectual property. To this end, Xilinx has created three new IP capabilities in Vivado, called IP Packager, IP Integrator and the Extensible IP Catalog.

"Today, it is hard to find an IC design that doesn't incorporate some amount of IP," said Feist. "By adopting industry standards and offering tools to specifically facilitate the creation, integration and archiving/upkeep of IP, we are helping IP vendors in our ecosystem and customers to quickly build IP and improve design productivity. More than 20 vendors are already offering IP supporting the new suite."

IP Packager allows Xilinx customers, IP developers and ecosystem partners to turn any part of their design—or indeed, the entire design—into a reusable core at any level of the design flow: RTL, netlist, placed netlist and even placed-and-routed netlist. The tool creates an IP-XACT description of the IP that users can easily integrate into future designs. For its part, the IP Packager specifies the data for each piece of IP in an XML file. Feist said that once you have the IP packaged, you can use the new IP Integrator to stitch it into the rest of your design.

"IP Integrator allows customers to integrate IP into their designs at the interconnect level rather than at the pin level," said Feist. "You can drag and drop the pieces of IP onto your design and it will check up front that the respective interfaces are compatible. If they are, you draw one line between the cores and it will automatically write the detailed RTL that connects all the pins."

Once you've merged, say, four or five blocks into your design with IP Integrator, he said, "you can take the output of that [process] and run it back through the IP Packager." The result "then becomes a piece of IP that other people can reuse," said Feist. "And this IP isn't just RTL, it can be a placed netlist or even a placed-and-routed IP netlist block, which further saves integration and verification time."

A third feature, the Extensible IP Catalog, allows users to build their own standard repositories from IP they've created or licensed from Xilinx and third-party vendors. The catalog, which Xilinx built to conform to the requirements of the IP-XACT standard, allows design teams and even enterprises to better organize their IP and share it across their organization. Feist said that the Xilinx System Generator and IP Integrator are part of the Vivado Extensible IP Catalog so that users can easily access catalogued IP and integrate it into their design projects.

"Instead of having third-party IP vendors deliver their IP in a zip file and with various deliverables, they can now deliver it to you in a unified format that is instantly accessible and compatible with the Vivado suite," said Ramine Roane, director of product marketing for Vivado.

VIVADO HLS TAKES ESL MAINSTREAM

Perhaps the most forward looking of the many new technologies in the Vivado Design Suite release is Vivado HLS (high-level synthesis), which Xilinx gained in its acquisition of AutoESL in 2010. Xilinx conducted an extensive evaluation of commercial electronic system-level (ESL) design offerings before acquiring the best in the industry. A study by research firm BDTI helped Xilinx's acquisition choice (see *Xcell Journal* issue 71, "BDTI Study Certifies High-Level Synthesis Flows for DSP-Centric FPGA Design," *http://www.xilinx.com/publications/archives/xcell/Xcell71.pdf*).

"Vivado HLS provides comprehensive coverage of C, C++ and SystemC, and does floating-point as well as arbitrary precision floating-point [calculations]," said Feist. "This means that you can work with the tool in an algorithm-development environment rather than a typical hardware environment, if you wish. A key advantage of doing this is that the algorithms you developed at that level can be verified orders of magnitude faster than at the RTL. That means you get simulation acceleration but also the ability to explore the feasibility of algorithms and make, at an architectural level, trade-offs in terms of throughput, latency and power."

Designers can use the Vivado HLS tool in many ways to perform a wide range of functions. But for demonstration purposes, Feist outlined a common flow users can employ for developing IP and integrating it into their designs.

In this flow, users create a C, C++ or SystemC representation of their design and a C testbench that describes its desired behavior. They then verify the system behavior of their design using a GNU Compiler Collection/G++ or Visual C++ simulator. Once the behavioral design is functioning satisfactorily and the accompanying testbench is ironed out, they run the design through Vivado HLS synthesis, which will generate an RTL design: Verilog or VHDL. With the RTL they can then perform Verilog or VHDL simulation of the design or have the tool create a SystemC version using the C-wrapper technology. Users can then perform SystemC architectural-level simulation and further verify the architectural behavior and functionality of the design against the previously created C testbench.

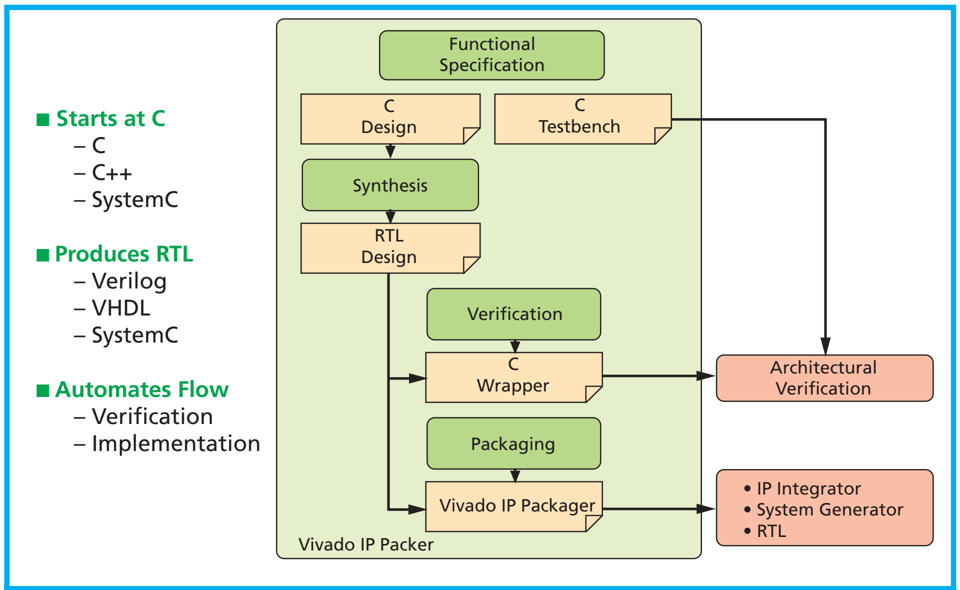


Figure 3 – Vivado HLS allows design teams to begin their designs at a system level.

Once the design has been solidified, users can put it through the Vivado Design Suite's physical-implementation flow to program their design into a device and run it in hardware. Alternatively, they can use the IP Packager to turn the design into a reusable piece of IP, stitch the IP into a design using IP Integrator or run it in System Generator.

This is merely one way to use the tool. In fact, in this issue of *Xcell Journal*, Agilent's Nathan Jachimiec and Xilinx's Fernando Martinez Vallina describe how they used the Vivado HLS technology (called AutoESL technology in the ISE Design Suite flow) to develop a UDP packet engine for Agilent.

VIVADO SIMULATOR

In addition to Vivado HLS, Xilinx also created a new mixed-language simulator for the suite that supports Verilog and VHDL. With a single click of the mouse, Feist said, users can launch behavioral simulations and view results in an integrated waveform viewer. Simulations are accelerated at the behavioral level using a new performance-optimized simulation kernel that executes up to three times faster than the ISE simulator. Gate-level simulations can also run up to 100 times faster using hardware co-simulation.

AVAILABILITY IN 2012

Where Xilinx offered the ISE Design Suite in four editions aimed at different types of designers (Logic, Embedded, DSP and System), the company will offer the Vivado Design Suite in two editions. The base Design Edition includes the new IP tools in addition to Vivado's synthesis-to-bitstream flow. Meanwhile, the System Edition includes all the tools of the Design Edition plus System Generator and Xilinx's new Vivado HLS.

The Vivado Design Suite version 2012.1 is available now as part of an early-access program. Customers should contact their local Xilinx representative for more information. Public access will commence with version 2012.2 in the middle of the second quarter, followed by WebPACK availability later in the year. ISE Design Suite Edition customers with current support will receive the new Vivado Design Suite Editions in addition to ISE at no additional cost.

Xilinx will continue to support and develop the ISE Design Suite for customers targeting devices prior to the 28-nm generation. To learn more about Vivado, please visit *www.xilinx.com/design-tools* or come see the suite in action at the Design Automation Conference (DAC), June 3-7 in San Francisco, Booth 730.

High-Level Synthesis Tool Delivers Optimized Packet Engine Design

AutoESL enabled the creation of an in-fabric, processor-free UDP network packet engine.

by Nathan Jachimiec, PhD
R&D Engineer
Agilent Technologies
Technology Leadership Organization
nathan_jachimiec@agilent.com

Fernando Martinez Vallina, PhD
Software Applications Engineer
Xilinx, Inc.
vallina@xilinx.com

Gigabit Ethernet is one of the most ubiquitous interconnect options available to link a workstation or laptop to an FPGA-based embedded platform due to the availability of the hardened tri-Ethernet MAC (TEMAC) primitive. The primary impediment in developing Ethernet-based FPGA designs is the perceived processor requirement necessary to handle the Internet Protocol (IP) stack. We approached the problem using the AutoESL high-level synthesis tool to develop a high-performance IPv4 User-Datagram Protocol (UDP) packet transfer engine.

Our team at Agilent's Measurement Research Lab wrote original C source code based on Internet Engineering Task Force requests for comments (RFCs) detailing packet exchanges among several protocols, namely UDP, the Address Resolution Protocol (ARP) and the Dynamic Host Configuration Protocol (DHCP). This design implements a hardware packet-processing engine without any need for a CPU. The architecture is capable of handling traffic at line rate with minimum latency and is compact in logic-resource area. The usage of AutoESL makes it easy to modify the user interface with minimum effort to adapt to one or more FIFO streams or to multiple RAM interface ports. AutoESL is a new addition to the Xilinx® ISE® Design Suite and is called Vivado™ HLS in the new Vivado Design Suite (see cover story).

IPV4 USER DATAGRAM PROTOCOL

Internet Protocol version 4 (IPv4) is the dominant protocol of the Internet, with version 6 (IPv6) growing steadily in popularity. When most developers discuss IP, they commonly refer to the Transmission Control Protocol, or TCP, a connection-based protocol that provides reliability and congestion management. But for many applications such as video streaming, telephony, gaming or distributed sensor networks, increased bandwidth and minimal latency trump reliability. Hence, these applications typically use UDP instead.

UDP is connectionless and provides no inherent reliability. If packets are lost, duplicated or sent out of order, the sender has no way of knowing and it is the responsibility of the user's application to perform some packet inspection to handle these errors. In this regard, UDP has been nicknamed the "unreliable" protocol, but in comparison to TCP, it offers higher performance. UDP support is available in nearly every major operating system that supports IP. High-level software programming languages refer to network streams as "sockets" and UDP as a datagram socket.

SENSOR NETWORK ARCHITECTURE

At Agilent, we developed a LAN-based sensor network that interfaces an analog-to-digital converter (ADC) with a Xilinx Virtex®-5 FPGA. The FPGA performs data aggregation and then streams a requested number of samples to a predetermined IP address—that is, a host PC. Because the block RAM of our FPGA was almost completely devoted to signal processing, we did not have enough memory to contain the firmware for a soft processor. Instead, we opted to implement a minimal set of networking functions to transfer sensor data via UDP back to a host. Due to the need for high bandwidth and low latency, UDP packet streaming was the preferred network mode.

Because of the time-sensitive nature of the data, a new set of sample data is more pertinent than any retransmission of lost samples. One of the two challenging issues we faced was to avoid overloading the host device. That meant we had to find a way of efficiently handling the large number of inbound samples. The second major challenge was quickly formatting the UDP packet and calculating the required IP header fields and the optional, but necessary, UDP payload checksum, before the next set of samples overflowed internal buffers.

INITIAL HDL DESIGN

An HDL implementation of the packet engine was straightforward given preexisting pseudocode, but not optimal for our FPGA hardware. C and pseudocode provided from various sources simplified verification. In addition, tools such as Wireshark, the open-source packet analyzer, and high-level languages such as Java simplified the process of simulation and in-lab verification.

Using provided pseudocode, the task of developing Verilog to generate the packet headers involved coding a state machine, reading the sample FIFO and assembling the packet into a RAM-based buffer. We broke the design into three main modules, RX Flow, TX Flow and LAN MCU, as shown in Figure 1. As packets arrive from the LAN, the RX Flow inspects them and passes them either to the instrument core or to the LAN MCU for processing, such as when handling ARP or DHCP packets.

The TX Flow packet engine reads N ADC samples from a TX FIFO and computes a running payload checksum for calculating the UDP checksum. The TX FIFO buffers new samples as they arrive, while the LAN MCU prepares the payload of a yet-to-be-transmitted packet. After fetching the last requested sample, the LAN MCU computes the remaining header fields of the IP/UDP packet. In network terminology, this procedure is a TX checksum offload.

Once the packet fields are generated, the LAN MCU sends the packet to the TEMAC for transmission but retains it until the TEMAC acknowledges successful transmission—not reception by the destination device. As this first packet is awaiting transmission by the TEMAC, new sensor samples are arriving into the TX FIFO. When the first packet is finished, our packet engine releases the buffer to prepare for the next packet. The process continues in a double-buffered fashion. If the TEMAC signals an error and the next transmit buffer overflow is imminent, then the packet is lost to allow the next sample set to continue, but an exception is noted. Due to time-stamping of the

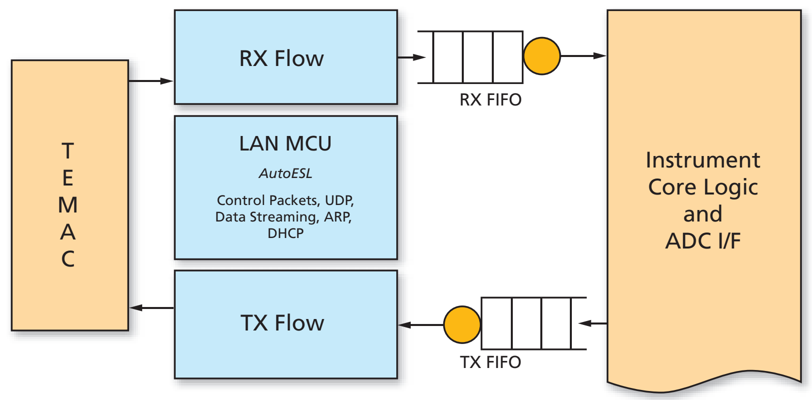

RX Flow
RX FIFO
T E M A C
LAN MCU
AutoESL
Control Packets, UDP, Data Streaming, ARP, DHCP
Instrument Core Logic and ADC I/F
TX Flow
TX FIFO

Figure 1 – Our UDP packet engine design consisted of three main modules: RX Flow, TX Flow and LAN MCU.

sample set incorporated into our packet format, the host will realize a discontinuity in the set and accommodate it.

The latency to transmit a packet is the number of cycles it takes to read in *N* ADC samples plus the cycles to generate the packet header fields, including the IPv4 flags, source and destination address fields, UDP pseudo header and both the IP and UDP checksums. The checksum computations are rather problematic since they require reading the entire packet, yet they lie before the payload bytes.

AutoESL's ability to abstract the FIFO and RAM interfaces proved to be one of the most beneficial optimizations for performance.

CODING HDL IN THE DARK

To support the high-bandwidth and low-latency requirements of the sensor network, we needed an optimal hardware design to keep up with the required sample rate. The straightforward approach we implemented first in Verilog failed to meet a 125-MHz clock rate without floorplanning, and took 17 clock cycles to generate the IP/UDP packet header fields. As we developed the initial HDL design, ChipScope™ was vital to understanding the nuances of the TEMAC interface, but it also impeded the goal of achieving a 125-MHz clock. The additional logic-capture circuits altered the critical path and would require manual floorplanning for timing closure.

The critical path was calculating the IP and UDP header checksums, because our straightforward design used a four-operand adder to sum multiple header fields together in various states of our design. Our HDL design attempted a "greedy" scheduling algorithm that tried to do as much work as possible per cycle of the state machine. By removing ChipScope on these operations and by floorplanning, we closed timing.

The HDL design also used only one port of a 32-bit-wide block RAM that acted as our transmit packet buffer. We chose a 32-bit-wide memory because that's the native width of the BRAM primitive and it allowed for byte-enable write accesses that would avoid the need for read-modify-write access to the transmit buffer.

Using byte enables, the finite state machine (FSM) writes directly to the header field bytes needing modification at a RAM address. However, what seemed like good design choices based on knowledge of the underlying Xilinx fabric and algorithm yielded a nonoptimal design that failed to meet timing without manual placement of the four-input adders.

Because the UDP algorithms were already available in various forms in C code or written as pseudocode in IP-related RFC documentation, recoding the UDP packet engine in C was not a major task and proved to yield a better insight to the packet header processing. Just taking the pseudocode and starting to write Verilog may have made for quicker coding, but this methodology would have sacrificed performance without fully studying the data and control flows involved.

ADVANTAGE AUTOESL

The ability for AutoESL to abstract the FIFO and RAM interfaces proved to be one of the most beneficial optimizations for performance. With the ability to code directly in C, we could now easily include both ARP and DCHP routines into our packet engine. Figure 2 shows a flowchart of our design. Our HDL design utilized a byte-wide FIFO interface that connected to the aggregation and sensor interface of our design, which remained in Verilog. Also, our Verilog design utilized a 32-bit memory interface that collected 4 bytes of sample data and then saved it in the transmit buffer RAM as a 32-bit word.

By means of its "array reshape" directive, AutoESL optimized the memory interface so that the transmit buffers, while written in C code as an 8-bit memory, became a 32-bit memory. This meant the C code could avoid having to do many bit manipulations of the header fields, as they would require bit shifting to place into a 32-bit word. It also alleviated little-endian vs. big-endian byte-ordering issues. This optimization reduced the latency of the TX offload function that computes the packet checksums and generates header fields from 17 clocks, as originally written in Verilog, to just seven clock cycles while easily meeting timing. AutoESL could do better in the future, since this current version does not have the ability to manipulate byte enables on RAM writes. Byte-enabled memory support is on the long-term road map for the tool.

Another optimization that AutoESL performed, which we found by serendipity, was to access both ports of our memory, since Xilinx block RAM is inherently dual-port. Our Verilog design reserved the second port of the transmit buffer so that its interface to the TEMAC would be able to access the buffer without any need for arbitration. By allowing AutoESL to optimize for our true dual-port RAM, it was capable of performing reads or writes from two different locations of the buffer. In effect, this wound up halving the number of cycles necessary to generate the header. The reduction in latency was well worth the effort in creating a simple arbiter in Verilog for the second port of the memory so that the TEMAC interface could access the memory port that AutoESL usurped.

We controlled the bit widths of the transmit buffer and the sample FIFO interfaces via directives. Unfortunately, AutoESL does not automatically optimize your design. Instead, you have to experiment with a variety of directives and determine through trial and error which of them is delivering an improvement. For our design, reducing the number of clock cycles to process the packet fields while operating at 125 MHz was the goal.

The "array reshape" and loop "pipeline" directives were important for optimizing the design. The reshape directive alters the bit width of the RAM and FIFO interfaces, which ultimately led to processing multiple header fields in parallel per clock cycle and writeback to memory. The optimal combination that yielded the least cycles was a transmit buffer bit width of 32. The width of the FIFO feeding ADC samples was not a factor in reducing the overall latency because it's impossible to force samples to arrive any faster.

The loop-pipelining directive is extremely important too, because it indicates to the compiler that our loops that push and pop from our FIFO interfaces can operate back-to-back. Otherwise, without the pipeline directive, AutoESL spent three to 20 clock cycles between pops of the FIFO due to scheduling reasons. It is therefore vital to utilize pipelining as much as possible to attain low latency when streaming data between memories.

Xilinx block RAM also has a programmable data output latency of one to three clock cycles. Allowing three cycles of read latency enables the minimum "clock to Q" timing. To experiment with different read latencies was only a matter of changing the "latency" directive for the RAM primitive or "core" resource. Because of the scheduling algorithms that AutoESL performed, adding a read latency of three cycles to access the RAM only tacked on one additional cycle of latency to the overall packet header generation. The extra cycle of memory latency allowed for more slack in the design, and that aided the place-and-route effort.

We also implemented ARP and DHCP routines in our AutoESL design that we had avoided doing before because of the level of effort required to code them in Verilog. While not difficult, both ARP and DHCP are extremely cumbersome to write in Verilog and would require a great number of states to perform. For instance, the ARP request/response exchange required more than 70 states. One coding error in the Verilog FSM would likely require multiple days to undo. For this reason alone, many designers would prefer just to use a CPU to run these network routines.

Overall, AutoESL excelled at generating a synthesizable netlist for the UDP packet engine. The module it generated fit between our two preexisting ADC and TEMAC interface modules and performed the necessary packet header generation and additional tasks. We were able to integrate the design it created into our core design and simulate it with Mentor Graphics' ModelSim to perform functional verification. With the streamlined design, we were able to reach timing closure with less synthesis, map and place-and-route effort than with our original HDL design. Yet we have significantly more functionality now, such as ARP and DHCP support.

Comparing our original design in Verilog with our hybrid design that utilized AutoESL to craft our "LAN MCU" and "TX Flow" modules yielded impressive results. Table 1 shows a comparison of lookup table (LUT) usage. Our HDL version of TX Flow was smaller by more than 37 percent, but our AutoESL design incorporated more functionality. Most impressive is that AutoESL reduced the number of cycles to perform our packet header generation by 59 percent. Table 2 shows the latency of the "TX Offload" algorithm.

The critical path of the HDL design was computing the UDP checksum. Comparing this with the AutoESL design shows that the HDL design suf-

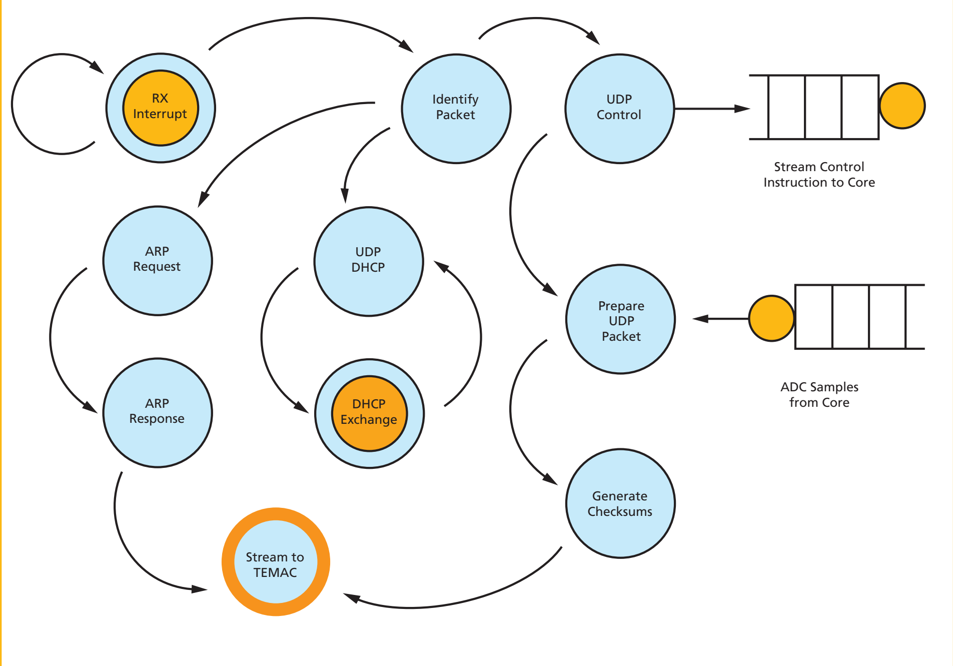


Figure 2 – Packet engine flowchart shows inclusion of ARP and DHCP.

TX Flow Resource Usage			
	HDL – TX	AutoESL - TX	% Increase
LUTs	858	1,372	37.5

Table 1 - The AutoESL design used more lookup tables but incorporated more functionality.

Latency			
	HDL	AutoESL	% Improved
Clock Cycles	17	7	58.8%

Table 2 – AutoESL improved the latency of the “TX Offload” algorithm.

fered from 10 levels of logic and a total path delay of 6.4 nanoseconds, whereas AutoESL optimized this to only three levels of logic and a path delay of 3.5 ns. Our development time for the HDL design was about a month of effort. We took about the same amount of time with AutoESL, but incorporated more functionality while gaining familiarity with the nuances of the tool.

LATENCY AND THROUGHPUT

AutoESL has a significant advantage over HDL design in that it performs control and data-flow analyses and can use this information to reorder operations to minimize latency and increase throughput. In our particular case, we used a greedy algorithm that tried to do too many arithmetic operations per clock cycle. This tool rescheduled our checksum calculations so as to use only two input adders, but scheduled them in such a way to avoid increasing overall execution latency.

Software compilers intrinsically perform these types of exercises. As state machines become more complex, the HDL designer is at a disadvantage compared to the omniscience of the compiler. An HDL designer would typically not have the opportunity to explore the effect of more than just two architectural choices because of time constraints to deliver a design, but this may be a vital task to deliver a low-power design.

The most important benefit of this tool was its ability to try a variety of scenarios, which would be tedious in Verilog, such as changing bit widths of FIFOs and RAMs, partitioning a large RAM into smaller memories, reordering arithmetic operations and utilizing dual-port instead of single-port RAM. In an HDL design, each scenario would likely cost an additional day of writing code and then modifying the testbench to verify correct functionality. With AutoESL these changes took minutes, were seamless and did not entail any major modification of the source code.

Modifying large state machines is extremely cumbersome in Verilog. The advent of tools like AutoESL is reminiscent of the days when processor designers began to employ microprogramming instead of hand-constructing the microcoded state machines of early microprocessors such as the 8086 and 68000. With the arrival of RISC architectures and hardware description languages, microprogramming is now mostly a lost art form, but its lesson is well learned in that abstraction is necessary to manage complexity. As microprogramming offered a higher layer of abstraction of state machine design, so too does AutoESL—or high-level synthesis tools in general. Tools of this caliber allow a designer to focus more on the algorithms themselves rather than the low-level implementation, which is error prone, difficult to modify and inflexible with future requirements.

USB 3.0 Board

Up to 270 MBytes/sec

Mars PM3 MX1 Kit	■ Mars MX1 SO-DIMM FPGA Module ■ Cypress FX3 USB 3.0 Controller ■ FMC LPC Connector ■ p-ITX Form Factor (100 x 72 mm)

ZYNQ Module

Designed for Linux

Mars ZX3	■ XILINX ZYNQ EPP ■ DDR3 SDRAM + NAND FLASH ■ Gigabit Ethernet + USB 2.0 OTG ■ SO-DIMM Form Factor (68 x 30 mm)

Drive FMC Card

Up to 756 Watts

FMC DR2	■ 6 Half-Bridges for 1-3 Motors ■ Supports DC, BLDC & Steppers ■ CAN PHY + 30V GPIO’s ■ FMC LPC Connector

Drive IP Core

■ Up to 8 DC, BLDC & Stepper Motors
■ Current, Velocity & Position Control
■ Field-Oriented Control & Micro-Stepping
■ Up to 100 kHz PID-FF Control Rate
■ Lowest Resource Usage & BOM Cost

www.enclustra.com

ENCLUSTRA
FPGA SOLUTIONS

FPGA Design Center

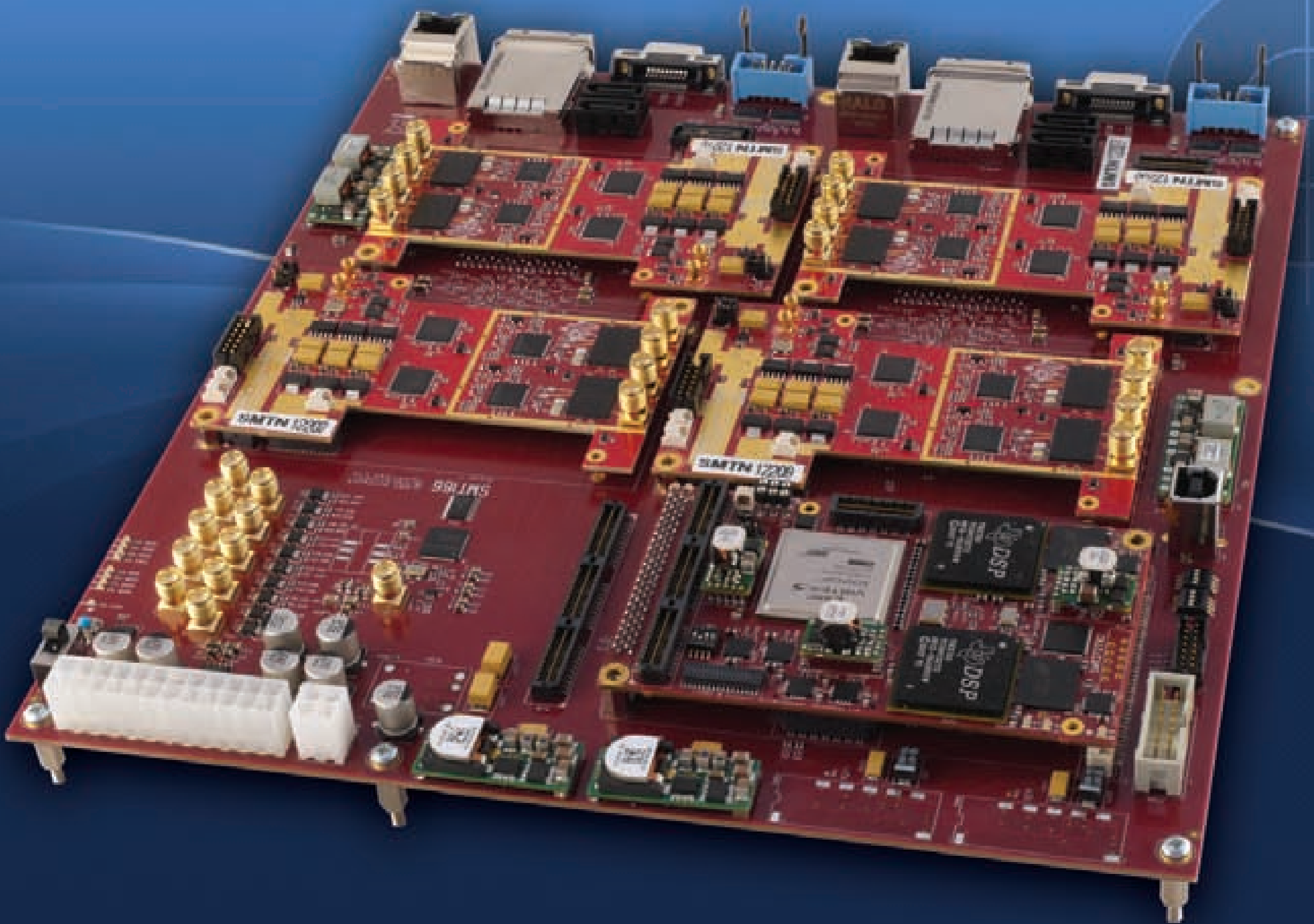
Sundance Multiprocessor Technology

●●● embedded signal processing solutions

SUNDANCE

FlexTiles Development Platform

Self-Adaptive Heterogeneous MultiCore Solutions based on DSPs and FPGAs



www.SUNDANCE.com

www.flextiles.eu

Test & Measurement instruments • Communications, MIMO & Satellite receivers • RADAR ECM systems
Space, Land, Sea, Air equipments • Research, Science & Technology • Industrial control & video solutions

Accelerating Distributed Computing with FPGAs

An SoC network that uses Xilinx partial-reconfiguration technology offers cloud computing for algorithms under test with large stimulus data sets.

by Frank Opitz, MSc
Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Computer Science
Opitz.frank@googlemail.com

Edris Sahak, BSc
Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Computer Science
edris.sahak@haw-hamburg.de

Bernd Schwarz, Prof. Dr.-Ing.
Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Computer Science
schwarz@informatik.haw-hamburg.de

Rather than install faster, more power-hungry supercomputers to tackle increasingly complex scientific algorithms, universities and private companies are applying distributed platforms upon which projects like SETI@home compute their data using thousands of personal computers. [1, 2] Current distributed computing networks typically use CPUs or GPUs to compute the project data.

FPGAs, too, are being harnessed in projects like COPACOBANA, which employs 120 Xilinx® FPGAs to crack DES-encrypted files using brute-force processing. [3] But in this case, the FPGAs are all collected in one place—an expensive proposition not appropriate for small university or company budgets. Currently FPGAs are not noted as a distributed computing utility because their use demands the involvement of a PC to continually reconfigure the whole FPGA with a new bitstream. But now, with the application of the Xilinx partial-reconfiguration technology, it's feasible to design FPGA-based clients for a distributed computing network.

Our team at the Hamburg University of Applied Sciences created a prototype for such a client and implemented it in a single FPGA. We structured the design to consist of two sections: a static and a dynamic part. The static part loads at startup of the FPGA, while its implemented processor downloads the dynamic part from a network server. The dynamic part is the partial-reconfiguration region, which offers shared FPGA resources. [4] With this configuration, the FPGAs may be situated anywhere in the world, offering computing projects access to a high amount of computing power with a lower budget.

DISTRIBUTED SOC NETWORK

With their parallel signal-processing resources, FPGAs provide four times the data throughput of a microprocessor by using a clock that is eight times slower and with eight times lower power consumption. [5] To leverage this computational power for high-data-input rates, designers typically implement algorithms as a pipeline, like DES encryption. [3] We developed the distributed SoC network (DSN) prototype to increase the speed of such algorithms and to process large data sets using distributed FPGA resources. Our network design applies a client-broker-server architecture so that we can assign all registered system-on-chip (SoC) clients to every network participant's computational project (Figure 1). This would be impossible in a client-server architecture, which connects every SoC client to only one project.

Furthermore, we chose this broker-server architecture to reduce the number of TCP/IP connections of each FPGA to just one. The DSN FPGAs compute the algorithms with dedicated data sets while the broker-server manages the SoC clients and the project clients. The broker schedules the connected SoC clients so that each project has nearly the same computing power at the same time, or uses time slices if there are fewer SoCs than projects with computational requests available.

The project client delivers the partial-reconfiguration module (PRM) and a set of stimulus input data. After connecting to the broker-server, the project client sends the PRM bit files to the server, which distributes them to SoC clients with a free partially reconfigurable region (PRR). The SoC client's static part, a MicroBlaze™-based microcontroller, reconfigures the PRR dynamically with the received PRM. In the next

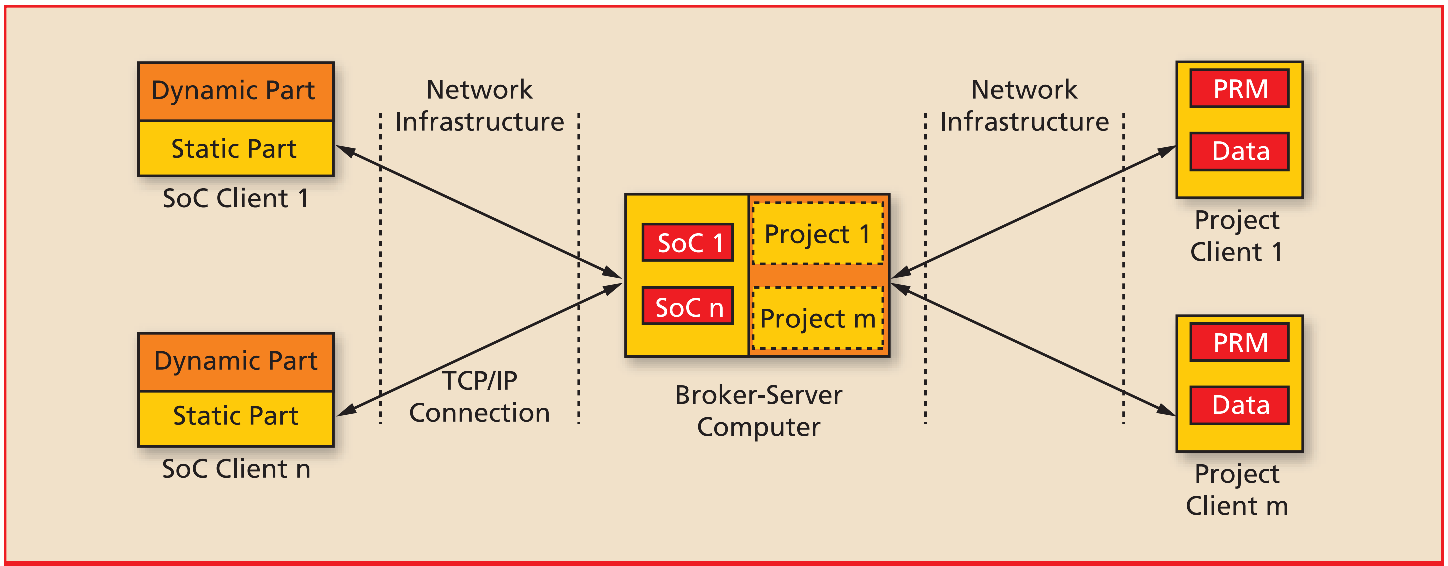


Figure 1 – Distributed SoC network with SoC clients provided by FPGAs and managed by a central broker-server. Project clients distribute the partial-reconfiguration modules and data sets. The dynamic part of an SoC client supplies resources via the PRR, and a microcontroller contained in the static part processes the reconfiguration.

A MicroBlaze processor runs the client's software, which manages partial reconfigurations along with bitstream and data exchanges.

step, the project client starts sending data sets and receives the computed response from the SoC client via the broker-server. Depending on the project client's intentions, it compares different computed sets or evaluates them for its computational aims, for example.

THE SOC CLIENT

We developed the SoC client for a Xilinx Virtex®-6 FPGA, the XC6VLX240T, which comes with the ML605 evaluation board. A MicroBlaze™ processor runs the client's software, which manages partial reconfigurations along with bitstream and data exchanges (Figure 2). A Processor Local Bus (PLB) peripheral that encapsulates the PRR in its user logic is the interface between the static and the dynamic parts. In the dynamic part reside the shared FPGA resources for accelerator IP cores supplied by the received PRM. To store received and computed data, we chose DDR3 memory instead of CompactFlash because of its higher data throughput and the unlimited amount of write accesses. The PRM is stored in a dedicated data section to control its size and to avoid conflicts with other data sets. The section is set to 10 Mbytes, which is big enough to store a complete FPGA configuration. Thus, every PRM should fit in this section.

We also created data sections for the received and the computed data sets. These are 50 Mbytes in size so as to ensure enough address space for images or encrypted text files, for example. Managing these data sections relies on an array of 10 administration structures; the latter contain the start and end addresses of each data set pair and a flag that indicates computed sets.

To connect the static part to the PRR, we evaluated IP connections given by the Xilinx EDK, such as the Fast Simplex Link (FSL), PLB slave and PLB master. We chose a PLB master/slave combination to get an easy-to-configure IP that sends and receives data requests without the MicroBlaze's support, significantly reducing the number of clock cycles per word transfer.

For the client-server communication, the FPGA's internal hard Ethernet IP is an essential peripheral of the processor system's static part. With the soft-direct-memory access (SDMA) of the local-link TEMAC to the memory controller, the data and bit file transfers produce less PLB load. After receiving a frame of 1,518 bytes, the SDMA generates an interrupt request, so that the lwip_read() function unblocks and can handle this piece of data. The lwip_write() function tells the SDMA to perform a DMA transfer over the TX channel to the TEMAC.

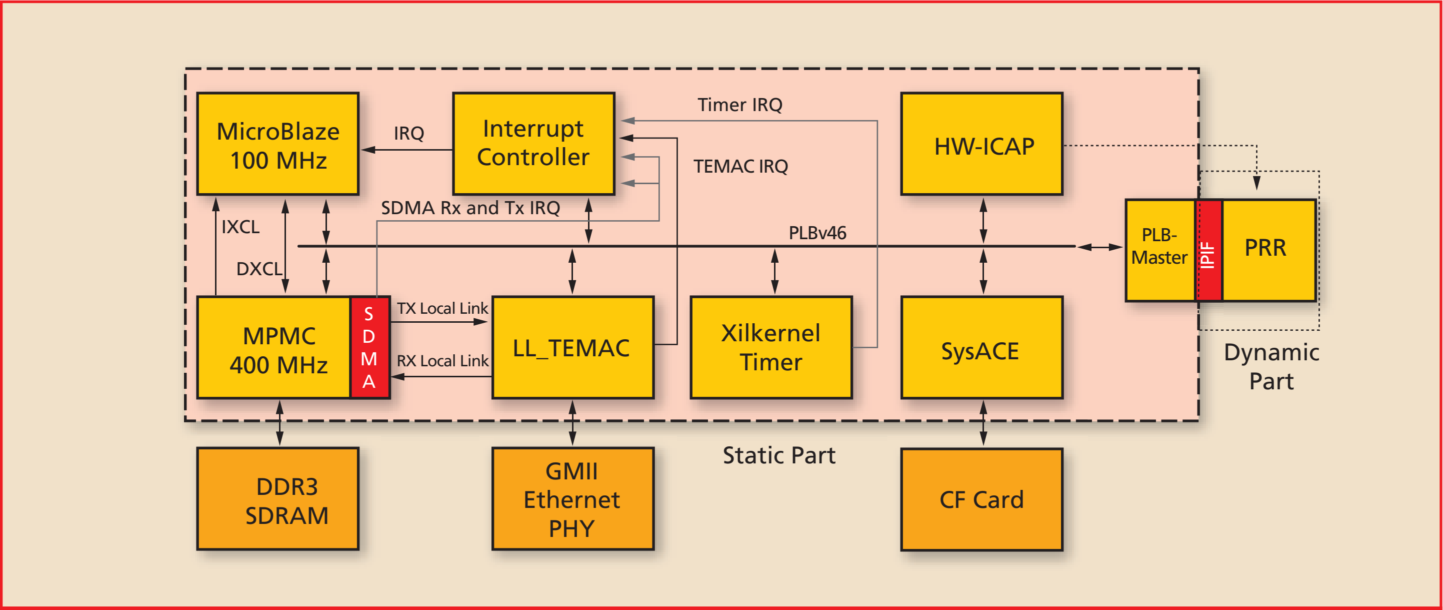


Figure 2 – The SoC client is a processor system with a static part and a bus master peripheral, which contains the partially reconfigurable region (PRR). Implemented with Virtex-6 FPGA XC6VLX240T on an ML605 board.

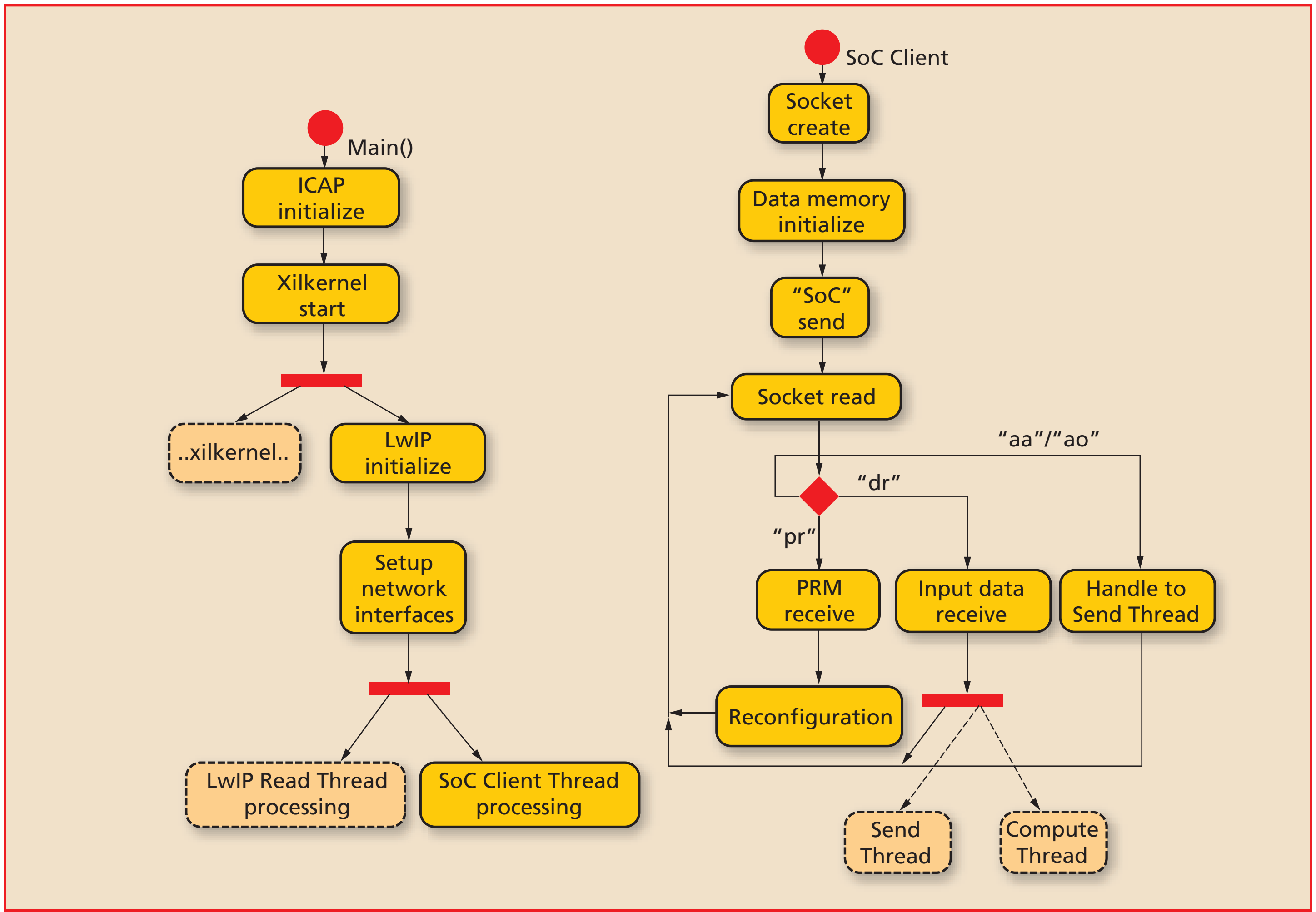


Figure 3 – SoC client's software initialization and processing cycles include reconfiguration of the PRR with a PRM, data set retrieved from the server, start of processing and the data set's return to the server threads. Black bars indicate thread creation by sys_thread_new() calls from the Xilkernel library.

We implemented the Xilkernel, a kernel for Xilinx embedded processors, as an underlying real-time operating system of the SoC client's software in order to utilize the lightweight TCP/IP stack (LwIP) library with the socket mode for the TCP/IP server connection. Figure 3 provides an overview of the client's threads initialization, creation, transmission and processing sequences. The SoC client thread initiates a connection to the server and receives a PRM bitstream ("pr"), which it stores in DDR3 memory, applying the XILMFS file system. Thereafter the Xps_hwicap (hardware internal configuration access point) reconfigures the PRR with the PRM. Finally, the bus master peripheral sets a status bit that instructs the SoC client to send a request to the server. The server responds with a data set ("dr"), which the SoC client stores in the onboard memory as well. These data files contain a content sequence such as output_length+"ol"+data_to_compute. The output_length is the byte length, which reserves the memory range for the result data followed by the character pair "ol." With the first received "dr" message, a compute and a send thread get created.

The compute thread transfers the addresses of the input-and-result data sets to the slave interface of the PRR peripheral and starts the PRM's autonomous data set processing. An administration structure provides these addresses for each data set and contains a "done" flag, which is set after the result data is completely available. In the current version of the client's software concept, the compute and send threads communicate via this structure, with the send thread checking the done bit repeatedly and applying the lwip_write() calls on results stored in memory.

When testing the SoC client, we determined that with all interrupts enabled while the reconfiguration of the PRR is in progress, this process gets stuck randomly after the Xilkernel's timer generates a scheduling call to the MicroBlaze. This didn't happen with all interrupts disabled or while using a standalone software module for the SoC client's MicroBlaze processor without the Xilkernel's support.

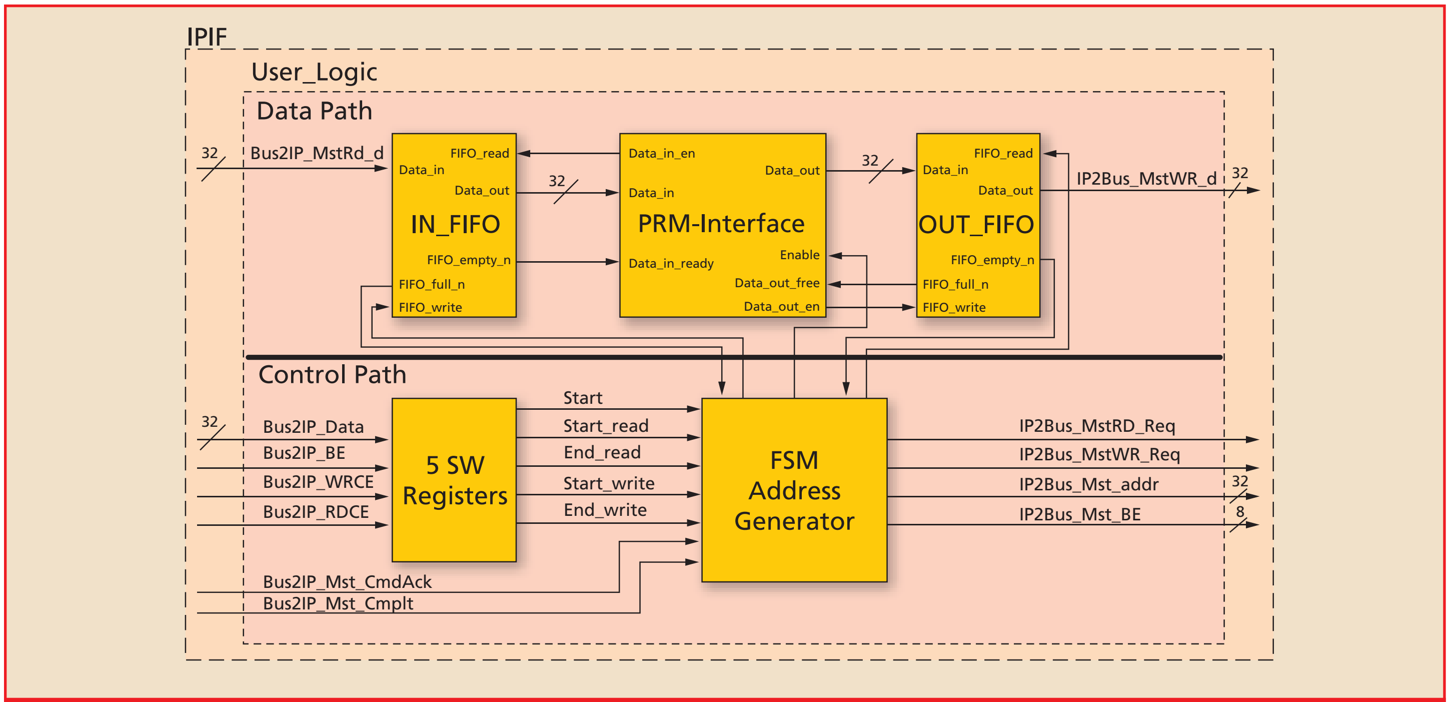


Figure 4 – Bus master peripheral operates as a processor element.
The PRM interface includes the dynamic part with a component instantiation of the PRM.

BUS MASTER PERIPHERAL WITH PRM INSTANTIATION

To achieve a self-controlled stimulus data and result exchange between the PRM and the external memory, we structured the bus master peripheral as a processor element with a data and a control path (Figure 4). Within the data path, we embedded the PRM interface between two FIFO blocks with a depth of 16 words each in order to compensate for communication and data transfer delays. Both FIFOs of the data path are connected directly to the PLB's bus master interface. In this way, we obtain a significant timing advantage from a straightforward data transfer operated by a finite state machine (FSM). No software is involved, so no intermediate data storage takes place in the MicroBlaze's register file. This RISC processor's load-store architecture always requires two bus transfer cycles for loading a CPU register from an address location and storing the register's content to another PLB participant. With the DXCL data cache link of the MicroBlaze to the memory controller as a bypass to the PLB, the timing of these load-store cycles would not improve. That's because the received data and the transmitted computing results are all handled once, word by word, without utilizing caching benefits. As a consequence, the PRR peripheral's activities are decoupled from the MicroBlaze's master software processing. Thus, the PRR data transfer causes no additional Xilkernel context switches. But there is still the competition of two masters for a bus access, which can't be avoided.

IN_FIFO	OUT_FIFO	Memory access	Next state
don't care	not empty	writing	WRITE_REQ
not full	empty	reading	READ_REQ
full	empty	—	STARTED

Table 1 – FSM control decisions in state STARTED with write priority

The peripheral's slave interface contains four software-driven registers that provide the control path with start and end addresses of the input and output data sets. Another software register introduces a "start" bit to the FSM, which initiates the master data transfer cycles. The status of a completed cycle of data processing is available with the address of the fifth software register to the client's software.

With the state diagram of the control path's FSM, the strategy to prioritize the write cycles to the PLB becomes clear (Figure 5). Pulling out the data from the OUT_FIFO dominates over filling the IN_FIFO, to prevent a full OUT_FIFO from stopping the PRM from processing the algorithm. Reading from or writing to the external memory occurs in alternate sequences, because only one kind of bus access at a time is available. When a software reset from the client's com-

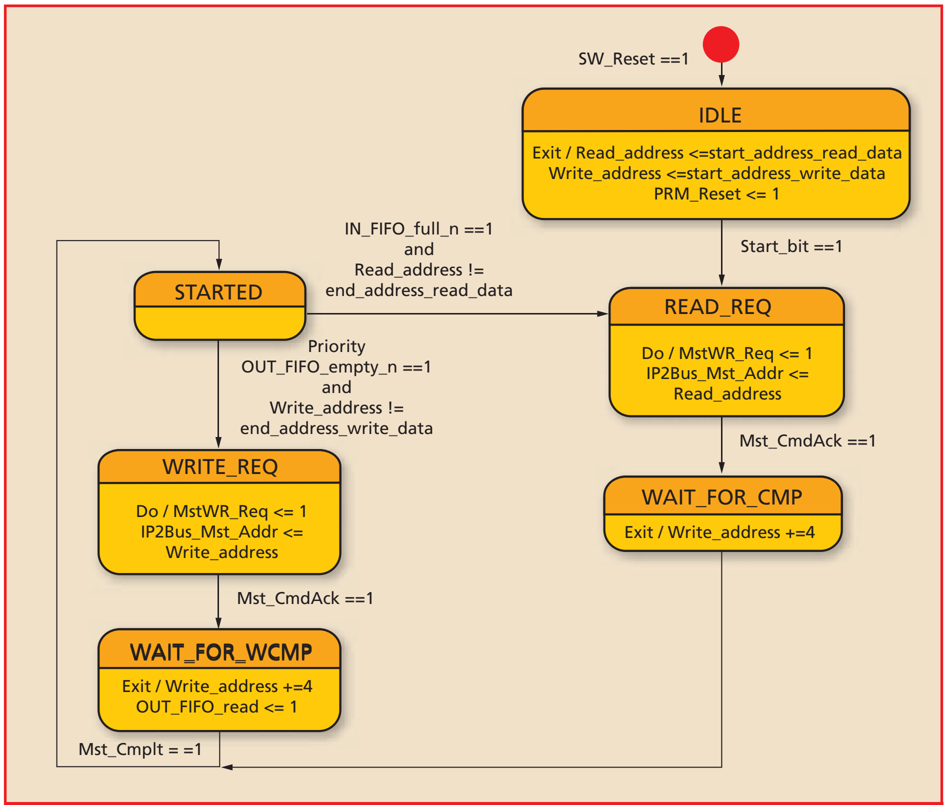


Figure 5 – State diagram of the bus master peripheral's control path. Data write requests to the bus are prioritized to keep a full OUT_FIFO from stopping PRM algorithms. UML modeling style: Moore outputs with label Do/ and Mealy outputs with label Exit/.

pute thread starts the FSM (Figure 3), the first thing that happens is a read from the external memory (state READ_REQ). From then on, the bus master follows the decision logic given by the transition conditions from state STARTED (Table 1).

The FSM Mealy outputs (label Exit/) prepare the address counters to increment when a bus transfer is completed. Here, the two counters are introduced directly into the FSM code. Usually we prefer timers and address counters as separate clocked processes enabled simply by FSM outputs, in order to keep the counter's transition logic small and free from unnecessary multiplexer inputs for counter state feedback. At this point, the XST synthesis compiler results present RTL schematics with a clear FSM extraction parallel to loadable counters, with clock-enable inputs driven by an expected state decoding logic. Despite a more readable behavioral VHDL-coding style, the FPGA resources and simple primitives get utilized without a loss of features.

DEFINING THE DYNAMIC PART WITH PLANAHEAD

The design flow for configuration of a static and a dynamic part within the FPGA is a complex development process that involves several steps with the physical-design constraints tool PlanAhead™. Our first effort was a script-based design flow for a PetaLinux-driven dynamic-reconfiguration platform implemented on an ML505 board. [6] With the current iteration, the design steps for integrating a PRR directly into a peripheral's user logic are much more practical than the former method of adding bus macros and a device control register (DCR) as a PLB interface for the PRM and an extra PLB-DCR bridge for enabling the bus macros.

Here is how we fixed the dynamic part's size and position with the AREA_GROUP constraints, which are included in the UCF file of the PlanAhead project, as shown in the code below.

```
INST "
dyn_interface_0/dyn_inter-
face_0/USER_LOGIC_I/PRR"
AREA_GROUP =
"pblock_dyn_interface_0_USE
R_LOGIC_I_PRR ";
AREA_GROUP "
pblock_dyn_interface_0_USER
_LOGIC_I_PRR "
RANGE=SLICE_X0Y0
:SLICE_X57Y239 ;
AREA_GROUP "
pblock_dyn_interface_0_USER
_LOGIC_I_PRR "
RANGE=RAMB18_X0Y0:RAMB18_X3
Y95 ;
AREA_GROUP "
pblock_dyn_interface_0_USER
_LOGIC_I_PRR "
RANGE=RAMB36_X0Y0:RAMB36_X3
Y47 ;
```

An instance name concatenation specifies the inner partial-reconfiguration region (prm_interface.vhd) with instance name PRR. For all FPGA resources we want to include in the desired PRR, we specify a rectangular region with its lower-left and upper-right coordinates.

This special choice covers slices and BRAM only, because the available DSP elements belong to dedicated

Resource	Amount
LUT	55,680
FD_LD	111,360
SLICEL	7,440
SLICEEM	6,480
RAMBFIFO36E1	192

Table 2 – Allocated resources for the dynamic part of the SoC client

clock regions and are utilized for the Multiport Memory Controller (MPMC) implementation (Table 2).

To prevent the PRM netlists that ISE® generates from using excluded resources, we set the synthesis options to dsp_utilization_ratio = 0; use_dsp48 = false; iobuf = false. Finally, the FPGA Editor offers an insight: that the static part's placement is located in an area separated completely from the PRR, which in this special case uses very few resources (Figure 6).

AN SOC CLIENT WITH IMAGE-PROCESSING PRM

We proved the SoC client's operation and its TCP/IP server communication with a Sobel/median filter combination implemented in a PRM (Figure 7). We developed the image-processing neighborhood operations with the Xilinx System Generator, which gave us the advantage of Simulink® simulation and automatic RTL code generation. A deserializer converted the input pixel stream to a 3 x 3-pixel array, which sequences like a mask over the whole image and provides the input to the filter's parallel sum of products or to the successive comparisons of the median filter. [7] Input and output pixel vectors of the filters have a width of 4 bits, so we inserted a PRM wrapper that multiplexes the eight nibbles of the 32-bit input vector from the synchronization FIFO. With a MATLAB® script, we convert an 800 x 600 PNG image to 4-bit gray-scale pixels for the PRM input stimulus. At the filter's output, eight 4-bit registers are successively filled and concatenated for the word transfer to the OUT-FIFO (Figure 4).

Table 3 summarizes the results of timing measurements performed with three operational steps of the SoC client: receiving a PRM bit file, reconfiguration of the PRR and image-processing sequences. We captured the receiving and image-processing cycles, from the first to the last data

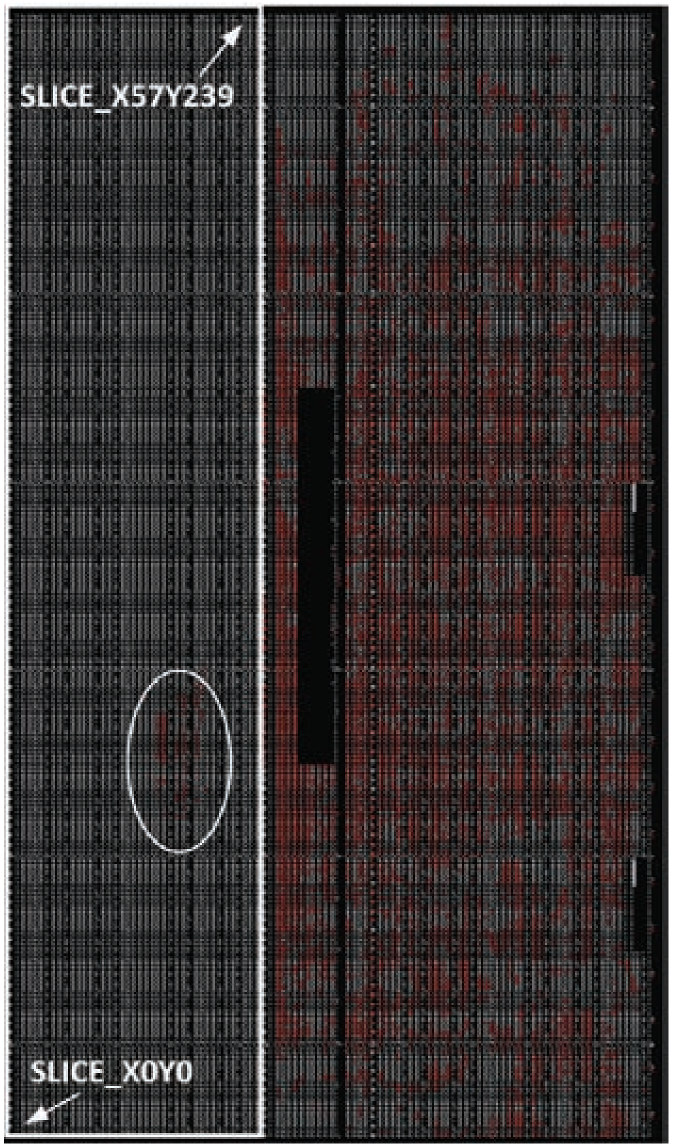


Figure 6 – Resource placement of the static part (right side) and dynamic part (left side, with white oval) according to the area specification for the PRR

		Interval duration		
Filter module	Slices	PRM receive (seconds)	Reconfiguration 3.5-Mbyte bit file (sec)	Image processing (ms)
Binarize	3	77	31.25	25.25
Erosion 3x3	237	73	31.25	85.93
Median 3x3	531	73	31.25	77.09
Sobel 3x3	479	73	31.25	86.45

Table 3 – Timing measurement results; reconfiguration with disabled interrupts. Processor and peripheral clock rate fclk = 100 MHz.

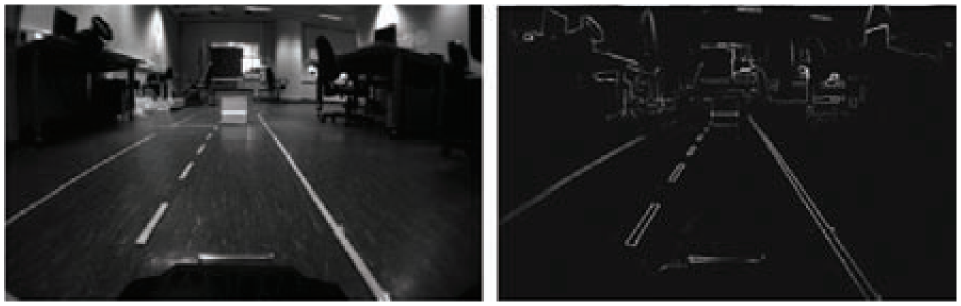


Figure 7 – PRM processing results for edge detection. Gray-scale input stimulus image to the PRM is shown at left, while the response from a PRM with a Sobel/median filter combination is seen at right.

transfer, with a digital oscilloscope measurement at a GPIO output toggled by XGpio_WriteReg() calls.

The reconfiguration intervals all have the same duration, because no Xilkernel scheduling events disrupted the software-driven HWICAP operation. An FSM-controlled HWICAP operation without MicroBlaze interaction will yield a shorter duration with a reconfiguration speed of more than 112 kbytes/second, even with enabled interrupts.

During PRM transmission from the broker to the SoC client, the connection soon aborted. With a 1-millisecond delay between each transmitted 100 bytes, the SoC client performed a nondisturbed communication. Parallel to the image-processing cycles, normal Xilkernel threading caused PLB access competition and therefore, the SoC client operated under typical conditions. The binarize sequence has a duration value of 600 x 800/100 MHz = 4.8 ms, because only a single comparison is active. This sequence is nested in two image transfers via the PLB, which take a minimum of five clocks per word, as extracted from a functional bus simulation: 2 x 5 x 600 x 800/(8 x 100 MHz) = 6 ms. Because all measurement numbers for the data transfers are larger than rough estimates led us to expect, we are in the midst of a detailed analysis of the full timing chain buildup by bus reading, FIFO filling and emptying, image-processing pipeline and bus writing.

POWER OF PARTIAL RECONFIGURATION

To compute complex algorithms, it is profitable to employ the power of distributed computing networks. State-of-the-art implementations of these networks operate with CPUs and GPUs only. Our prototype of an FPGA-based distributed SoC network architecture utilizes the parallel signal-processing features of FPGAs to compute complex algorithms.

The Xilinx partial-reconfiguration technology holds the key to utilizing shared FPGA resources all over the world. In our architecture, the static part of the SoC client reconfigures the dynamic part of the FPGA with updated accelerators in a self-controlled way. We have to improve the SoC client to run the HWICAP with enabled interrupts, so that it keeps fully reactive. A step in that direction is an FSM-controlled reconfiguration, which puts no load on the processor. But we need to analyze the influence of PLB transfers and the MPMC bottleneck as well.

To manage the SoC client, a Xilkernel linked with the LwIP supplies concurrency with threads for the reconfiguration drivers, the dynamic part's bus interface and other applications. We further concentrate on timing analysis of the client-server system and the dynamic part's processing cycles in order to identify the software/RTL-model configuration with an improved data throughput and a reliable communication.

For the next stage of our SoC client design, we have to take the AXI4 bus features into account. In general, PRM exchanges can be treated as additional hardware tasks operating in conjunction with a set of software tasks. Last but not least, we are still refining the server's software design to achieve improved user satisfaction.

References

1. *Unofficial BOINC Wiki, "Boinc FAQ: Introduction to boinc,"* http://www.boinc-wiki.info/

2. *Markus Tervooren, all project stats.com,* http://www.allprojectstats.com/

3. *S. Kumar, J. Pelzl, G. Pfeiffer, M. Schimmler and C. Paar, "Breaking ciphers with COPACOBANA, a cost-optimized parallel code breaker, or how to break DES for 8,980 eur,"* http://www.copacobana.org/paper/CHES2006_copacobana_slides.pdf

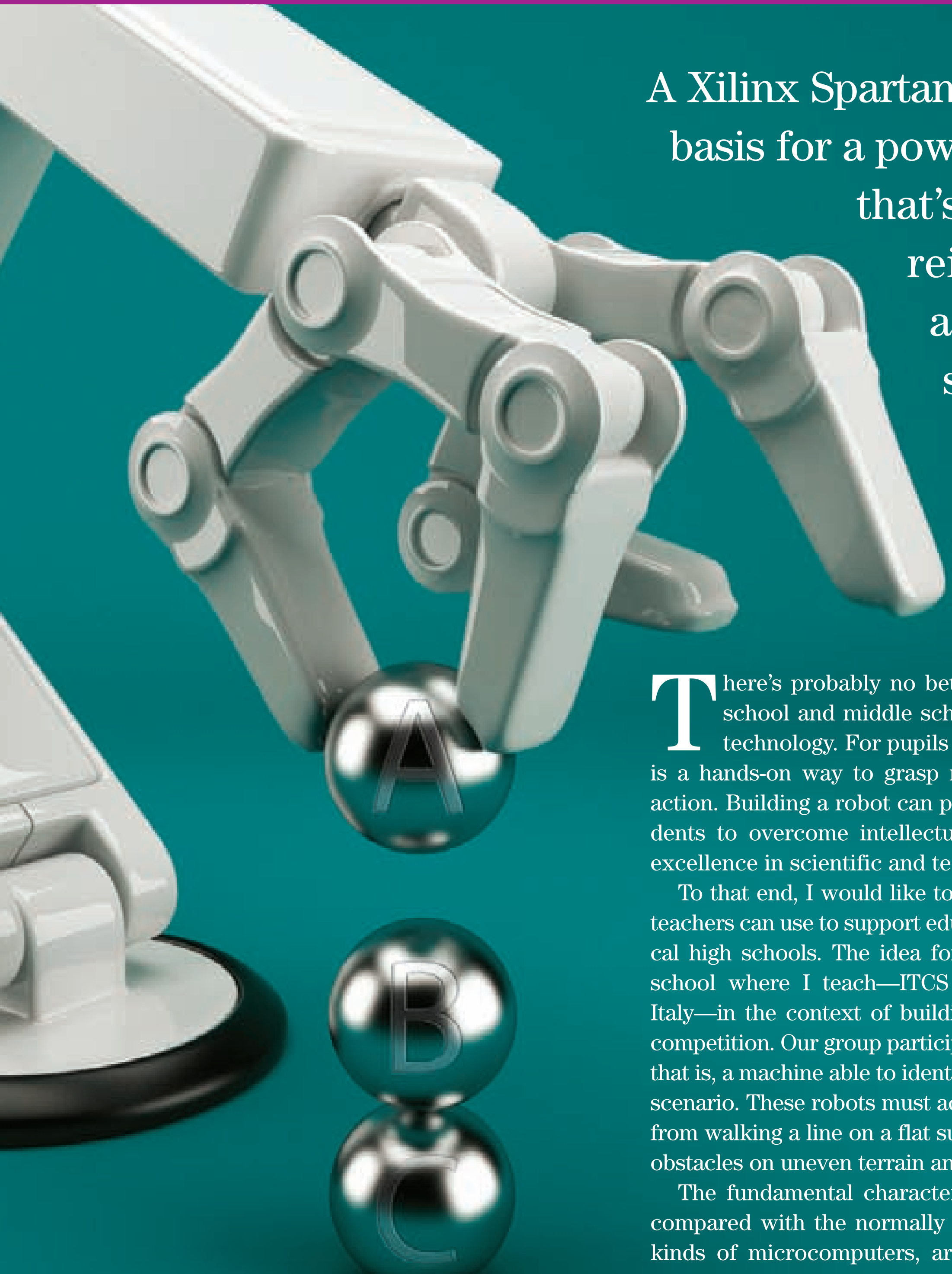
4. *Frank Opitz, "Development of an FPGA-based distributed computing platform." Master's thesis, HAW Hamburg, 2011,* http://opus.haw-hamburg.de/volltexte/2012/1450/pdf/Masterarbeit_Frank_Opitz.pdf

5. *Ivo Bolsens, "Programming Modern FPGAs,"* http://www.xilinx.com/univ/mpsoc2006keynote.pdf

6. *Armin Jeyrani Mamegani, "Implementation and evaluation of methods for partial and dynamic reconfiguration of SoC- FPGAs." Master's thesis, HAW Hamburg, 2010,* http://opus.haw-hamburg.de/volltexte/2010/1083/pdf/MA_A_Jeyrani.pdf

7. *Edris Sahak, "Partial reconfiguration of an SoC-based image-processing pipeline." Bachelor's thesis, HAW Hamburg, 2011,* http://opus.haw-hamburg.de/volltexte/2011/1420/pdf/BA_E_Sahak.pdf

FPGAs Enable Flexible Platform for High School Robotics



A Xilinx Spartan FPGA forms the basis for a powerful teaching tool that's able to evolve, reinventing itself according to student needs.

by Giulio Vitale
Professor and FPGA Design Consultant
ITCS Erasmo da Rotterdam, Bollate, Italy
gvitale@tiscali.it

There's probably no better vehicle than a robot to get high school and middle school students hooked on science and technology. For pupils in grades 8-12, tinkering with a robot is a hands-on way to grasp new ideas and to see technology in action. Building a robot can provide a powerful motivation for students to overcome intellectual challenges and achieve levels of excellence in scientific and technical subjects.

To that end, I would like to present an educational platform that teachers can use to support educational robotics activities for technical high schools. The idea for this platform was born at the high school where I teach—ITCS Erasmo da Rotterdam, near Milan, Italy—in the context of building an entry for the RoboCup Junior competition. Our group participated in the category "rescue robot"—that is, a machine able to identify victims within a re-created disaster scenario. These robots must accomplish tasks varying in complexity from walking a line on a flat surface up to negotiating paths through obstacles on uneven terrain and saving some well-defined victims.

The fundamental characteristics of our FPGA-based design, if compared with the normally diffuse platforms based on different kinds of microcomputers, are openness, flexibility, capability to evolve and reusability.

We built the 2011 version of the robot on a Xilinx® Spartan®-3E device, using the Digilent Nexys2 educational board. As of this writing, we are porting this version—which will compete in the next RoboCup Junior Italy in April 2012—to a Spartan-6 FPGA. The next, 2013 version is scheduled to run on a Zynq™-7000 Extensible Processing Platform device.

Working within this paradigm, we were able to design a rescue robot that evolved, year after year, from the first prototype to the current version, named Nessie 2011 (a pun on both Nexys and the Loch Ness monster, whose long neck is reminiscent of our robot's). The flexibility of the FPGA allowed a complete remodeling of the robot's architecture, following the progression of the students' knowledge, while leaving its basic physical structure substantially unaltered and maintaining the same design infrastructure.

THE CHALLENGE

The ITCS Erasmo da Rotterdam is a technical high school located in a suburb of Milan in northern Italy, with a student population that is strongly heterogeneous and, often, not so easy to coax into deepening their scientific and technological knowledge.

Starting four years ago, I decided to activate an open space, which I named the Permanent Laboratory for Didactic Robotics, where students can experiment in a different way from the standard classroom. Here, they approach the technical disciplines in an interactive environment in which pupils can negotiate some unexpected aspects of the subject matter, make choices regarding the subjects they will pursue, organize their own jobs and receive direct feedback from the results of their actions. In other words, they get to experiment in an "active learning space" based on the old and well-known model of "situated cognition" [1], where students collaborate with one another and with their instructor while pursuing a common goal with a shared understanding.

In this learning space, pupils can practice a "cognitive apprenticeship" that uses problem-solving methodologies. The teacher assumes the role of a "professional expert" who proposes specific processes involving authentic tasks and strategies, and allows students to try them independently, coaching only as needed.

Robotics was the natural choice to provide a fertile breeding ground for the convergence of different disciplines and the exchange of knowledge. We decided that the fun of taking part in the RoboCup Junior competition would provide a strong stimulus to incentivize student participation.

THE SOLUTION

I understood that to be effective, I would have to propose topics normally covered in regular classes on digital electronics and informatics, but aimed at more complex applications than those the students would be able to solve alone. Instead, they would need to work in groups or with the support of an experienced teacher who could propose appropriate models.

I knew what to build but I did not know how to build it. Everything had to be born and developed in the laboratory, with students discussing the design and seeking solutions together.

After some deliberation, I came to the conclusion that the most likely solution was one based on a flexible platform, such as an FPGA, rather than on standard microcomputers. That's because an FPGA was the only device able to provide the required characteristics, and to keep pace with the dynamic and evolutionary scope of the laboratory activities.

I chose, initially, to use an educational card based on the Spartan-3E because it could provide the necessary characteristics we were seeking—namely, openness, flexibility, ability to evolve, reusability of the hardware and richness of performance.

- **Openness,** because students must actively participate in the entire design flow, from the sensor interface to the CPU and from this to the actuators.
- **Flexibility,** because the complete architecture of the system and the nature and type of devices should not be fixed in advance, but must emerge from the research process activated within a creative context for learning.
- **Ability to evolve**, because after each RoboCup competition, students must learn the shortcomings of their work and know how to make the appropriate modifications to try to reach more advanced solutions. The system must grow in parallel with the students' expertise.

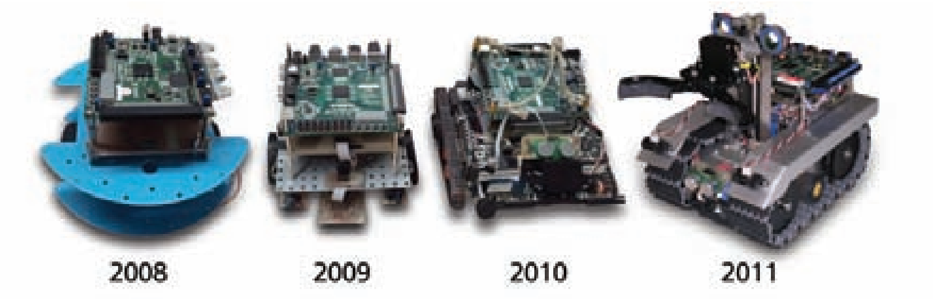


Figure 1 – Thanks to FPGA flexibility, the same platform can evolve in parallel with student understanding, reusing the same hardware and redesigning as needed. Photos show Nessie's evolution from 2008 through 2011.

- **Reusability**, in order to avoid unnecessary waste of the hardware and the school budget.
- **High performance at an affordable cost**. We had to control a large number of devices and peripherals that were not fully defined, but needed to operate with a high degree of parallelism. The CPU should be very powerful but relatively simple in its architecture and easy to interface.

NESSIE 2012: BLOCK DIAGRAM AND DESCRIPTION

We achieved our goal using a Digilent card carrying a Spartan 3E-1200 onboard, which was the common thread of the four-year project development. As you can see in Figure 1, the rescue robot the students designed showed a clear evolution from a machine that barely moved in 2008 to one that, in 2011, made us one of just 15 teams, out of 65 participants, to reach the finals.

The level of student expertise has grown from year to year, laying a foundation for further improvements that we are planning for this year's RoboCup Junior competition in April.

First, we have moved from the Spartan-3E to the Spartan-6 family, converting the bus infrastructure of the standard Processor Local Bus to the AXI4 interface. Second, we have modified some critical sensors for tracking of the reference line and redesigned the motor interface, porting a PID algorithm for automatic speed control directly into the FPGA fabric.

Figure 2 illustrates the complete block diagram of the system, as it appears in the current design. Looking at it, you can clearly see the richness of didactic topics that it enables a teacher to cover in a course on digital control systems. Equally clear is the high level of parallelism that the system can obtain in terms of performance, compared with a robotic platform built on a standard microcomputer.

Moreover, the activity in the robotics laboratory has had an interesting impact on the ordinary educational courses at our school. The FPGA has become a tool for rapid and effective implementation of the theoretical aspects of technology, sparking students' keen interest in the topics covered.

MANAGING NESSIE'S WALKING PROBLEM

Armed with the rich supply of resources available in the Spartan-6

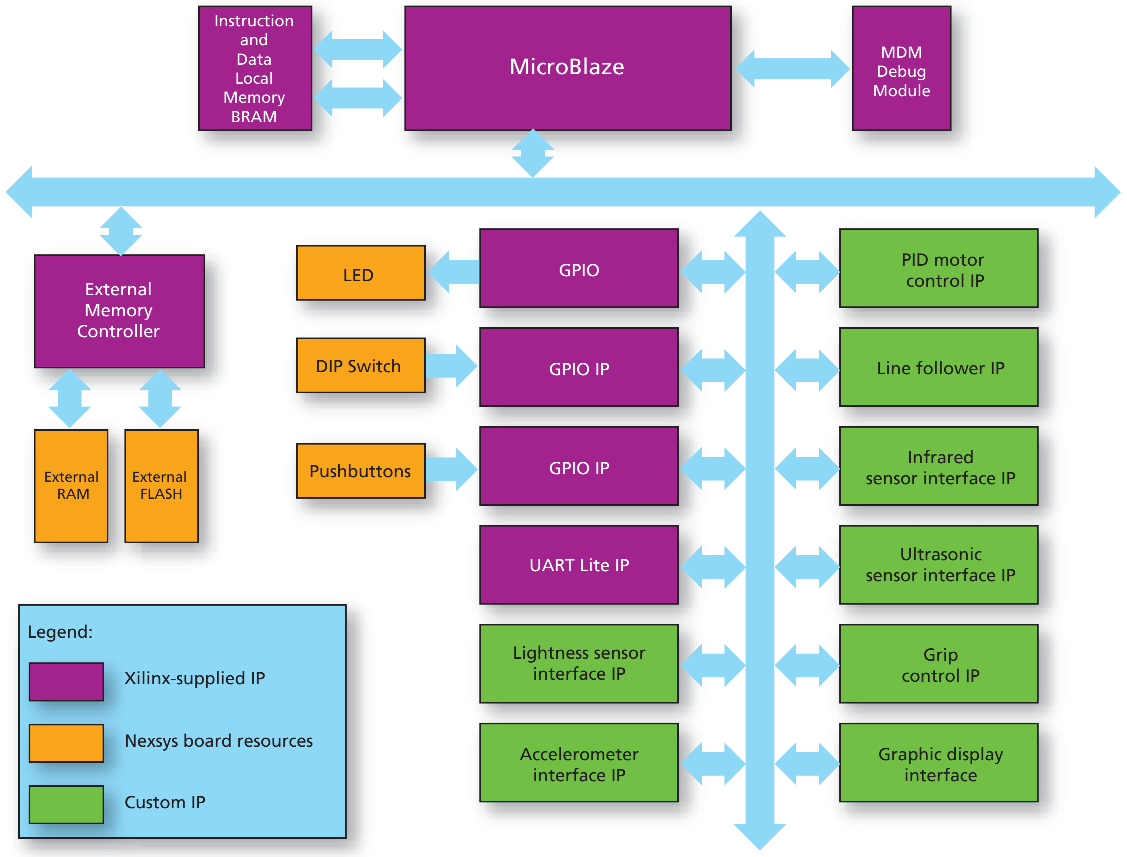


Figure 2 – This block diagram shows the current Spartan-6 reconfiguration, Nessie 2012, which will compete in the RoboCup Junior spring event.

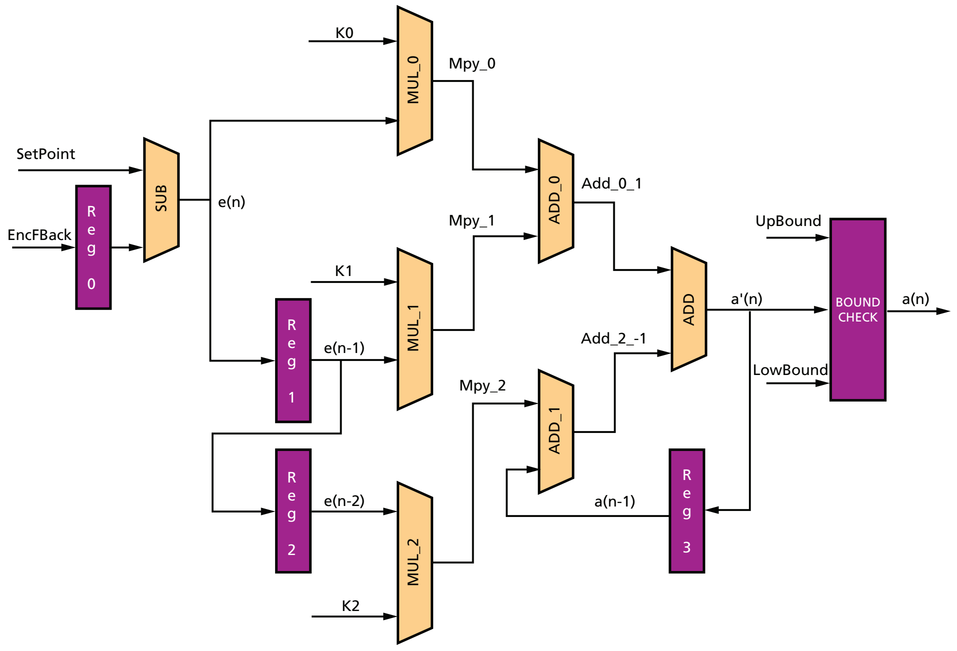


Figure 3 – RTL sketch of the PID algorithm that controls the speed of the two motors of the robot

family, I faced the problem of how to bring PID control from the analysis conducted on continuous systems to an actual realization with digital systems, giving the students the hands-on opportunity to solve the problem of digitizing a system of equations and immediately translate them in terms of the elements of digital electronic circuits.

I started from the classical PID equation:

$$a(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{de(t)}{dt}$$

And I converted it into another classic finite-difference algorithm,

$$a(n) = K_P \left[e(n) + \frac{T_C}{T_I} \sum_{j=0}^{n} e(j) + \frac{T_D}{T_C}(e(n) - e(n-1)) \right]$$

where K_P is the proportional gain, T_I and T_D the time constant of integrative and derivative actions, and T_C the sampling period. Using suggestions found in the paper "FPGA implementation of a closed-loop control system for a small-scale robot," written by W. Zhao and others, [2] I converted the model into the following very simple iterative algorithm:

$$\Delta a(n)=a(n)-a(n-1)$$

with $\Delta a(n)$ computable as

$$\Delta a(n)=K_0 \cdot e(n)+K_1 \cdot e(n-1)+K_2 \cdot e(n-2)$$

and the iteration step:

$$a(n)=a(n-1)+\Delta a(n)$$

The Ki coefficients are calculable as

$$K_0 = K_P \cdot \left(1 + \frac{T_C}{T_I} + \frac{T_D}{T_C}\right)$$
$$K_1 = -K_P \cdot \left(1 + 2\frac{T_D}{T_C}\right)$$
$$K_2 = K_P \cdot \frac{T_D}{T_C}$$

with the PID parameters empirically tunable using the Ziegler Nichols procedure.

The PID RTL model for the FPGA implementation is outlined in Figure 3. The way we implemented the complete IP for controlling how Nessie will walk into the RoboCup 2012 arena is shown in Figure 4.

What is extremely productive in this approach, from a teacher's point of view, is the linearity of the transformations and the relative transition from concept to its immediate implementation in a physical system that can be easily realized. This process encourages the students to try more experiments and further work with, and learn from, the system.

HOW TO GIVE LIGHT TO NESSIE'S EYES

As a rescue robot, Nessie's mission is to identify victims within an artificially created disaster scenario. The way the robot moves on uneven terrain, overcoming obstacles and debris and looking for the victims to be saved, is to visually follow a black line on a white background.

An efficient tracking system is crucial in determining the level of performance obtained during the RoboCup Junior competitions. A flawless execution of the guided tour is the necessary starting point if the robot is to have enough time to win the rest of the challenges, including avoiding obstacles and rubble, and rescuing victims.

The idea we came up with to solve the problem of Nessie's vision capabilities was to use a 128 x 1 linear sensor array with an internal light integration and holding circuit, namely, the Taos 1401R-LH.

The linear array is composed of a block of 128 photodiodes and an analog circuit that uses two capacitors for integration and maintenance of the charge generated by the photodiodes. The first capacitor gathers the current and the second copies it back and keeps it during the scanning and capture of the next load. The sensor, in effect, performs two operations simultaneously, accomplishing the integration and acquisition of a new measure while also reading, by scanning, the charge accumulated in the previous cycle. The charge integrated in one scan cycle is transferred at the end into the holding capacitors.

Each cycle begins on the rising edge of a signal of Start Integration. At the first clock cycle, this signal throws an internal switch in order to isolate the holding capacitor and, in parallel, to delete the stored charge contained in the integration capacitor. This

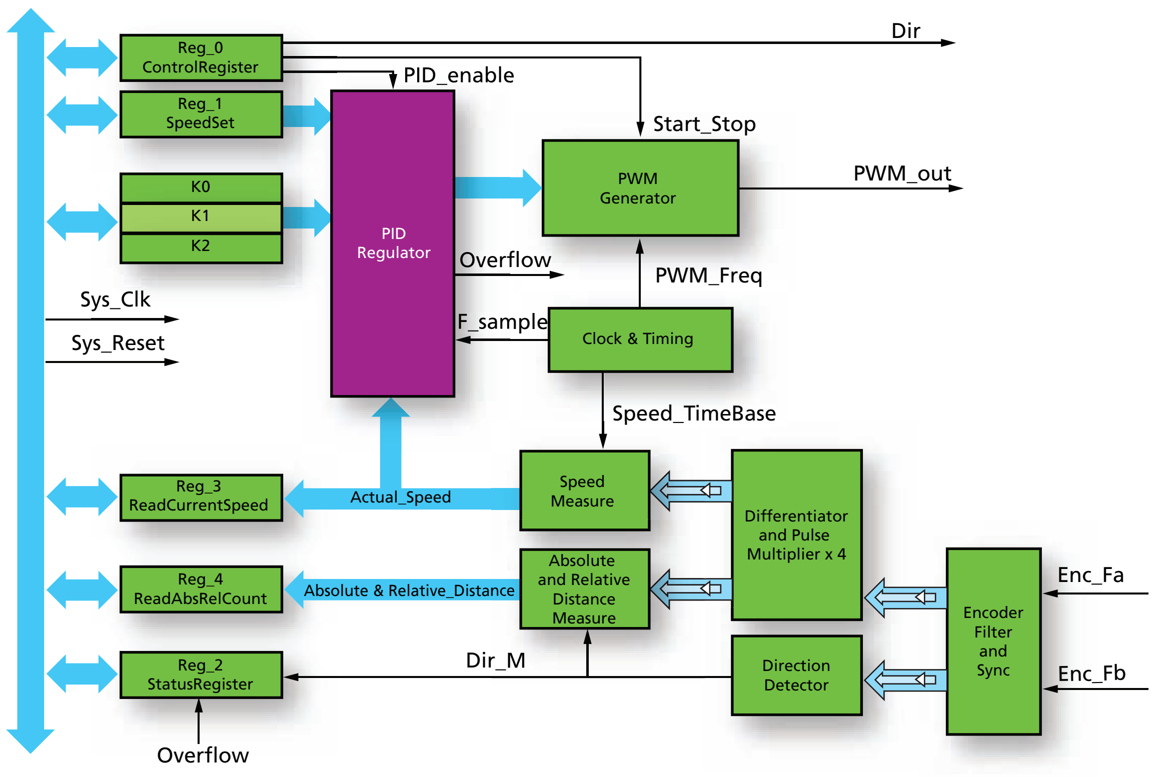


Figure 4 – The complete block diagram of the automatic motor control IP we designed for Nessie 2012

process occurs in parallel on all 128 photodiodes. The cycle continues with a new integration phase, during which the sensor will read all the 128 values of the new brightness data. Meanwhile, in parallel, on each rising edge, the charge contained in the holding capacitors is exposed to an output amplifier that allows the voltage across it to be read on an analog output pin. The voltage is readable as a shift register. In this way, it is possible to simultaneously read the voltage due to charges accumulated during previous exposure to light radiation, while the current acquisition value is accumulated in the capacitor integration.

Here again the FPGA proves its worth, freeing the CPU of the load of refreshing the accumulated current and saving the converted value to make it available for further handling. Figure 5 depicts the IP we designed for this purpose. It also illustrates the "double buffer" mechanism that internal BRAM, available on the Spartan devices, allows in order to overlap the acquisition of a new frame with the manipulation of the last one acquired. In this way we optimize the time needed for processing the images and to extract all the information for guiding the robot.

DIGITAL PROCESSING AT THE HIGH SCHOOL LEVEL

True to the didactic spirit of this robotic activity, I also used this design to introduce the students to some simple elements of digital image processing.

As can be seen in Figure 2, we added a small graphics display to the design in order to show what the Taos sensor was viewing and to understand how to process it for extracting the information necessary to guide Nessie to achieve its purpose. Figure 6 shows the raw image captured in one frame.

In working with Nessie, students must discover how to manipulate the scanned line so as to discern, between possible debris and garbage, the location of the black line and to define an algorithm for controlling the speed of the two motors to accordingly follow it. In this way the students grasp simple issues of how pixels are processed and learn the results of different mathematical operations in pursuit of that task.

In particular, to locate the black line, they will discover some fundamental operations: thresholding, salt-and-pepper filtering, edge detection and line segmentation.

The scanned line needs to be organized as a collection of segments of different levels of light and different lengths, and all these objects must be correctly ordered to identify the direction and guide the robot. This represents a second control loop, managed via software, outside the hardware PID loop, that allows an optimal movement during the task of line following, adjusting the relative speed of the two motors according to the angle between the sensor axis and the black line, and the relative dimension of the lateral white spaces.

MOTIVATIONAL TOOL

This level of system complexity has turned out to be age-appropriate for my students and has proven to be a

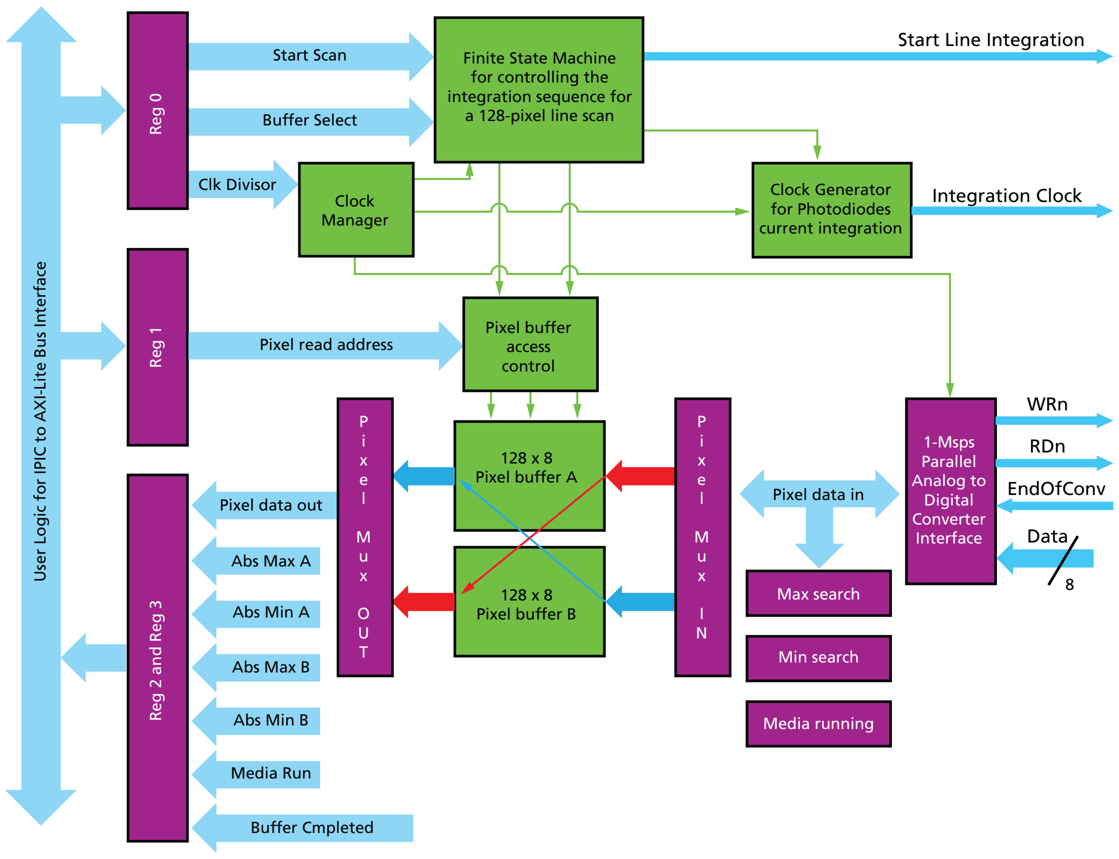


Figure 5 – Block diagram of the interface between the MicroBlaze® and the 128 x 1 Taos 1401 linear-array light sensor

good way to introduce them to higher levels of difficulty, such as two-dimensional image processing. These kinds of starting problems are more accessible and can serve to motivate the students to continue improving their technical and scientific knowledge and gaining new skills as they find solutions to more complex issues. With my long experience in teaching in high school, I can safely say that after this kind of experience, students acquire greater interest and motivation to continue their technical studies and can address both university study and the search for a job in their specific sector with greater security.

In my three years of teaching robotics using this flexible FPGA-based platform, I have seen my students attain a greater perception of the internal organization of "real-time" computer systems. They also acquire a knowledge of how to design the inner peripherals along with a greater sense of authorship over the finished product, an enhanced ability to work in a team and a better capacity to address problems and find appropriate solutions on their own.

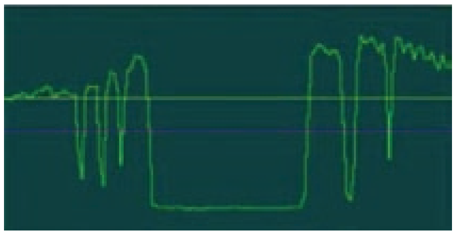


Figure 6 – Raw pixel line captured with a black line over a white background and some debris near it, before the thresholding and the salt-and-pepper filtering

Given the educational purpose of this activity, our group at the ITCS Erasmo da Rotterdam is encouraging students to use their work on Nessie as the basis for passing their final State Exam, a necessity for graduation in Italy. The exam has an individualized component in that students must discuss a personal research path. Nessie was, in fact, the inspiration for some students to achieve an electronics technician diploma, with honors.

References

1. J. S. Brown, A. Collins and P. Duguid, "Situated cognition and the culture of learning," Educational Researcher, Vol. 18, No. 1. (Jan. - Feb. 1989), http://people.ucsc.edu/~gwells/Files/Courses_Folder/ED%20261%20Papers/Situated%20Cognition.pdf

2. W. Zhao, B. H. Kim, A. C. Larson and R. M. Voyles, "FPGA implementation of a closed-loop control system for small-scale robot," Proceedings from 12th International Conference on Advanced Robotics (ICAR 05), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.8719

XILINX.
KINTEX 7™

inrevium
A Revolutionary Innovation Platform

TB-7K-325T-IMG

USB3.0 Device Interface Card

Kintex™-7 FPGA ACDC
(Acquisition, Contribution, Distribution and Consumption)
1.0 Baseboard

The Kintex-7 FPGA ACDC (Acquisition, Contribution, Distribution and Consumption) 1.0 Baseboard. The processing performance and flexibility provided by 28nm Kintex-7 FPGAs allows manufacturers to build stunningly immersive technology into new consumer TV screens that come with features such as multiple windows/picture-in-picture, 3D graphics for games and ultra realistic viewing that goes far beyond HDTV.

- **FPGA : Kintex-7 FPGA XC7K325T-FFG900**
- **Memory : DDR3 2Gbit x 4** (address shared)
- **FMC connector**
  - HPC (High Pin Count) x 2
  - LPC (Low Pin Count) x 2
- **SERDES : 16ch x 12.5Gbps** (GTX)

™ **TOKYO ELECTRON DEVICE LIMITED**

Headquarter : Yokohama , Japan
E-mail:psd-sales@teldevice.co.jp
http://solutions.inrevium.com/

US Offices : Fremont , CA
E-mail:psd-nasales@teldevice.co.jp

China Offices : Shanghai
E-mail:sales@teldevice.cn

XILINX® ALLIANCE PROGRAM PREMIER MEMBER

DIGILENT BEYOND THEORY
Buy Now
http://www.digilentinc.com/INREVIUM

Design and Verification for Complex FPGA/SoC Designs

Supporting the latest Xilinx devices

www.aldec.com

A Proven EDA Solutions Provider makes all the difference.

Aldec's industry leading, mixed language HDL simulator, combined with Xilinx ISE™, enables both software and hardware designers to address core functionality requirements associated with ARM-based embedded system designs.

Functional Verification solutions from Aldec scale to meet the growing complexity and density of the most complex Xilinx devices.

- Verilog/SystemVerilog, VHDL, SystemC
- UVM/OVM, VMM, OS-VVM
- SecureIP and AXI support
- Code and Functional Coverage

Supporting:

ZYNQ™
ARTIX™ 7
KINTEX™ 7
VIRTEX® 7

XILINX® ALLIANCE PROGRAM

ALDEC® The Design Verification Company

proud global sponsor 2012 X fest

Aldec, Inc.
Corporate - **N. America**
+1.702.990.4400
sales@aldec.com
Visit us at www.aldec.com

Aldec **Israel**
+972.072.2288316
sales-il@aldec.com

Aldec **China**
+86.21.6875.2030
info@aldec.com.cn

Aldec **Europe**
+44.1295. 20.1240
sales-eu@aldec.com

Aldec **India**
+91.80.3255.1030
sales-in@aldec.com

Aldec **Japan**
+81.3.5312.1791
sales-jp@aldec.com

Aldec **Taiwan**
+886.2.2659.9119
sales-tw@aldec.com

Smart, Fast Trading Platforms Start with FPGAs

A new framework speeds application development of ultralow-latency financial systems.

by Rafeh Hulays, PhD
Vice President, Business Development
AdvancedIO Systems Inc.
rhulays@advancedio.com

Since the advent of electronic trading, a race for speed has ensued to build the fastest and smartest trading platforms. Smart and fast translates into money. The trading platform that is able to identify a trading opportunity and execute on it first will win the day. Response time has decreased from seconds, to milliseconds, to microseconds. The drive for microsecond and submicrosecond response time is simply not possible with traditional software or simple hardware architectures, a fact that is propelling the adoption of FPGA technology in ultralow-latency systems. However, programming FPGAs requires the development and migration of existing trading strategies (largely written in C and C++) into HDL machine language. This is a fundamentally different set of skills than standard C and C++ programming and requires a new investment in people, tools and time.

C-to-HDL compiler tools that port existing C code to a hardware description language (HDL) are available, and are faster than developing the code using HDL from scratch. However, the person working on porting the code must be familiar with hardware programming or must follow specific guidelines in order to generate acceptable code. Even then, the resulting code may be opaque and less efficient than code developed natively in HDL.

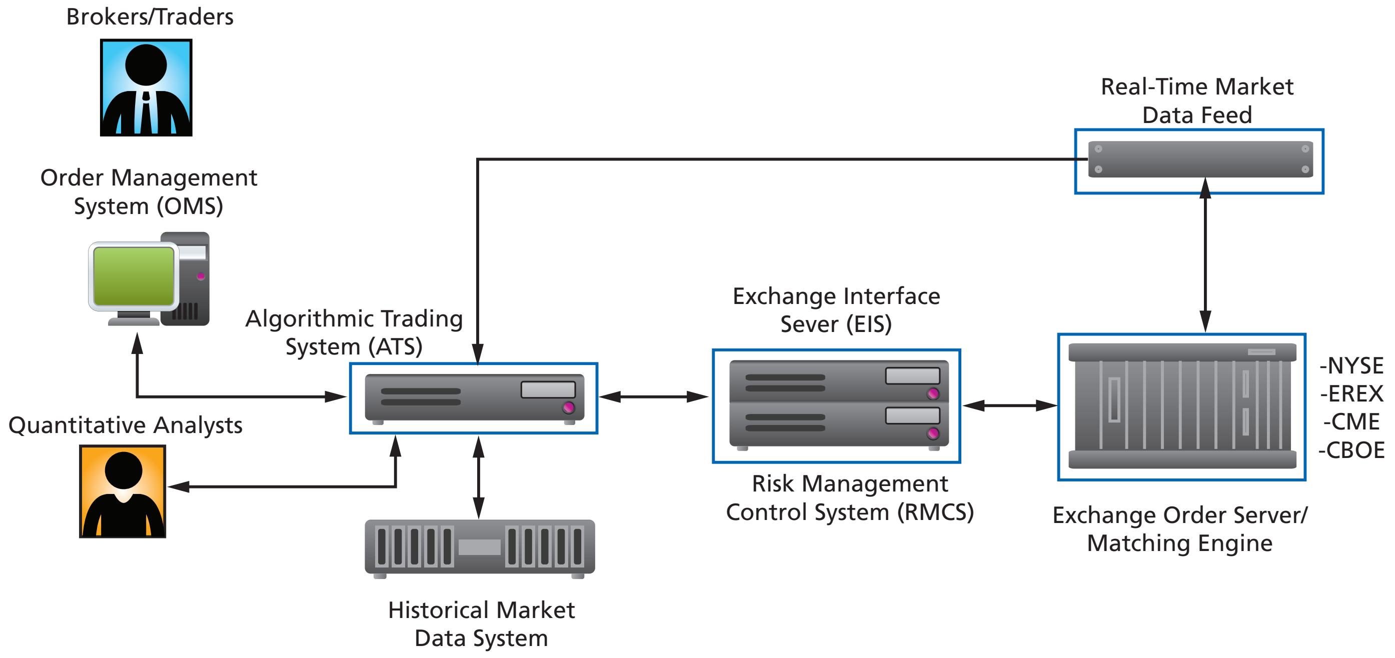
To reduce the risk involved in developing HDL code natively on an FPGA Ethernet card while also slashing development time, AdvancedIO has pioneered the use of FPGA frameworks for 10-Gigabit Ethernet (10GE) communications. Our expressXG development framework tool set provides the infrastructure necessary to ensure rapid deployment of financial services and allows seamless portability to the latest generation of FPGA cards. It integrates and optimizes all necessary core functionality required to develop applications, minimizing third-party licensing costs and shaving months off the development cycle. Development teams can then integrate a C-to-HDL compiler tool into the framework and thereby have a choice of options for developing different parts of the applications.

Indeed, both methods have a place within a project. Where efficiency and compact code are essential or where components are used in multiple projects, it's best to develop HDL components natively using the expressXG framework. Where time-to-market and customization are necessary, and where code is available in C, it may be desirable to use a C-to-HDL compiler tool. This approach addresses the fear barrier that many in the financial industry have with regard to FPGAs, without compromising performance.

AN OVERVIEW OF ALGORITHMIC TRADING

Trading in securities, derivatives, futures and other financial instruments invokes images of trading floors filled with hundreds of people shouting, milling about or peering intently at computer screens. The reality is that today, algorithms running on computer servers conduct more than 70 percent of trades in the United States. In the financial industry, speed matters. The company that can first detect and act upon an opportunity stands to profit handsomely. For some trading strategies, getting in the queue first wins the trade. No wonder, then, that companies go to great lengths to ensure they have an edge—even if just a microsecond—over the competition.

In an algorithmic trading system (ATS), algorithms running on high-performance computers handle trades and make decisions. Such systems use multiple techniques to minimize latency and gain a competitive edge. These techniques include co-location with the different exchanges, use of shortest-path networks and, increasingly, utilizing 10GE to achieve lower latency. The major trading companies are adopting all of these techniques to gain an advantage over the competition.



Brokers/Traders
Order Management System (OMS)
Algorithmic Trading System (ATS)
Quantitative Analysts
Historical Market Data System
Exchange Interface Sever (EIS)
Risk Management Control System (RMCS)
Real-Time Market Data Feed
Exchange Order Server/ Matching Engine
-NYSE
-EREX
-CME
-CBOE

Figure 1 – A simplified diagram of a trading ecosystem. The boxes outlined in blue are points where FPGAs will significantly improve performance.

An FPGA brings a level of parallelism to financial applications that is impossible to match with general-purpose processors. Designers now are striving to pack as much functionality as possible into these FPGAs.

Trading on a stock exchange is restricted to broker-dealers and market-making firms that are members of said exchange. Some of these firms provide services to other trading companies and enable them to use Direct Market Access to the exchange order book through their accounts. Firms providing Direct Market Access must implement pretrade risk-management controls in order to limit their financial exposure and to meet regulatory requirements. To do so, they must deploy servers with software to check on all trades going through their account. This must happen at wire speed, because any delay will put them and their clients at a disadvantage. Existing software-based pretrade risk-management platforms take on the order of tens of microseconds to perform the required policy check on financial transactions. Clearly, this is not fast enough for today's traders.

Figure 1 shows the elements of a simplified trading system. In reality, an ATS may have many more elements, such as additional servers or computing clouds to perform more-sophisticated algorithms that we will not discuss here. A trading system may connect to multiple exchanges directly or through one or more exchange interface servers, electronic communications networks or other means. In addition, firewalls and intrusion-detection systems are key components of a robust trading infrastructure.

LIMITATIONS OF SOFTWARE-BASED SOLUTIONS

An ATS must be able to receive multiple market feeds, decode the different protocols used (and there are several, with colorful names like FIX, FAST, ITCH and OUCH), filter the desired trading symbols and send them to predefined "baskets" that the trading algorithms (TAs) will analyze. Once the TAs decide that a trade must be acted upon, they send a trade request to the order-management system (OMS), which communicates with the exchange interface server or with the exchanges themselves, places orders and receives confirmations. An ATS server can also stamp the incoming data with an ultra-accurate time stamp and send it for archiving. Time-stamping and market data capture can be done on either the same server (with the data going to a database server over an internal network) or a separate server.

An ATS may receive data from multiple exchanges at an aggregate burst data rate exceeding 6 Gbits/second (Gbps). Recently, Nasdaq has declared its intention to offer connectivity at 40 Gbps to its data center customers, subject to regulatory approval. This increasing rate of data is putting a significant strain on system processors and is taking away from the important task of analyzing the data and making decisions on trades. Tests on file servers using commodity 10GE network interface cards have shown that when performing TCP transfers through their native stacks, even 2.2-GHz processors can be close to 100 percent utilized just processing the IP protocols, while only achieving data rates of less than 5 Gbps. To remedy this situation, one approach is to use a piece of hardware called a TCP offload engine (TOE) to accelerate the network stack. This yields some improvement in response time.

Even with TOE being done on the network card, the amount of data that passes to the main processor is so large as to cause unacceptable delay. To reduce

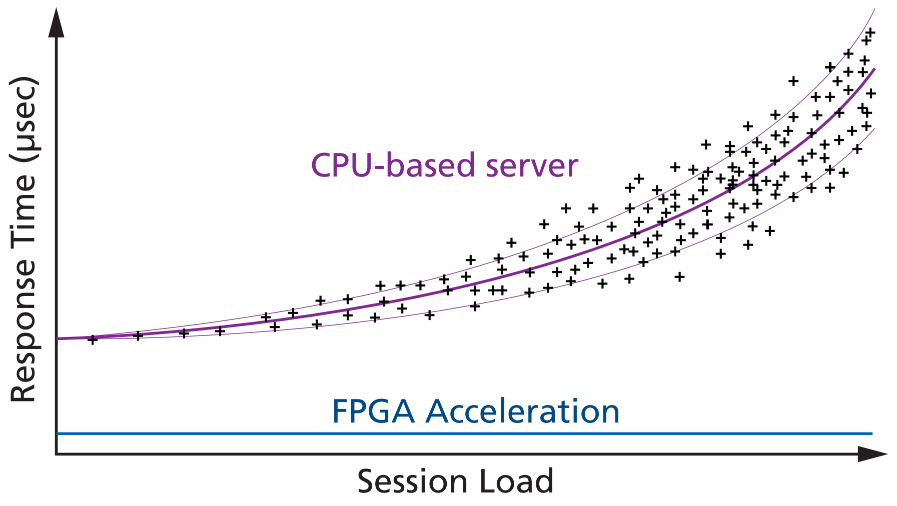


Figure 2 – Response time for applications run on a CPU-based server and those on an FPGA Ethernet card (FPGA acceleration)

latency, it is necessary to filter the relevant data at the network card before passing it to the main system processor. When all of the workload is done at the FPGA level, the response time is substantially faster. In addition, an FPGA will provide the same response time regardless of the level of load. This is not true with a regular processor. When a processor is lightly loaded, the response time is predictable, but at high load the response time increases and becomes unpredictable.

Today, communication between processors and Ethernet cards takes place via the PCI Express® bus. In theory, an eight-lane PCI Express Gen 2 bus offers a peak throughput of 4 Gbps. However, PCI Express suffers from latencies inherent in device drivers and operating-system interrupt handling. There are clear advantages to performing financial computations on the network card without having to cross the PCI Express bus to the host processor.

FPGA technology is an excellent candidate for financial applications because of its ability to process massive amounts of data in real time, with low latency and with consistency. Simply put, an FPGA brings a level of parallelism that is impossible to match with general-purpose processors. Because of the increased performance and blazing speed they are bringing to trading systems, designers are striving to pack as much functionality as possible (such as the OMS, which communicates with the exchanges, and some of the algorithms) into these FPGAs in order to reduce response time. Figure 2 illustrates the difference in response time between an FPGA-based and a CPU-based system. Not only is the FPGA considerably faster, but it also behaves consistently with increasing load, whereas the response time in a CPU-based solution increases significantly with load.

FPGA TECHNOLOGY ADVANTAGES

FPGA technology offers an unmatched capability to provide optimized solutions to the challenging problems encountered in high-bandwidth, real-time and computationally intensive financial trading applications. A solution where an FPGA executes trades at the Ethernet card has many advantages:

- Trade execution on an FPGA located close to the network's physical interface eliminates the latencies caused by the host bus, the host processor and the operating system. This drastically improves trading response time.
- FPGAs provide wire-speed performance, allowing the execution of the part of the algorithm that detects and acts on trading opportunities almost instantaneously, before others even notice the opportunity.
- FPGAs can be reprogrammed during operation, making it possible to change parameters and update algorithms to keep ahead of competitors.
- FPGAs are excellent at parallel processing, enabling them to act on multiple trades simultaneously.

Ultralow-latency financial applications are challenging to implement since they must operate at wire speed at very high bandwidth and must execute complex algorithms. Special architectures, designed with the application in mind, are required to achieve cutting-edge performance. These must take into account the need to balance computationally intensive algorithms with the need for blazing response speed to trades. Designers must make a special effort to remove bottlenecks and minimize communication and processing latencies. To avoid latencies associated with the host system bus, it is important to choose powerful FPGAs to implement the essential parts of trading algorithms, along with generous amounts of SRAM and SDRAM memory and low-latency communication ports.

Programming FPGAs using an off-the-shelf hardware module typically requires more effort and a specialized skill set than that required to create software for single or multicore processors. In addition, significant effort and investment are required to learn the documented and undocumented nuances of implementing FPGA solutions, especially at very high speeds. These characteristics may increase FPGAs' perceived risk and implementation time frames, causing some project managers and teams to avoid them and to settle instead for suboptimal software implementations. Moreover, existing trading algorithms are written almost exclusively in C and C++, and porting these into HDL code is not trivial.

There have been many approaches to simplify the task of programming FPGAs, such as embedded hard cores, software cores and tools to port C code to HDL languages. While each of these schemes has its place and each accelerates the deployment of applications, they all suffer from performance drawbacks. C-to-HDL tools are emerging as clear candidates to simplify the task of developing applications using FPGAs. However, the code these

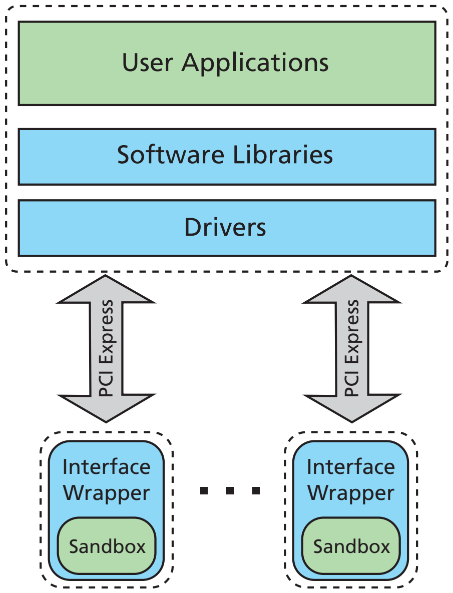


Figure 3 – A complete system-level FPGA development framework solution includes optimized low-latency Linux drivers and APIs as well as the optimized controllers shown in Figure 4.

tools generate is largely opaque and is rather difficult to optimize. Simply put, at times there is no substitute for direct HDL programming.

NATIVE HDL VS. C-TO-HDL TOOLS

A study done at the George Washington University [1, 2] reviewed the high-level language (HLL) tools that generate HDL code for FPGAs used in high-performance reconfigurable computers, and developed metrics to measure the efficiency and productivity of the various tools vs. native HDL. The researchers selected four workloads and drafted users of different levels of experience to implement the code using various tools. The results show that the HLL-to-HDL tools invariably shortened the development time by as much as 61 percent depending on the tool used. However, the results also show that the resulting code is invariably less efficient than native HDL code, with the area of utilization increasing by up to 36 percent and a frequency reduction of up to 50 percent. In addition, the research shows a considerable reduction in throughput when using some of these tools. [2] Moreover, the code these tools generated is opaque, making debugging on the HDL level challenging.

We should note, however, that the four designs used in the test were relatively simple. Some of the financial algorithms and applications are substantially more complex, and implementing them in native HDL is more challenging. We expect that as more-complex algorithms are implemented in FPGA, we will see more savings in time when using HLL-to-HDL tools. This is especially true when the algorithms are already available in C and need to be ported into HDL rather than being written in C first and then ported.

It is important to remember that C and HDL are fundamentally different, and not everything that is written in C translates well into HDL. As seen in Table 1, there are currently several variants or subsets of the C languages that vendors are using. There is an attempt under way to standardize on OpenCL, an open standard for cross-platform, parallel programming of modern processors that is being adapted for use on FPGAs. [3]

FPGA DEVELOPMENT FRAMEWORK IN HDL

Another approach to simplifying and shortening the development cycle is to use a framework written in native HDL which is highly optimized for latency and performance (Figures 3 and 4). The framework abstracts the details of Ethernet protocols and interfaces, memory controllers and host fabric interfaces, thereby reducing the development effort and schedule for designers to implement custom algorithms. This allows developers to focus 100 percent of their time on application development and integration rather than on getting all the external interfaces working on the FPGA card. A proper development framework must ensure application portability among FPGA device families and within the same family of cards. This significantly reduces the costs of future migration or upgrade cycles.

We have implemented and validated our development framework on the various families of high-performance 10GE cards from AdvancedIO Systems, which are used for multiple applications in the

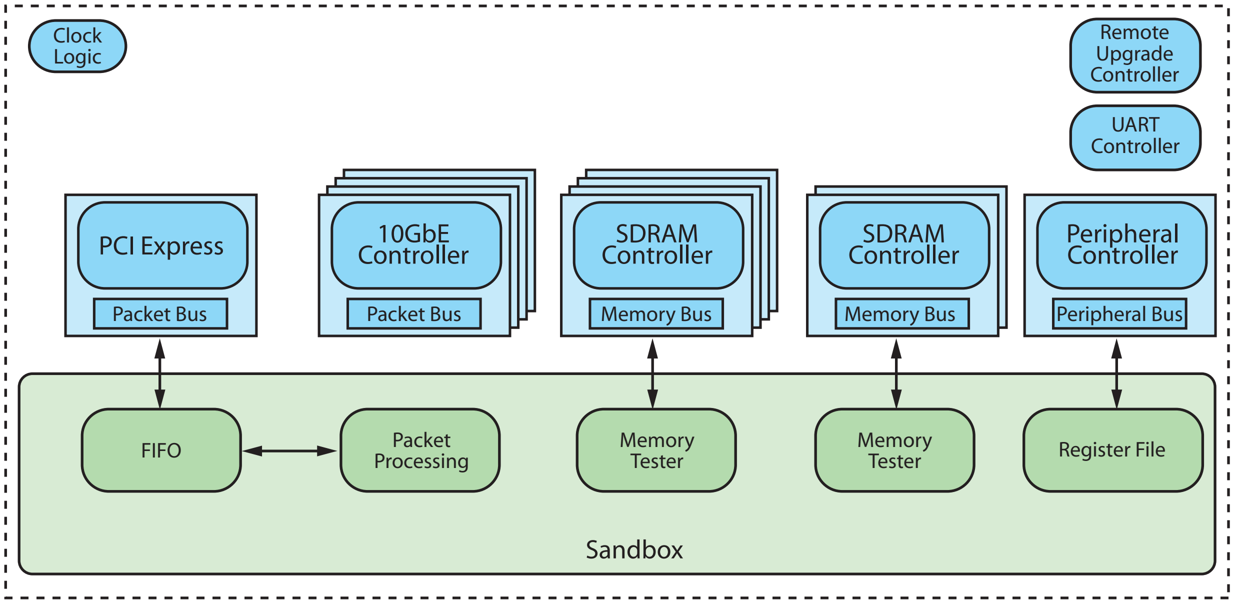


Figure 4 – A high-level view of the expressXG FPGA development framework from AdvancedIO. The top-row components (in blue) are the controllers integrated into the hardware. The components at bottom provide an infrastructure for development.

	Tool	Development Time (hrs)	Frequency (MHz)	Area (% Utilization)
1	C	3.0	–	–
2	Impulse-C	6.0	125	21.25
3	Handel-C	7.5	200	20.25
4	Carte-C	6.5	100	19.50
5	Mitrion-C	10.75	100	22.75
6	SysGen	9.0	200	19.00
7	RC Toolbox	9.0	200	19.00
8	HDLs	15.25	200	16.75

Table 1 – This comparison of HLL-to-HDL tools shows that while they deliver a considerable saving in time, the resulting code is not as efficient as that created natively in HDL. [1]

defense, financials and telecommunications markets.

The V5022 card, optimized for financial trading, features the Xilinx® Virtex®-6 HXT family of FPGAs and has all the necessary elements that an application architect needs when implementing a high-capacity, ultralow-latency trading solution. The Virtex-6 HXT family offers a large number of logic resources for complex algorithm implementations. It has two independent banks of up to 8-Gbyte, 533-MHz DDR3 SDRAM and four independent banks of up to 144-Mbit, 350-MHz QDRII+ SRAM, ideal for advanced algorithms requiring buffering or ultrafast lookup tables. The V5022 card has four 10GE ports and offers a significant reduction in latency measured from the optical cable to the MAC interface (Layer 2) inside the FPGA device.

The V5022 supports a PCI Express Gen 2 host interface. An intercard high-speed port ensures ultrafast communications between different cards in the system without the need to use the host system bus, providing a further reduction in latency. This supplies the trading platform with additional high-speed processing capability for the implementation of more-complex trading algorithms.

The development framework provides the major functionality that must be integrated into the board to ensure

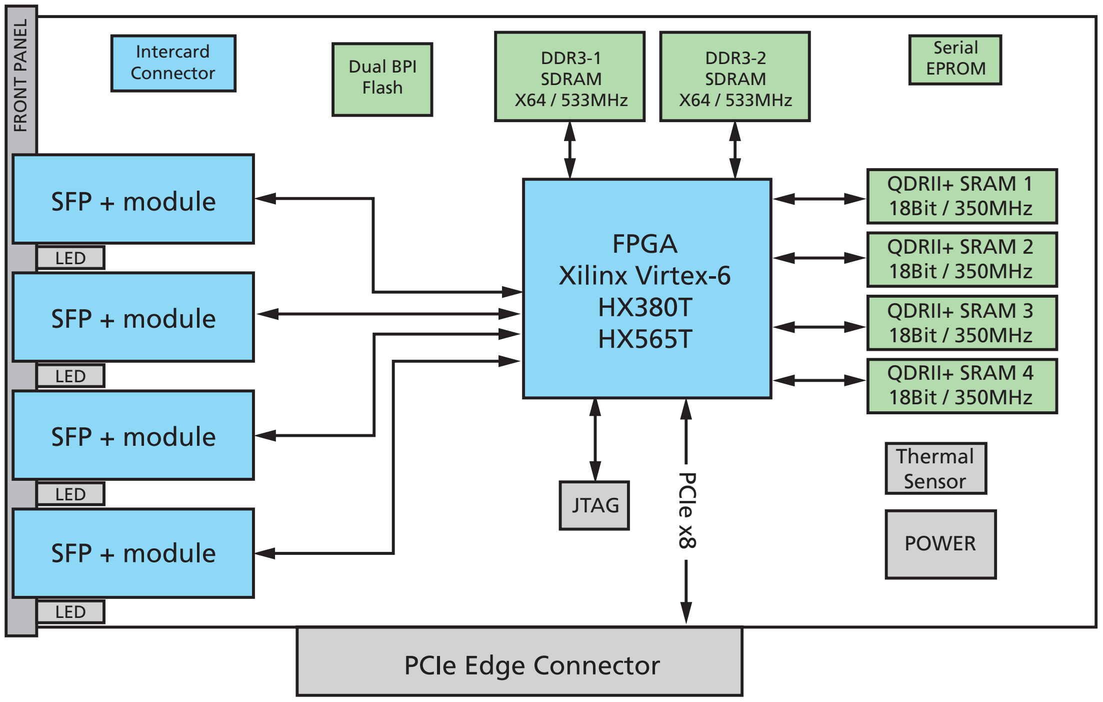

FRONT PANEL
Intercard Connector
Dual BPI Flash
DDR3-1 SDRAM X64 / 533MHz
DDR3-2 SDRAM X64 / 533MHz
Serial EPROM
SFP + module
LED
SFP + module
LED
SFP + module
LED
SFP + module
LED
FPGA Xilinx Virtex-6 HX380T HX565T
QDRII+ SRAM 1 18Bit / 350MHz
QDRII+ SRAM 2 18Bit / 350MHz
QDRII+ SRAM 3 18Bit / 350MHz
QDRII+ SRAM 4 18Bit / 350MHz
Thermal Sensor
JTAG
PCIe x8
POWER
PCIe Edge Connector

Figure 5 – A high-level diagram for the V5022 shows the different elements that an FPGA designer needs to communicate with when implementing a solution.

that everything works the way it should. Suffice it to say that the logistics of acquiring, integrating and optimizing such functionality is a costly and time-consuming proposition. Hence, the development framework offers a great value for project managers and help with time-to-market.

AdvancedIO has done extensive work to ensure that its framework is optimized to provide the best performance possible and that it occupies a small footprint and performs efficiently. This leaves the bulk of the FPGA resources available for the development of applications. For example, on the V5022 and when using the Xilinx Virtex-6 HX565T, the framework occupies less than 7.5 percent of the resources of the FPGA. This includes a PCIe® interface, four 10GE interfaces, two SDRAM controllers and four SRAM controllers.

The development framework provides a "sandbox" where programmers can develop their applications. It has easy-to-understand interfaces to the outside world, allowing the quick integration of applications on the FPGA card. Sample code and examples are provided to showcase how to exercise the interfaces and get data flowing through the system right out of the box, giving developers more confidence.

SHAVING MONTHS OFF DEVELOPMENT CYCLE

To reduce the risk involved in developing HDL code on FPGA devices and to reduce development time as well, our FPGA development framework integrates and optimizes all the controllers necessary to develop applications on the FPGA card. This shaves at least many months off the development of projects. We have also proposed the integration of a C-to-HDL compiler tool into the development framework, as studies have shown that while the code it generates is less efficient than native HDL coding, it delivers a significant reduction in development time.

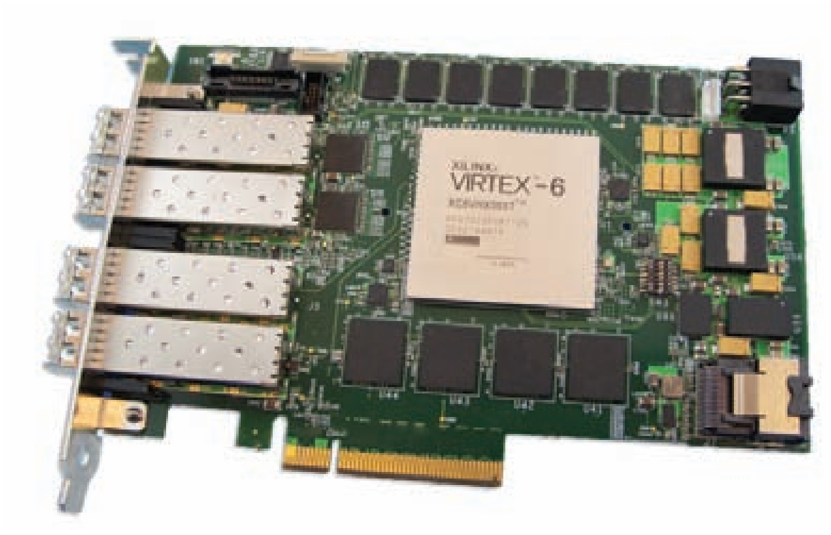


Figure 6 – The V5022 is a single-slot, half-length card deployed worldwide in financial trading systems.

In real-world scenarios, there is no single approach that works best for all situations. A design team should have many tools and choices when weighing development options. Where efficiency and compact code are essential or where components are used in multiple projects, the best choice is to develop HDL components using the FPGA development framework. Where speed getting to market and customization are necessary, and where code is already available in C, it may be desirable to use a C-to-HDL compiler tool. It's best to develop fixed-function modules such as TCP/IP and UDP/IP stacks in HDL using the FPGA development framework, while algorithms that require frequent changes can make use of high-level language tools such as the Xilinx AutoESL [4] or the Impulse [5] high-level synthesis tools.

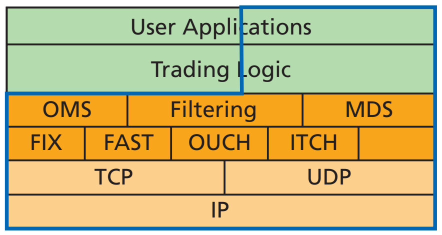


Figure 7 -- Among the elements of an ATS, the blue lines indicate the software components in which FPGAs will increase performance.

APPLICATION TO FINANCIAL TRADING SYSTEMS

Figure 7 shows the different parts that an algorithmic trading system needs to implement at a very high level. An ATS has to be able to read one or more market data feeds, perform filtering on this data against different baskets, undertake analysis, make trading decisions and communicate with one or more exchanges.

Trading logic and strategy components are subject to frequent changes and are specific to different trading companies. They are perfect candidates to be implemented using a C-to-HDL compiler in order to meet time-to-market requirements and, if the market demands, move to a more-efficient implementation at a later stage (using the FPGA development framework). On the other side, network and financial protocols do not change often and an efficient implementation of such protocols significantly affects the performance of the system. Therefore, we recommend implementing these natively in HDL using the expressXG development framework.

SPEED, RESPONSIVENESS AND PREDICTABILITY

In the financial trading sector, the drive for speed and ultralow latency is necessitating the adoption of FPGA technology. Users in this sector face many challenges, including unfamiliarity with FPGA design, different skill sets and a large existing code base in high-level languages, among others. Our expressXG FPGA development framework promises to simplify and shorten the development of applications on FPGA-powered high-performance Ethernet PCI Express cards. To assist in the porting of existing C code or where time-to-market is paramount, we propose integrating a C-to-HDL compiler within this framework.

We believe that expressXG, integrated with a C-to-HDL compiler, will assist in the adoption of FPGA technology, improving the speed, responsiveness and predictability of trading systems. For more details on the expressXG system, see *http://www.advancedio.com/.*

References

1. *E. El-Araby, S.G. Merchant and T. El-Ghazawi, "A Framework for Evaluating High-Level Design Methodologies for High-Performance Reconfigurable Computers."* IEEE Transactions on Parallel and Distributed Systems, *Vol. 22, No. 1., pp. 33-45, January 2011. Abstract available at* http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5445087.

2. *E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi and G.B. Newby, "Comparative Analysis of High-Level Programming for Reconfigurable Computers: Methodology and Empirical Study."* Proceedings of the Third Southern Conference on Programmable Logic *(SPL '07), February 2007. Abstract available at* http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4234328.

3. *Chronus Group,* http://www.khronos.org/opencl/

4. *Xilinx, Inc.,* http://www.xilinx.com/tools/autoesl.htm

5. *Impulse Accelerated Technologies,*

ATLYS™

Comprehensive Spartan®-6 Design Platform

- Xilinx Spartan-6 XC6SLX45 FPGA
- 1Gbit DDR2, 128Mbit x4 Quad SPI Flash
- 4 HDMI ports (2 in, 2 out)
- Trimode (10/100/1000) Ethernet Interface
- Audio codec with in, out, mic & headphone ports
- USB-UART and USB-HID host ports
- On-board USB-based FPGA configuration circuitry
- On-board, real-time power supply monitors

$199 (Academic Price)

JTAG-HS1™

The perfect programming solution for Xilinx® devices

- Compatible w/ all Xilinx tools & devices
- Up to 30 Mbit / sec on JTAG bus
- Supports 6 & 14-pin headers
- Volume pricing available

$49.99

NEXYS 3™

Spartan®-6 Power Unleashed

- Xilinx Spartan6 XC6LX16 FPGA
- 16Mbyte Micron Cellular RAM
- 16Mbyte Micron Parallel PCM
- 16Mbyte Micron Quad-mode SPI PCM
- 16Mbyte Quad-mode SPI PCM
- 8-bit VGA
- 10/100 SMSC LAN8710 PHY
- Digilent USB port for power, config & data
- USB-UART, type-A USB host
- Four Pmod™ connectors, one VHDC connector

$119 (Academic Price)

JTAG-SMT1™

Embedded programming for Xilinx® devices

- Compatible w/ all Xilinx tools & devices
- Up to 30 Mbit / sec on JTAG bus
- Surface mounted
- Volume pricing available

$27.99 (100+)

DIGILENT® BEYOND THEORY www.digilentinc.com (509) 334-6306

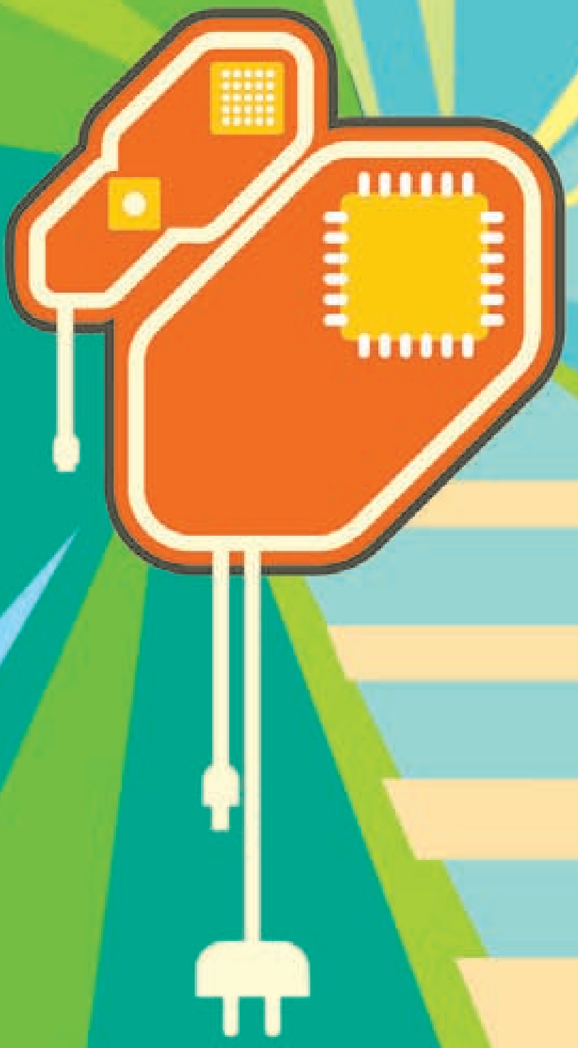
Accelerate Partial Reconfiguration with a 100% Hardware Solution

For timing-critical applications like video, a hardware-only solution improves operational capabilities for Xilinx FPGAs.

by Sébastien Lamonnier
FPGA Designer
Sagem DS (Safran Group)
sebastien.lamonnier@sagem.com

Marc Thoris
FPGA Project Manager
Sagem DS (Safran Group)
marc.thoris@sagem.com

Marlène Ambielle
FPGA Designer
Sagem DS (Safran Group)
marlene.ambielle@sagem.com



In many modern applications such as video processing, minimizing FPGA reconfiguration time is critical in order to avoid losing too many images. Partial reconfiguration is a technique that allows users to reconfigure a small part of the FPGA without impacting logical elements around it. For the human eye to see an image without flicker, the reconfiguration time must be less than 40 milliseconds. That's very little time to reconfigure an entire device, save for the smallest FPGAs; and in certain specific applications, this reconfiguration time must be even less. Hence the appeal of partial reconfiguration: Because a partial bitstream is smaller than a full one, it takes less time to reconfigure.

At Sagem DS, we have devised a technique that allows FPGA designers to accomplish partial reconfiguration very fast. The ML507 [1] was the Xilinx® reference board we used for testing and validating the solution and to measure timing. Typically, the components on this board are a Virtex®-5 FPGA (XC5VFX70T-FFG1136), a CPLD (used as a routing component) and two XCF32P memories (Xilinx Platform Flash).

A MICROBLAZE VS. HARDWARE SOLUTION

In many documents, partial reconfiguration (PR) uses an internal controller like a MicroBlaze® or an external processor. Implementing a processor within the FPGA takes its toll in development time and consumes significant device resources, depending upon the configuration. Likewise, an external processor costs money and board space. Moreover, buses such as PLB or AXI have latency that degrades performance in terms of reconfiguration time.

For those reasons, we adopted an all-hardware solution based on a little state machine and the Internal Configuration Access Port (ICAP) interface for bitstream loading. This approach offers many advantages. Because there is no latency, it consumes few resources (less than 300 lookup tables on the FPGA), and designers can optimize the timing of the partial reconfiguration.

DEVELOPMENT FLOW OVERVIEW

From VHDL conception to bitstream and partial-bitstream creation, our hardware-based partial-reconfiguration flow is the same as the generic process described in Xilinx tutorials, user guides [2] and application notes, except that there is no embedded processor. Users must define reconfigurable areas in PlanAhead™ and import reconfigurable modules for each area. For all configuration runs, you import static logic from a previous run.

During partial reconfiguration, the FPGA must be in slave mode. That is to say, the only usable interfaces are JTAG, slave serial, slave SelectMap or ICAP. An external component should drive the FPGA CCLK for reconfiguration; ICAP is not accessible at the first FPGA boot. To achieve a timing-efficient reconfiguration, there can be no serial interfaces. That leaves us with at least two interesting interface choices: SelectMap and ICAP.

The first option is to use the SelectMap interface for full and partial bitstream loading. This method of configuration necessitates adding a Bitgen option (-g Persist) when creating the bitstreams. Here, the FPGA keeps the SelectMap pins under its control to load the partial bitstream. In addition, with SelectMap there is no signal to indicate the end of the process very precisely (like the DONE signal for full configuration). So it is difficult to know exactly when the partial reconfiguration has ended. Users must create a module that estimates when all configuration data has been sent.

This is why we opted instead to use the ICAP primitive to load the partial bitstreams. ICAP is not a self-configuring interface like SelectMap, so we do use SelectMap to realize the first boot for the FPGA. After that, the user code fully controls the ICAP primitive.

ICAP offers two advantages over a full SelectMap design. First, the commutation (SelectMap for first boot, ICAP for partial-bitstream loading) is transparent for users and there is no need for a Bitgen option. Thus, the user can control the memory pins. The second, and most important, advantage is that ICAP continually pushes a status on its output. This status changes if the FPGA is in a reconfiguration mode. That allows the user to "see" the end of partial reconfiguration.

RECONFIGURABLE FUNCTIONS AND COMPONENTS

In our company, we develop video-processing applications using FPGAs. These functions use logic elements, some BRAM components and a lot of

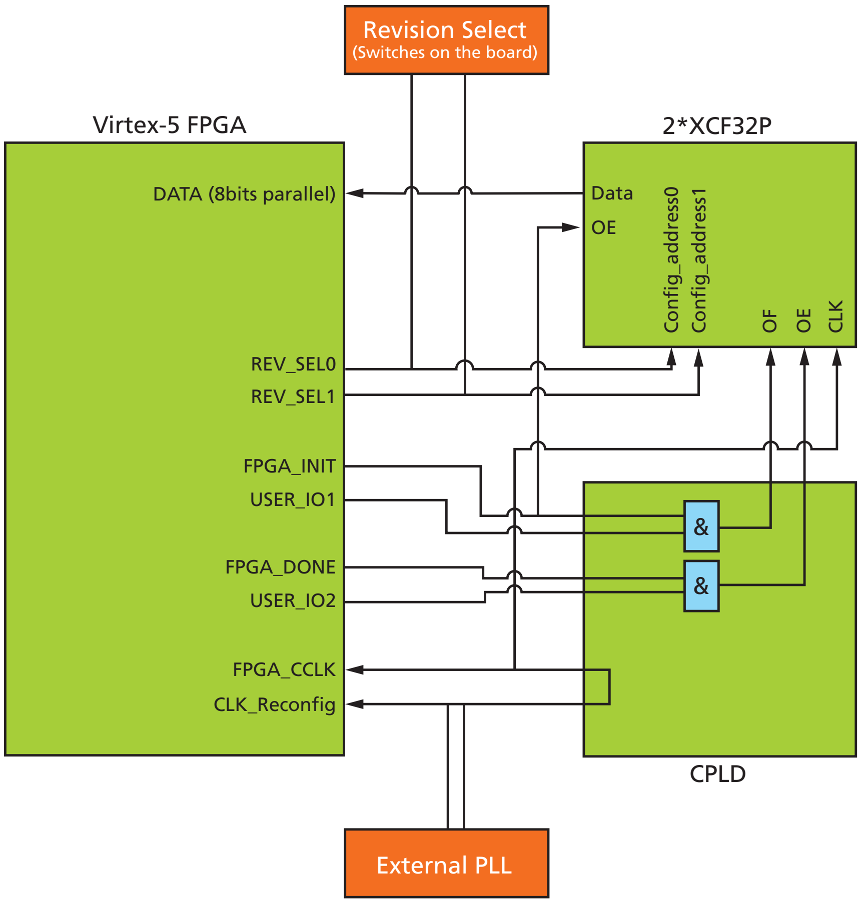


Figure 1 – Partial-bitstream loading, ICAP synchronization and desynchronization

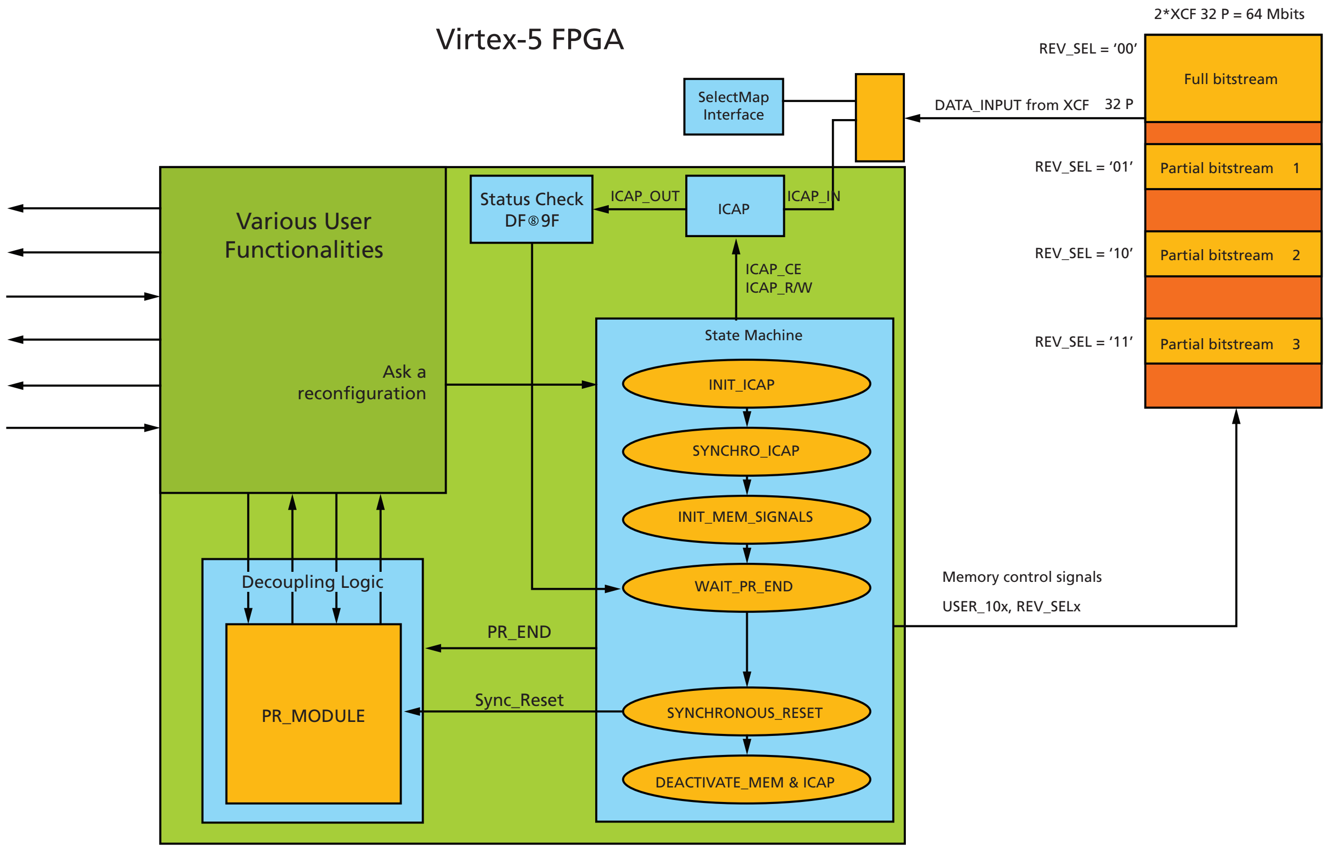


Figure 2 – Hardware PR control module within the FPGA

DSPs. It is very interesting to study the reconfiguration time of these three elements because it will guide us in sizing a reconfigurable area based on the answers to two questions: How will we size the area to respect the 40-millisecond image refresh time our applications demand? And, how many different elements must we include in this area? On the ML507, we answer those questions by testing various types of configurations.

As shown in Figure 1, on the ML507 we chose the external phase-locked loop (PLL) as a reference clock for partial-reconfiguration control. Some connections are not our choice but are dictated by board design. The schematic does not show safeguards against conflicting levels on nets. The clock has a frequency of 33 MHz, which is the maximum speed of the two XCF32P memories on the board. [3] The data bus is 8 bits wide, which allows a data transfer rate of up to 264 Mbits/second.

On an FPGA, the smallest reconfigurable area is a frame. Frame size differs according to the type of FPGA. For example, on the Virtex-5 the frame is 20 CLBs high, [4] but on the Virtex-6 it is twice that number. The partial bitstream is a combination of elementary frames, BRAM, DSP and a few configuration words. The configuration interface and partial bitstream structure (number of elements contained) are the keys to achieve a timing-efficient partial reconfiguration.

RECONFIGURATION PROCESS

Figure 2 illustrates the nonexhaustive architecture of an FPGA designed for partial reconfiguration. All things begin when you switch on the board. The FPGA loads the full bitstream through the parallel SelectMap interface. After that, the user code controls all modules present in the light- and dark-green zones of the illustration. The decoupling logic is a specific zone designed around the PR module.

During the reconfiguration process, unknown states can appear on connection nets. This zone allows you to maintain a known state on the connection nets between the static zone and reconfigurable zone. The decoupling logic is in fact very small, consisting simply of a D flip-flop and a multiplexer on each net. The signal "PR_END" controls the multiplexer.

Now it's time to reconfigure the PR_MODULE (for example, a histogram correction function) with a new configuration contained in a partial bitstream in memory. The signal "Ask for a reconfiguration" begins the partial-reconfiguration process, which follows a succession of states (the list given in Figure 2 is not exhaustive).

- The first state, ICAP initialization, exists to give a default value to the control signals (CE and RW high).
- The second state prepares ICAP to receive data from memory. ICAP is

now activated and is in write mode (write configuration in the FPGA).

- The third state controls memory signals and initializes the partial-bitstream transfer sequence from memory to FPGA. In parallel, decoupling logic retains signals between the static and reconfigurable regions. This prevents undesirable data from propagating in user functions.

- The fourth state is a wait state. During this time, ICAP loads configuration frames into the FPGA, while the "Status Check" block reads the ICAP output status. Detection of a "desynchronize" word releases the wait state.

- The fifth state pushes a synchronous reset in the PR_MODULE to reset and give a known state to the new logical elements.

- Finally, the last state releases decoupling logic and deactivates ICAP and memory.

INSIDE ICAP

Users need not know or care precisely how ICAP works. That's because the partial bitstream provides all the things ICAP needs for this application. Nevertheless, it's important to understand two 32-bit words in ICAP: synchronize and desynchronize.

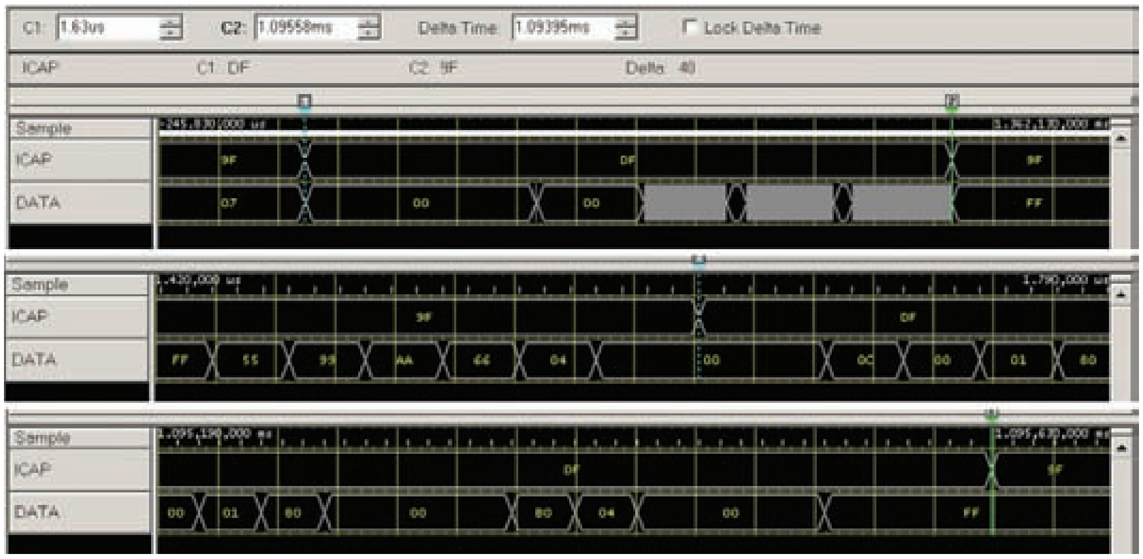
The first word, "synchronize," is 5599AA66h on input. This word makes ICAP output changes from 9Fh to DFh (DFh means "component synchronized"). During all the time the ICAP output status is DFh, the FPGA loads in new configuration frames.

When configuration data is sent, the bitstream contains a "desynchronize" word, which is 000000B0. When ICAP receives this word, its output status changes to 9Fh, indicating that the component is desynchronized. So, checking the ICAP output status gives us a precise idea of the partial-reconfiguration time. Figure 3 shows the reconfiguration time for a frame with relevant signals.

EXPERIMENTS AND RESULTS

Designing a video-processing function such as automatic histogram correction (AHC) requires logic cells, BRAMs and a lot of DSP. Because corrections are not the same for illumination systems, thermic or TV video, the FPGA must adapt itself to respond in a minimum amount of time (less than 40 ms—that is to say, the time it takes the eye to register one image). Results seen in Figure 4(a), (b) and (c) give us a precise idea of the reconfiguration time according to our interface; we measured these timing results with little reconfigurable modules (one with logic only, one with logic and BRAM and the last with logic and DSP). The results show us that the time is linear by component category. Of course, the results are only valid with our interface specification (clock at 33 MHz and a bus that's 8 bits wide).

Reconfiguring a bigger function, such as one to handle AHC, is more relevant for our products and puts the FPGA through its paces in operational conditions. To explain the AHC function briefly: A TV image has three components, Y, Cr and Cb. Y is luminance and Cr and Cb are red and blue colors. An AHC module must, for example, convert an image in RGB format and apply specific treatment on each color. In the case of black-and-white images, we can do a min, a max and an average, and then we apply a gain, an offset or both to add more luminosity (in case of dark images) and contrast. In terms of FPGA resources, this correction costs around 5,000 lookup tables


C1: 1.63us C2: 1.09558ms Delta Time: 1.09395ms Lock Delta Time

ICAP C1: DF C2: 9F Delta: 40

Figure 3 – ICAP timing for one reconfigured frame region consisting of six slices

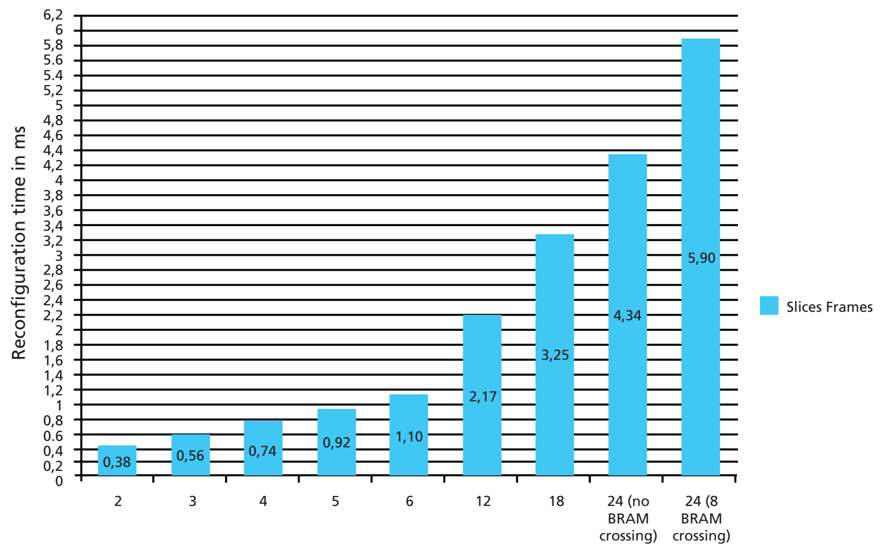


Figure 4(a) – Reconfiguration time for slice frames only

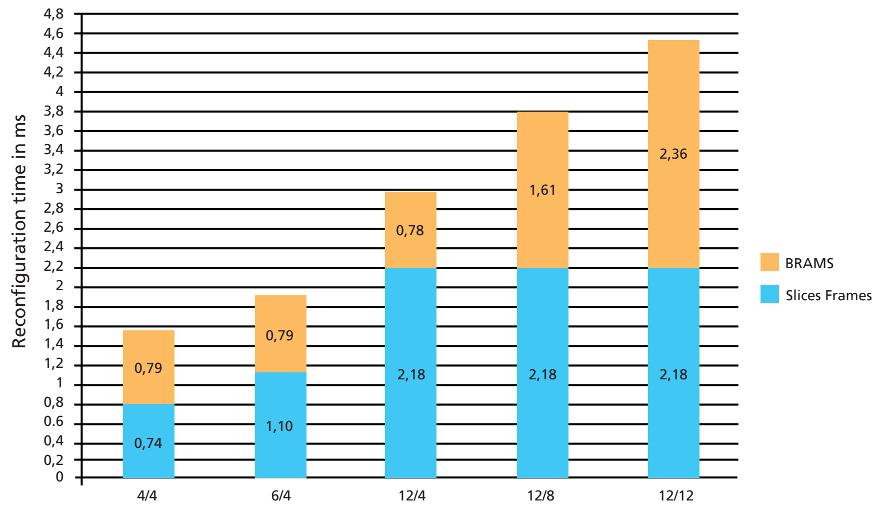


Figure 4(b) – Reconfiguration time for slice frames and BRAMs

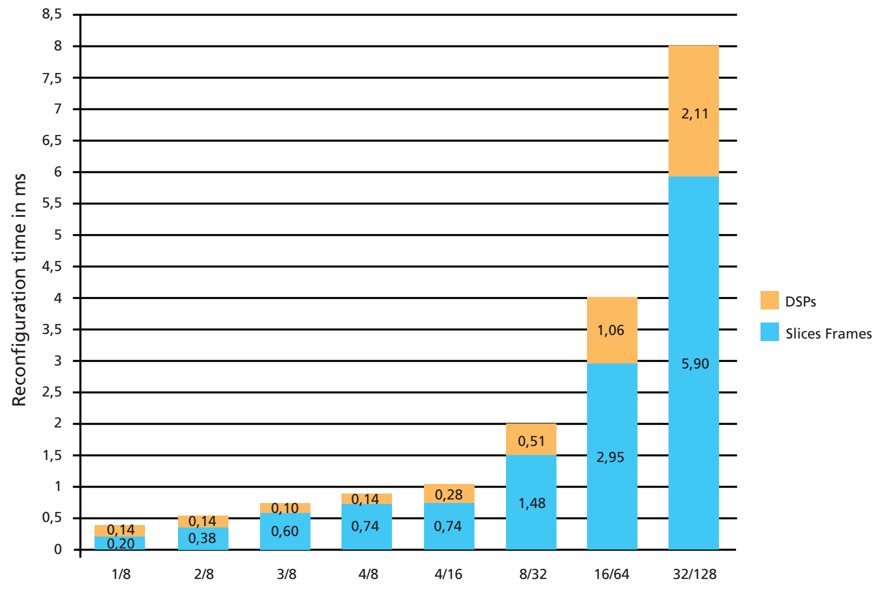


Figure 4(c) – Reconfiguration time for slice frames and DSPs

(LUTs), 4,000 flip-flops, 100 DSP slices and 20 BRAM slices.

Figure 3 shows the reconfigurable zone we chose for our AHC function. This zone contains 7,840 LUTs and flip-flops, 112 DSP and 28 BRAM slices. With such a design, we respect Xilinx's recommendation that logic density in the reconfigurable zone must not exceed 80 percent. The Virtex-5 architecture has four LUTs and four flip-flops in a slice. A configurable logic block (CLB) has two slices, so a frame contains 160 LUTs and 160 flip-flops. Our region contains 49 frames.

According to Figure 4(a), (b) and (c), we can calculate reconfiguration time this way: Logic needs 9.8 ms, 112 DSP needs 2 ms and 28 BRAM needs 6.4 ms. In all, then, the region needs 18.2 ms to reconfigure. That is far less than the 40 ms it takes the eye to register one image. Figure 5 shows the PlanAhead view with the reconfigurable zone.

ELIMINATION OF SOFTWARE DEVELOPMENT

As shown in Figure 1, we can replace the CPLD with logical components on a custom board. For now, the solution works fine, but it is not optimal. We can improve the reconfiguration interface by using a wider bus (ICAP supports up to 32 bits parallel) and a higher frequency.

The concept of 100 percent hardware partial reconfiguration represents a gain in term of system development cost because there is no need for software development, as would be the case in a solution using the MicroBlaze. By using the ICAP interface, a designer has total control of what is done in the FPGA, and ICAP greatly improves the performance and functionality of the system. Finally, for our video-processing applications in particular, hardware partial reconfiguration is a relevant feature that will enhance our company's products.

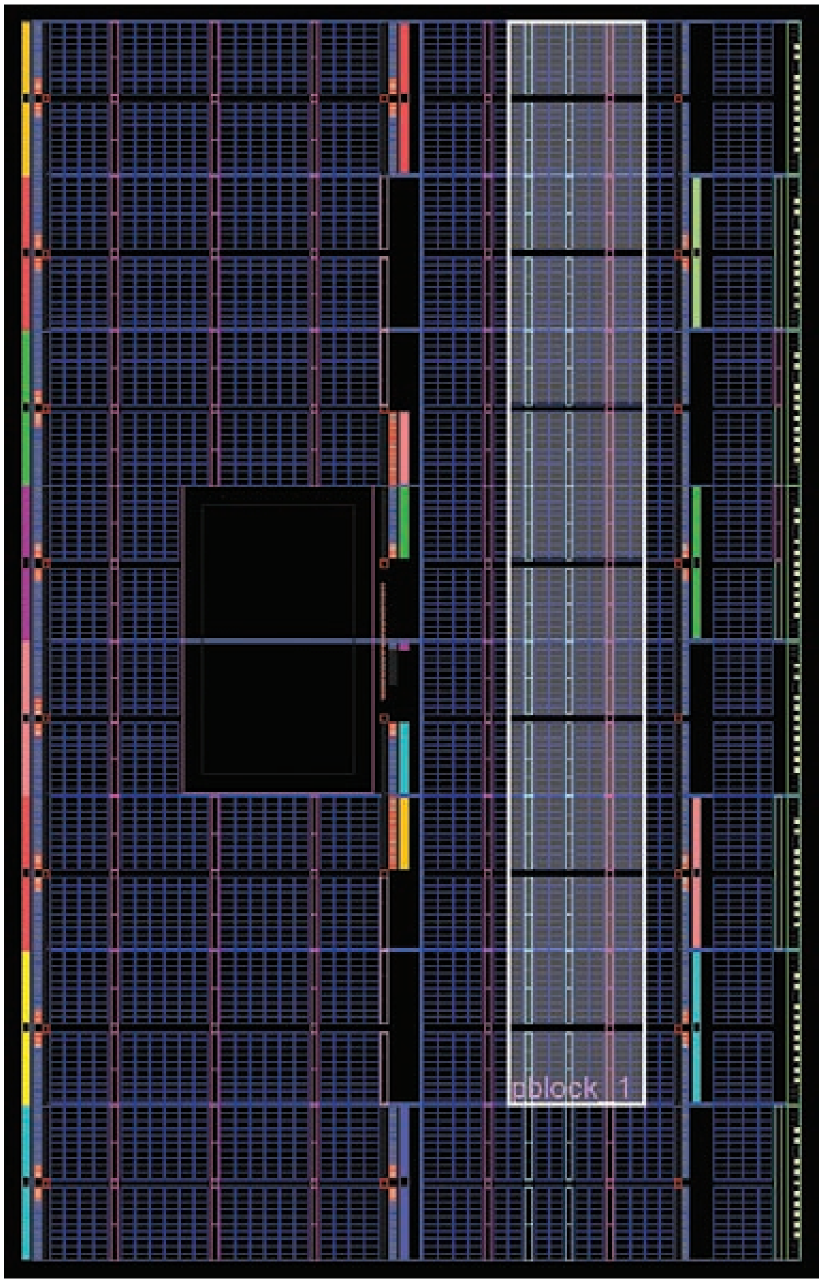


Figure 5 – PlanAhead view with reconfigurable zone

References

1. *ML505/6/7 Block Diagram, Version A, Rev. 02, Xilinx, January 2008,* http://www.xilinx.com/support/documentation/boards_and_kits/ml50x_schematics.pdf

2. *Partial Reconfiguration User Guide, UG702, v13.1, Xilinx, March 2011,* http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf

3. *Platform Flash In-System Programmable Configuration PROMs, DS123, v2.18, Xilinx, May 2010,* http://www.xilinx.com/support/documentation/data_sheets/ds123.pdf

4. *Virtex-5 FPGA Configuration User Guide, UG191, v3.9.1, Xilinx, August 2010,* http://www.xilinx.com/support/documentation/user_guides/ug191.pdf

GigaBee
Spartan-6 LX

- Scalable: 45K to 150K Logic Cells
- Gigabit Ethernet MAC and PHY
- 2 independent DDR3 Memory Banks
- LVDS IO/s
- Very Small: 40 mm x 50 mm

TE0320
Spartan-3A DSP

- High-Speed USB 2.0
- 32-bit, 128 MByte DDR RAM

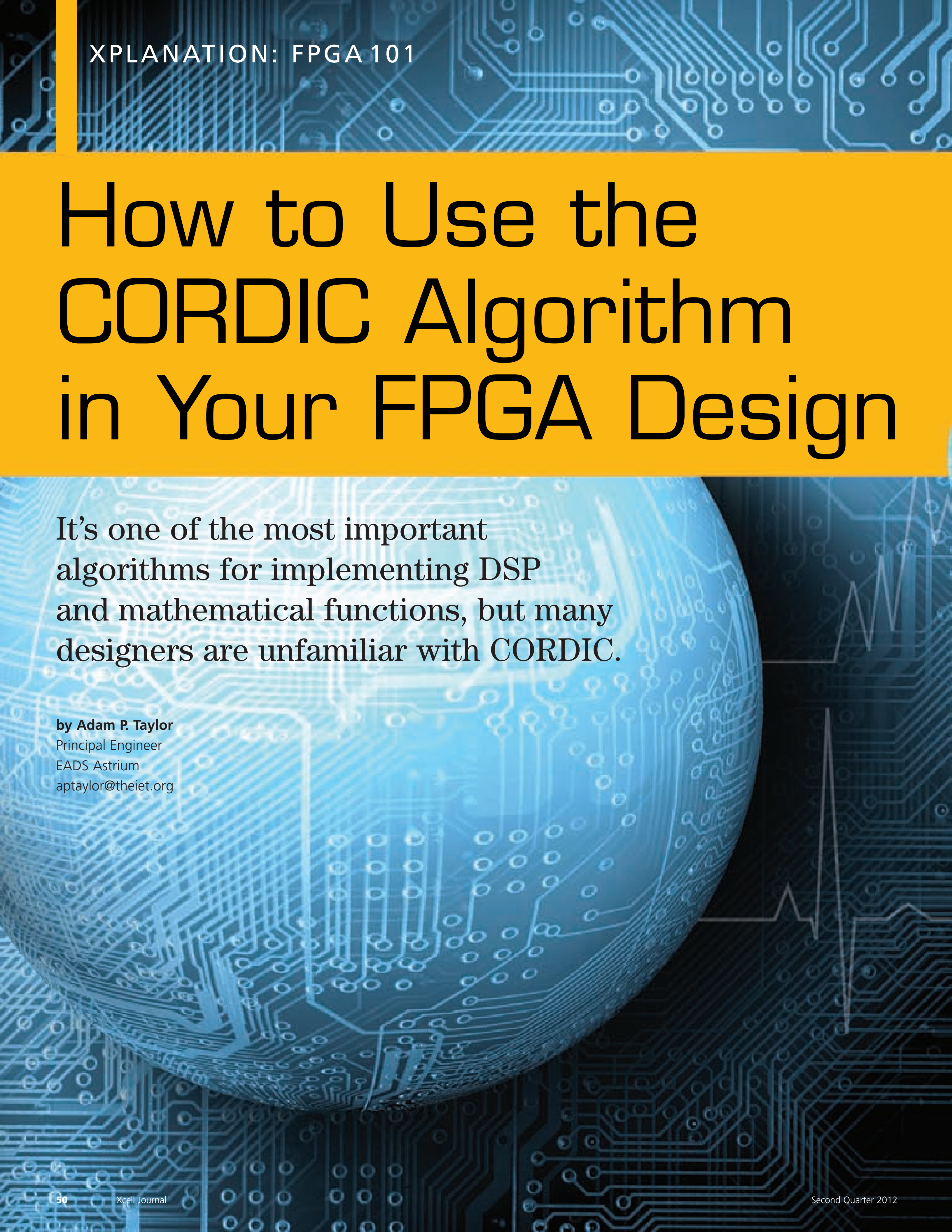
Common Module Properties
- On Board Power Supply
- Very Low Cost
- Long-Term Available
- Free Reference Designs
- Ruggedized for Industrial Usage
- Customized Versions Available
- Custom Integration Services

Development Services
- Hardware Design
- HDL Design
- Software Development

trenz electronic

www.trenz-electronic.de

How to Use the CORDIC Algorithm in Your FPGA Design



It's one of the most important algorithms for implementing DSP and mathematical functions, but many designers are unfamiliar with CORDIC.

by Adam P. Taylor
Principal Engineer
EADS Astrium
aptaylor@theiet.org

M

Most engineers tasked with implementing a mathematical function such as sine, cosine or square root within an FPGA may initially think of doing so by means of a lookup table, possibly combined with linear interpolation or a power series if multipliers are available. However, in cases like this the CORDIC algorithm is one of the most important tools in your arsenal, albeit one that few engineers are aware of.

Invented by Jack Volder while designing a new navigation computer at Convair for the B-58A Hustler program in 1959, CORDIC—it stands for Coordinate Rotation Digital Computer—is a simple algorithm designed to calculate mathematical, trigonometric and hyperbolic mathematical functions.

The real beauty of this algorithm is that you can implement it with a very small FPGA footprint. CORDIC requires only a small lookup table, along with logic to perform shifts and additions. Importantly, the algorithm requires no dedicated multipliers or dividers.

This algorithm is one of the most helpful for DSP and industrial and control applications. Perhaps its most common use is in implementing traditional mathematical functions as shown in Table 1, where multipliers, dividers or more interesting functions are required in devices that lack dedicated multipliers or DSP blocks. For example, designers use CORDICs in many small industrial controllers to implement mathematical transfer functions and true RMS measurement. Engineers are also using CORDICs in biomedical applications to compute fast Fourier transforms (FFTs) to analyze the frequency content of many physiological signals. In this application, along with the traditional mathematical functions, designers use the CORDIC to implement the FFT twiddle factors.

Configuration	Rotation	Vectoring
Linear	Op Y = X * Y	Op Z = X/Y
Hyperbolic	Op X = CosH(X) Op Y = SinH(Y)	Op Z = ArcTanH
Circular	Op X = Cos(X) Op Y = Sin(Y)	Op Z = ArcTanH (Y) Op X = SQR($X^2 + Y^2$)

Table 1 – Functions resulting from CORDIC configurations and mode

INSIDE CORDIC

The CORDIC algorithm can operate in one of three configurations: linear, circular or hyperbolic. Within each of these configurations the algorithm functions in one of two modes—rotation or vectoring. In rotation mode, the input vector is rotated by a specified angle, while in vectoring mode the algorithm rotates the input vector to the x axis while recording the angle of rotation required.

Additionally, you can derive other functions from the CORDIC outputs as well. In many cases you could even implement these through the use of another CORDIC in a different configuration, as shown here.

```
     Tan   = Sin / Cos
    TanH = Sinh / Cosh
  Exponential = Sinh + Cosh
 Natural Logarithm = 2 * ArcTanH
(where X = Argument +1 Y = Argument – 1)

     SQR = (X² – Y²)
(where X = Argument +0.25 Y = Argument –
0.25)
```

The unified algorithm seen below covers all three CORDIC configurations. The algorithm has three inputs, X, Y and Z. Table 2 defines the required lookup table precalculated values depending upon configuration, while Table 3 addresses how these are initialized at startup depending upon the mode of operation (vectoring or rotation).

$$X_{i+1} = X_i - m * Y_i * d_i * 2^i$$
$$Y_{i+1} = Y_i + X_i * d_i * 2^i$$
$$Z_{i+1} = Z_i - d_i * e_i$$

where m defines the configuration for either hyperbolic (m = -1), linear (m = 0) or circular (m = 1). Correspondingly, the value of e_i as the angle of rotation changes depending upon the configuration. The value of e_i is normally implemented as a small lookup table within the FPGA and is defined in Table 2.

Configuration	e_i
Linear	2^{-i}
Hyperbolic	ArcTanH(2^{-i})
Circular	ArcTan(2^{-i})

Table 2 – CORDIC angle of rotation

In the equation, d_i is the direction of rotation, which depends upon the mode of operation. For rotation mode $d_i = -1$ if $Z_i < 0$ else +1, while in vectoring mode $d_i = +1$ if $Y_i < 0$ else -1.

When configured in either a circular or hyperbolic configuration using rotation mode, the output results will have gain that can be precalculated using the number of rotations defined in the equation

$$A_n = \sum\sqrt{(1+2^{-2i})}$$

This gain is typically fed back into the initial setting of the algorithm to remove the need for post-scaling of the result.

Working designers must keep in mind that the CORDIC algorithm only operates within a strict convergence zone. You may have to do some prescaling to ensure it performs as expected. It is worth noting that the algorithm will get more accurate the more iterations (serial) or stages (paral-

Mode	Y	X	Z
Circular rotation	A_n	0	Argument Z
Circular vectoring	1	Argument X	0
Hyperbolic rotation	A_n	0	Argument Z
Hyperbolic vectoring	Argument + 1	Argument - 1	0
Linear rotation	0	Argument X	Argument Z
Linear vectoring	Argument Y	Argument X	0

Table 3 – Initialization settings depending upon mathematical operation as demonstrated in Table 1

Iteration	x	y	z	Direction	Gain
Initial	0.607253	0.000000	0.017453		
0	0.607253	0.607253	-0.767945	1	0.707107
1	0.910879	0.303626	-0.304298	-1	0.894427
2	0.986786	0.075907	-0.059319	-1	0.970143
3	0.996274	-0.047442	0.065036	-1	0.992278
4	0.999239	0.014826	0.002617	1	0.998053
5	0.998776	0.046052	-0.028623	1	0.999512
6	0.999496	0.030446	-0.012999	-1	0.999878
7	0.999734	0.022637	-0.005186	-1	0.999969
8	0.999822	0.018732	-0.001280	-1	0.999992
9	0.999859	0.016779	0.000673	-1	0.999998
10	0.999842	0.017756	-0.000304	1	1
11	0.999851	0.017268	0.000185	-1	1
12	0.999847	0.017512	-0.000060	1	1
13	0.999849	0.017390	0.000063	-1	1
14	0.999848	0.017451	0.000001	1	1
	Cos	Sin		A_n	0.607253
CORDIC	0.999848	0.017451			
Actual	0.999848	0.017452			

Table 4 – Excel Model of CORDIC configured for circular rotation

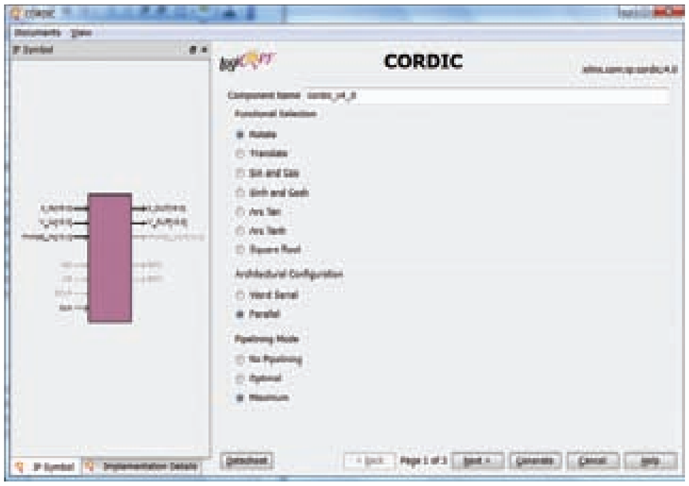
lel) you decide to implement. A general rule of thumb is that for n bits of precision, you will require n iterations or stages. All of this is, however, easily modeled in simple tools such as Excel or MATLAB® prior to cutting code to ensure you will obtain accurate results with the selected iterations.

The CORDIC algorithm as defined will only converge (work) across a limited range of input values. For circular configurations of CORDIC algorithms, convergence is guaranteed for the angles below the sum of the angles in the lookup table—that is, between -99.7 and 99.7 degrees. For angles outside of this, you must use a trigonometric identity to translate an angle within this range. This is also true for convergence within the linear configuration. However, to gain convergence in hyperbolic mode, you must repeat certain iterations (4, 13, 40, K… 3K+1). In this case the maximum input of θ is approximately 1.118 radians.

WHERE IS CORDIC USED?

Designers use CORDIC algorithms in a wide range of applications, from digital signal processing and image processing to industrial control. The most basic way of using a CORDIC is to combine it with a phase accumulator and generate sine and cosine waves for use in I and Q modulation. Using the algorithm to generate these waveforms can, if correctly done, result in a high spurious-free dynamic range. Good SFDR performance is required for most signal-processing applications.

Within the field of robotics, CORDICs are used within kinematics , where they are useful for determining the position and movement of robotic joints and limbs. In this application, you can use a circular CORDIC in vectoring mode to easily add coordinate values with new coordinate values. Within the field of image processing, three-dimensional operations such as lighting and vector rotation are the perfect candidates for implementation using the algorithm.



Functional Selection
[✓ Rotate] Translate Sin and Cos Sinh and Cosh Arc Tan Arc Tanh Square Root

Architectural Configuration
Word Serial [✓ Parallel]

Pipelining Mode
No Pipelining Optimal [✓ Maximum]

Figure 1 – CORE Generator allows simple implementations of many popular CORDIC configurations.

MODELING IN EXCEL

One of the most straightforward methods of modeling your CORDIC algorithm before cutting code is to put together a simple Excel spreadsheet. It allows you to model the number of iterations and gain (A_n) initially using a floating-point number system and later in a scaled, fixed-point system, providing a reference for verification of the code during simulation.

As you can see from the Excel implementation in Table 4, the initial X input is set to A_n to reduce the need for postprocessing the result. The initial argument is set in Z, which is defined in radians, as are the results.

IMPLEMENTING YOUR CORDIC

Unless there is a good reason not to, the simplest method of implementing a CORDIC algorithm within an FPGA is to utilize a tool such as Xilinx® CORE Generator®. CORE Generator provides a comprehensive interface that allows you to define the exact functionality of the CORDIC (rotate, vector, etc.), as seen in Figure 1.

Unfortunately, the CORE Generator does not provide options for working with CORDICs within the linear mode (the tool does provide separate cores to perform these functions). However, you can write the VHDL code required to implement the algorithm in very few lines, as the simple example of a circular implementation below shows.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY synth_cordic IS PORT(

clk      :  IN   std_logic;
resetn   :  IN   std_logic;
z_ip    :  IN   std_logic_vector(16 DOWNTO
0); --1,16
x_ip : IN  std_logic_vector(16 DOWNTO 0);
--1,16
y_ip : IN  std_logic_vector(16 DOWNTO 0);
--1,16
cos_op : OUT std_logic_vector(16 DOWNTO
0); --1,16
sin_op : OUT std_logic_vector(16 DOWNTO
0)); --1,16
END ENTITY synth_cordic;

ARCHITECTURE rtl OF synth_cordic IS

TYPE signed_array IS ARRAY (natural RANGE
<> ) OF signed(17 DOWNTO 0);

--ARCTAN Array format 1,16 in radians
CONSTANT tan_array : signed_array(0 TO 16)
:= (to_signed(51471,18),
to_signed(30385,18),
to_signed(16054,18),to_signed(8149,18),
to_signed(4090,18), to_signed(2047,18),
to_signed(1023,18), to_signed(511,18),
to_signed(255,18), to_signed(127,18),
to_signed(63,18), to_signed(31,18),
to_signed(15,18),
to_signed(7,18),to_signed(3,18),
to_signed(1,18), to_signed(0, 18));

SIGNAL x_array : signed_array(0 TO 14) :=
(OTHERS => (OTHERS =>'0'));
SIGNAL y_array : signed_array(0 TO 14) :=
(OTHERS => (OTHERS =>'0'));
SIGNAL z_array : signed_array(0 TO 14) :=
(OTHERS => (OTHERS =>'0'));

BEGIN

--convert inputs into signed format

PROCESS(resetn, clk)
  BEGIN
    IF resetn = '0' THEN
      x_array <= (OTHERS => (OTHERS =>
'0'));
      z_array <= (OTHERS => (OTHERS =>
'0'));
      y_array <= (OTHERS => (OTHERS =>
'0'));
    ELSIF rising_edge(clk) THEN
      IF signed(z_ip)< to_signed(0,18)
THEN
        x_array(x_array'low) <=
signed(x_ip) + signed('0' & y_ip);
        y_array(y_array'low) <=
signed(y_ip) - signed('0' & x_ip);
        z_array(z_array'low) <=
signed(z_ip) + tan_array(0);
      ELSE
        x_array(x_array'low) <=
signed(x_ip) - signed('0' & y_ip);
        y_array(y_array'low) <=
signed(y_ip) + signed('0' & x_ip);
        z_array(z_array'low) <=
signed(z_ip) - tan_array(0);
      END IF;
```

```
        FOR i IN 1 TO 14 LOOP
          IF z_array(i-1) < to_signed(0,17)
THEN
            x_array(i) <= x_array(i-1) +
(y_array(i-1)/2**i);
            y_array(i) <= y_array(i-1) -
(x_array(i-1)/2**i);
            z_array(i) <= z_array(i-1) +
tan_array(i);
          ELSE
            x_array(i) <= x_array(i-1) -
(y_array(i-1)/2**i);
            y_array(i) <= y_array(i-1) +
(x_array(i-1)/2**i);
            z_array(i) <= z_array(i-1) -
tan_array(i);
          END IF;
        END LOOP;
      END IF;
    END PROCESS;
cos_op <=
std_logic_vector(x_array(x_array'high)(16
DOWNTO 0));
sin_op <=
std_logic_vector(y_array(y_array'high)(16
DOWNTO 0));

END ARCHITECTURE rtl;
```

There are two basic topologies for implementing a CORDIC in an FPGA: either a state machine-based approach or a pipelined approach.

If the processing time is not critical, implement the algorithm as a state machine that computes one CORDIC iteration per cycle until the desired number of cycles has been completed. If you require a high calculation speed, then a parallel architecture is more appropriate. The code above implements a 15-stage parallel CORDIC operating within the rotation mode. It uses a simple lookup table of ArcTan(2-i) as demonstrated in Table 2 for a circular configuration, coupled with a simple array structure to implement the parallel stages.

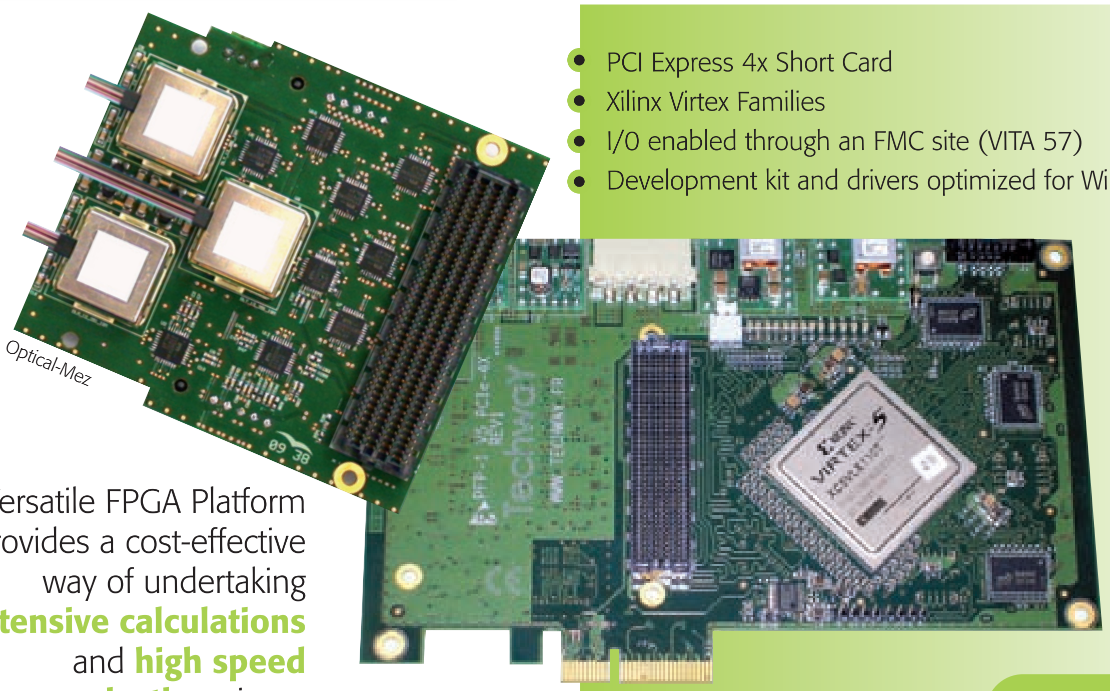
In all of these ways, the CORDIC algorithm proves its merit as a simple but powerful algorithm that all FPGA designers should be aware of. Even better, you can implement the CORDIC very simply using either core-generation tools or hand coding.

TechwaY

The way of innovation

Versatile FPGA Platform

- PCI Express 4x Short Card
- Xilinx Virtex Families
- I/O enabled through an FMC site (VITA 57)
- Development kit and drivers optimized for Windows and Linux



The Versatile FPGA Platform provides a cost-effective way of undertaking **intensive calculations** and **high speed communications** in an industrial environment.

www.techway.eu

Using Formal Verification for HW/SW Co-verification of an FPGA IP Core

A new formal-verification technique allowed a group of academic and industry researchers to holistically verify tightly coupled hardware and firmware within a Xilinx soft core.

by Markus Wedler
Prof. Dr.-Ing.
Berlin Institute of Technology
markus.wedler@tu-berlin.de

Eric Crabill
IP Design Engineer
Xilinx, Inc.
eric.crabill@xilinx.com

Graham Schelle
Senior Staff Research Engineer
Xilinx, Inc.
graham.schelle@xilinx.com

Patrick Lysaght
Senior Director
Xilinx, Inc.
patrick.lysaght@xilinx.com

An ever-growing number of products—from mobile devices to automobiles to industrial machinery—employ sophisticated processing in which hardware and software function together to perform a range of remarkable tasks. But as these systems become more complex, their designers face growing challenges in verifying that the hardware and software work together properly. Over the past few decades, companies and researchers have created a number of ways to verify the proper functioning of both hardware and software. But the ability to verify that the two will work together as intended is very challenging.

For well over a decade, one of the most promising technologies design teams have employed to verify the correctness of their hardware has been a technique called formal verification. Intel, for example, recently revealed that this was the methodology its engineers adopted after rectifying the costly floating-point unit problem it had with its Pentium I microprocessor in the 1990s. [1] Formal verification has since gained adherents at Intel and many other hardware design groups, although it remains something of a niche methodology. But formal techniques for HW/SW co-verification are still not widely used in industry.

OneSpin Solutions, the University of Kaiserslautern and Xilinx researchers recently undertook an investigation into how to apply formal techniques to the holistic verification of a Xilinx® soft IP core that contains both embedded firmware and hardware components. We found that it was possible to capture the firmware and hardware interaction in a scalable formal-verification environment. The work, which arose from a joint collaboration between industry and academia, exploits recently reported advances in the formal verification of hardware-dependent firmware and is based on a form of bounded model checking called interval property checking (IPC).

MODEL CHECKING AND IPC

Formal verification uses mathematical techniques to ensure that a design conforms to some rigorous specification of functional correctness. It starts with a model of a design, a description of the environment in which the system operates and a set of properties the design is expected to meet. It proceeds to verify whether the properties hold in all cases or whether there are conditions that may cause them to fail. Increasingly, designers use formal verification and more conventional simulation-based verification techniques in tandem, exploiting the best features of both. Equivalence checking and assertions, for example, have their origins in formal verification but are widely integrated in verification flows that are simulation based.

Model checking is the leading technique for automatic formal verification of systems that can be represented as finite state machines. A system specification is captured as a suite of properties in a temporal logic. Each property specifies valid (or invalid) logical behaviors pertaining to sets of system states over time. For example, the RESET property may assert that when the RESET signal is active, the system will return to the RESET state, irrespective of what state it has transitioned to over time. These properties may be expressed in familiar language formats such as SystemVerilog assertions, or SVAs (see sidebar, "Inside an Interval Property").

The suite of properties forms an abstract model of the system specification. The model-checking software processes each of the properties, exhaustively traversing all the reachable states of the design in order to verify the properties. When a property fails, the model checker returns a counterexample that represents a trace from reset for which the property is not satisfied.

Model checking has the benefits of being automatic and executing relatively fast. Unlike simulation-based verification, it exhaustively tests the system model with the result that it frequently exposes hard-to-find bugs. Sometimes these bugs arise in obscure corner cases; sometimes the verification engineer who created the stimulus test patterns and assertions for the simulation testbench simply overlooked the conditions under which the bugs arise.

Modern systems with large numbers of state variables frequently exhibit the so-called "state-space explosion" problem, in which the number of reachable states grows exponentially. The complexity of their state-spaces can easily exceed the capacity of conventional model checkers, thereby limiting the practical utility of the technique. This restriction has prompted the development of newer forms of model checking, including interval property checking, the methodology we used in this work. [2]

IPC is a specialized example of what are known as bounded model checkers. In common with other bounded model checkers, IPC restricts the scope of properties to a finite number of clock cycles to limit overall complexity and uses Boolean satisfiability (SAT) solvers to perform the actual model checking. IPC differs from conventional bounded model checkers by allowing the window of clock cycles over which a property may be asserted to start at an arbitrary point in time.

IPC also introduces an intuitive approach to overcoming the state-space explosion challenge. The IPC methodology exploits design abstraction to guide the user in creating a con-

INSIDE AN INTERVAL PROPERTY

The following example shows the pseudocode of a simple SVA-style property that describes the operation of an ADD instruction in a processor. As you can see, at time 0, an ADD instruction and its two operands are fetched, and two cycles later, the register is correctly updated with the sum of the two operands.

```
Property
ADD_INSTRUCTION;
 t##0 ready_to_issue_instr() and
 t##0 decode(ADD,op1,op2,res)
implies
 t##1  pc_updated() and
 t##2  reg(res)=reg(op1)+reg(op2)
endproperty
```

ceptual state machine (CSM) representing the key functionality of the design under verification. The CSM captures all the essential operations or transactions within a given design. At the highest level of abstraction, each transition between states in the CSM represents a single atomic action that is covered by a unique property. In reality, because the CSM is an abstract view of the design from which it is extracted, each CSM transaction can correspond to as many as a couple of hundred clock cycles of related signal activity in the corresponding RTL code. A well-formed property captures all of the most relevant state, as well as input and output signal behaviors at the cycle-accurate level for the complete interval it abstracts. Each property specifies the expected behaviors within a multicycle interval corresponding to exactly one transition or operation of the CSM. Hence the methodology is known as interval property checking or, equivalently, operation-style interval property checking.

The CSM abstraction deliberately does not attempt to capture every detail of the underlying RTL design. Instead, the methodology reduces the state-space and state-transition complexity by elaborating only the subset of control states that is important for capturing complete system behavior. In this way, IPC is different from standard assertion-based verification, where a designer adds local signal-level assertions to the RTL code and checks them using simulation or with formal techniques whenever possible. Very often such assertions check preconditions that the designer has inferred when implementing the corresponding portion of the RTL code. It can be unclear how these local assertions relate to the specification of the IP block. Additionally, these hand-generated assertions are sprinkled throughout the code base, making it difficult to reason about the extent to which the assertions cover the entire design functionality.

Next, all viable state transitions are automatically and exhaustively explored and their corresponding properties are verified. Any failures are highlighted along with details of the counterexamples that triggered the failures. Operation-style IPC with the OneSpin 360 MV formal-verification tool also allows for an automatic proof-of-completeness analysis. This is accomplished with the same state-of-the-art property-checking technology (based on efficient solvers for the Boolean satisfiability problem) that is used for proving the CSM properties.

The proof of completeness guarantees that for every possible input sequence, a unique chain of operation properties determines the behavior of the design. The properties in this chain are linked together via their abstract starting and ending states in the CSM, and the input triggers match the respective input trace under consideration. Moreover, the analysis checks that each property individually specifies all relevant output signals within a particular time interval, and that these intervals hook together without any gap in describing output behavior.

In this way, this proof of completeness is a check of the set of properties as a whole. If the check succeeds, then there is no input scenario for which the behavior of the design remains undetermined by the set of properties; thus, the property set captures the complete functionality of the design.

The challenge of applying formal methods to the growing task of verifying embedded firmware and its interaction with hardware is an area of active and ongoing research. It's been the norm to separately verify the firmware and hardware components, a process that then requires final and time-consuming full-system integration. Until now, verification engineers have applied interval property checking mainly to the formal verification of hardware subsystems. Recently reported research, however, has outlined ways to extend the methodology to more complete systems consisting of both hardware and firmware. The focus of this work has been to evaluate a unified, formal-verification environment that describes both firmware and hardware and their interactions by applying IPC to a commercially available, soft IP core from Xilinx called the Soft-Error Mitigation (SEM) core (*http://www.xilinx.com/products/intellectual-property/SEM.htm*). The core incorporates register-transfer-level (RTL) logic and a PicoBlaze™ microcontroller.

THE SEM CORE

To better understand the verification task, let's first look at the soft IP core in more detail. This Xilinx soft IP core can detect, classify and correct errors in the configuration memory of Xilinx FPGAs. It can also inject errors for testing purposes.

Modern semiconductor devices are susceptible to disturbances resulting from high levels of radiation. A high-energy neutron, for example, can impact a chip by causing a single-event upset (SEU), resulting in a change of state in a configuration memory cell. To mitigate these effects, the memory cells of the configuration frames in Xilinx FPGAs are protected with single-error correcting, double-error detecting hardware blocks. For certain applications, such as the instrumentation in spacecraft, even more elaborate schemes are necessary to offset the extremely high levels of radiation. The SEM core provides a solution in these cases by monitoring the status of the error-correcting code (ECC) blocks via the FRAME_ECC port within Xilinx FPGAs.

The SEM core uses a Xilinx PicoBlaze microcontroller to monitor the status of the configuration memory cells and to take corrective action as required. Designers can incorporate the SEM core into a design and combine it with additional circuitry of their own to implement higher levels of protection against radiation effects as mandated by the application. Should it detect a configuration memory error, the SEM core can resolve it directly or communicate the error to the larger system design.

The proper operation of the SEM core is vital because its sole purpose is to ensure the correctness of users' FPGA circuits. While Xilinx has fully validated this core, it is a challenging piece of IP to verify for a variety of reasons.

The core communicates with several interfaces including a UART, the previously mentioned FRAME_ECC, the FPGA's Internal Configuration Access Port (ICAP) and a custom error-injection interface. These interfaces, while well understood, are difficult to exercise with every possible combination of input combinations to stress the embedded PicoBlaze microcontroller. Formal verification of the SEM core's functionality requires that we capture the interaction among three things: the PicoBlaze firmware, written in assembly code; the "wrapper" logic; and the external interfaces to system resources described in specification documents.

To accomplish these tasks, we applied interval property checking to verify the correctness of the hardware-dependent software in the SEM core.

USING IPC TO FORMALLY VERIFY FIRMWARE AND HARDWARE

Design teams face a major challenge when it comes to formally verifying low-level firmware such as boot code or drivers for bus protocols in conjunction with the hardware on which they are running. Recently, a team from the Electronic Design Automation Group at the University of Kaiserslautern in Germany has reported on how to extend interval property checking to make it tractable for hardware/software co-verification. [3] The problem that needs solving is how to handle the state-space explosion that arises when working with programs that contain hundreds or even thousands of lines of code.

The team's key insight was to extend operation-style property checking by using interval properties to abstract sets of finite-state sequences, which are represented by many lines of code. The technique thus combines hardware and software events in a single conceptual state machine. The common computational model for the first time permits the representation and debugging of the hardware and software interplay using the IPC methodology. Of course, the method has limitations and is not germane to all classes of software. In particular, code that makes extensive use of recursion falls outside the scope of the current work.

When verifying the SEM core, we used the firmware control flow graph (CFG) to generate property templates. Each transition between basic blocks is treated as a separate property that is triggered either by registers internal to the PicoBlaze microcontroller or by outside events. The functions describing abstract starting and ending states at any given cycle simply depend on the PicoBlaze architectural state and any external stimulus.

In describing the SEM core state at the beginning and the end of an assertion, as required by IPC, we needed to examine both PicoBlaze firmware and the RTL code for the FPGA logic. Figure 1 shows the combination of both firmware and hardware state. It is important to note that the PicoBlaze microcontroller is truly implementing a software finite state machine and its behavior is directly tied to the wrapper hardware. Verifying that firmware in isolation, while possible, would require a hardware bus-functional model (BFM) interface—in essence creating yet another behavioral testbench. Extending IPC, however, makes it possible to model full-system behavior, including hardware and firmware, within a single verification environment. We could have run and compiled the firmware within an instruction-set simulator, but we would never fully capture the hardware and firmware system behavior in one environment. In contrast, the use of IPC gives us a single hardware/firmware verification environment.

In constructing the properties, you can restrict their complexity iteratively

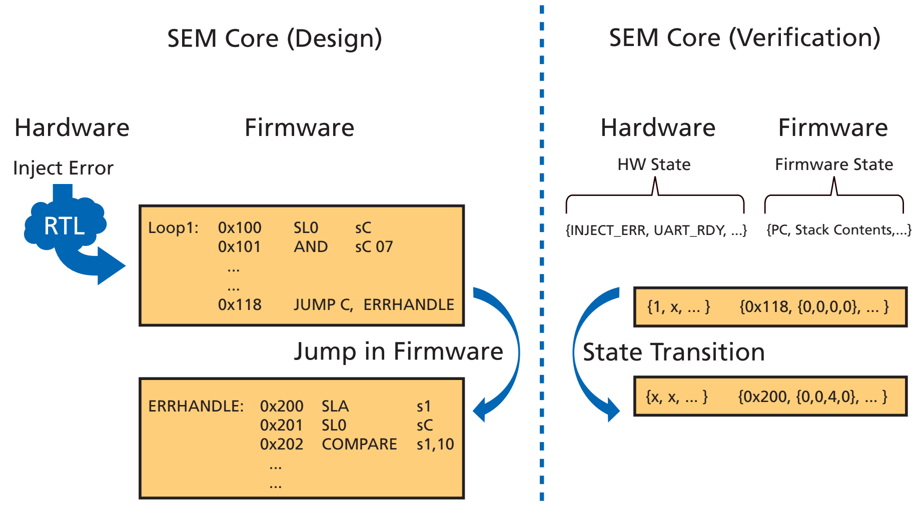


Figure 1 – An example translation of hardware and firmware state of the SEM core to a combined verification state within the OneSpin 360 MV tool

The PicoBlaze microarchitecture has several characteristics that simplify its integration into a formal-verification tool flow. Instructions execute in order, and every instruction takes two cycles to execute. Also, there is a finite stack depth, a feature that simplifies proofs.

to establish an acceptable trade-off between the complexity of any given property and its evaluation time in the OneSpin 360 MV tool. For this work, we found that keeping the property interval under 100 cycles made it possible to prove properties in 30 minutes.

There are further design factors within the SEM core that contribute to keeping the property complexity low. Aspects of the SEM core firmware and the PicoBlaze microarchitecture, and how their implementations simplify property creation, are pertinent to the discussion.

First, we can exploit some characteristics of the SEM core's embedded firmware:

- This firmware image implements a software finite state machine, which has several characteristics beneficial to formally describing the state of the processor. In particular, the firmware will neither dynamically allocate memory nor call unbounded recursions. This is typical of low-level, embedded software in general. These two aspects of the firmware greatly simplify modeling the processor state and its memory.
- An explicit memory map for global and local variables is available to the verification engineer. All external variables in the wrapper RTL that's used to trigger state transitions in the firmware are available inside the SEM core for exploration with the OneSpin 360 MV tool.
- Finally, the firmware machine code is stored in a ROM within the SEM core. Since the ROM is visible to the verification process, there is no need to completely verify any program that may run on the PicoBlaze microcontroller, but rather only the actual program stored in the ROM. In other words, we are not re-verifying that the field-proven PicoBlaze microcontroller works for every piece of firmware. Instead, we can strictly focus on the interplay of the PicoBlaze microcontroller with the SEM core's firmware and wrapper RTL.

Second, we are able to exploit aspects of the PicoBlaze microarchitecture in order to describe the state of the firmware image. The PicoBlaze microarchitecture has several characteristics that simplify its integration into a formal-verification tool flow:

- Instructions execute in order. With no interrupts in the SEM core, we can know exactly when an instruction will start in relation to other instructions in the firmware.
- Every instruction takes two cycles to decode/execute. With this fixed latency on the operations within the SEM core firmware, we can create properties that encapsulate multiple instructions, knowing these instructions will execute with a deterministic aggregate latency.
- The PicoBlaze microarchitecture has a finite stack depth. The stack content is a key part of the SEM core state, but with a finite depth, that state will not grow past a set depth. Since the complexity of proving properties rises with increasing amounts of state, a set depth within the stack simplifies proofs.

When describing the state of the processor at some cycle, it is relatively simple to describe this amount of architectural state. This manageable state description feeds directly into the OneSpin 360 MV's property-generation tools.

With these simplifying factors in place, we must describe the relevant state transitions. We can do this directly by mapping the basic blocks of the firmware as individual design states. By definition, a basic block consists of a linear sequence of instructions. Each conditional jump or conditional function call determines the end of a basic block and two potential successor blocks. The destination addresses of these instructions mark starting points of new basic blocks. The control flow graph consists of the basic blocks and their successor relationship. Each edge in the graph corresponds to a conditional branch of the firmware and is labeled with a branching condition.

If the firmware is implemented using a high-level language, a compiler may generate the CFG automatically. However, with the help of a (symbolic) instruction-set simulator, you may likewise extract the CFG from the assembly code. This technique would also allow for co-verification of legacy platforms where only assembly code may be available.

When verifying the SEM core, we used the firmware CFG to generate property templates. Each transition between basic blocks was treated as a separate property triggered by either registers internal to the PicoBlaze microcontroller or by outside events. The functions describing abstract starting and ending states at any given cycle simply depend on the PicoBlaze architectural state and any external stimulus.

As part of that mapping of external stimulus to architectural state, it proved necessary to specify the branching condition of every conditional jump or function call. As we were interested in the overall behavior of the design, we specified this condition in terms of external input events or important state registers of the wrapper circuit rather than local status registers of the embedded processor. The registers of the wrapper that are evaluated in trigger conditions become part of the abstract state definitions.

With simplifying conditions in both the SEM core firmware and the PicoBlaze microcontroller itself, we could define all state transitions of the firmware in terms of processor state and external stimulus. This state included firmware and hardware that relate the overall behavior of the design to an abstract conceptual finite state machine within the OneSpin 360 MV tool.

IPC'S EUROPEAN ROOTS

Operation-style interval property checking originated from industrial research at Infineon Technologies in Europe. The core technology was spun out and commercialized by OneSpin Solutions, which continues to develop and refine it in addition to providing a comprehensive CAD environment called 360 MV to make the methodology more accessible to nonspecialists. Since the inception of IPC, researchers at the University of Kaiserslautern, Germany, have collaborated with Infineon Technologies and OneSpin Solutions to further develop the underlying theory of IPC and to broaden its applicability.

PROPERTY GENERATION

We found that it was possible to describe the SEM core's firmware and hardware in 1,700 properties that captured end-to-end functionality of the design and its interfaces. We used the OneSpin 360 MV to check the properties and to explore the completeness of the entire property set. The properties could be proven in parallel, and the longest property took approximately 30 minutes to prove within a server cluster.

At a first glance a total of 1,700 properties seems to be a large number. Most of these properties (approximately 1,500), however, dealt only with wait loops for the sequential UART that is used for dumping state information out of the controller, mainly for debugging purposes. Remember, each property is a basic software block ending with a conditional jump; therefore, each UART wait loop creates a unique property within our formal model. For a complete verification of the design, we proved that unexpected side effects during these wait loops did not corrupt the abstract state content.

In total, the generated properties cover the complete control flow of the firmware. They describe the input/output behavior of the design during execution of the corresponding firmware basic blocks. For this case study, these properties complement existing verification testbenches, giving extremely high confidence on the core's functionality.

The generated properties revealed a scenario whereby the SEM core's error-injection test feature could inject an error into an unintended configuration memory location. This situation would happen only if the error-injection port was exercised in a manner that was contrary to the recommendations of the SEM core product documentation. Although this scenario could never occur under normal operating conditions, Xilinx nonetheless updated the SEM core to eliminate this possibility completely.

COMMITMENT TO QUALITY

In our investigation, we demonstrated the use of interval property checking to formally verify the tightly coupled hardware and firmware of the Xilinx SEM core. Within a single verification environment, the SEM core was holistically verified using 1,700 properties that were proven in parallel. This work takes advantage of state-of-the-art techniques and tools in formal verification. The results have provided increased confidence in the functional correctness of the SEM core and demonstrate Xilinx's continued commitment to quality IP distribution.

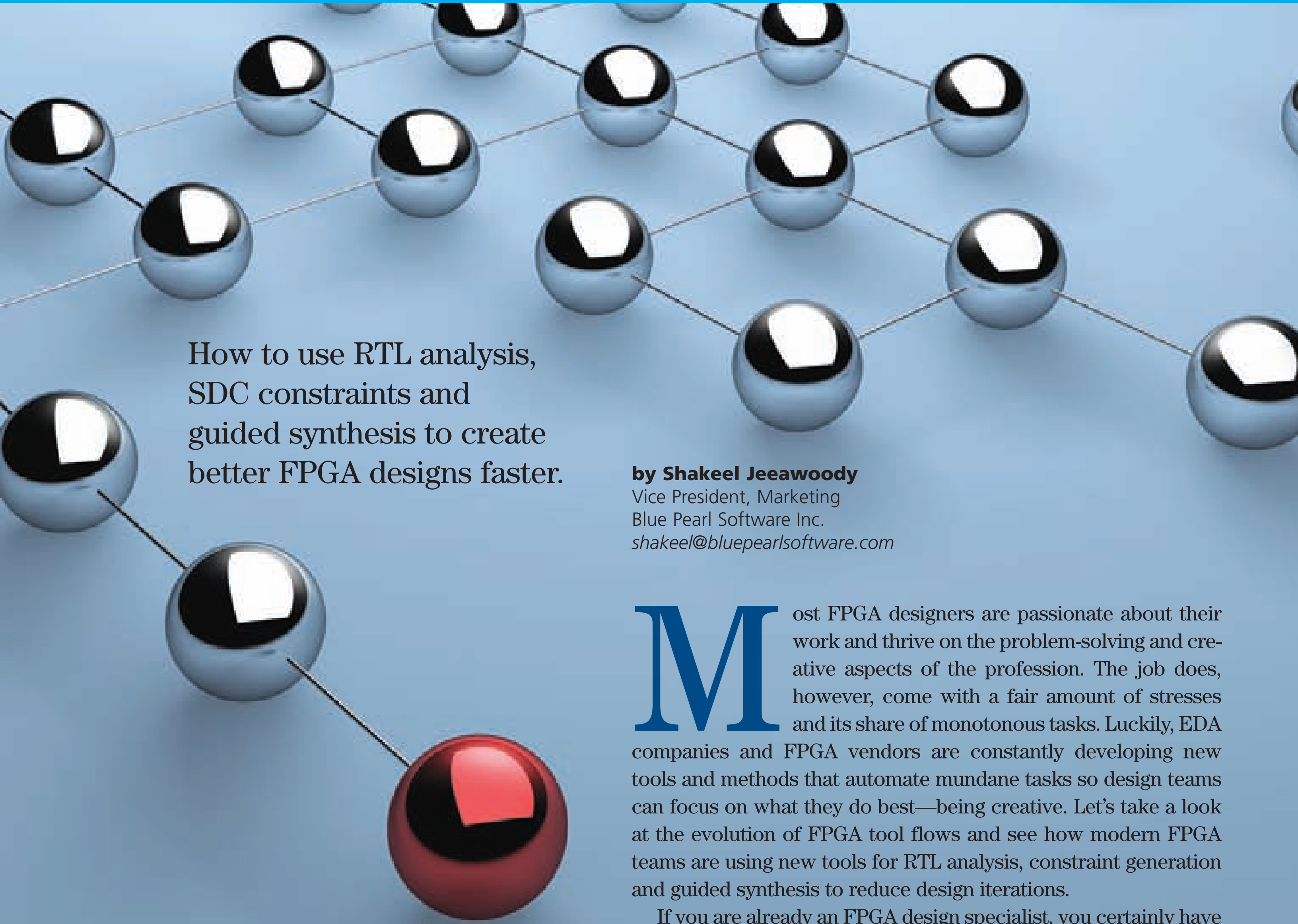
References

1. *Jones, R.B.; O'Leary, J.W.; Seger, C.-J.H.; Aagaard, M.D.; Melham, T.F.; "Practical formal verification in microprocessor design,"* Design & Test of Computers, *IEEE, vol. 18, no. 4, pp. 16-25, July/August 2001*

2. *Nguyen, M.D.; Thalmaier, M.; Wedler, M.; Bormann, J.; Stoffel, D.; Kunz, W.; "Unbounded protocol compliance verification using interval property checking with invariants,"* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, *vol. 27, no. 11, pp. 2068-2082, November 2008*

3. *Nguyen, M.D.; Wedler, M.; Stoffel, D., and Kunz, W.; "Formal hardware/software co-verification by interval property checking with abstraction," Design Automation Conference (DAC), 48th ACM/EDAC/IEEE, pp. 510-515, June 5-9, 2011*

New Tools Take the Pain out of FPGA Synthesis



How to use RTL analysis, SDC constraints and guided synthesis to create better FPGA designs faster.

by Shakeel Jeeawoody
Vice President, Marketing
Blue Pearl Software Inc.
shakeel@bluepearlsoftware.com

Most FPGA designers are passionate about their work and thrive on the problem-solving and creative aspects of the profession. The job does, however, come with a fair amount of stresses and its share of monotonous tasks. Luckily, EDA companies and FPGA vendors are constantly developing new tools and methods that automate mundane tasks so design teams can focus on what they do best—being creative. Let's take a look at the evolution of FPGA tool flows and see how modern FPGA teams are using new tools for RTL analysis, constraint generation and guided synthesis to reduce design iterations.

If you are already an FPGA design specialist, you certainly have a bright future ahead of you, because an increasing number of designs that would have typically been implemented in an ASIC are now being implemented on FPGAs. Designing ASICs is becoming exponentially more expensive with each introduction of a new silicon process technology. Meanwhile, FPGA vendors implement each new generation of their devices on these new process technologies but do not burden customers with exorbitant costs.

The bad news, however, is that FPGA designs can be so complex that they require tool flows as advanced as ASIC flows and often demand the efforts of a design team rather than a single designer. As such, FPGA design teams need to seriously analyze their current tool sets as they embark on ECOs or new projects. The good news? There are numerous new-generation EDA tools available to help them. Designers can choose easy-to-install and easy-to-use products that employ standard data formats, which are simpler to integrate in a flow and run natively on their platform of choice, whether Windows or Linux.

THE EVOLUTION OF FPGA TOOL FLOWS

As FPGA designs have become more complicated over the years, the tool flows have evolved accordingly and become more ASIC-like. In the 1990s, FPGA flows (see Flow A in Figure 1) became RTL based and used synthesis and place-and-route tools, just like simple ASIC flows of the time. As designs got more complex, FPGA teams added timing analysis to their flows to help customers ensure that designs could perform at the required frequency. Today's FPGAs have become giant system platforms, so now it's commonplace for design teams to employ RTL analysis to minimize design iterations and ensure their designs achieve performance goals.

Further, because today's FPGA design projects can be so large and complex, designers need ways to better understand the scope and complexity of their design and to better control the tools in their flow to get their designs to market quickly. An emerging way that modern FPGA design teams do this is by using constraints throughout their design flows. Let's take a look at one of the most popular constraint methodologies, the Synopsys Design Constraint (SDC) format, now supported in the new Xilinx® Vivado™ flow, and at how you can use SDC to benefit your design projects.

WHAT IS SDC?

SDC is a Tcl-based format used to specify design intent, including the timing, power and area constraints for a design. There are several products that either read or write SDCs. Some sample SDC constraints include timing constraints, such as create_clock, create_generated_clock, set_input_delay and set_output_delay; and timing exceptions, such as set_false_path, set_max_delay, set_min_delay and set_multicycle_path. These SDC constraints are typically applied to design objects like registers, clocks, ports, pins and nets.

It must be noted that even though SDC is a standard format, there are slight variations (among different tools) between generated and read SDCs. Understanding these minor variations and dealing with them in a timely manner helps to avoid surprises.

SDC SHOULD NOT IMPLY PAIN

One of the most popular uses of SDCs is to constrain synthesis. In general, designers have a feel for what areas of their design need to be constrained and thus start by writing SDCs for them. They would typically execute the flow described in Flow B and inevitably not close timing the first time around. It then becomes an iterative, manual, shooting-in-the-dark process of adding SDCs to close timing or get the design to work at the desired frequency. Many designers who have executed this process complain about spending

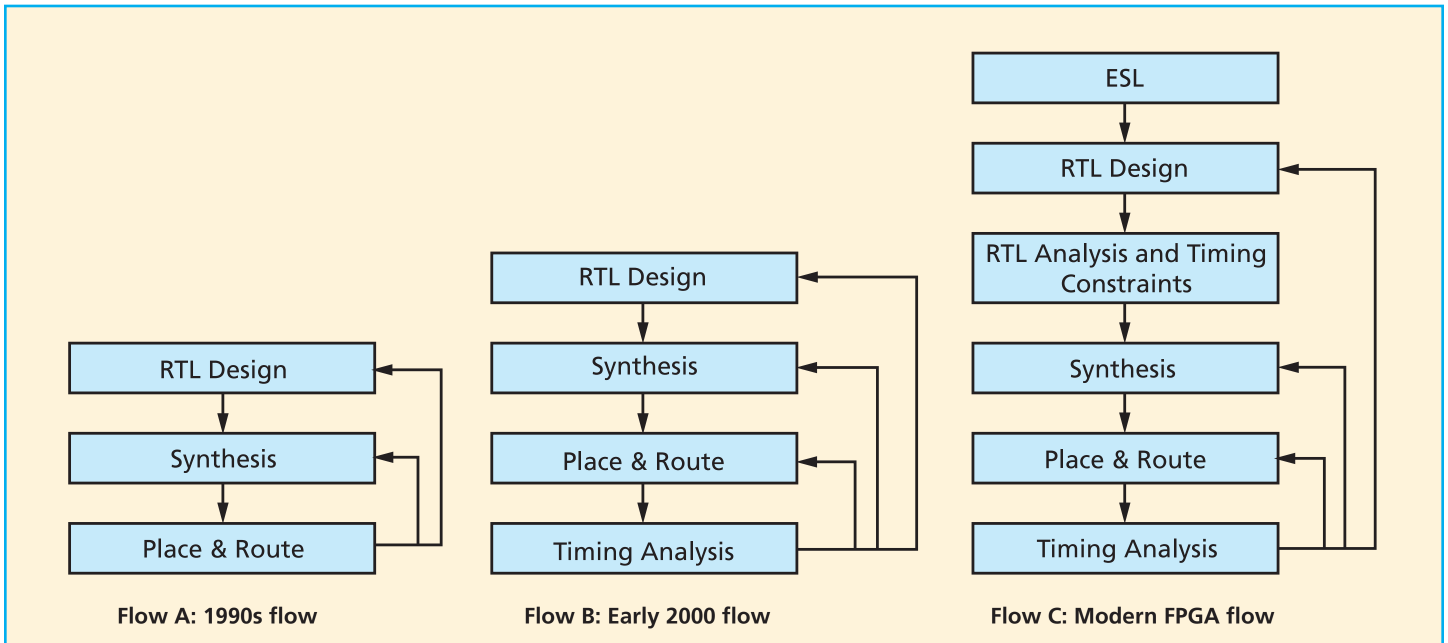


Figure 1 – FPGA tool flows have evolved over time to become more ASIC-like.

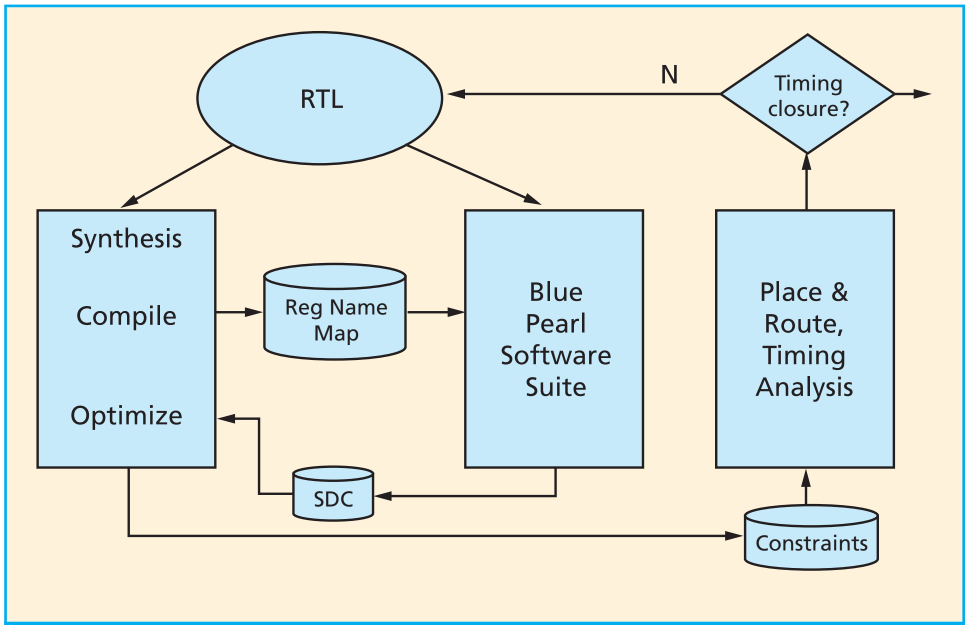
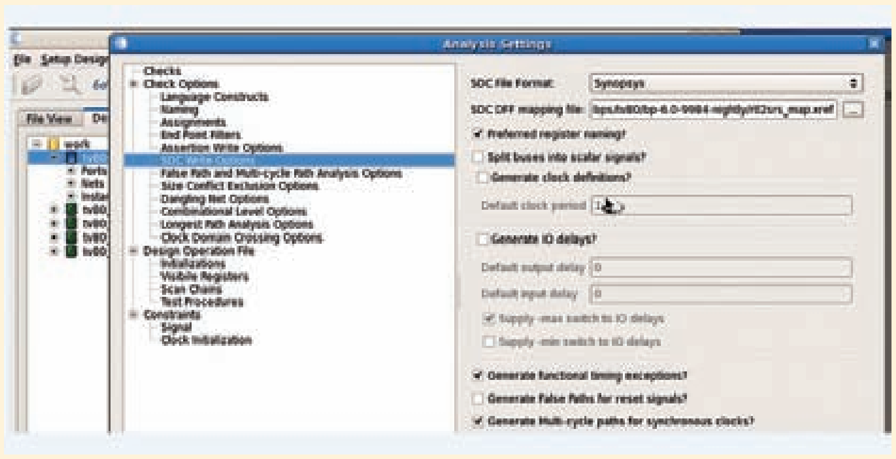


Figure 2 – How the Blue Pearl Software Suite and Vivado work together



SDC File Format: Synopsys

SDC DFF mapping file: bps/tv80/bp-6.0-9984-nightly/rtl2srs_map.xref

[✓] Preferred register naming?

[ ] Split buses into scalar signals?

[ ] Generate clock definitions?

Default clock period: 1

[ ] Generate IO delays?

Default output delay: 0

Default input delay: 0

[✓] Supply -max switch to IO delays

[ ] Supply -min switch to IO delays

[✓] Generate functional timing exceptions?

[ ] Generate False Paths for reset signals?

[✓] Generate Multi-cycle paths for synchronous clocks?

Figure 3 – The Blue Pearl software simplifies formatting issues.

weeks iterating the design, often missing schedules in the process.

Another issue associated with the iterations is that several designers working on different blocks and possibly at different locations are contributing to the SDCs. This process more often than not becomes so complex that design teams need a methodology to verify the SDCs and remove conflicts in hierarchical names or pins as they assemble the design at the chip level. It is very important to have the right tools and methodologies for collaborative design to be effective.

The modern flow described in Flow C, which uses analysis, SDC constraints and high-level synthesis in addition to the tools in Flow B, makes great strides at solving both of these issues.

GUIDING SYNTHESIS

For a typical FPGA design, the synthesis solution space, being heuristic, has multiple local maxima and minima, irrespective of whether we are optimizing for area, speed or power. By using smart guidance, we can achieve an optimal solution rather than having the synthesis tool converge on an arbitrary local minimum. One of the most effective guidance techniques includes the usage of false paths and multicycle paths that keep the synthesis tool from spending precious optimization time on unnecessary elements.

However, finding all the false paths (FPs) and multicycle paths (MCPs) in a design is not a trivial task. You'll find some simple FPs and MCPs if you spend enough time looking for them, but complex ones involving state machines and counters (especially within multiple levels of hierarchy) are nearly impossible to ferret out. Fortunately for FPGA designers, innovative companies like Blue Pearl Software have tools to perform automatic FP and MCP generation that is complete, comprehensive and accurate. Furthermore, these tools provide different mechanisms for each FP and MCP—including schematic, assertion and audit trail—by which the user may verify its correctness.

Because FPGA and commercial EDA vendors have been increasingly working together and using common interfaces, design teams can integrate the Blue Pearl Software Suite into the flow they like the best. Since Xilinx's new Vivado Design Suite supports SDC, it is very simple to communicate design intent between the tools (Figure 2).

In addition to working with Xilinx and other FPGA vendors, Blue Pearl also closely collaborated with Synopsys. The two companies came together to investigate what needs to be done so that the synthesis tool can accept a maximum number of the automatically generated SDCs, without forcing the designer to perform any manual changes. Due to minor differences in different tools' usage of the SDC format, the team quickly identified naming convention as a major challenge for smooth interoperability.

The Blue Pearl Software Suite delivers a 20 percent QoR improvement in our sample design, which contains multiple IP cores including an R1200 in Verilog and AES encryption in VHDL.

The solution involved intercepting the name mapping that happens after the first phase (compilation) of synthesis, to use the names (see Figure 3) within the Blue Pearl Software Suite SDC generation tool, and then providing the proper SDC to the second phase (optimization) of the synthesis tool. This approach gives the FPGA designer an optimized solution without having to waste time with formatting issues.

For example, a nonoptimized constraint that reads:

```
set_false_path -from
[get_cells
{i_tv80_core.SP[*]}] -to
[get_cells
{i_tv80_core.i_reg.RegsL}]
```

might be optimized to read:

```
set_false_path -from
[get_cells
{i_tv80_core.SP[*]}] -to
[get_cells
{i_tv80_core.i_reg.RegsL_2[
7:0]}]
```

WHAT ABOUT THE TANGIBLE BENEFITS?

Even though the Blue Pearl Software Suite automates several tasks, designers will be pleased with its quality of results (QoR). Table 1 shows that using the automatically generated SDCs from the Blue Pearl Software Suite delivers a 20 percent QoR improvement for this sample design, which contains multiple IP cores including an R1200 in Verilog and AES encryption in VHDL.

Run 1, without the Blue Pearl software, did not achieve timing closure; the designer could easily spend weeks iterating through the RTL design or tool constraints to meet the 60-MHz requirement. In Run 2, the Blue Pearl Software Suite generated the SDCs in minutes and the automatically generated SDCs were sufficient to guide the downstream tools to meet timing.

Clearly, for an FPGA designer, one way to reduce stress and simplify your life is to learn from others and add RTL analysis, SDC generation and guided synthesis tools to your tool box. For more information, visit *www.bluepearlsoftware.com.*

	Run 1	Run 2
Flow	Like **Flow B** (see Figure 1)	Like **Flow C** (see Figure 1)
Tools involved	**Synopsys:** Synplify Pro **Xilinx:** Virtex-5, XC5VLX50, FF1153,-1	**Synopsys:** Synplify Pro **Xilinx:** Virtex-5, XC5VLX50, FF1153,-1 **Blue Pearl Software:** Release 6.0
Design frequency	Set to **60 MHz** globally from Synplify Pro	Set to **60 MHz** globally from Synplify Pro
Blue Pearl SDC	**NO**	**YES**
Setup violation (after Place & Route)	**-3.57 ns**	**NONE**

Table 1 -- Comparing two runs shows an advantage in adding the Blue Pearl suite to the flow.

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 application notes.

XAPP588: VIRTEX-5QV FPGA EXTERNAL CONFIGURATION MANAGEMENT

http://www.xilinx.com/support/documentation/application_notes/xapp588_v5qv_ex_config_mgmt.pdf

The Xilinx® Virtex®-5QV device is high in logic density and radiation-hardened by design, and contains 12-transistor (12T) configuration memory cells. The result is the fastest and largest FPGA available for high-reliability space applications. This makes the Virtex-5QV FPGA the programmable logic device of choice for orbital applications. This FPGA has an extremely low upset rate of five events per year. However, if needed, designers can implement configuration management to ensure reliable functionality over long operational periods by periodically performing a soft reboot, as Y.C. Yang explains in this application note.

Orbital applications carry the risk that high-energy charged particles will cause a single-event upset (SEU) in a programmable logic device's configuration memory. When an SEU affects memory cells essential to the design, the result can impact the application functionality. Additionally, an SEU in the FPGA control circuitry can cause a single-event functional interrupt (SEFI). Since both of these scenarios are undesirable, designers need a way to resume normal and expected functionality in the form of configuration management.

The configuration management scheme outlined in this application note involves using an external device to monitor and scrub the Virtex-5QV FPGA to mitigate SEU and SEFI effects. This external device—which is assumed to be a space-qualified FPGA or ASIC, although it could also be a microprocessor—performs SEU detection and correction, partial reconfiguration, blind scrubbing, and SEFI detection and mitigation.

The reference design is based on the configuration monitor IP used during radiation testing of the Virtex-5QV device. Because a Virtex-II Pro FPGA is the external configuration manager in the Xilinx test apparatus, Yang implemented the reference design using ISE® 10.1.03 software. Nonetheless, the reference code should be 100 percent portable to later versions of ISE, since it infers design primitives and is synthesizable in XST.

XAPP553: SCALABLE SERDES FRAMER INTERFACE (SFI-S) FOR 7 SERIES FPGAS

http://www.xilinx.com/support/documentation/application_notes/xapp553-scalable-serdes-framer-interface.pdf

The Scalable Serdes Framer Interface (SFI-S) is an Optical Internetworking Forum (OIF) standard that defines the electrical connections between devices on a typical optical-communications line card. An n-bit-wide SFI-S configuration contains n data channels and one control channel for interface skew compensation. This application note by Julian Kain describes a 10-data-channel SFI-S design targeting Xilinx 7 series FPGAs using GTX or GTH serial transceivers to implement an aggregate 111.8-Gbit/second bidirectional interface. The hardware-verified Verilog HDL reference design provides significant skew compensation and fine-grained control of skew tracking, and the Verilog HDL source can be readily modified if necessary. A synthesizable example design with PRBS31 generator and checker logic enables simple simulation and a hardware demonstration of the reference design.

XAPP521: BRIDGING XILINX STREAMING VIDEO INTERFACE WITH THE AXI4-STREAM PROTOCOL

http://www.xilinx.com/support/documentation/application_notes/xapp521_XSVI_AXI4.pdf

The ARM® core AMBA® specification (version 4.0) AXI interconnect standard includes three Advanced Extensible Interface v4 (AXI4) interconnect protocols—AXI4 interconnect, AXI4-Lite and AXI4-Stream. The Xilinx AXI video direct-memory access (VDMA) core comes with an AXI4-Stream interface for video data. Video applications that require the video data to be buffered in memory need an AXI4-Stream interface to connect with the AXI VDMA. Previous versions of Xilinx video IP cores used a protocol called the Xilinx Streaming Video Interface (XSVI) for video data. In this application note, authors Steve Elzinga and Chris Martin tell how to bridge an XSVI to an AXI4-Stream interface, enabling video designs with Xilinx video IP cores and XSVI interfaces to use the AXI VDMA. The authors include an embedded reference design to illustrate the functionality of the bridges described. Designers can use this solution to build video systems when the Xilinx video IP does not contain an AXI4-Stream interface.

XAPP888: MMCM AND PLL DYNAMIC RECONFIGURATION

http://www.xilinx.com/support/documentation/application_notes/xapp888_7Series_DynamicRecon.pdf

This application note provides a method to dynamically change the clock output frequency, phase shift and duty cycle of the Xilinx 7 series FPGA's mixed-mode clock manager (MMCM). Similarly, it shows how to change the phase-locked loop (PLL) through the dynamic reconfiguration port (DRP). After explaining the behavior of the internal DRP control registers, author Jim Tatsukawa offers a reference design that uses a state machine to drive the DRP, which ensures the registers are controlled in the correct sequence. The reference design supports two reconfiguration state addresses, but thanks to its modular nature, designers can extend it to support additional states. Each state does a full reconfiguration of the MMCM or PLL so that most parameters can be changed. The design uses minimal 7 series FPGAs resources, consuming only 27 slices.

XAPP520: INTERFACING 7 SERIES FPGA HIGH-PERFORMANCE I/O BANKS WITH 2.5V AND 3.3V I/O STANDARDS

http://www.xilinx.com/support/documentation/application_notes/xapp520_7Series_HPIO.pdf

The I/Os in Xilinx 7 series FPGAs are classified as either high-range (HR) or high-performance (HP) banks. HR I/O banks can operate from 1.2 to 3.3 volts, whereas HP I/O banks are optimized for operation between 1.2V and 1.8V. This application note describes methodologies for interfacing 7 series HP I/O banks with 2.5V and 3.3V systems. There are different options for interfacing depending on performance needs, function and signal type (input, output or bidirectional). Authors John Rinck and Austin Tavares explore options such as added resistors, field-effect transistor (FET) switches, level translators and even other Xilinx FPGAs. Together, these strategies can accommodate virtually all design, cost and performance needs.

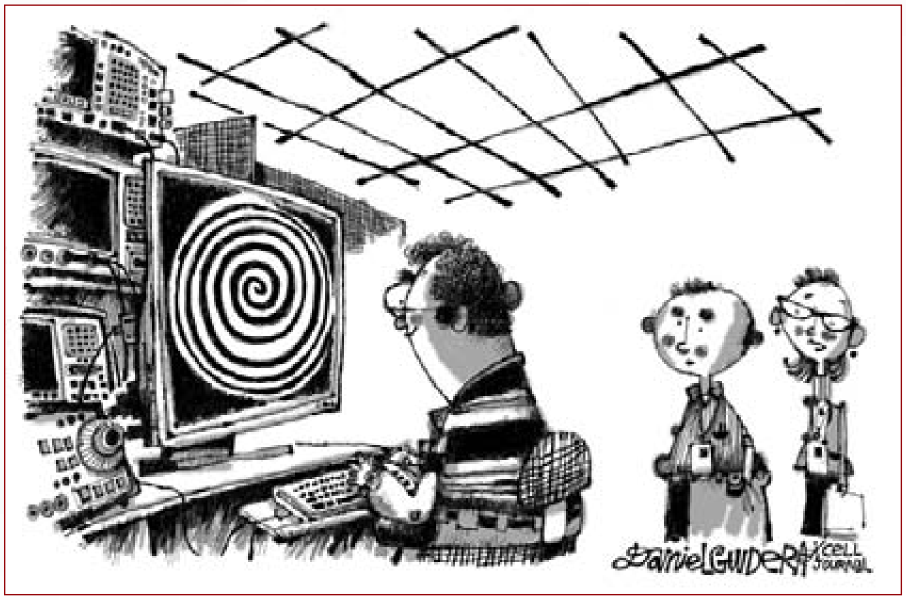
XAPP517: DUAL USE OF ICAP WITH SEM CONTROLLER

http://www.xilinx.com/support/documentation/application_notes/xapp517_DualUse_ICAP_SEM.pdf

This application note by John Ayer Jr. includes a method for sharing an Internal Configuration Access Port (ICAP) between the user design and the soft-error mitigation (SEM) controller in the Spartan®-6 and Virtex-6 devices. There are various reasons why the user might want to do so. The reference design uses an example of reading back the device IDCODE. This methodology also enables customers to create multiboot designs that require access to the ICAP.

This design demonstrates that it is possible to share the ICAP used by the SEM controller with other designated user functions by implementing multiplexers on the ICAP control signals and switching control away from the SEM IP only when the controller is in the idle state. Using the Xilinx ML605 and SP605 reference boards and the ChipScope™ Pro analyzer, the SEM controller can relinquish the ICAP to allow the device IDCODE to be read back, take control again and successfully correct injected errors.

Xpress Yourself in Our Caption Contest



DANIEL GUIDERA

Are you feeling sleepy yet? If you sense a post-hypnotic suggestion to Xercise your funny bone, here's your opportunity. We invite readers to step up to our verbal challenge and submit an engineering- or technology-related caption for this cartoon showing a mesmerizing moment in the lab. The image might inspire a caption like "Jason goes into a trance whenever he's running eye diagrams."

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*. The winner and two runners-up will each receive an Avnet Spartan®-6 LX9 MicroBoard, an entry-level development environment for evaluating the Xilinx® Spartan®-6 family of FPGAs (*http://www.xilinx.com/products/boards-and-kits/AES-S6MB-LX9.htm*).

The deadline for submitting entries is 5:00 pm Pacific Time (PT) on June 25, 2012. So, get a grip, and get writing!

CARL GRECO, professor of electrical engineering at Arkansas Tech University (Russellville, Ark.), won first place with this caption for the cartoon of a giant bird clutching an unwary victim in Issue 78 of *Xcell Journal*:

"OK, now that Henry has been relieved, who would like to take over project leader position?"

Congratulations as well to our two runners-up. Like our winner, they will each receive an Avnet Spartan®-6 LX9 MicroBoard.

"The company accountant demonstrates how you can still fly in for meetings under the new budget."

– Clyde Schlabach, design engineer, Avtec Inc., Gilbert, S.C.

"When Larry said he'd be flying by the seat of his pants during the presentation, I thought he was kidding."

– R. Scott Brandt, design engineer, AuthenTec, Inc., Satellite Beach, Fla.

No Room for Error

Synopsys FPGA Design Tools are an Essential Element for Today's High Reliability Designs

There is no room for error in today's communications infrastructure systems, medical and industrial applications, automotive, military/aerospace designs. These applications require highly reliable operation and high availability in the face of radiation-induced errors that could result in a satellite failing to broadcast, a telecom router shutting down or an automobile failing to respond to a command.

To learn more about Synopsys FPGA Design tools visit www.synopsys.com/fpga

SYNOPSYS®
Predictable Success

2012 X-fest

COMING TO A CITY NEAR YOU!

From April through September 2012, Avnet and Xilinx will co-host X-fest technical training events throughout Asia, Europe, North America and Japan.

WHAT YOU CAN EXPECT

» **A fully customizable training experience – pick up to four courses from any of the three unique training tracks.**

- The **FPGA Fundamentals** track focuses on common design issues facing both new and advanced FPGA designers. Courses include:
  - » Designing with the Xilinx 7 Series FPGAs PCIe Embedded Block
  - » High Performance Clocking with the Xilinx 7 Series FPGAs
  - » Optimal Memory Interface Design with Xilinx 7 Series
  - » Powering Xilinx 7 Series FPGAs
- The **Applications** track delves deeper into design methodologies and techniques required when designing with the Zynq™-7000 Extensible Processing Platform (EPP) and 7 series FPGAs. Courses include:
  - » Designing High Pixel-Rate Video Systems with Xilinx FPGAs
  - » Designing Wireless Communication Systems in Xilinx FPGAs
  - » FPGA-Based Motor Control
  - » Sampling and Processing Real-World Data with Xilinx 7 Series FPGAs
- The **Zynq-7000 EPP** track provides four courses dedicated to the ARM® dual core Cortex™-A9 MPCore™ based device:
  - » ARM® Software Development for Zynq-7000 EPP
  - » Introduction to the Zynq-7000 EPP
  - » Software Acceleration in Zynq-7000 EPP
  - » Zynq-7000 EPP in Action: Exploring the Low Cost ZedBoard Kit

» **Partner demonstrations and exhibits**

- Xilinx
- Avnet
- Analog Devices
- Maxim
- Micron
- TE Connectivity
- Texas Instruments
- Cypress Semiconductor
- Aldec
- ARM
- MathWorks
- Spansion
- FCI
- Samtec
- PLDA

Learn more at
www.em.avnet.com/xfest

XILINX® 2012 X fest AVNET®

DON'T MISS ANY X-FEST UPDATES. JOIN THE DIALOGUE TODAY!

@avnetxfest facebook.com/xfest2012 Join the Avnet X-fest group!

PN 2513