Xcell journal

ISSUE 87, SECOND QUARTER 2014

SOLUTIONS FOR A PROGRAMMABLE WORLD

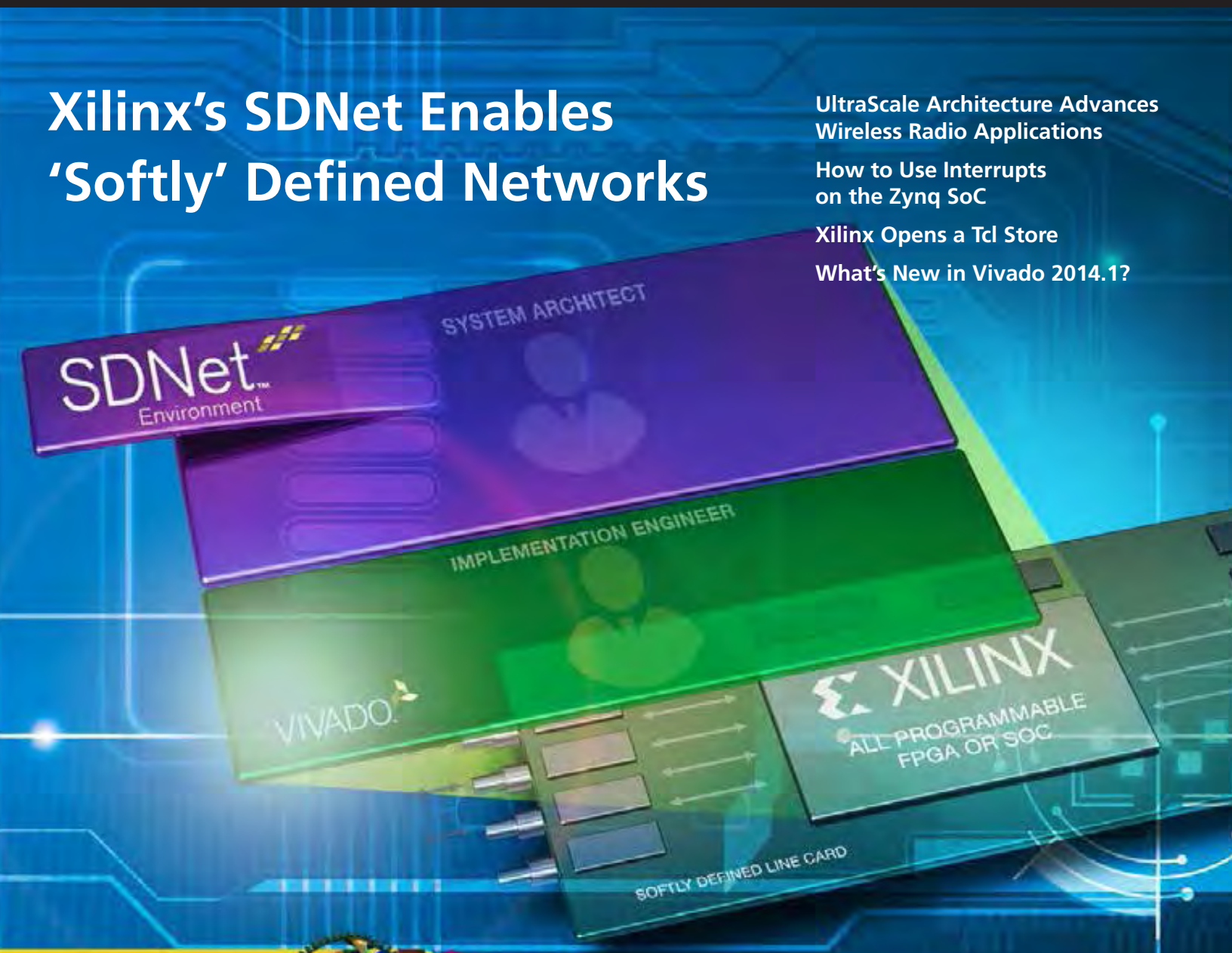
Xilinx's SDNet Enables 'Softly' Defined Networks

UltraScale Architecture Advances Wireless Radio Applications

How to Use Interrupts on the Zynq SoC

Xilinx Opens a Tcl Store

What's New in Vivado 2014.1?



Motor Drives Migrate to Zynq SoC with Help from MATLAB 32

XILINX
ALL PROGRAMMABLE™

www.xilinx.com/xcell

XILINX
ALL PROGRAMMABLE™

AVNET®
electronics marketing

Xilinx® SpeedWay Design Workshops™

Featuring MicroZed™, ZedBoard™ & the Vivado® Design Suite

ZYNQ®

VIVADO™

Avnet Electronics Marketing introduces a new global series of Xilinx® SpeedWay Design Workshops™ for designers of electronic applications based on the Xilinx Zynq®-7000 All Programmable (AP) SoC Architecture. Taught by Avnet technical experts, these one-day workshops combine informative presentations with hands-on labs, featuring the ZedBoard™ and MicroZed™ development platforms. Don't miss this opportunity to gain hands-on experience with development tools and design techniques that can accelerate development of your next design.

em.avnet.com/xilinxspeedways

© Avnet, Inc. 2014. All rights reserved. AVNET is a registered trademark of Avnet, Inc.
Xilin, Zynq and Vivado are trademarks or registered trademarks of Xilinx, Inc.

DESIGNED BY AVNET

Announcing
HAPS Developer eXpress Solution

Pre-integrated hardware and software for fast prototyping of complex IP systems

- ☑ 500K-144M ASIC gates
- ☑ Ideal for IP and Subsystem Validation
- ☑ Design Implementation and Debug Software Included
- ☑ Flexible Interfaces for FMC and HapsTrak
- ☑ Plug-and-play with HAPS-70

Designs come in all sizes. Choose a prototyping system that does too. HAPS-DX, an extension of Synopsys' HAPS-70 FPGA-based prototyping product line, speeds prototype bring-up and streamlines the integration of IP blocks into an SoC prototype.

To learn more about Synopsys FPGA-based prototyping systems, visit www.synopsys.com/haps

SYNOPSYS®
Accelerating Innovation

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

Tomoko Suto, Japan
tomoko@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/

© 2014 Xilinx, Inc. All rights reserved. XILINX, the Xilinx Logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

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

Here's to 30 Years of Innovation and to Many Decades More

Xilinx celebrated its 30th anniversary in February. As someone who is a relative newcomer, having joined the company in 2008, I find it remarkable how far and fast Xilinx and the devices we make have advanced in just the last six years. I can't imagine what it must be like for the employees who have been with the company essentially from the beginning, when Xilinx was a tiny startup on Hamilton Avenue in San Jose offering what to many seemed like a crazy technology with an even crazier business model.

FPGAs and the fabless semiconductor business model are mainstays of the industry today, but back in 1984 that wasn't the case. Years ago, I had the pleasure of interviewing Bill Carter, who in the Xilinx world is a bit of a living legend, having laid out the industry's very first FPGA, the XC2064, and later becoming Xilinx's first CTO. Of the many great recollections Bill shared regarding his years at Xilinx, one of the most memorable was the story of his job interview with Xilinx's co-founders: Ross Freeman (who invented the FPGA), Bernie Vonderschmitt and Jim Barnett. Basically, the founders laid out their plans to come up with a reprogrammable device and have it manufactured by Seiko-Epson.

"I took one look at the architecture and thought they were nuts," said Carter. "Back then, silicon real estate was precious and their new circuit structure took so many transistors to implement. But the idea of being able to reprogram the hardware was revolutionary."

The business model was sort of nuts too for that time. Back in 1984, everyone manufactured their own chips and the biggest barrier to entry into the business was securing enough funding to build a factory. Carter recalled that Vonderschmitt knew manufacturing from his days at RCA and could call on his friends at Seiko (to whom he had shown the silicon-manufacturing ropes while working at RCA) to produce the chips.

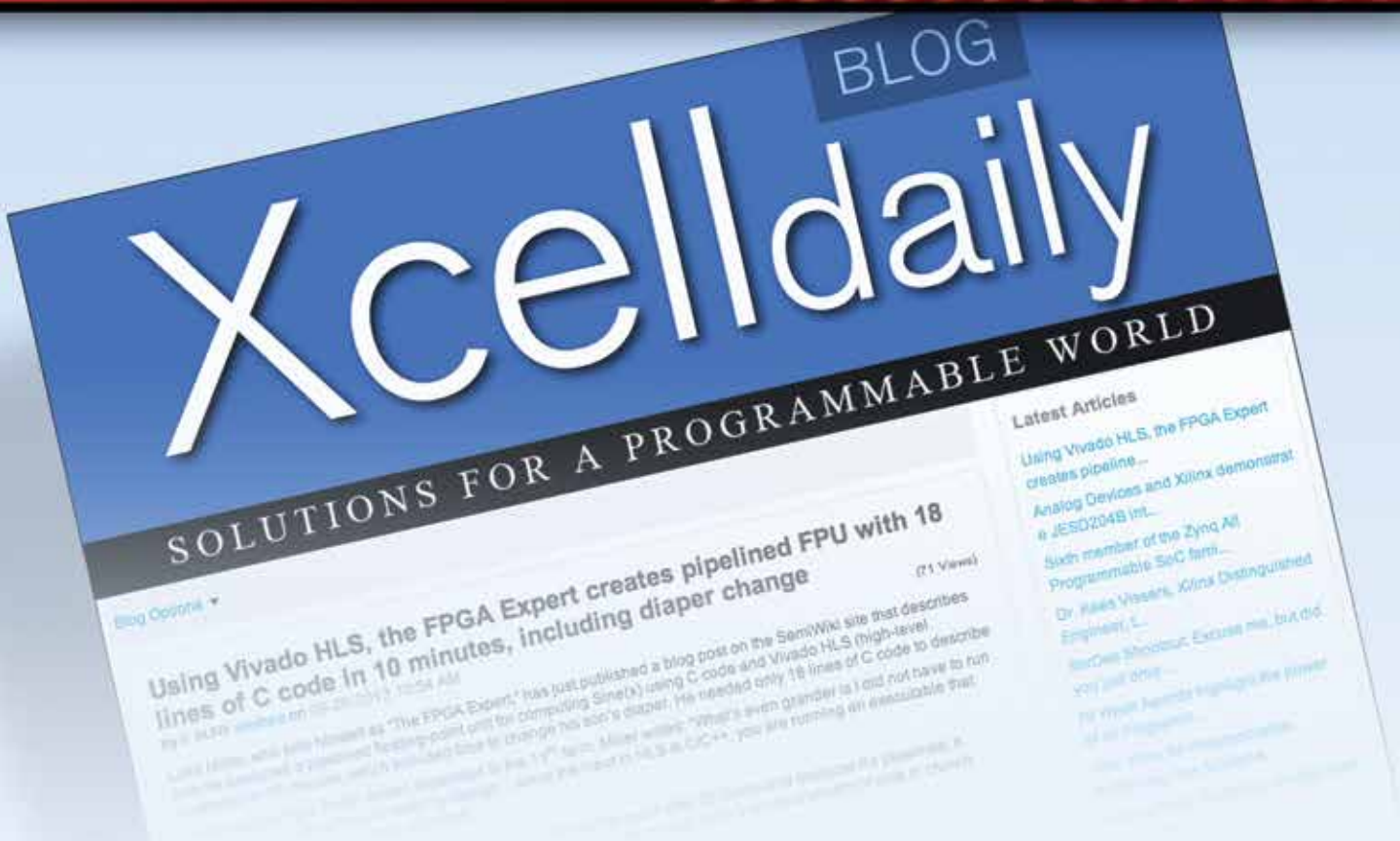
So here was Bill Carter, with a young family and a mortgage to pay, working a solid job for established Zilog, creator of the Z80. He walked away to take a chance with Xilinx. The rest is history. Xilinx introduced FPGAs to the world in 1985, and years later the fabless model under the leadership of TSMC became the mainstream way to produce chips.

What I realize 30 years after the fact is that while many personalities have come and gone and shaped the character of Xilinx, the risk-taking, innovative spirit that launched the company is still thriving here. In the last six years, I've been an eyewitness to Xilinx gaining a Generation Ahead lead over the competition at the 28-nanometer node by means of innovative silicon (7 series All Programmable FPGAs, SoCs and 3D ICs) and tools (the Vivado® Design Suite). What's more, Xilinx is stretching that lead with first-to-market 20-nm UltraScale™ devices. And you can count on even bigger innovations coming down the pike as the era of the FinFET arrives. This pioneering spirit is charged by the remarkable innovations Xilinx customers have created with our chips over the last 30 years. I'm sure we'll see even more customer innovations in the decades ahead.

Editor's Note: If you are interested in reading about the early pioneering years of Xilinx and the fabless industry, I recommend this great piece, "Xilinx and the Birth of the Fabless Semiconductor Industry," written by Xcell Daily's *editor, Steve Leibson. If you want to keep up to date with the future Xilinx innovations, please keep reading and contributing great technical content to* Xcell Journal*—now proudly in our 26th year of publishing.*

Mike Santarini
Publisher

Xcell Journal Adds New Daily Blog



Xilinx has extended the Award Winning Journal and added an exciting new *Xcell Daily Blog*. The new site provides dedicated readers with a frequent flow of content to help engineers leverage the flexibility and extensive capabilities of Xilinx products, ecosystem, and customers to create All Programmable and Smarter Systems.

Recent

- ***Is DDR4 the last SDRAM protocol? Yes, says SemiWiki's Eric Esteve. Then what are the alternatives?***
- ***Access memory-mapped devices in Linux without writing drivers on the Zynq®-based ZedBoard***
- ***Jan Gray's New LUT Math: How many 32-bit RISC CPUs fit in an FPGA? Now vs. 1995?***
- ***For Zynq SoC Developers: Latest version of ARM® Cortex™-A Programmer's Guide just published. Download now!***
- ***XIMEA CB200 5K digital video camera pumps 1.7Gbytes/sec down a 300M optical cable using FPGA-based PCIe***

Visit Blog: www.forums.xilinx.com/t5/Xcell-Daily/bg-p/Xcell

XILINX
ALL PROGRAMMABLE.

VIEWPOINTS

Letter From the Publisher

Here's to 30 Years of Innovation and to Many Decades More… **4**

XCELLENCE BY DESIGN APPLICATION FEATURES

Xcellence in Wireless

Xilinx's 20-nm UltraScale Architecture Advances Wireless Radio Applications…**14**

Xcellence in Industrial

Angle Measurement Made Easy with Xilinx FPGAs and a Resolver-to-Digital Converter…**24**

Xcellence in Industrial

Motor Drives Migrate to Zynq SoC with Help from MATLAB…**32**

Cover Story

8 Xilinx's New SDNet Environment Enables 'Softly' Defined Networks

THE XILINX XPERIENCE FEATURES

Xplanation: FPGA 101

How to Use Interrupts on the Zynq SoC… **38**

Xplanation: FPGA 101

Calculating Mathematically Complex Functions… **44**

Xplanation: FPGA 101

Make Slow Software Run Fast with Vivado HLS… **50**

Tools of Xcellence

Xilinx Opens a Tcl Store… **54**

XTRA READING

Xtra, Xtra The latest Xilinx tool updates and patches, as of April 2014… **60**

Xpedite Latest and greatest from the Xilinx Alliance Program partners… **62**

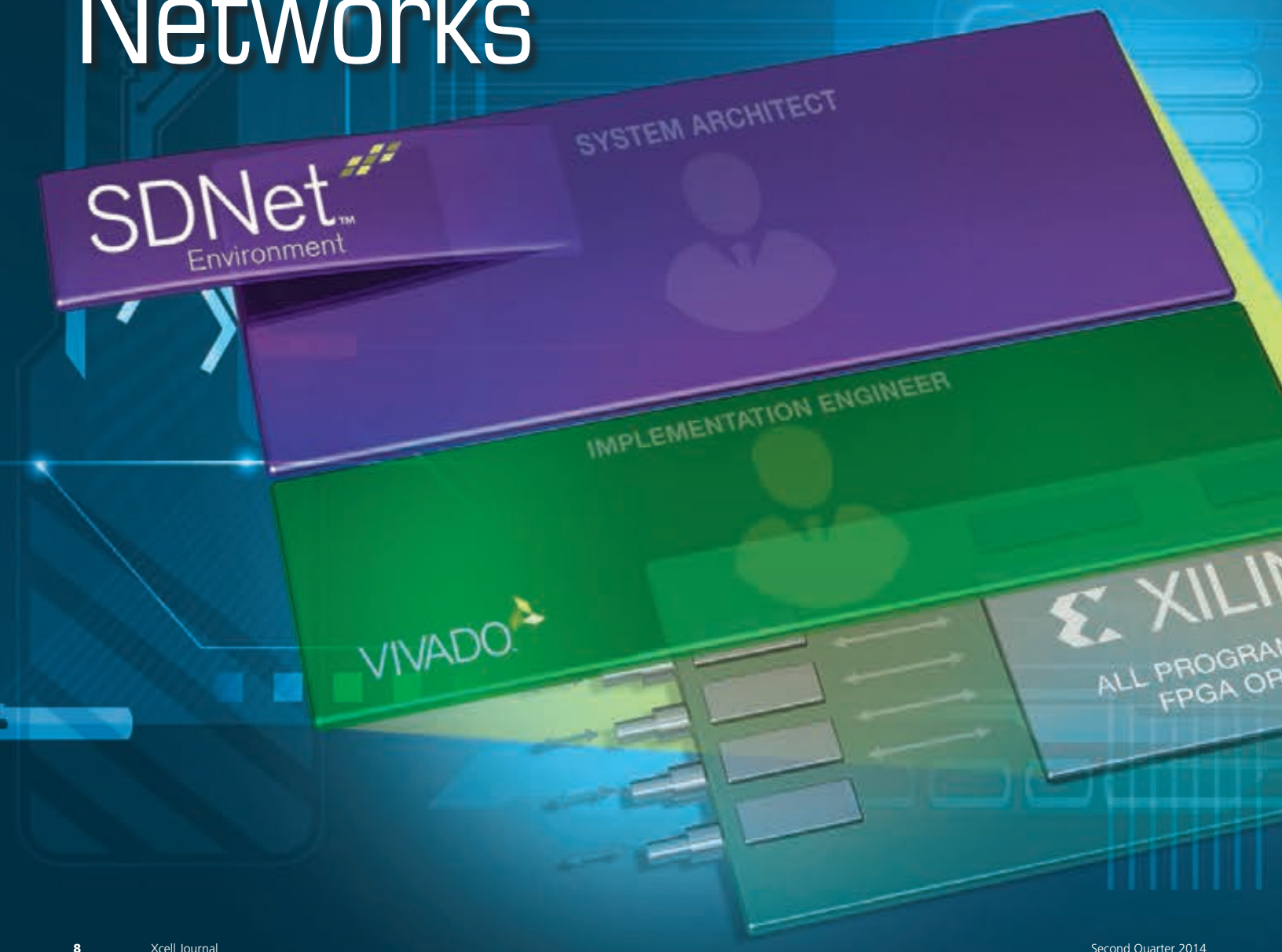
Xamples A mix of new and popular application notes… **64**

Xclamations Share your wit and wisdom by supplying a caption for our wild and wacky artwork… **66**

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

APEX®
AWARDS FOR PUBLICATION EXCELLENCE
Excellence in Magazine & Journal Design
2010, 2011, 2012

Xilinx's New SDNet Environment Enables 'Softly' Defined Networks



With Xilinx technology, design teams can now build a line card on a chip and tailor their hardware for specific network services and applications.

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

At a time when communications architectures are rapidly evolving, driven by consumer demand for greater bandwidth and better, more reliable and secure services, Xilinx has innovated a game-changing technology and design approach that will enable its customers to quickly produce and upgrade next-generation line cards for wired and wireless networks as well as data centers. The new technology is SDNet, a software-defined specification environment. When used with Xilinx® All Programmable FPGAs and SoCs, SDNet allows communications design groups to apply a revolutionary approach that Xilinx calls "Softly" Defined Networks to the design and upgrade of line cards for the next generation of software-defined network architectures.

FROM FIXED NETWORKS TO SDN

The communications architectures of the past 20 years have mainly comprised fixed control and data planes that didn't expand as network requirements evolved, said Nick Possley, vice president of communications IP and services at Xilinx. This rigid architecture required carriers to replace equipment frequently if they wanted to expand network functionality and increase overall bandwidth. The line cards at the heart of these systems were largely based on a mix of highly specialized ASICs, ASSPs and memory ICs. FPGAs served to accelerate and bridge communications among the chips on the line card.

As the pace of demand quickened, carriers and the communications systems companies that serve them sought better alternatives. In the last few years, they have turned to software-defined networks (SDN) and network functions virtualization (NFV). These architectures separate the control and data planes, and add more software virtualization to the control

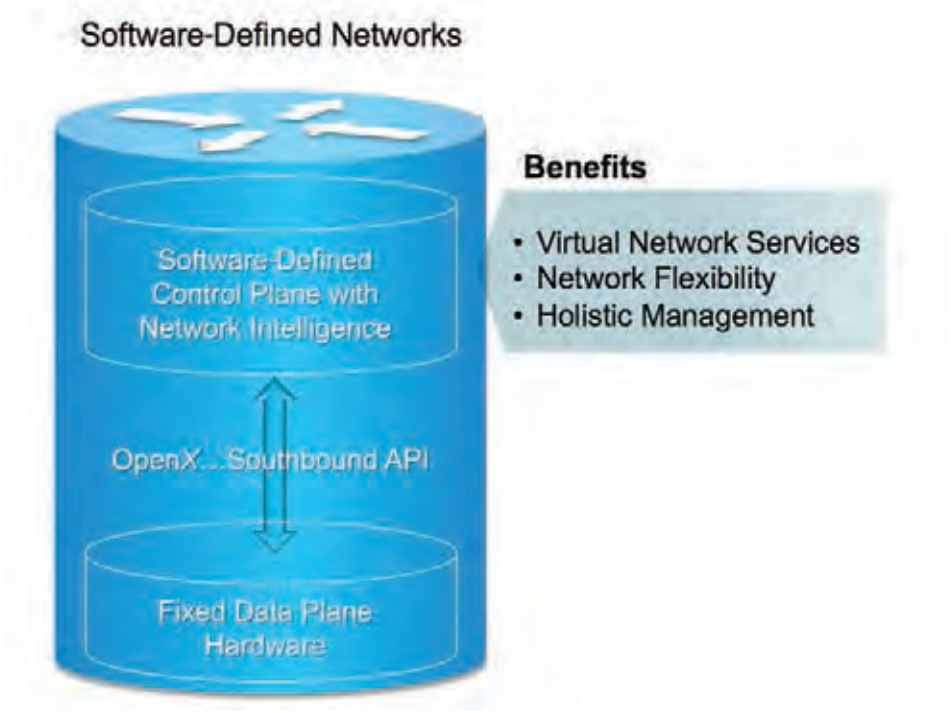


Figure 1 – Today's software-defined networks separate the control and data planes but still have fixed data planes, minimal differentiation and short life cycles.

plane. As a result, carriers can rapidly deploy new applications, and network equipment is easier to upgrade than in traditional networks. This improves longevity (and profitability) and simplifies network management (Figure 1).

But Possley said that even the most recent SDN and NFV architectures are too rigid in that the data planes are not programmable and the designs typically are based on off-the-shelf ASSPs. The line cards at the heart of the network use discrete off-the-shelf packet-processor and traffic-manager ASSPs connected to optics, along with coprocessors and external memory. The cards also include FPGAs to accelerate communications among all of these chips.

The latest versions of ASSPs that various chip makers have created for SDN and NFV architectures do comply with SDN specifications. But because the suppliers make the same ASSPs generally available to all network systems companies, these chips provide no competitive product differentiation or feature expansion. As a result, network system vendors are forced to compete on lowest pricing to carriers.

On the surface, one would think the carriers would love this lower equipment pricing. But in reality, fixed data-plane designs even in ASSP-based SDN architectures are still so rigid that carriers will have to make expensive in-field line-card swaps when they find out the ASSP's fixed hardware functionality can't accommodate ever-changing applications, protocol updates and new feature requirements. These line-card swaps require networks to shut down while technicians remove obsolete cards and install new ones. What's more, ASSP vendors tend to overbuild the functionality of their designs in an attempt to address a broad number of markets with a single device. As a result, these ASSP-based line cards tend to be power hungry and thus run hot, so carriers must take extra measures to keep the equipment cool. The cost of cooling, of course, adversely affects operating expenditures and further cuts into a carrier's bottom-line profitability.

A BETTER SOLUTION: SOFTLY DEFINED NETWORKS

With SDNet and Xilinx's revolutionary softly defined network approach, communications systems companies can develop integrated, low-power, All Programmable line cards that boast far more than a software-defined control plane with network intelligence required by SDN architectures. This new technology will also let vendors differentiate their systems with software-defined data-plane hardware that has content intelligence, meaning that design teams can tailor the hardware to the exact network services and applications their systems require (Figure 2).

Traditionally, network architects (who typically don't have hardware design backgrounds) express the requirements of particular protocols in English-language descriptions, such as Internet requests for comment (RFCs) or ISO standards documents.

They then have to rely on specialized engineers who are very well-versed in the underlying architecture of the target device to manually turn those requirements into low-level, implementation-specific, descriptions (typically using highly specialized microcode). These hardware engineers will either specify how the general-purpose processors or specialized network processors should perform the packet processing, or they will design the functionality into a custom ASIC.

Network design teams then have to verify that hardware achieves the architect's original design intent or can at least accommodate the most recent version of the protocol they intend the card to use. If the line card doesn't meet the requirements, they have to repeat the design process until they get it to work properly. This process is complicated by the fact that the relationship between the desired specification and the microcode is not intuitive and the underlying architecture has performance limitations and capabilities that vary based on the services companies are targeting.

SDNet's softly defined network approach goes to the root of this problem and allows network system design teams to quickly design line cards that are correct by construction. In particular, SDNet focuses on automating the most complex aspect of line card design—namely, the design and programming of the packet-processor and traffic-manager functions in modern line cards (Figure 3).

Instead of having two separate, discrete ASSPs handling these functions, network systems teams can integrate packet processing and traffic management as well as other line card functionality on a single Xilinx All Programmable FPGA or SoC. They can ensure they are creating optimal implementations for their targeted applications. In addition to integrating the functionality of many chips into one All Programmable device, SDNet streamlines the creation of a high-level behavioral specification of the line card and automatically generates RTL blocks for implementation in Xilinx All Programmable devices, firmware and a validation testbench.

"With SDNet, system architects specify the 'what,' not the 'how,'" said Possley. "System architects specify the exact services they are looking to deploy without regard as to how they are being deployed in the underlying silicon."

In the SDNet flow, system architects define line-card functionality using a high-level functional specification (Figure 4). SDNet allows architects to describe the required behavior of various types of packet-processing engines, including parsing, editing, search and quality-of-service (QoS) policy engines. Architects can describe engines hierarchically in terms of simpler subengines that they can interconnect and arrange into packet data flows. These subengines can include user-provided engines. The SDNet specification environment contains no implementation details. That gives customers the freedom to scale the performance and resources of their design without the need to understand the details of the underlying architecture. The SDNet specifications are also not limited to any specific network protocols.

Possley said that SDNet is simple, and the select few customers with whom Xilinx beta tested it have found it very intuitive and easy to use. "It dramatically cuts the amount of code they have to produce into a simple and intuitive specification and is, therefore, orders-of-magnitude less effort compared with microcoding a network processor," he said.

Once architects have finished defining the system engines and flows in the SD-

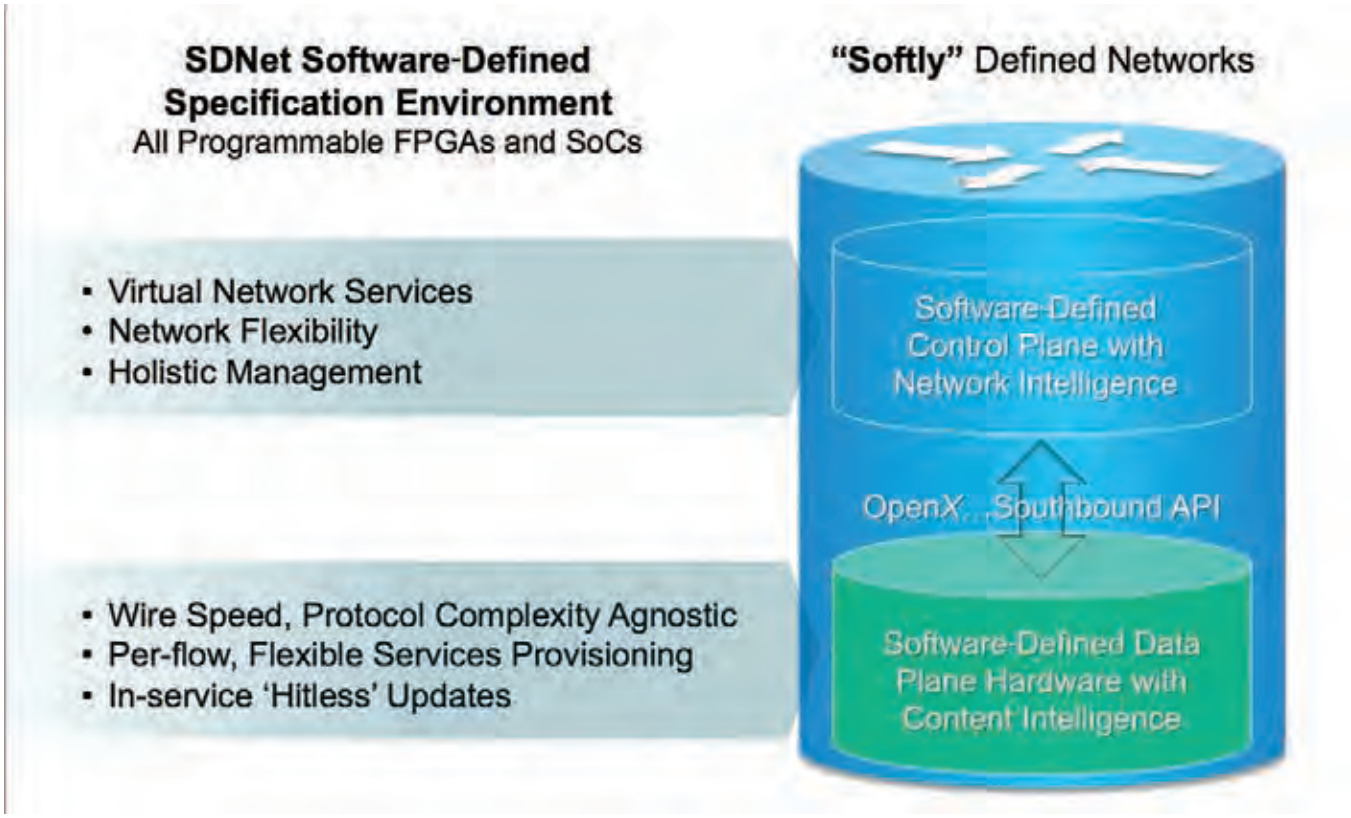


Figure 2 – SDNet brings flexibility and automation to the data plane, enabling a softly defined network approach for the design and upgrade of next-generation networks.

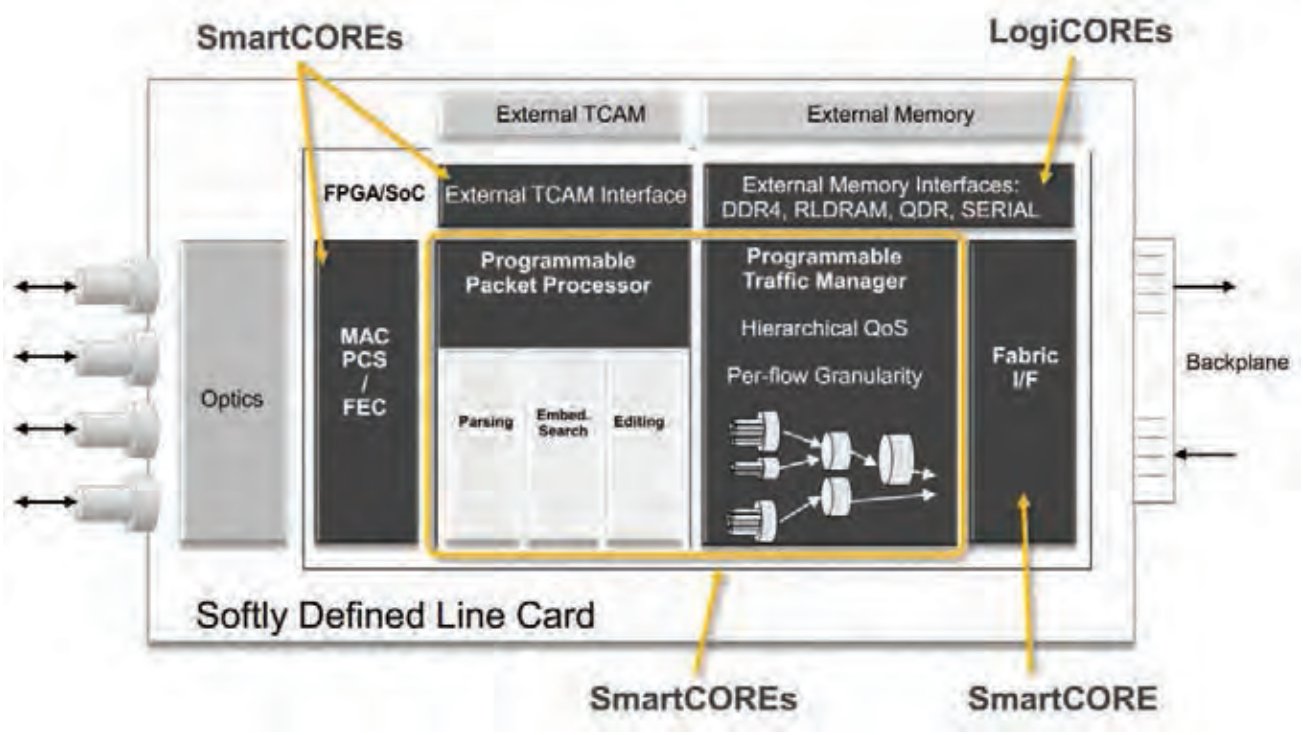


Figure 3 – With SDNet, companies can create a highly integrated All Programmable line cards.

Net specification environment, they provide SDNet's compiler with throughput and latency requirements and run-time programmability requirements that influence the optimized hardware architecture generated by the compiler. They then execute a command, and SDNet's compiler automatically generates the RTL for the hardware blocks the design requires. The compiler also generates firmware and a verification/validation testbench. The SDNet design environment includes integration of Xilinx-optimized SmartCOREs for networking and LogiCOREs™ for connectivity, external memory control and embedded processors.

After compilation, network engineers can then finish the implementation of the design in the Vivado® Design Suite using the IP Integrator (IPI) tool. They first use the Vivado tools and IPI to transform the RTL architecture description the SDNet compiler has generated into an optimized Xilinx FPGA implementation. They can then integrate any additional line-card functionality into the FPGA, given sufficient resources on the device they've selected, essentially

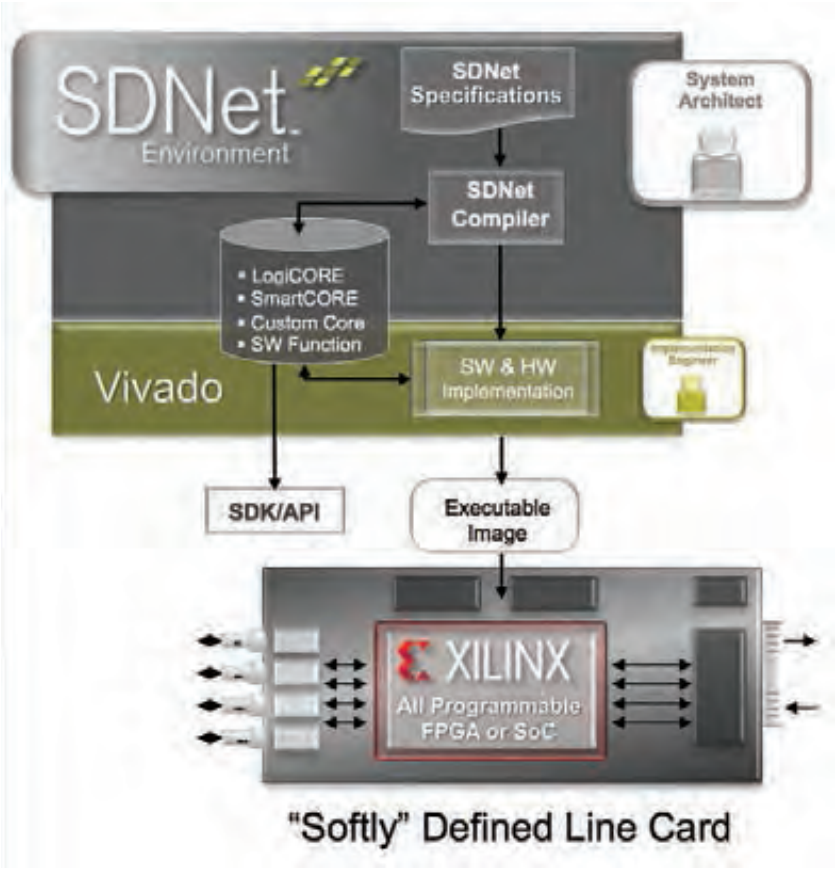


Figure 4 – The SDNet-based implementation flow enables correct-by-construction design of an All Programmable line card.

creating an All Programmable line card on a chip.

What's more, SDNet generates data for functional verification and validation to guide correct-by-construction design. Specifically, SDNet's compiler accepts a collection of test packets for testing input and output of the design. Architects can use the packets in the specification-definition phase of the process to ensure they are creating an accurate interpretation of the SDNet description. Network engineers can use test packets during the simulation of the RTL description generated by the SDNet compiler. Last but not least, the packets can help with hardware validation of the final implementation of the design using network test equipment. In addition, SDNet will generate corresponding contents for search engine lookup tables. This verification-and-validation ability vastly reduces design time and eliminates iterations between system architects and network hardware engineers, allowing the teams to get highly differentiated products to market faster.

Gordon Brebner, distinguished engineer at Xilinx, said the compiler automatically generates custom firmware operations and their binary encodings for each individual component in the architecture. "This gives architects an intimate level of control over the processing," he said. SDNet has a utility that keeps a record of runs and stores details of the generated architecture and its firmware. When users rerun the compiler with an updated SDNet description as input, it determines whether the change can be accommodated with a firmware update only (without generation of new hardware), or whether a regeneration of the hardware (and firmware) is needed. In most cases, medium-scale updates, such as adding or subtracting a protocol the line card will handle, can be done through firmware updates only.

"The intimate connection between the firmware and the architecture, which are both generated by SDNet's compiler, means that users can perform hitless upgrades, whereby the firmware is changed and placed into service without disrupting the flow of packets," said Brebner. "In this way, companies can perform significant service upgrades without any interruption to the service. This revolutionary development is achieved through the unique nature of the SDNet technology and its coupling of high-level specifications with Xilinx All Programmable devices" (Figure 5).

"SDNet's ability to generate datapath processing functions that support hitless, in-service updates is unique," said Possley. "Carriers can update line card components with new features or capabilities using a software controller via standard SDNet APIs. They can run the updating software on an embedded soft processor or on an external processor." Of course, if they implemented the design on a Xilinx Zynq®-7000 All Programmable SoC, he added, they can run the software on the device's embedded ARM® processor.

"SDNet offers full hardware programmability under software control, which is why we call it 'softly' defined networking," Possley said.

For more information on the SDNet specification environment, including a video demonstration of SDNet in action, visit *www.xilinx.com/sdnet*. At the same site, you will find an in-depth white paper entitled "Software Defined Specification Environment for Networking (SDNet)."

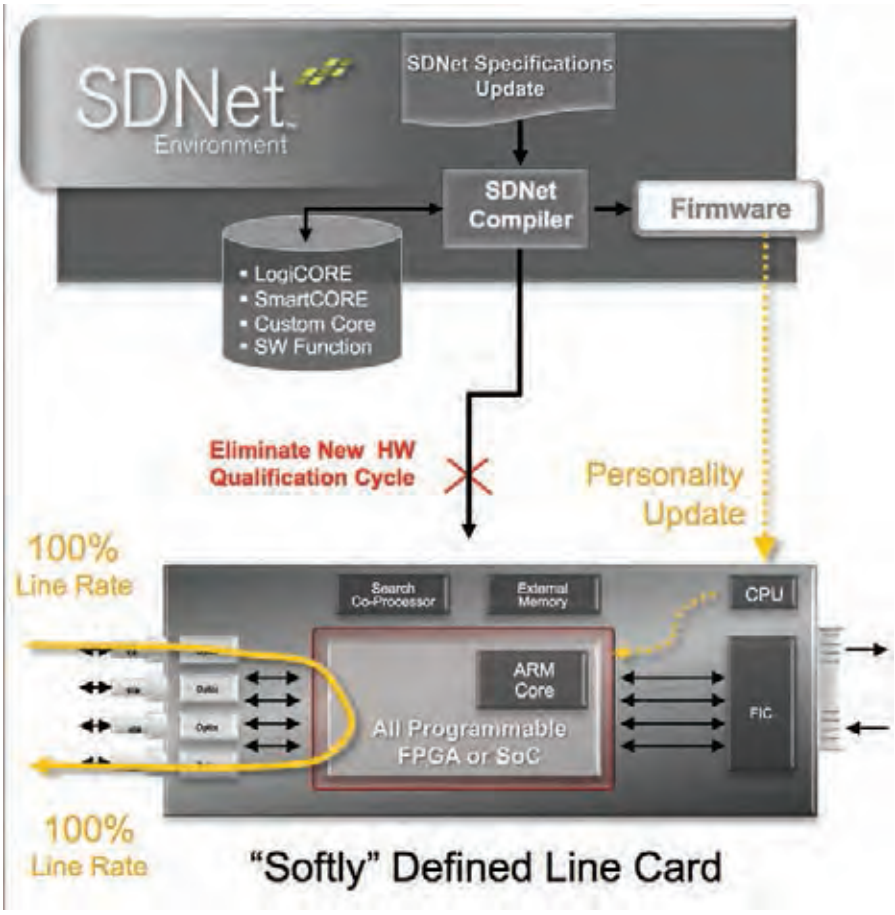
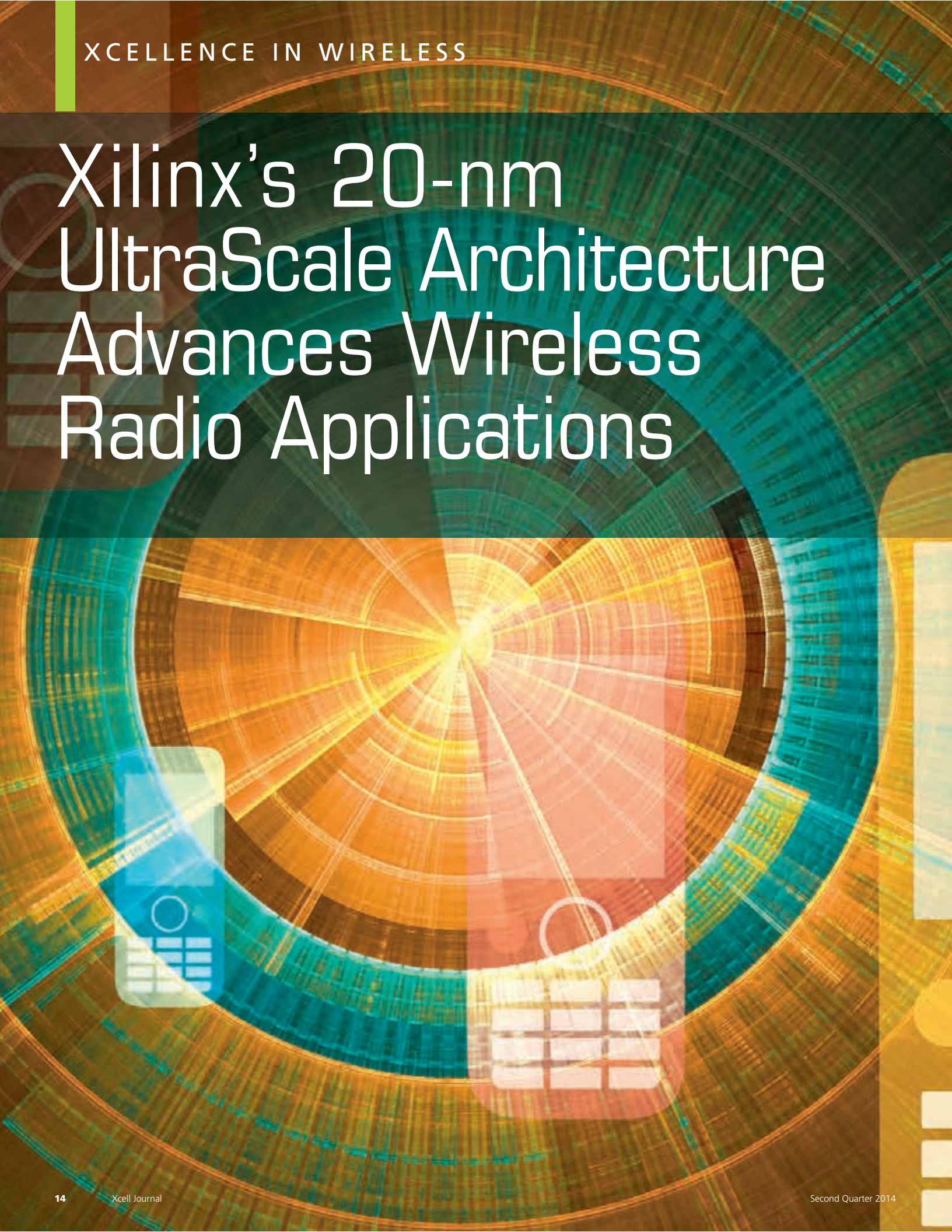


Figure 5 – After deployment, SDNet allows vendors to update protocols on line cards without interrupting service.

Xilinx's 20-nm UltraScale Architecture Advances Wireless Radio Applications



Next-generation 5G systems will be complex to design. UltraScale devices have built-in functionality that will make the job easier.

by Michel Pecot
Wireless Systems Architect
Xilinx, Inc.
michel.pecot@xilinx.com

Upcoming 5G wireless communications systems will likely be required to support much wider bandwidths (200 MHz and larger) than the 4G systems used today, along with large antenna arrays, enabled by higher carrier frequencies, that will make it possible to build much smaller antenna elements. These so-called massive MIMO applications, together with more stringent latency requirements, will increase design complexity by an order of magnitude.

At the end of last year, Xilinx announced the 20-nanometer UltraScale™ family and the first devices are now shipping [1,2,3]. This new technology brings many advantages over the previous 28-nm 7 Series generation, especially for wireless communications. Indeed, the combination of this new silicon and the tools of the Xilinx® Vivado® Design Suite [4, 5] is a perfect fit for high-performance signal-processing designs such as next-generation wireless radio applications.

Let's look at the benefits of the UltraScale devices for such designs, with a focus on architectural aspects—specifically, the advantages of the enhancements brought to the DSP48 slices and Block RAMs for the implementation of some of the most common functionalities used in radio digital front-end (DFE) applications. The UltraScale family offers much denser routing and clocking resources compared with the 7 Series devices, enabling better device utilization, especially for high-speed designs. However, these features do not usually have a direct impact on design architectures, so we will not address them here.

OVERVIEW OF ENHANCEMENTS TO ULTRASCALE FABRIC

Moving to 20 nm not only enables the higher integration capabilities, improved fabric performance and lower power consumption that come with any geometry node migration, but the UltraScale 20-nm architecture also includes several new, greatly enhanced features that directly support DFE applications. This is especially true for the UltraScale Kintex® devices, which Xilinx has highly tuned to the needs of this type of design.

First, these devices contain up to 5,520 DSP48 slices. That's almost three times more than the maximum count of 1,920 available on 7 Series FPGAs (2,020 for the Zynq®-7000 All Programmable SoC). Higher levels of integration are therefore possible. For example, you can implement a complete 8Tx/8Rx DFE system with instantaneous bandwidth of 80 to 100 MHz in a single midrange UltraScale FPGA, while a two-chip solution is necessary on the 7 Series architecture, with each chip effectively supporting a 4x4 system. For a detailed functional description of such designs, read the Xilinx white paper WP445, "Enabling High-Speed Radio Designs with Xilinx All Programmable FPGAs and SoCs" [6].

The serdes can support a throughput of 12.5 Gbps on slowest-speed-grade devices, enabling the maximum speed of JESD204B interfacing.

With the thermal constraints imposed by passively cooled remote radios, the integration of complex designs into a single device requires a significant power reduction to be able to dissipate the heat. The UltraScale family offers such a capability, with 10 to 15 percent less static power compared with 7 Series devices of the same size, and 20 to 25 percent less dynamic power for similar designs. Furthermore, Xilinx has also significantly lowered serdes power consumption in the UltraScale product line.

There is a performance advantage as well. The slowest-speed-grade UltraScale devices support designs with clock rates higher than 500 MHz, while midspeed grade is required for the 7 Series devices. However, even here, Block RAMs are still demanding from a timing perspective, and WRITE_FIRST or NO_CHANGE modes need to be selected to reach this kind of performance. You cannot use READ_FIRST, since it is limited to around 470 MHz, while 530 MHz is achievable for the other two modes. NO_CHANGE is your best choice whenever possible, since it also minimizes the power consumption.

Similarly, the serdes can support a throughput of up to 12.5 Gbps on the slowest UltraScale speed grade, hence enabling the maximum speed of JESD204B interfacing, which should be soon available on most DACs and ADCs. Similarly, the lowest speed grade can also support the two highest CPRI rates (rates 7 and 8, with respective throughputs of 9.8304 and 10.1376 Gbps) as well as 10GE interfaces, which are commonly used in DFE systems.

In addition, the UltraScale Kintex resource mix is better suited for radio applications, which results in a more optimal usage of the logic resources. The DSP-to-logic ratio, especially, is much more closely in line with what is typically required for DFE designs. More precisely, UltraScale Kintex devices have eight to 8.5 DSP48 slices per 1K lookup tables (LUTs), while this number is only around six on 7 Series devices.

Xilinx has also significantly increased the clocking and routing resources in the UltraScale architecture. This increase enables higher device utilization, especially for high-clock-rate designs. In effect, routing congestion is reduced, and designers can achieve better design packing and LUT utilization. In particular, LUT/SRL compression is more efficient. This is an interesting fabric feature that users can exploit to better pack their designs and consequently optimize resource utilization as well as dynamic power consumption, which can be reduced by a factor of up to 1.7 for the related logic. The principles of LUT/SRL compression involve using the two outputs of the LUT6 to pack two different functions in a single LUT. In this way, you can pack two LUT5s, implementing a logic function or a memory, into a single LUT6, provided they share the same inputs or read/write address for a memory. Similarly, you can pack two SRL16s into a single LUT6.

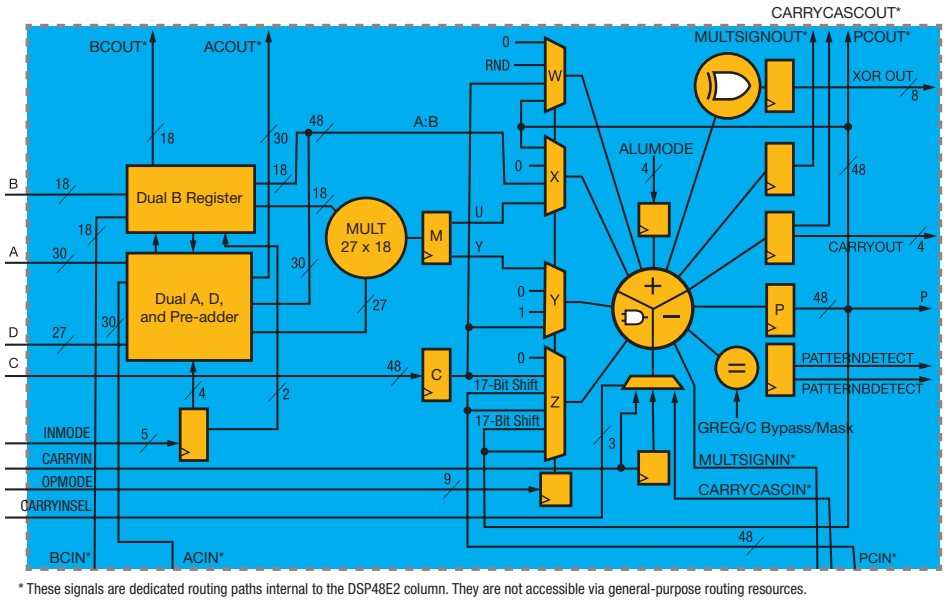
This feature is quite useful for digital radio designs, which usually integrate many small memories sharing the same address—for instance, ROMs storing filter coefficients—and a lot of short delay lines (less than 16 cycles) to time-align different signal paths. Functions for data multiplexing, especially two-input multiplexers, can benefit from this feature too. LUT/SRL compression, however, must be used carefully when targeting high clock rates. First, you must use the two flip-flops connected to the O6/O5 LUT outputs to avoid any timing issues. For the same reasons, it is recommended to apply this capability to related logic only, a strategy that also has the advantage of limiting routing congestion.

The clocking architecture and configurable logic block (CLB) also contribute to better device utilization in the UltraScale devices. Although the CLB is still based on that of the 7 Series architecture, there is now a single slice per CLB (instead of two), integrating eight, six-input LUTs and 16 flip-flops. The carry chain is consequently 8 bits long and a wider output multiplexer is available. In addition, Xilinx has also increased the control-set resources (that is, the clock, clock-enable and reset signals shared by the storage elements within a CLB).

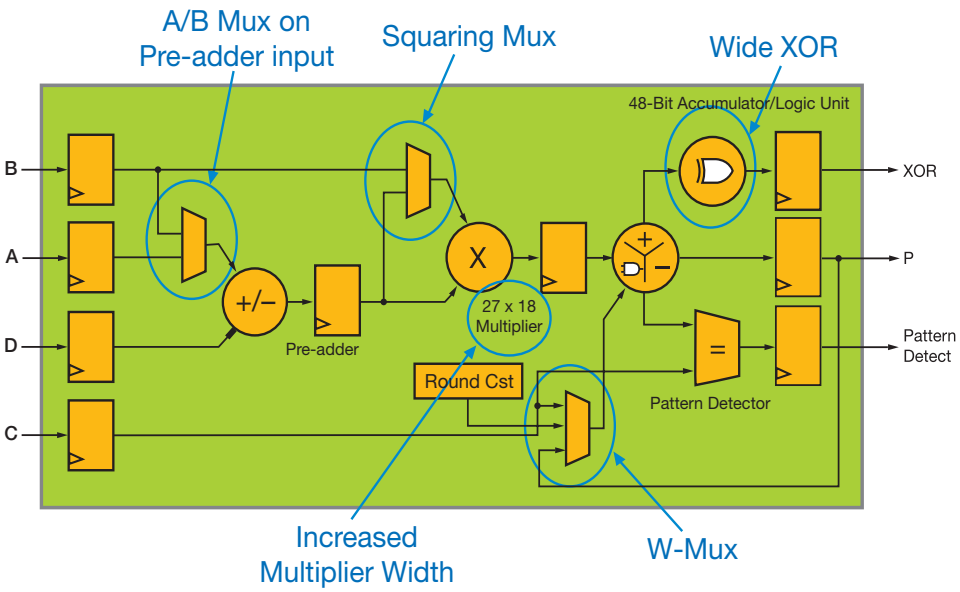
However, it is essentially the improvements to the DSP48 slice and Block RAM that have the most impact on radio design architectures. Let's look at them more closely.

BENEFITS OF THE ULTRASCALE DSP48 SLICE ARCHITECTURE

Figure 1 shows a view of the UltraScale DSP48 slice (DSP48E2). The top diagram (labeled "a") describes the detailed architecture, while the bottom part ("b") highlights the functional enhancements compared with the 7 Series slice (DSP48E1).



(a) Detailed DSP48E2 architecture



(b) DSP48E2 high-level functional view

Figure 1 – Architecture of the UltraScale DSP48 slice

The Xilinx user guide UG579 offers a comprehensive description of the DSP48E2 capabilities [7]. The major enhancements in the UltraScale architecture are:

- Xilinx has increased the multiplier width from 25x18 to 27x18, and the pre-adder width rises accordingly to 27 bits.
- You can select the pre-adder input to be either A or B, and some multiplexing logic has been integrated on output, which allows feeding D±A or D±B on any of the multiplier inputs (27-bit or the 18-bit input).
- The pre-adder output can feed both multiplier inputs (with appropriate MSB truncation on the 18-bit input), hence allowing the computation of $(D\pm A)^2$ or $(D\pm B)^2$ for up to 18-bit data.
- A fourth operand is added to the arithmetic logic unit (ALU), through the extra W-mux multiplexer, which can take as input either C, P or a constant value (defined at FPGA configuration). This makes it possible to perform a three-input operation when the multiplier is used, such as A*B+C+P or A*B+P+PCIN. It is worth noting that the W-mux output can only be added within the ALU (subtraction is not permitted).
- Xilinx has integrated additional logic to perform a wide XOR between the 96 bits of any two of the X, Y or Z multiplexer outputs. Four different modes are actually available, offering 1x 96-bit, 2x 48-bit, 4x 24-bit or 8x 12-bit XOR operation.

Increasing the multiplier size from 25x18 to 27x18 has minimal impact on the silicon area of the DSP48 slice, but significantly improves the support for floating-point arithmetic. First, it is worth pointing out that the DSP48E2 can in effect support up to 28x18-bit or 27x19-bit signed multiplication. This is achieved by using the C input to process the additional bit, as described in Figure 2, which shows the multiplication of a 28-bit operand, X, with an 18-bit operand, Y.

The 45 most significant bits (MSBs) of the 46-bit output are computed as:

$$Z[45:1] = X[27:1]*Y[17:0] + X[0]*Y[17:1]$$

The 27 MSBs of X and 18 bits of Y are directly fed into the DSP48E2 multiplier inputs, while X[0]*Y[17:1] is derived from an external 17-bit AND operator and sent to the C input after a single pipelining stage to match the DSP48E2 latency. The AND operator can actually be omitted by directly feeding Y[17:1] into a register with the reset pin controlled by X[0]. Similarly, an external 1-bit AND operator and a three-clock-cycle delay for latency balancing are used to compute the LSB of Z, Z[0].

You can therefore implement a 28x18-bit multiplier with a single DSP48E2 slice and 18 LUT/flip-flop pairs. The same applies for a 27x19-bit multiplier, using 27 additional LUT/flip-flop pairs. In both cases, convergent

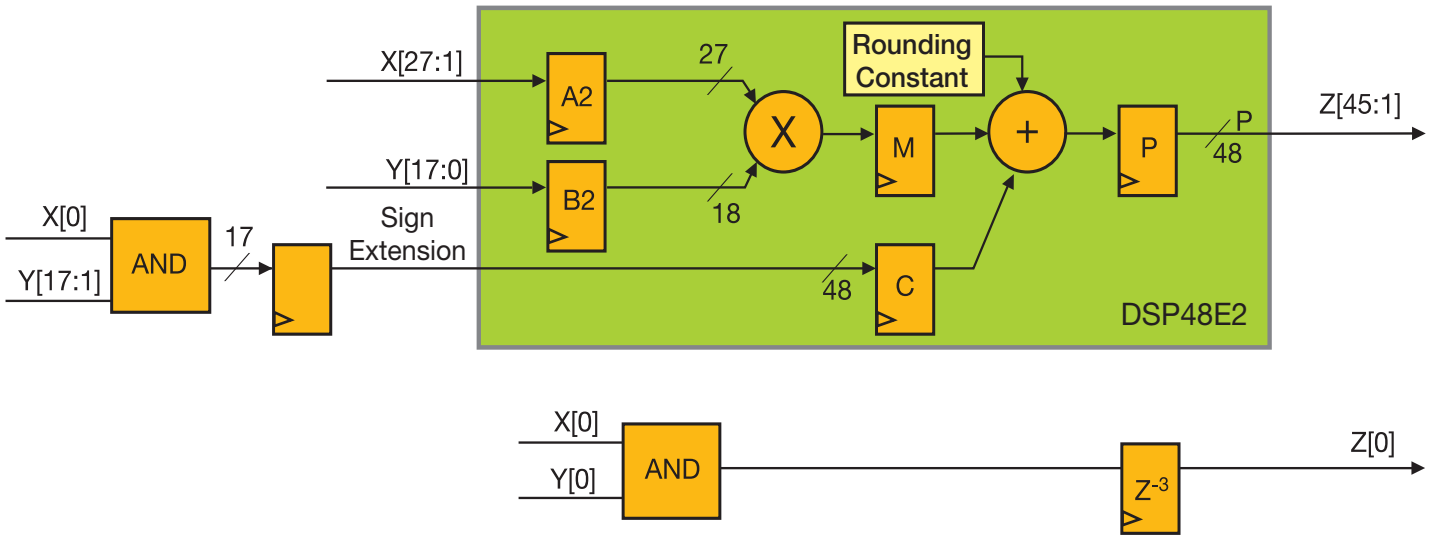


Figure 2 – A 28x18-bit signed multiplication with convergent rounding capability of the output

rounding of the result can still be supported through the W-mux.

A double-precision floating-point multiplication involves the integer product of the 53-bit unsigned mantissas of both operators. Although a 52-bit value (m) is stored in the double-precision floating-point representation, it describes the fractional part of the unsigned mantissa, and it is actually the normalized 1+m values, which need to be multiplied together; hence the additional bit required by the multiplication. Taking into account the fact that the MSBs of both 53-bit operands are equal to 1, and appropriately splitting the multiplication to optimally exploit the DSP48E2 26x17-bit unsigned multiplier and its improved capabilities (e.g., the true three-input 48-bit adder enabled by the W-mux), it can be shown that the 53x53-bit unsigned multiplication can be built with only six DSP48E2 slices and a minimal amount of external logic. It is out of the scope of this article to provide all the details of such an implementation, but a similar approach would require 10 DSP48E1 slices on the previous-generation 7 Series devices; hence there is a 40 percent gain brought by the UltraScale architecture.

The 27x18 multiplier of the DSP48E2 is also very useful for applications based on fused data paths. The concept of a fused multiply-add operator has been recently added to the IEEE floating-point standard [8]. Basically, it consists of building the floating-point operation A*B+C, without explicitly rounding, normalizing and de-normalizing the data between the multiplier and the adder. These functions are indeed very costly when using traditional floating-point arithmetic and account for the greatest part of the latency. This concept may be generalized to build sum-of-products operators, which are common in linear algebra (matrix product, Cholesky decomposition). Consequently, such an approach is quite efficient for applications where cost or latency are critical, while still requiring the accuracy and dynamic range of the floating-point representation. This is the case in radio DFE applications for which the digital predistortion functionality usually requires some hardware-acceleration support to improve the update rate of the nonlinear filter coefficients. You can then build one or more floating-point MAC engines in the FPGA fabric to assist the coefficient-estimation algorithm running in software (e.g. on one of the ARM® Cortex™-A9 cores of the Zynq SoC).

For such arithmetic structures, it has been shown that a slight increase of the mantissa width from 23 to 26 bits can provide even better accuracy compared with a true single-precision floating-point implementation, but with reduced latency and footprint. The UltraScale architecture is again well adapted for this purpose, since it takes only two DSP48 slices to build a single-precision fused multiplier, whereas three are required on 7 Series devices with additional fabric logic.

The pre-adder, integrated within the DSP48 slice in front of the multiplier, provides an efficient way to implement symmetric filters that are commonly used in DFE designs to realize the digital upconverter (DUC) and downconverter (DDC) functionality. For an N-tap symmetric filter, the output samples are computed as follows:

$$y(n)=\sum_{k=0}^{\left\lceil\frac{N-1}{2}\right\rceil} h(k).(x(n-k)+x(n-N+1+k))$$

where $x(n)$ represents the input signal and $h(n)$ the filter impulse response, with $h(n)=h(N-1-n)$.

Pairs of input samples are therefore fed into the pre-adder and the output is further multiplied with the appropriate filter coefficient. On the 7 Series architecture, the pre-adder must use the 30-bit input (A) of the DSP48E1, together with the 25-bit input (D), and its output

The addition of a fourth input operand to the ALU through the extra W-mux multiplexer brings the most benefit for radio applications.

is connected to the 25-bit input of the multiplier, while the B input is routed to the 18-bit multiplier input. Consequently, when building symmetric filters, the coefficients cannot be quantized on more than 18 bits, which limits the stopband attenuation to around 85 to 90 dB. This may be an issue for next-generation 5G radio systems, which are likely to operate in environments with a very high interference level, and may therefore need filters with greater attenuation.

The UltraScale architecture overcomes this problem because the pre-adder input can be selected as either A or B, and some multiplexing logic has been integrated on the output to allow feeding D±A or D±B to any of the multiplier inputs (27-bit or the 18-bit input). As a consequence, symmetric filters with up to 27-bit coefficients can be supported.

Another feature Xilinx has added to the DSP48E2 slice is the capability to connect the pre-adder output to both inputs of the multiplier (with appropriate MSB truncation on the 18-bit input). This makes it possible to perform operations such as $(D\pm A)^2$ or $(D\pm B)^2$ for up to 18-bit data, which can be used efficiently when evaluating sums of squared-error terms. Such operations are quite common in optimization problems, for example when implementing least-square solutions to derive the coefficients of the equalizer in a modem, or to time-align two signals.

It is indisputably the addition of a fourth input operand to the ALU, through the extra W-mux multiplexer, which brings the most benefit for radio applications. This operand can typically save 10 percent to 20 percent of the DSP48 requirements for such designs compared with the same implementation on a 7 Series device.

The W-mux output can only be added within the ALU (subtraction is not permitted), and can be set dynamically as the content of either the C or P register or as a constant value, defined at FPGA configuration (e.g. the constant to be added for convergent or symmetric rounding of the DSP48 output), or simply forced to 0. This allows performing a true three-input operation when the multiplier is used, such as A*B+C+P, A*B+C+PCIN, A*B+P+PCIN, something that is not possible with the 7 Series architecture. Indeed, the multiplier stage generates the last two partial-product outputs, which are then added within the ALU to complete the operation (see Figure 1). Therefore, when enabled, the multiplier uses two inputs of the ALU, and a three-input operation cannot be performed on 7 Series devices.

Two of the ost significant examples that benefit from this additional ALU input are semi-parallel filters and complex multiply-accumulate (MAC) operators. Let's take a closer look at both of them.

OF FILTERS AND MACS

Linear filters are the most common processing units of any DFE application. When integrating such functionality on Xilinx FPGAs, it is recommended [6], as far as possible, to implement multi-channel filters for which the composite sampling rate (defined as the product of the number of channels by the common signal-sampling frequency of each channel) is equal to the clock rate at which the design is running. In a so-called parallel architecture, each DSP48 slice supports a single filter coefficient per data channel, which greatly simplifies the control logic and hence minimizes the design resource utilization.

However, with higher clock-rate capabilities (for example, more than 500 MHz on lowest-speed-grade UltraScale devices), and for filters running at a relatively low sampling rate, it is often the case that the clock rate can be selected as a multiple of the composite sampling rate. It's desirable to increase the clock rate as much as possible to further reduce the design footprint, as well as the power consumption. In such situations, a semi-parallel architecture is built where each DSP48 processes K coefficients per channel, where K is the ratio between the clock rate and the composite sampling rate. The most efficient implementation then consists of splitting the filter into its K phases, each DSP48 processing a specific coefficient of these K phases.

At each clock cycle, the successive phases of the filter output are computed and need to be accumulated together to form an output sample (once every K cycle). Consequently, an additional accumulator is required at the filter output compared with a parallel implementation. This full-precision accumulator works on a large data width, equal to $b_S+b_C+b_F$, where b_S and b_C are respectively the bit widths of the data samples and coefficients, and $b_F=Log_2N$ is the

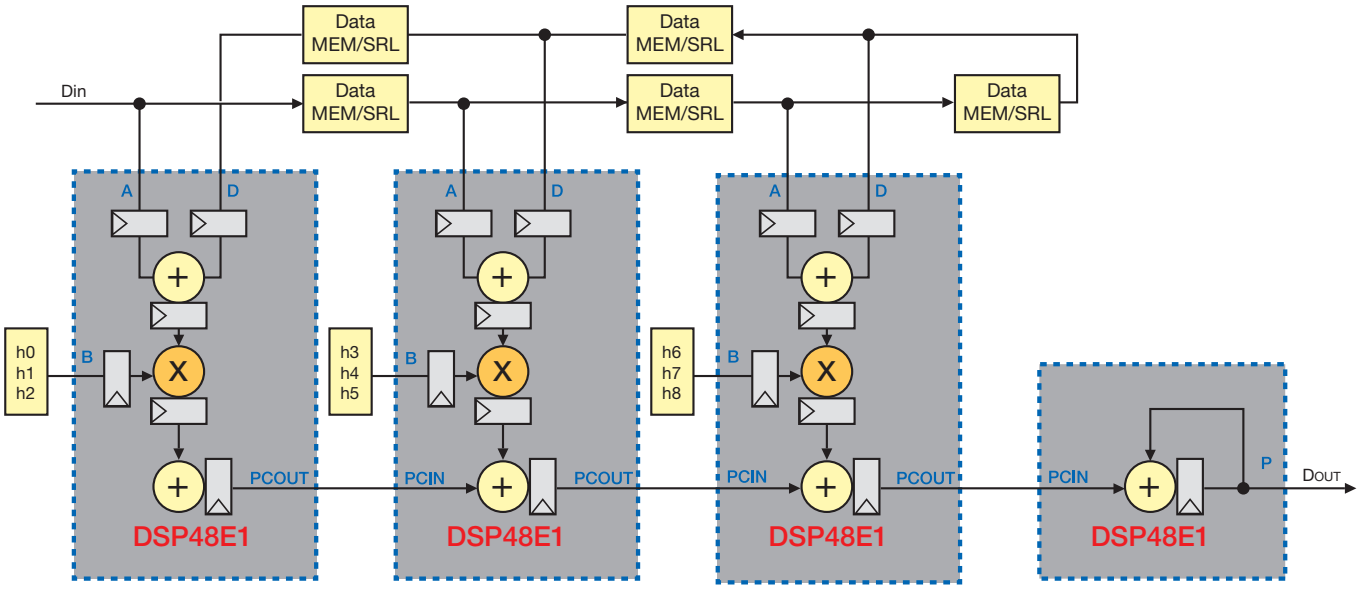
filter bit growth, N being the total number of coefficients. Normal practice is therefore to implement the accumulator within a DSP48 slice to ensure support for the highest clock rate while minimizing footprint and power.

It should be noted that semi-parallel architectures can be derived for any type of filter: single-rate, integer or fractional-rate interpolation and decimation. Figure 3 shows a simplified block diagram for both 7 Series and UltraScale implementations. It clearly highlights the advantage of the UltraScale solution, since the phase accumulator is absorbed by the last DSP48 slice thanks to the W-mux capability.

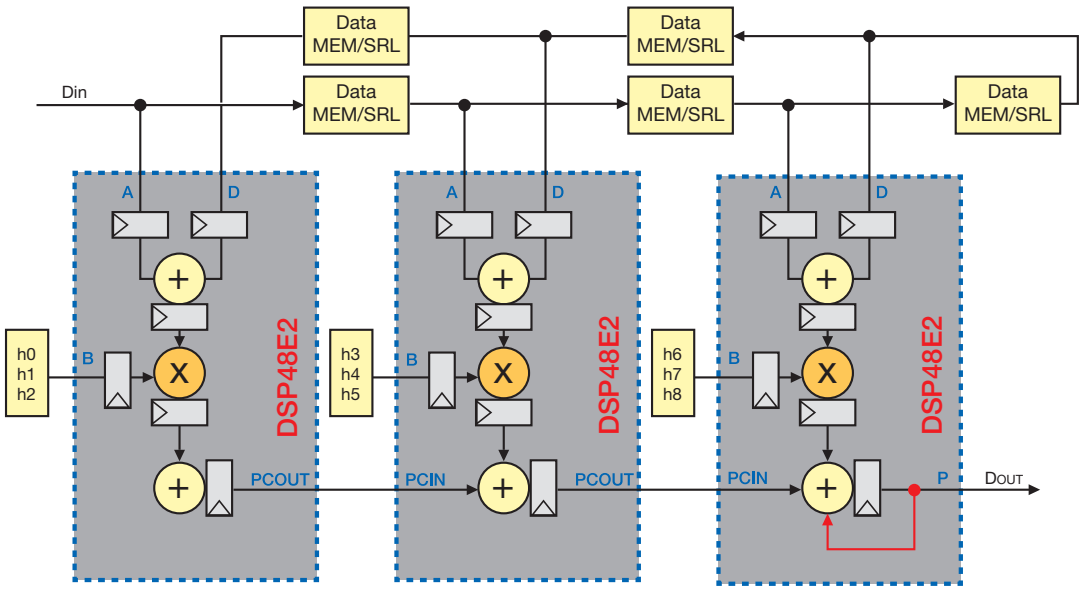
Let's now consider the implementation of a fully parallel complex MAC operator generating one output every clock cycle. It is well known that you can rewrite the equation of a complex product, $P_I + j.P_Q = (A_I + j.A_Q).(B_I + j.B_Q)$, so as to use only three real multiplications, according to:

- $P_I = P_1 + A_I.(B_I - B_Q)$
- $P_Q = P_1 + A_Q.(B_I + B_Q)$

where $P_1 = B_Q.(A_I - A_Q)$.

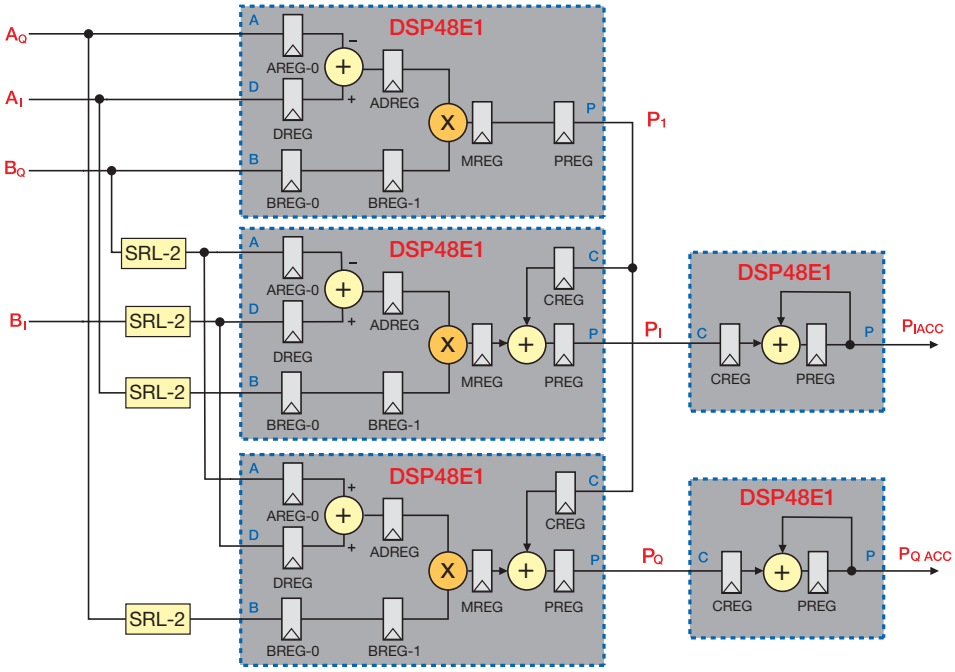


(a) 7 Series implementation

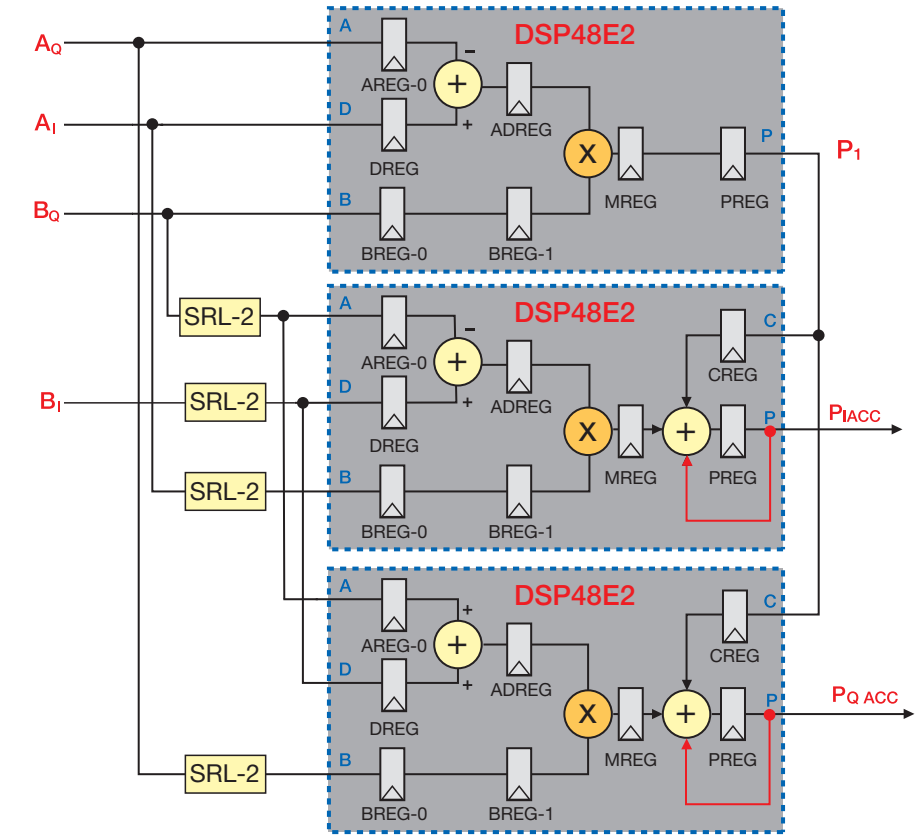


(b) UltraScale implementation

Figure 3 – Implementation of a semi-parallel filter on 7 Series and UltraScale architectures



(a) 7 Series implementation



(b) UltraScale implementation

Figure 4 – Implementation of a complex MAC on 7 Series and UltraScale architectures

Consequently, by exploiting the built-in pre-adder, you can implement a complex multiplier with three DSP48s only—one to compute P_I and the other two to handle the P_I and P_Q outputs. Depending on the latency requirements, which also dictate the speed performance, some logic needs to be added to balance the delays between the different data paths. To get maximal speed support, the DSP48 must be fully pipelined, which results in an overall latency of six cycles for the operator. A two-cycle delay line is consequently added on each input to correctly align the real and imaginary data paths. Those are implemented with four SRL2 per input bit, which are in effect packed into two LUTs by taking advantage of the SRL compression capabilities.

The complex MAC is finally completed by adding an accumulator on each of the P_I and P_Q outputs. Again this accumulator works on large data widths and is therefore better integrated within a DSP48 slice. The corresponding implementations for 7 Series and UltraScale devices are shown in Figure 4, which once again demonstrates the benefit of the W-mux integration. The P_I and P_Q DSP48E2 slices absorb the accumulators, with 40 percent resource savings. It is worth mentioning that the latency is also reduced, which may be beneficial for some applications.

Using a similar construction, you can build a complex filter (one with complex data and coefficients) with three real filters, as depicted in Figure 5. The real and imaginary parts of the input signal are fed into two real filters, with coefficients derived respectively as the difference and sum of the imaginary and real parts of the filter coefficients. The third filter processes the sum of the input real and imaginary parts in parallel, using the real part of the coefficients.

The outputs of these three filters are finally combined to generate the real and imaginary components of the output, which can again benefit from the W-mux, when parallel filters need to be built, which is typically the case for the equalizers used in DFE applications.

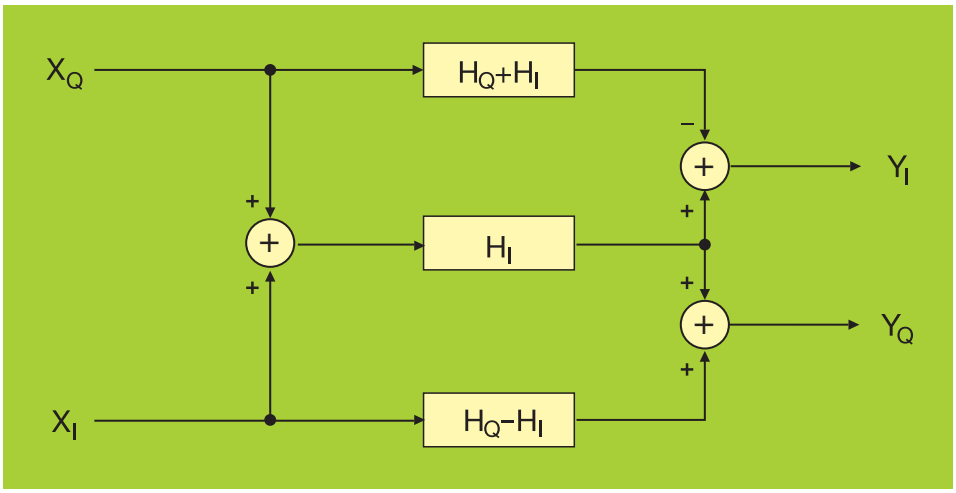


Figure 5 – Implementation architecture of a complex filter

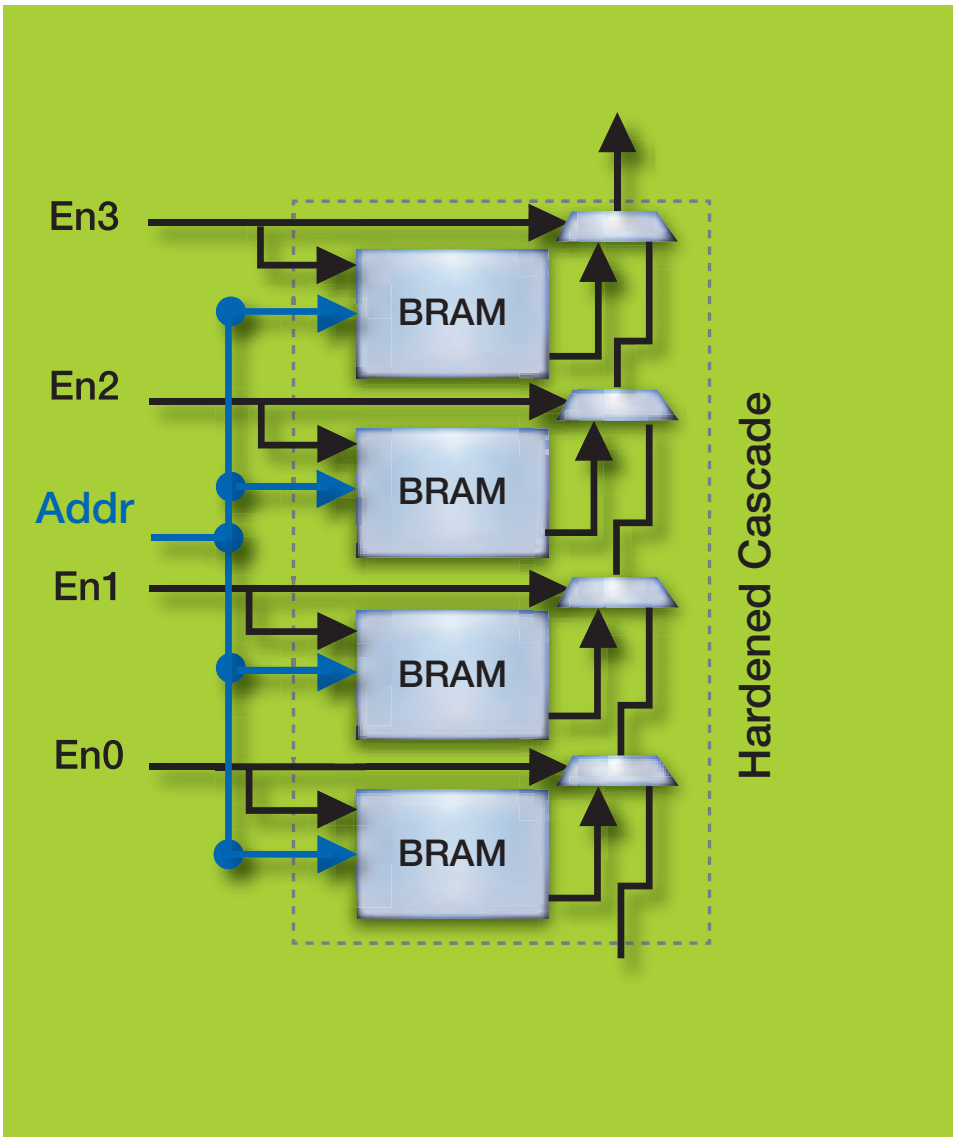


Figure 6 – BRAM cascade on UltraScale devices

BENEFITS OF THE ULTRASCALE MEMORY ARCHITECTURE

The Block RAMs integrated in UltraScale devices are essentially the same as in the 7 Series, but the new architecture introduces a hardware data-cascading scheme, together with a dynamic power-gating capability. Figure 6 illustrates this cascade, showing the data multiplexers embedded between every lower and upper adjacent Block RAM in a column. Larger memories can therefore be built in a bottom-up fashion without additional use of logic resources.

The cascade covers each entire column across the device, but its usage is better limited to a single clock region (that is, 12 successive BRAMs) to avoid clock skew and to maximize timing performance. Full flexibility is also available to support different implementations of the cascade feature. In effect, you can apply the multiplexer either to the Block RAM data input or to the output after or before the optional register.

The cascade opens up the possibility of building large memories requiring more than one BRAM, while simultaneously supporting minimal footprint, highest clock rate and minimal power, which is not feasible with 7 Series devices. For example, a 16K memory storing 16-bit data is better implemented with eight BRAMs (36K) configured as 16Kx2-bit on a 7 Series device to avoid external data multiplexing, which would add logic resources and latency, and could impact timing and routing congestion. This is unfortunately the less-efficient approach from a dynamic-power perspective, since the eight Block RAMs are enabled during any read or write operation. The optimal solution consists of using a 2Kx16-bit configuration, since only a single BRAM is then enabled, which divides the dynamic power by a factor 8. This is precisely what the cascade feature enables on the UltraScale devices, together with the dynamic power-gating capability.

Another direct application of the Block RAM cascade is related to the implementation of I/Q data-switching

functionality, commonly integrated with the baseband CPRI interfacing of DFE systems. Figure 7 shows the high-level switching architecture, which essentially consists of an NxM memory array. The successive data on the N ingress streams are written into the appropriate Block RAM in a line according to their output destination, and the M egress streams are read out from the appropriate Block RAM in a column. Consequently, each column can effectively be implemented with the BRAM cascade.

For more information on the 20-nm UltraScale family, visit *http://www.xilinx.com/products/silicon-devices/fpga/index.htm*.

References

1. Xilinx backgrounder, "Introducing UltraScale Architecture: Industry's First ASIC-Class All Programmable Architecture," July 2013
2. Xilinx white paper WP435, "Xilinx UltraScale: The Next Generation Architecture for Your Next Generation Architecture," July 8, 2013
3. Xilinx datasheet DS890, "UltraScale Architecture and Product Overview," Feb. 6, 2014
4. Xilinx backgrounder, "9 Reasons why the Vivado Design Suite Accelerates Design Productivity," July 2013
5. Xilinx user guide UG949, "Design Methodology Guide for the Vivado Design Suite," July 5, 2013
6. Xilinx white paper WP445, "Enabling High-Speed Radio Designs with Xilinx All Programmable FPGAs and SoCs," Jan. 20, 2014
7. Xilinx user guide UG579, "UltraScale Architecture—DSP Slice, Advance Specification User Guide," Dec. 10, 2013
8. IEEE Computer Society, "IEEE Standard for Floating Point Arithmetic, IEEE Std 754-2008," Aug. 29, 2008

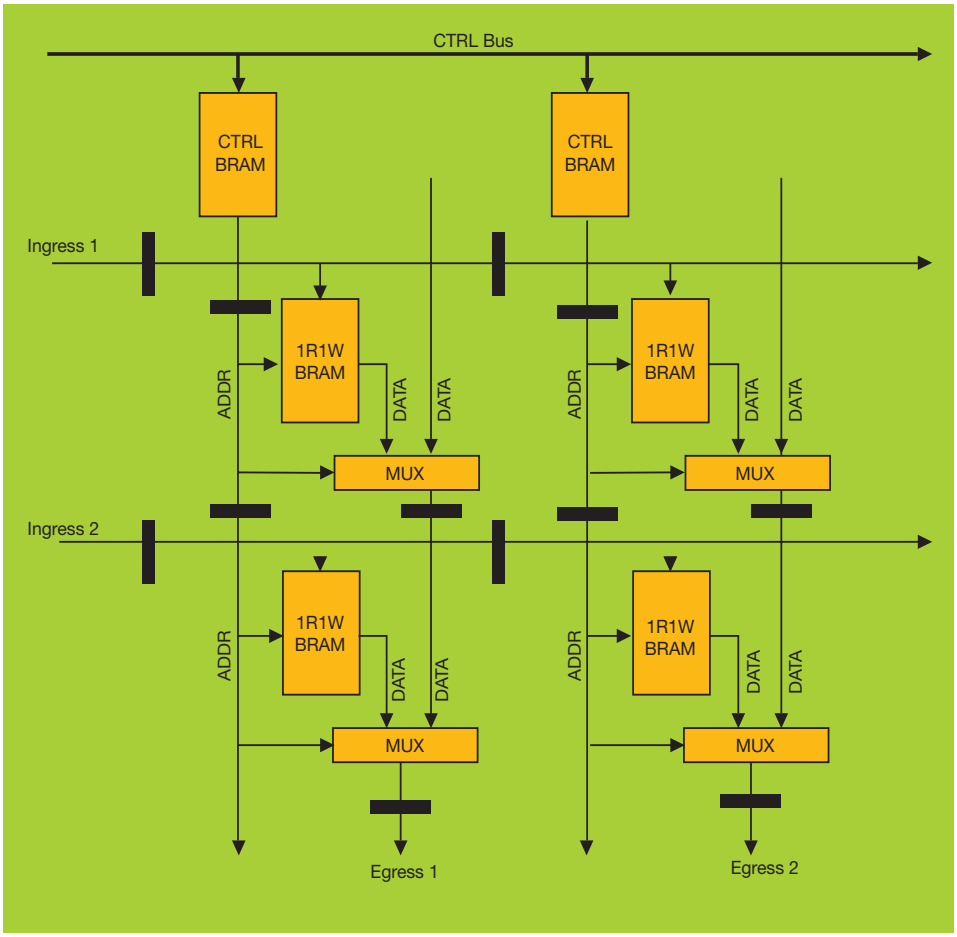


Figure 7 – Data-switching high-level architecture

All Programmable
FPGA and SoC modules

rugged for harsh environments
extended device life cycle

Available SoMs:
ZYNQ KINTEX 7
ARTIX 7 SPARTAN 6

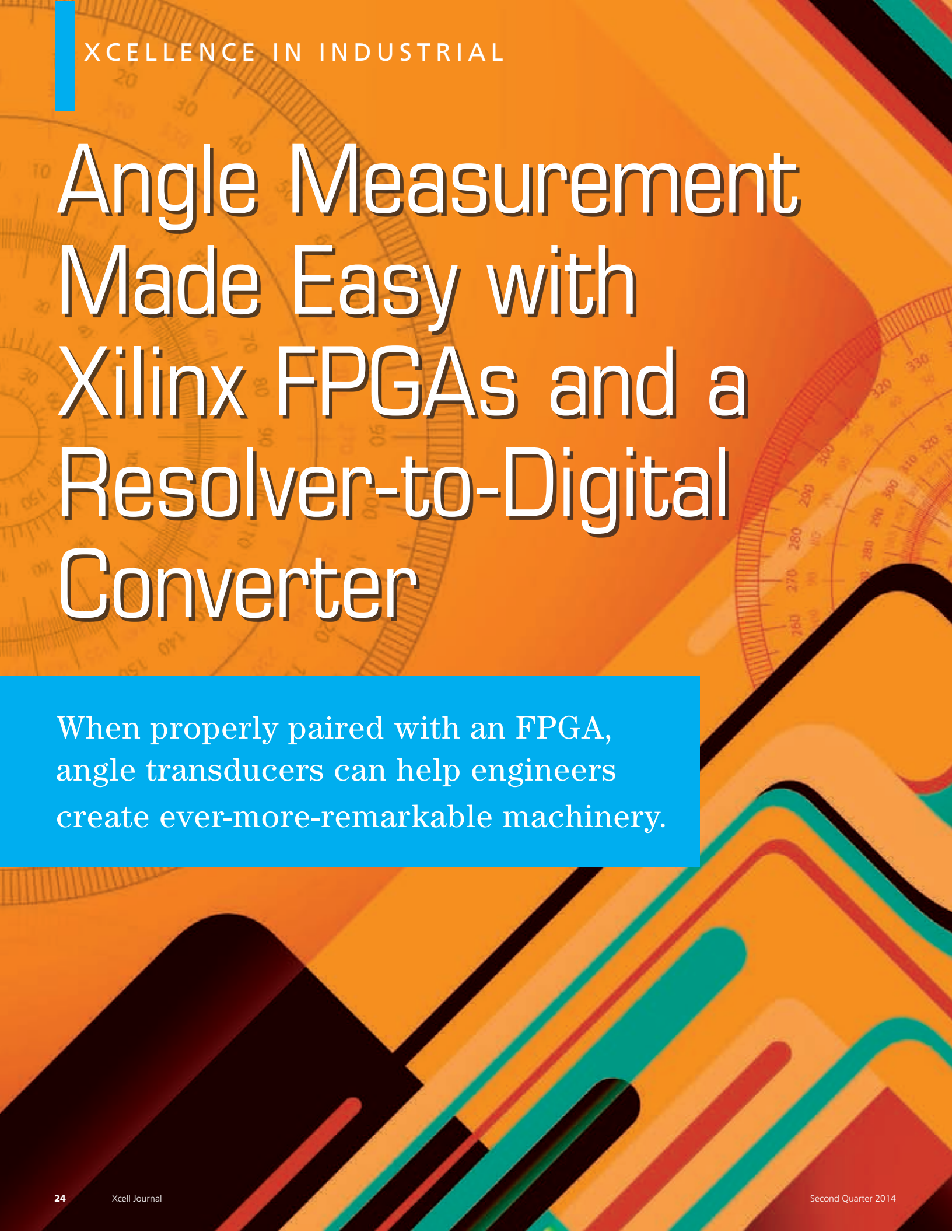
Platform Features
- 4×5 cm compatible footprint
- up to 8 Gbit DDR3 SDRAM
- 256 Mbit SPI Flash
- Gigabit Ethernet
- USB option

XILINX
ALLIANCE PROGRAM
CERTIFIED MEMBER — BASE

Design Services
- Module customization
- Carrier board customization
- Custom project development

trenz electronic
difference by design
www.trenz-electronic.de

Angle Measurement Made Easy with Xilinx FPGAs and a Resolver-to-Digital Converter



When properly paired with an FPGA, angle transducers can help engineers create ever-more-remarkable machinery.

by N. N. Murty
Scientist "F"

S. B. Gayen
Scientist "F"

Manish Nalamwar
Scientist "D"
nalamwar.manishkumar@rcilab.in

K. Jhansi Lakshmi,
Technical Officer "C"
Radar Seeker Laboratory
Research Centre IMARAT
Defense Research Development Organization
Hyderabad, India

Ever since humans invented the wheel, we have wanted to know, with varying degrees of accuracy, how to make wheels turn more efficiently. Over the course of the last few centuries, scientists and engineers have studied and devised numerous ways to accomplish this goal, as the basic principles of the wheel-and-axle system have been applied to virtually every mechanical system, from cars to stereo knobs to cogs in all forms of machinery, to the humble wheelbarrow [1].

Over these many eras, it turns out the most essential element in making a wheel turn efficiently is not the wheel itself (why reinvent it?) but the shaft angle of the wheel. And the most effective way to measure and optimize a shaft angle today is through the use of angle transducers. There are many types of angle transducers that help optimize wheel efficiency through axle monitoring and refinements, but by applying FPGAs to the task you can achieve remarkable results and improve axle/wheel efficiencies in a broad number of applications.

Before we get into the details of how engineers are doing this optimally with Xilinx® FPGAs, let's briefly review some basic principles of angle transducers. Today there are two widely used varieties: encoders and resolvers.

TYPES OF ENCODERS AND RESOLVERS

Encoders fall into two basic categories: incremental and absolute. Incremental encoders monitor two positions on an axle and create an A or B pulse each time the axle passes those positions. A separate external electric counter then interprets those pulses for speed and rotational direction. Incremental counters are useful in a number of applications, but they do have some disadvantages. For example, when the axle is powered off, an incremental encoder must first calibrate itself by returning to a desig-

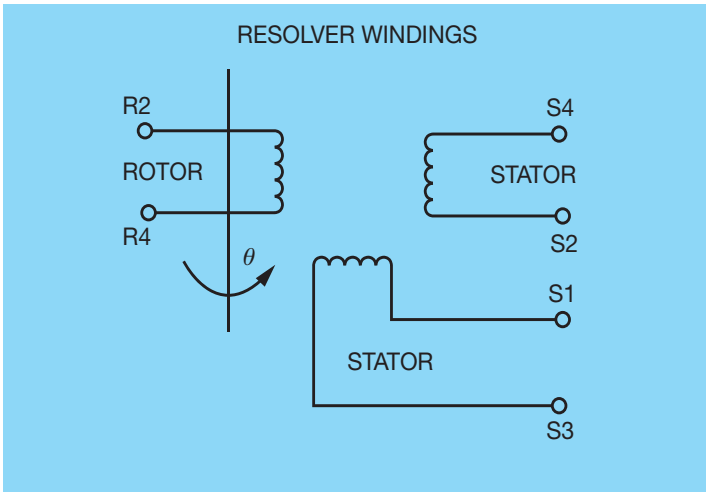


Figure 1 – Excitation to the rotor of a resolver

nated calibration point before beginning operation. Incremental counters are also susceptible to electrical interference, which can result in inaccuracies in pulses they send to the system and thus in rotation counts. Moreover, many incremental encoders are photoelectric devices, which precludes their use in radiation-hazardous areas, if that is a concern for your targeted application.

Absolute encoders are sensor systems that monitor the rotation count and direction of an axle. In an absolute-encoder-based system, users typically attach a wheel to an axle that has an electrical contact or photoelectric reference. When the axle is in operation, the absolute-encoder-based system records the rotation and direction of the operation and generates a parallel digital output that is easily translated into code, most commonly binary or Gray code. Absolute encoders are useful in that they need to be calibrated only once—typically in the factory—and not before every use. Moreover, they are typically more reliable than other encoders. That said, absolute encoders are typically expensive, and they are not great with parallel data transmission, especially if the encoder is located far away from the electronic system measuring its readings.

A resolver, for its part, is a rotary transformer—an analog device whose output voltage is uniquely related to the input shaft angle it is monitoring. It is an absolute-position transducer with 0° to 360° of rotation that connects directly to the axle and reports speed and positioning. Resolvers have a number of advantages over encoders. They are robust devices that can withstand harsh environments marked by dust, oil, temperature extremes, shock and radiation. Being a transformer, a resolver provides signal isolation and a natural common-mode rejection of electrical interference. In addition to these features, resolvers require only four wires for the angular data transmission, which suits them for everything from heavy manufacturing to miniature systems to those used in the aerospace industry.

A further refinement is the brushless resolver, which does not require slip-ring connections to the rotor. This type of resolver is therefore even more reliable and has a longer life cycle.

Resolvers use two methods to obtain output voltages related to the shaft angle. In the first method, the rotor winding, as shown in Figure 1, is excited by an alternating signal and the output is taken from the two stator windings. As the stator windings are mechanically positioned at right angles, the output signal amplitude is related by the trigonometric sine and cosine of the shaft angle. Both the

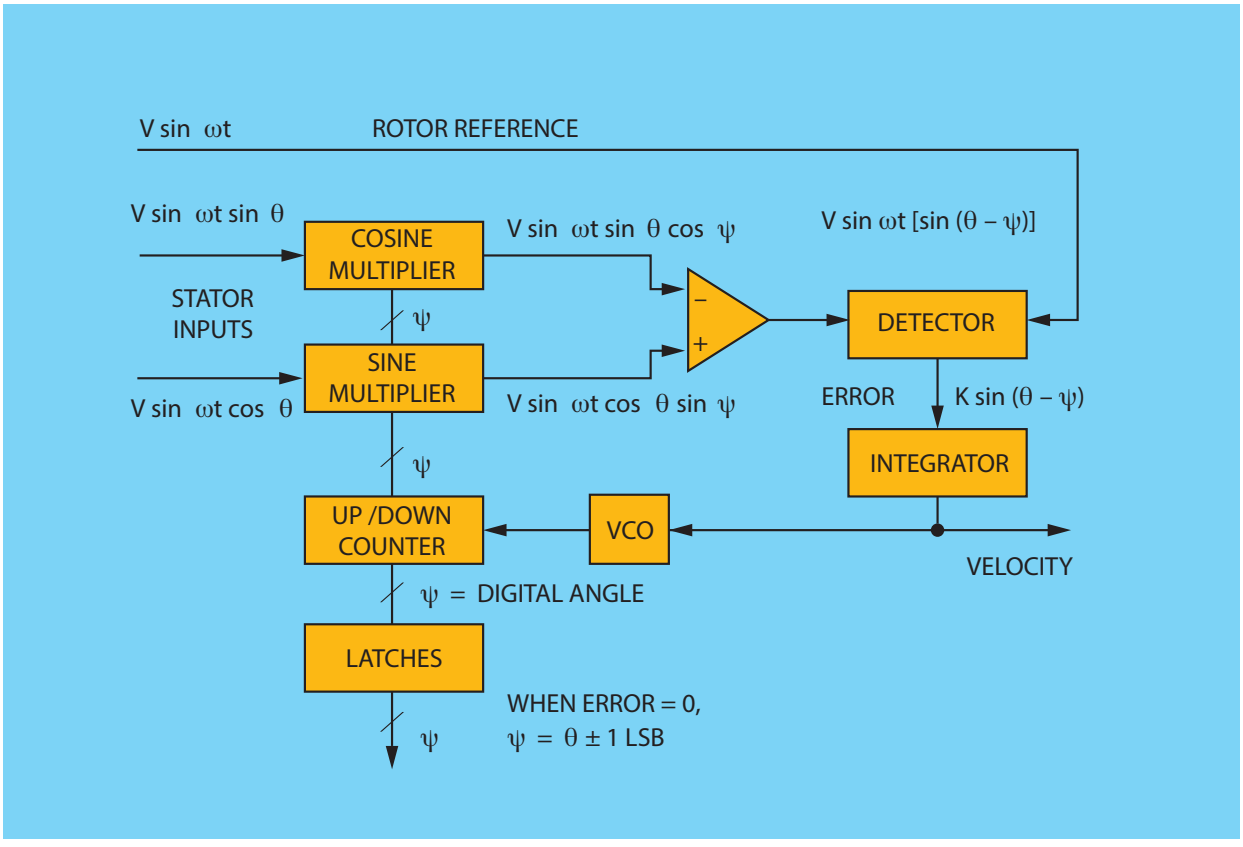


Figure 2 – Resolver-to-digital converter (RDC) block diagram

sine and cosine signals have the same phase as the original excitation signal; only their amplitudes are modulated by sine and cosine as the shaft rotates.

In the second method, a stator winding is excited with the alternating signals, which are in phase quadrature to each other. Then a voltage induces in the rotor winding. The winding's amplitude and frequency are fixed, but its phase shift varies with the shaft angle.

The resolver can be positioned where the angle needs to be measured [2]. The electronics, generally a resolver-to-digital converter (RDC), can be positioned where the digital output needs to be measured. Analog output from the resolver, which contains the angular position information of the shaft, is then transformed in digital form using the RDC.

FUNCTIONALITY OF THE TYPICAL RDC

In general, the two outputs of a resolver are applied to the sine and cosine multiplier of the RDC [3]. These multipliers incorporate sine and cosine lookup tables and function as multiplying digital-to-analog converters. Figure 2 shows their functionality.

Let us assume, in the beginning, that the current state of the up/down counter is a digital number representing a trial angle, ψ. The converter seeks to adjust the digital angle, ψ, continuously to become equal to and track θ, the analog angle being measured.

The stator output voltage of the resolver is:

$V1 = V \sin\omega t \sin\theta$ Eq. 1

$V2 = V \sin\omega t \cos\theta$ Eq. 2

where θ is the angle of the resolver's rotor. The digital angle ψ is applied to the cosine multiplier and its cosine is multiplied by V1 to produce the term:

$V \sin\omega t \sin\theta \cos\psi.$ Eq. 3

The digital angle ψ is also applied to the sine multiplier and multiplied by V2 to produce the term:

$V \sin\omega t \cos\theta \sin\psi.$ Eq. 4

These two signals are subtracted from each other by the error amplifier to yield an ac error signal of the form:

$(V \sin\omega t \sin\theta\cos\psi - V \sin\omega t \cos\theta \sin\psi)$ Eq. 5

$V \sin\omega t (\sin\theta \cos\psi - \cos\theta \sin\psi)$ Eq. 6

From trigonometric identity, this reduces to:

$V \sin\omega t [\sin(\theta - \psi)]$ Eq. 7

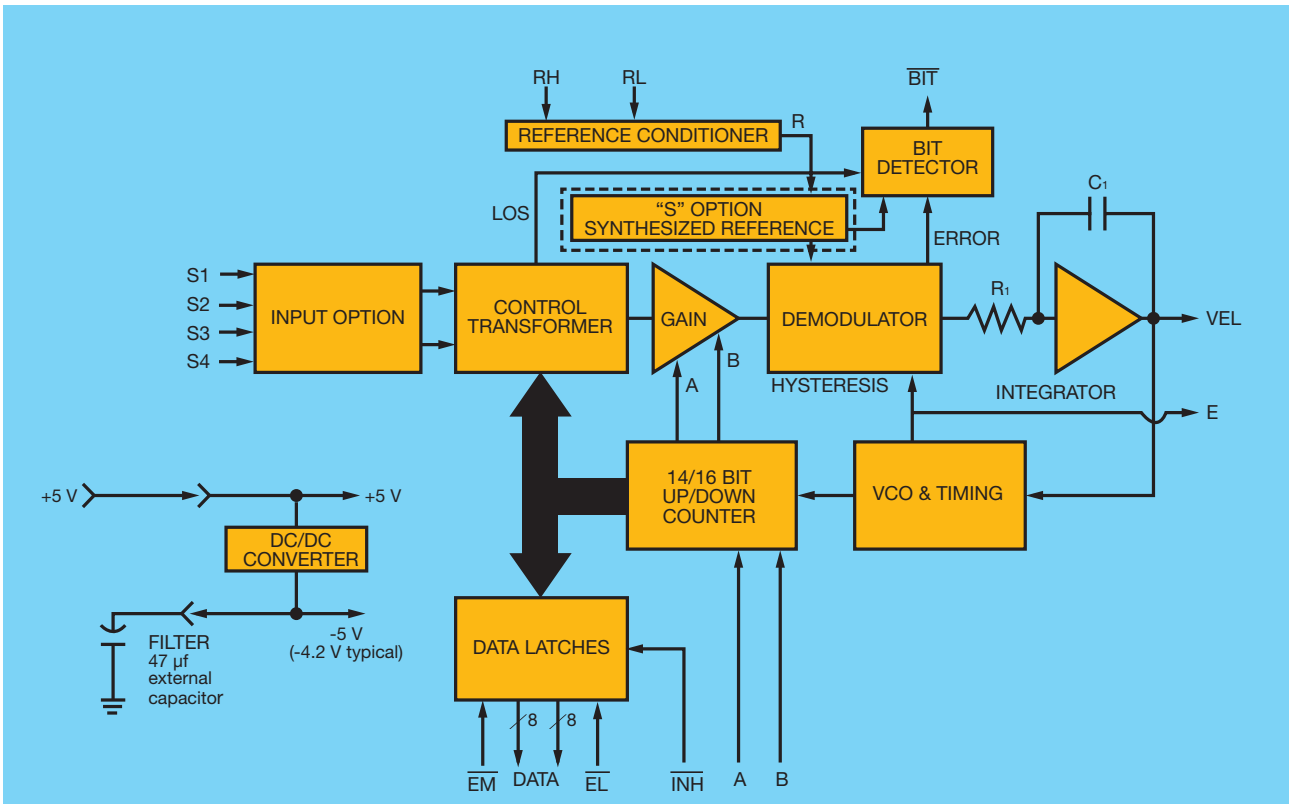


Figure 3 – SD-14620 block diagram (one channel)

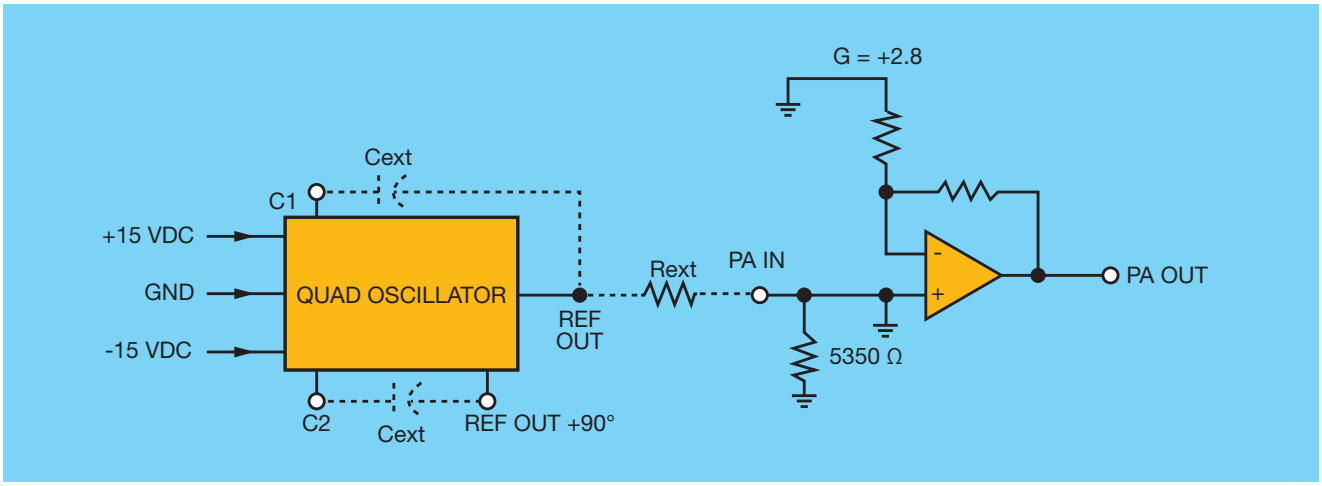


Figure 4 – Block diagram of the OSC-15802 reference oscillator

The detector synchronously demodulates this ac error signal, using the resolver's rotor voltage as a reference. This results in a dc error signal proportional to sin (θ -ψ).

The dc error signal feeds an integrator, the output of which drives a voltage-controlled oscillator. The VCO, in turn, causes the up/down counter to count in the proper direction to cause:

$$\sin(\theta - \psi) \rightarrow 0. \quad \text{Eq. 8}$$

When this result is achieved,

$$\theta - \psi \rightarrow 0, \quad \text{Eq. 9}$$

and therefore

$$\theta = \psi \quad \text{Eq. 10}$$

in one count. Hence, the counter's digital output, ψ, represents the angle θ. The latches make it possible to transfer this data externally without interrupting the loop's tracking.

This circuit is equivalent to a Type 2 servo loop, as it has in effect two integrators. One is the counter, which accumulates pulses; the other is the integrator at the output of the detector. In a Type 2 servo loop with a constant-rotational-velocity input, the output digital word continuously follows, or tracks, the input, without needing externally derived conversion.

TYPICAL EXAMPLE OF AN RDC: THE SD-14621

The SD-14621 is a small, low-cost, RDC from Data Device Corp. (DDC). It has two channels with programmable resolution control. Resolution programming allows selection of 10-, 12-, 14- or 16-bit modes [4]. This feature allows low resolution for fast tracking or higher resolution for higher accuracy. Thanks to its size, cost, accuracy and versatility, this converter is suitable for high-performance military, commercial and position-control systems.

A single +5 V is required for device operation. The converter has velocity outputs (VEL A, VEL B) of a voltage range of ±4 V with respect to analog ground, which can be used to replace a tachometer. Two built-in test outputs are provided for two channels (/BIT A and /BIT B) to indicate loss of signal (LOS).

This converter has three main sections: an input front end, an error processor and a digital interface. The front end differs for synchro, resolver and direct inputs. An electronic Scott-T is used for synchro inputs, a resolver conditioner for resolver inputs and a sine-and-cosine voltage follower for direct inputs. These amplifiers feed the high-accuracy control transformer (CT). The other input of the CT is a 16-bit digital angle ψ and the output is an analog error angle, or a difference angle, between the two inputs. The CT performs the ratiometric trigonometric computation of SINθ COSψ - COSθ SINψ = Sin(θ-ψ) using amplifiers, switches, logic and capacitors in precision ratios.

Compared with a conventional precision resistor, these capacitors are used in precision ratios to get enhanced accuracy. Further, these capacitors (which are used with an op amp as a computing element) sample at high rates to eliminate drift and op-amp offsets.

The DC error processing is integrated, yielding a velocity voltage that drives a voltage-controlled oscillator. This VCO is an incremental integrator when it is combined with the velocity integrator: a Type 2 servo feedback loop.

REFERENCE OSCILLATOR

The power oscillator in our design, also from DDC, is the OSC-15802. This device is suitable for RDC, synchro, LVDT, RVDT and inductosyn applications [5]. The frequency and amplitude outputs are programmable with capacitors and resistors, respectively. The output frequency range is 400 Hz to

The I/O voltage of the FPGA is 3.3 V, while the RDC's voltage is 5 V. We used voltage transceivers to achieve voltage compatibility between the two devices.

10 kHz, with an output voltage of 7 Vrms. Figure 4 shows a block diagram of the device.

The oscillator output, which is given to the resolver and the RDC, works as a reference signal.

VIRTEX-5 FX30T FPGA AND RDC INTERFACE

For our design, we used a Xilinx Virtex®-5 FX30T FPGA [6]. The I/O voltage of the FPGA is 3.3 V, while the RDC's voltage is 5 V. We used voltage transceivers to achieve voltage compatibility between the two devices. Internal connections with the FPGA are established through the GPIO IP core provided by Xilinx, as seen in Figure 5.

For simplicity's sake, Figure 5 shows just one channel with the single resolver interface. You will find the pin details of the RDC and corresponding pin locking with the FPGA in the Xilinx Board Description (XBD) file that accompanies this article. The details are listed in Section 1 of that document.

DETAILS OF THE DEVICE DRIVER

In this case, we used an external input clock of 20 MHz for the FPGA. This FPGA has a hard PowerPC® 440 core that is running at a 200-MHz frequency. The timing diagram of the RDC is shown in Figure 6 and Figure 7.

In accordance with the timing diagram of the RDC, we developed, tested and confirmed correct functionality with the actual hardware [4]. The actual code of the device driver is included in the separate XBD file. As per the timing diagram, we generated required delays using for loops. When processing is running at 200 MHz, each count corresponds to a delay of 5 nanoseconds.

The device driver has three sections of code: RDC initialization; generation of a control signal and reading from channel A of the RDC; and generation of a control signal and reading from channel B. RDC initialization is the point at which the direction of the signal and default values are

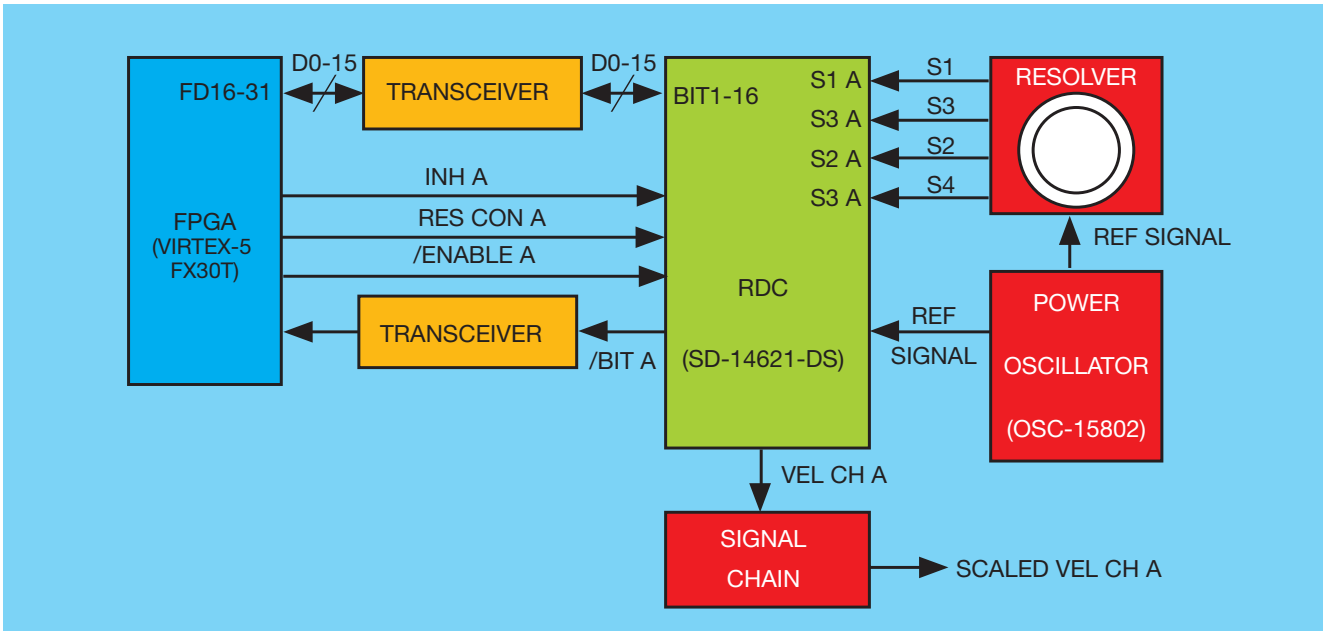


Figure 5 – RDC interface with the Virtex-5 FPGA (single channel)

set. For example, with the following statement, the direction is set as "out" from the FPGA to the RDC.

```
XGpio_WriteReg(XPAR_RESOLUTION_CNTRL_CH_A_
  BASEADDR,XGPIO_TRI_OFFSET,0x000);
```

With the next statement, the 16-bit resolution is set by writing "0x3" (that is, pulling high):

```
XGpio_WriteReg(XPAR_RESOLUTION_CNTRL_CH_A_
  BASEADDR,XGPIO_DATA_OFFSET,0x03);
```

Figure 8 shows a snapshot of the coding. Note: for simplification, we have included code for only one channel.

As we have seen, angle transducers help engineers create a better wheel and thus a plethora of more efficient machinery. Resolvers are an especially useful type of angle transducer, and when properly paired and controlled with an FPGA, can help engineers create even more remarkable machinery.

References

1. "Synchro/Resolver Conversion Handbook," Data Device Corp.
2. John Gasking, "Resolver-to-Digital Conversion: A Simple and Cost-Effective Alternative to Optical Shaft Encoders," AN-263, Analog Devices
3. Walt Kester, "Resolver to Digital Converter," MT-030, Analog Devices
4. SD-14620 Series Data Sheet, Data Device Corp.
5. OSC-15802 Data Sheet, Data Device Corp.
6. "Virtex-5 Family Overview," Xilinx

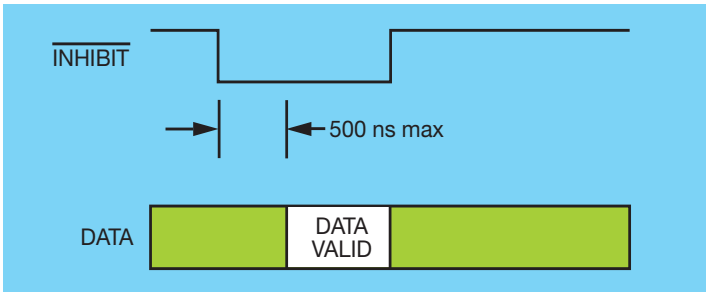


Figure 6 – INHIBIT timing

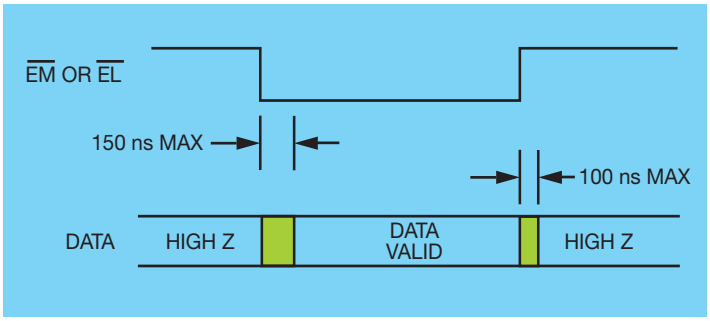


Figure 7 – ENABLE timing

```
seXGpio_WriteReg(XPAR_INHIBIT_CH_A_
BASEADDR,XGPIO_DATA_OFFSET,0x01);
for(i=0;i<=5;i++); //gives delay of 25 ns
          XGpio_WriteReg(XPAR_ENABLE_LSB_
CH_A_BIT_BASEADDR,XGPIO_DATA_OFFSET,0x01);

for(i=0;i<=5;i++);
XGpio_WriteReg(XPAR_INHIBIT_CH_A_BASEAD-
DR,XGPIO_DATA_OFFSET,0x00);
for(i=0;i<=2;i++);
XGpio_WriteReg(XPAR_ENABLE_LSB_CH_A_BIT_
BASEADDR,XGPIO_DATA_OFFSET,0x00);

for(i=0;i<=2;i++);
lsb_val=XGpio_ReadReg(XPAR_RDC_DATA_15_
TO_0_PINS_BASEADDR,XGPIO_DATA_OFFSET);

XGpio_WriteReg(XPAR_INHIBIT_CH_A_BASEAD-
DR,XGPIO_DATA_OFFSET,0x01);
      for(i=0;i<=5;i++);
XGpio_WriteReg(XPAR_ENABLE_LSB_CH_A_BIT_
BASEADDR,XGPIO_DATA_OFFSET,0x01);
for(i=0;i<=25;i++);

XGpio_WriteReg(XPAR_INHIBIT_CH_A_BASEAD-
DR,XGPIO_DATA_OFFSET,0x01);
for(i=0;i<=5;i++);
XGpio_WriteReg(XPAR_ENABLE_MSB_CH_A_BIT_
BASEADDR,XGPIO_DATA_OFFSET,0x01);
for(i=0;i<=5;i++);

XGpio_WriteReg(XPAR_INHIBIT_CH_A_BASEAD-
DR,XGPIO_DATA_OFFSET,0x00);
      for(i=0;i<=2;i++);
XGpio_WriteReg(XPAR_ENABLE_MSB_CH_A_BIT_
BASEADDR,XGPIO_DATA_OFFSET,0x00);
for(i=0;i<=2;i++);
msb_val=XGpio_ReadReg(XPAR_RDC_DATA_15_
TO_0_PINS_BASEADDR,XGPIO_DATA_OFFSET);

  lsb_val=lsb_val & 0x00ff;

  msb_val=msb_val & 0xff00;

  rdccount_cha = msb_val | lsb_val;

XGpio_WriteReg(XPAR_INHIBIT_CH_A_BA-
SEADDR,XGPIO_DATA_OFFSET,0X01);
for(i=0;i<=5;i++);

XGpio_WriteReg(XPAR_ENABLE_MSB_CH_A_
BIT_BASEADDR,XGPIO_DATA_OFFSET,0x01);
for(i=0;i<=20;i++);
```

Figure 8 – A snapshot of RDC device driver code

TRACE32®

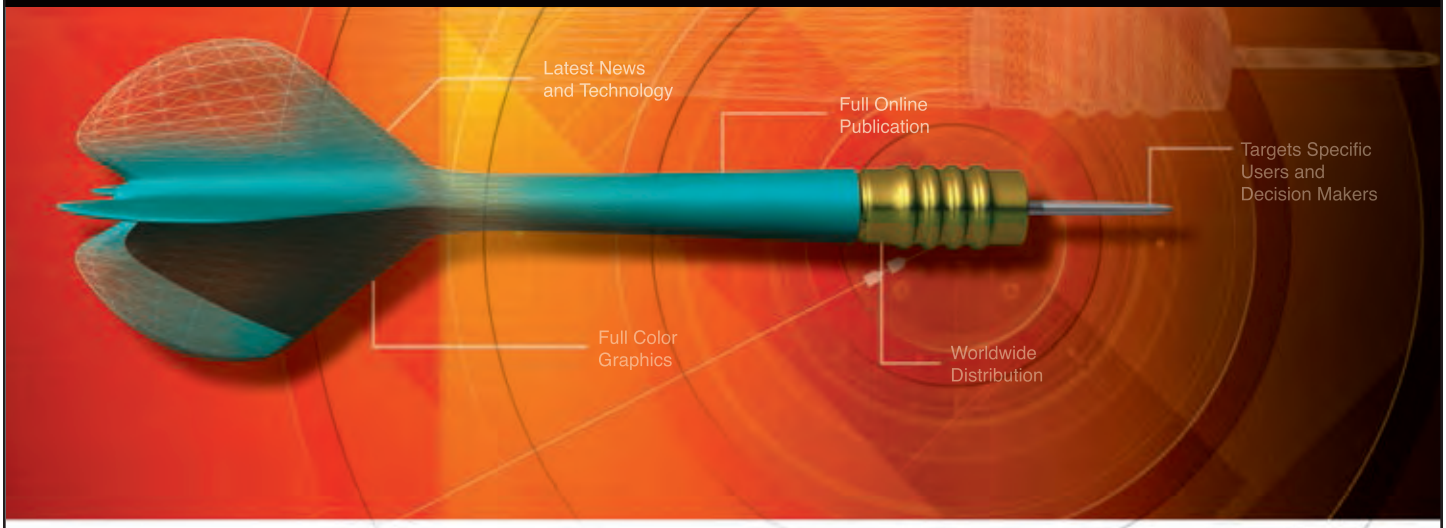
Debugging Xilinx's Zynq™ -7000 family with ARM CoreSight

- RTOS support, including Linux kernel and process debugging
- SMP/AMP multicore Cortex™-A9 MPCore™s debugging
- Up to 4 GByte realtime trace including PTM/ITM
- Profiling, performance and statistical analysis of Zynq's multicore Cortex™ -A9 MPCore™

LAUTERBACH
DEVELOPMENT TOOLS

www.lauterbach.com

Is your marketing message reaching the right people?



Hit your target by advertising your product or service in the Xilinx *Xcell Journal*, you'll reach thousands of qualified engineers, designers, and engineering managers worldwide.

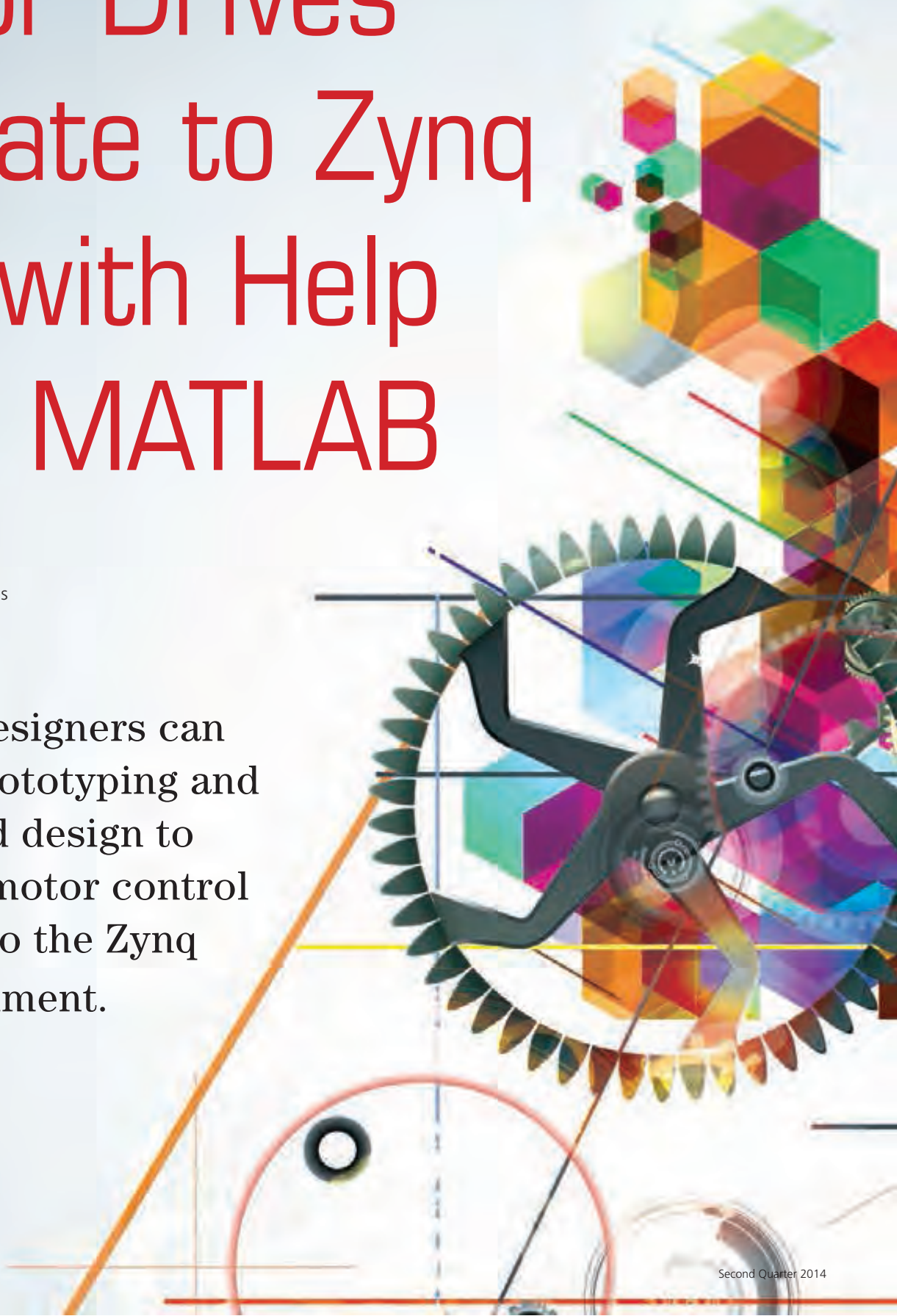
Call today: (800) 493-5551 or e-mail us at *xcelladsales@aol.com*

Xcell PUBLICATIONS

www.xilinx.com/xcell

Motor Drives Migrate to Zynq SoC with Help from MATLAB

by Tom Hill
Senior Manager, DSP Solutions
Xilinx, Inc.
tom.hill@xilinx.com



Industrial designers can use rapid prototyping and model-based design to move their motor control algorithms to the Zynq SoC environment.

Since the 1990s, developers of motor drives have been using a multichip architecture to implement their motor control and processing requirements. In this architecture, a discrete digital signal-processing (DSP) chip executes motor control algorithms, an FPGA implements high-speed I/O and networking protocols, and a discrete processor handles executive control. With the advent of the Xilinx® Zynq®-7000 All Programmable SoC, however, designers have the means to consolidate these functions into a single device while integrating additional processing tasks. The reduction in parts count and complexity makes it possible to lower system cost while improving performance and reliability.

But how can drive developers evolve their established design practices to leverage the Zynq SoC?

Industrial designers have long embraced model-based design for the development of custom motor algorithms on DSP chips through the use of simulation and C-code generation. Now, a new workflow from MathWorks—developed in conjunction with Xilinx—extends model-based design to the processing system and programmable logic available with the Zynq-7000 All Programmable SoC.

ZYNQ SOCS FOR MOTOR CONTROL

Today's advanced motor control systems are a combination of control algorithms and industrial networks, including EtherCAT, Profinet, Powerlink and Sercos III, that draw processing bandwidth from the computing resources. Moreover, other requirements are converging into the control system including motion-control layers, PLC layers, diagnostic layers and user interfaces for commission and maintenance or remote monitoring. These requirements translate into logical and physical partitions with elements that fit naturally into the processing systems while other elements best fit into the hardware-assisted offloading and acceleration.

The hardware platform you select should provide a robust and scalable system. Xilinx's Zynq SoCs fulfill these requirements by supplying a high-performance processing system to address the networking, motion, soft-PLC, diagnostic and remote-maintenance functions combined with programmable logic to accelerate performance-critical functions in hardware. On the processing side, the Zynq SoC

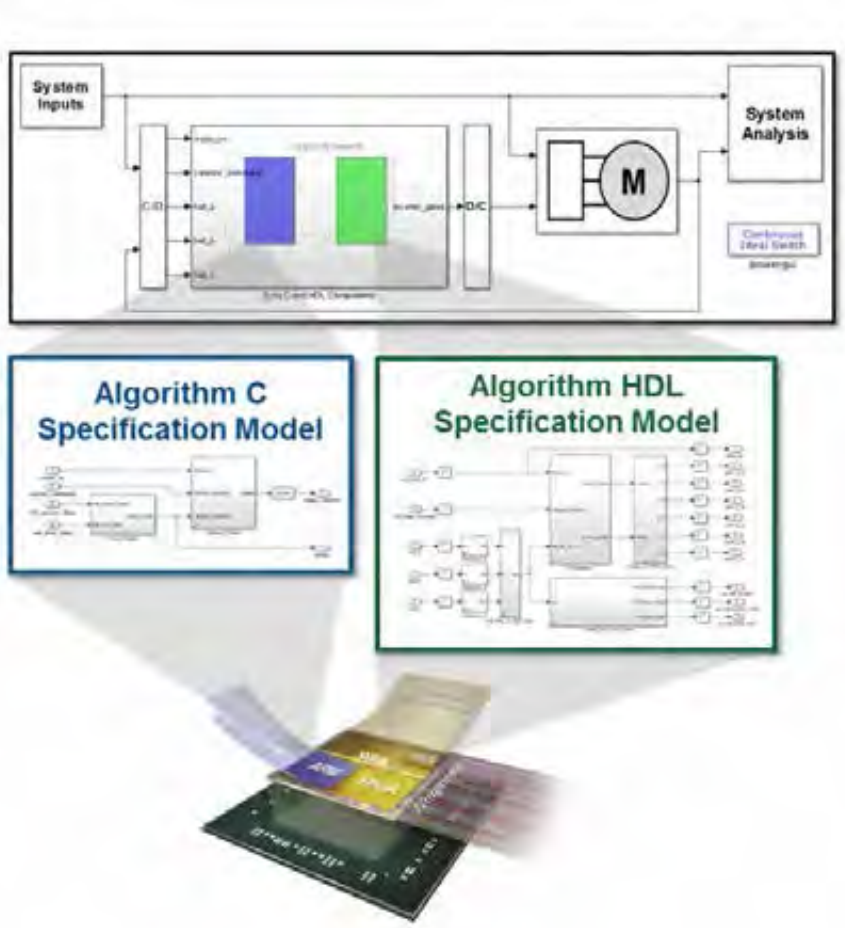


Figure 1 – MathWorks' workflow targeting the Zynq SoC, using C and HDL code generation

combines a dual-core ARM® Cortex™-A9 processing system with a NEON coprocessor and floating-point extensions to accelerate software execution. On the programmable logic side, the device has up to 444,000 logic cells and 2,200 DSP48 slices that supply massive processing bandwidth. Five high-throughput AMBA®-4 AXI high-speed interconnects tightly couple the programmable logic to the processing system with the equivalent of more than 3,000 pins of effective bandwidth.

Table 1 lists the processing performance that Zynq SoC devices can achieve.

PLANT AND MOTOR MODELING USING SIMULINK AND THE CONTROL SYSTEMS TOOLBOX

Modern control algorithms have system times and system variables that span several orders of magnitude, making hardware/software partitioning a daunting, time-consuming and iterative task. Figure 2 depicts a typical electrical drive. The power source is normally 50 to 60 Hz and is rectified to achieve a continuous voltage (DC). This DC voltage is then converted into a variable frequency that controls the power stage that feeds the motor terminals. The controller also must read the motor's basic variables including current and voltages. It likewise must read or establish the shaft position including its speed and handling commands originating from the communication network or supervising controller.

Simulink® provides a block-diagram environment for multidomain system simulation and model-based design that is well suited to simulating systems that include control algorithms and plant models. MathWorks products such as the Control Systems Toolbox provide a variety of "apps" based on widely used methods of systematically analyzing, designing and tuning control systems modeled in Simulink. Performing system modeling in Simulink can accelerate development of motor control systems while reducing risk in the following ways:

- **Reduces risk of damage** – Simulation allows thorough examination of new control system algorithms before they are tested on production hardware, where there are risks of damaging drive electronics, motors and other system components.
- **Accelerates system integration** – Support staff must integrate new control system algorithms into the

Elements	Performance (up to)
Processors (each)	1 GHz
Processors (aggregate)	5,000 DMIPs
DSP (each)	741 MHz
DSP (aggregate)	2,662 GMACs
Transceivers (each)	12.5 Gbps
Transceivers (aggregate)	200 Gbps
Software acceleration	10x

Table 1 – Processing performance of the Zynq SoC

production system, meaning that deploying new controllers can consume their limited time and can make the deployment a protracted process.

- **Reduces dependency on equipment availability** – The production environment itself may not be available, such as in cases where custom drive electronics or electric motors are under development or are not located where control system designers can access them.

Given these factors, simulation provides an excellent alternative to testing on production hardware. Simulation environments such as Simulink provide a framework for creating plant models from preexisting libraries of building blocks of electromechanical components for the evaluation of new control system architectures against plant models.

Risk to the schedule is further reduced by linking the system model to a rapid-prototyping environment as well as the final production system. The rapid-prototyping flow enables algorithm developers to prototype without having to depend on hardware designers. Instead they use a platform-specific support package in a highly automated process that deploys the hardware and software components of the system to a design template that can be compiled to a specific hardware development platform. The hardware and software design teams can reuse these same hardware and software components in the final production systems without modification to accelerate development and reduce errors.

RAPID PROTOTYPING USING THE AVNET INTELLIGENT DRIVES KIT

Designers can pair the Avnet Zynq-7000 AP SoC / Analog Devices Intelligent Drives Kit with Simulink and the Zynq SoC workflow for a complete rapid-prototyping system for motor control applications. This kit combines the Zynq SoC with the latest generation of Analog Devices' high-precision data converters and digital isolation. The kit enables high-performance motor control and dual Gigabit Ethernet industrial networking connectivity (*http://www.xilinx.com/products/boards-and-kits/1-490M1P.htm*).

It comes with an Avnet ZedBoard 7020 baseboard; Analog Devices' AD-FMCMOTCON1-EBZ module, which is capable of driving brushless DC and stepper motors with a 24-volt external power supply (included with the kit); and a 24-V BLDC motor rated for 4,000 RPM and equipped with Hall-effect sensors and a 1,250-CPR indexed encoder. Also included are a Zynq SoC reference design of field-oriented control and Analog Devices' Ubuntu Linux framework including drivers, application software and source code.

EXAMPLE: TRAPEZOIDAL MOTOR CONTROL

Let's apply this workflow to the trapezoidal motor control system in Figure 1 using simulation in Simulink to evaluate a controller with a simulated plant, then prototype the controller using the Intelligent Drives Kit. As a final step, we will validate the Simulink model using results from hardware testing.

In this example, we will use the kit to drive an inertial load in the form of an aluminum disc, with a basic trapezoidal controller. The controller's main components are as follows:

- Hall-effect sensor – detects the motor position
- Velocity estimator – computes rotor velocity based on the sensor signal
- Six-step commutator – computes the phase voltages and inverter enable signals based on rotor position and velocity
- Pulse-width modulation (PWM) – drives the controller outputs out through the drive circuitry

We start by using a behavioral, control-loop model of the system suited to control-loop analysis. First we will evaluate the model in simulation by subjecting it to a pulse test, commanding a rotational rate of 150 radians per second for 2 seconds and then returning to a stop. Through tuning of the control loop's proportional-integral (PI) controller gains, we can achieve a settling time of 1.2 seconds with negligible overshoot (control-loop simulation results appear as the purple-shaded signal in Figure 3; details on this example are available at *mathworks.com/zidk*).

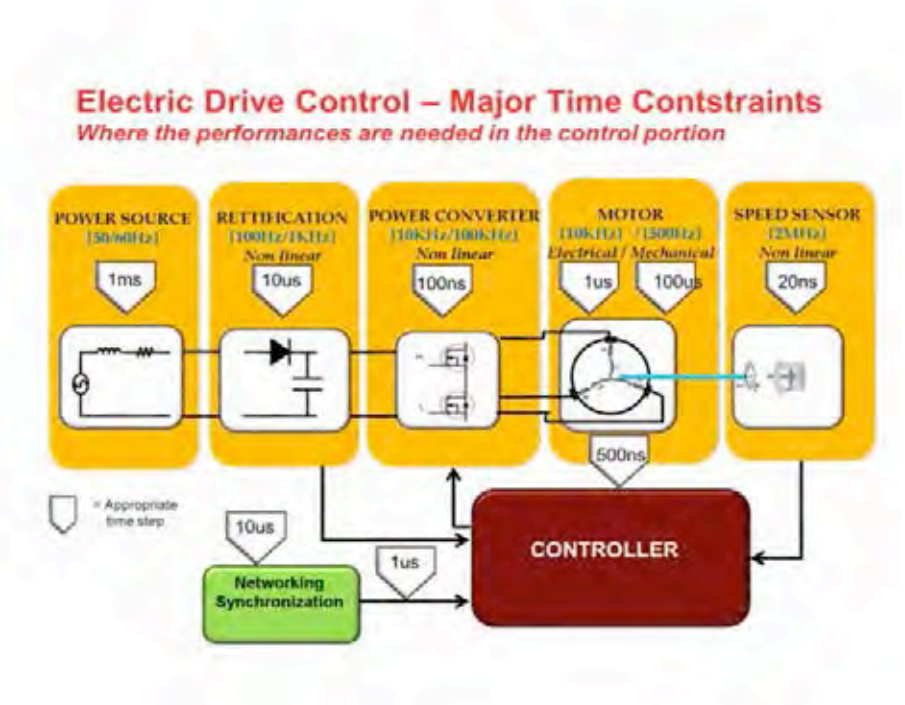


Figure 2 – Major time constraints of electrical drive controllers

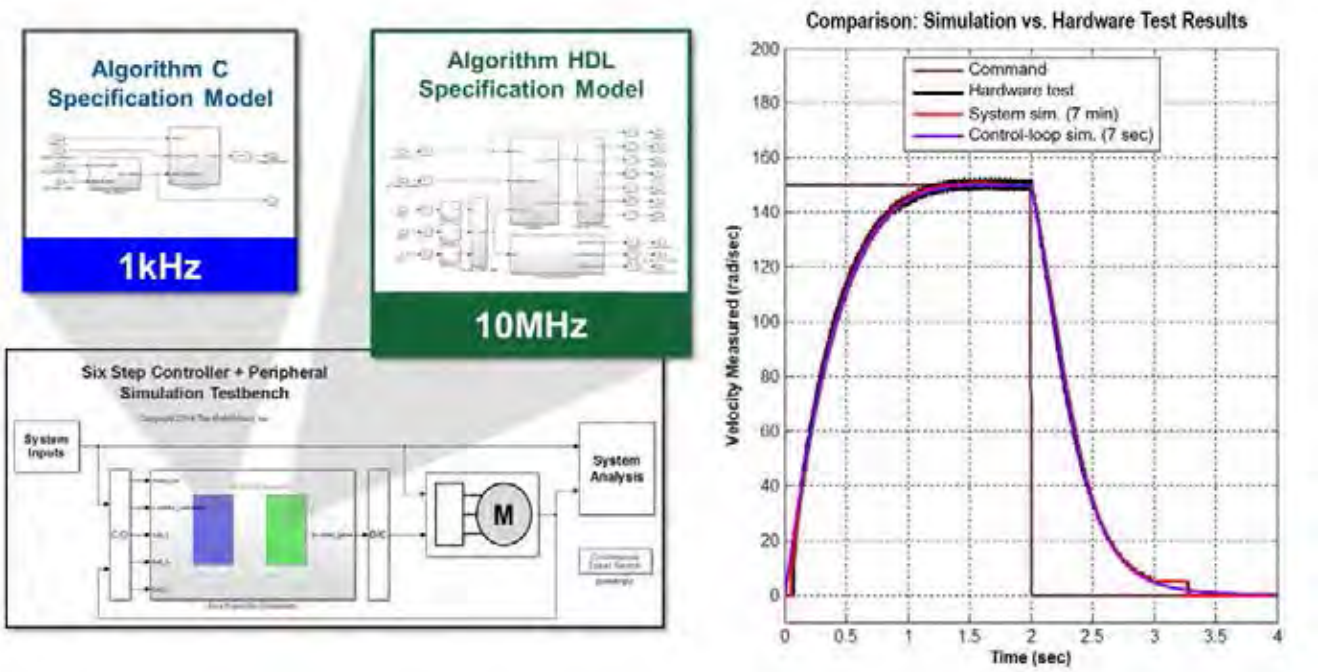


Figure 3 – Hardware and software simulation models used to validate against hardware results

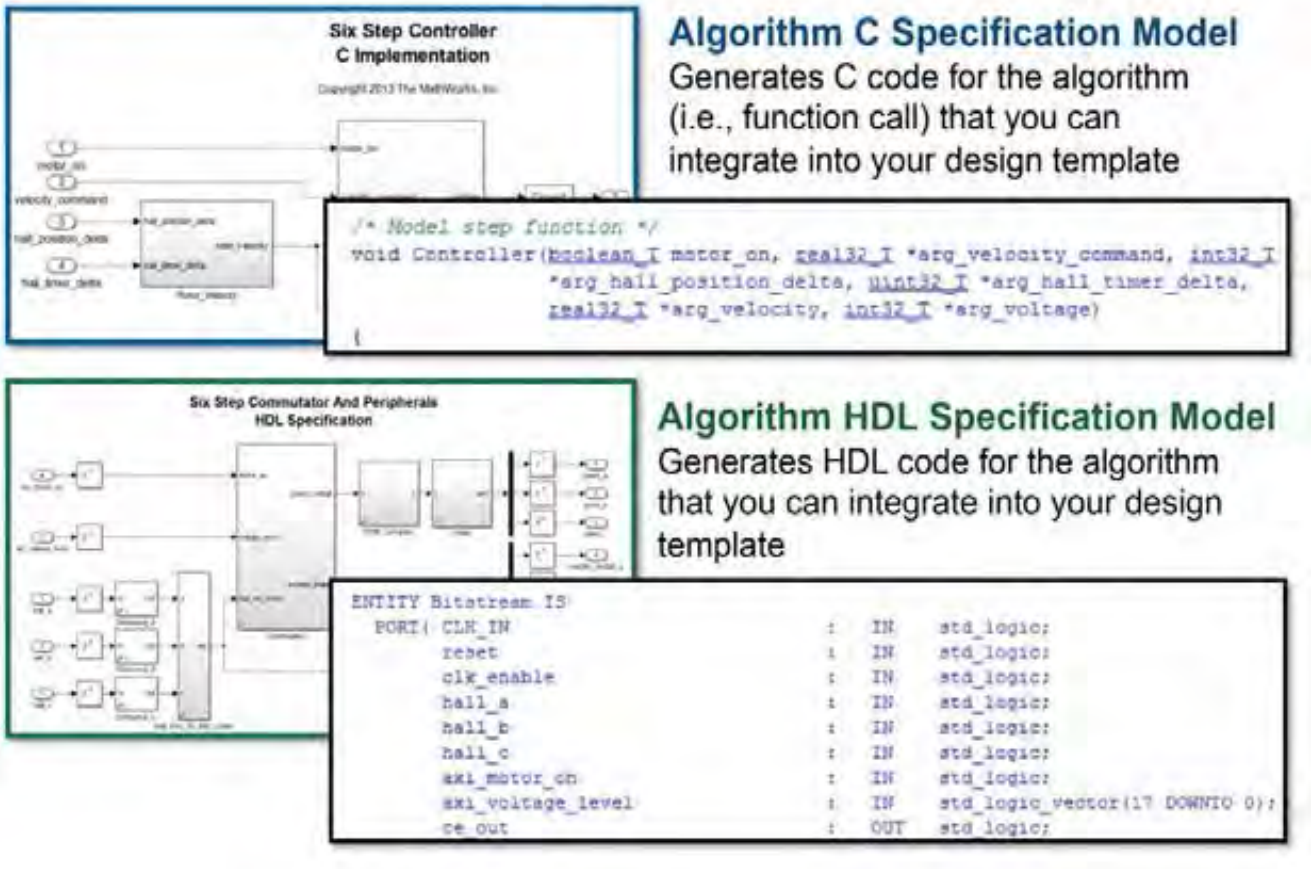


Figure 4 – C and HDL code generated from partitioned Simulink model

With the control-loop gains set, we now can test on a more accurate system model for the controller. In contrast to the control-loop model, the system model incorporates more detailed models of the drive electronics and, more significantly, it includes detailed models that specify the implementation of the controller and peripherals, including timing-accurate models for PWM and Hall-effect sensor processing.

We have partitioned the controller for the Zynq SoC, with the velocity controller and velocity estimator running on an ARM core at 1 kHz and the commutator, Hall sensor and PWM running on the Zynq SoC's programmable logic.

We can compare simulation results for the control-loop and system models (system model results appear as the red signal in Figure 3). In general we get very good agreement between the waveforms, except as the motor's rate approaches zero. At such points the coarseness of the Hall sensor, with only six index pulses per revolution of the motor's shaft, becomes evident. This high-fidelity system model runs the 4-second simulation in 7 minutes, compared with a run-time of only 7 seconds for the lower-fidelity control-loop model. For control system designers, the takeaway here is that these simulation results give us more confidence that the control-loop model is sufficiently accurate for further evaluation of controller alternatives, which can be validated before hardware testing using the system model.

Armed with these findings, we are prepared to prototype the controller on the Intelligent Drives Kit. Through the Zynq SoC guided workflow, we can generate C and HDL code from Simulink models that have been partitioned into subsystems targeting an ARM core and programmable logic (Figure 4).

With this workflow, we use the HDL Coder from MathWorks to generate an IP core that will run in the Zynq SoC device's programmable logic to build an executable running on an ARM core and to establish the interfaces between core and executable over the AXI bus.

With the bitstream loaded into the programmable logic and the executable running on an ARM core, we can run a hardware-in-the-loop test. For this test, we use a modified Simulink testbench model from which we have removed the models for the drive electronics, motor and sensors, since we are using hardware-in-the-loop in place of simulated plant models. To help us check the outcome of the test—and compare it with our simulation results—we can set up the Zynq SoC to store motor shaft velocity measurements and other data in the memory of an ARM core (the black-shaded signal in Figure 3 shows results from hardware testing). Doing so enables us to upload the results to a MATLAB session for processing and visualization at the conclusion of the test by applying a pulse input in the testbench. In this way, we can exactly repeat in hardware the test we've done in simulation. The results from prototyping align very closely with our simulation results, including the discontinuity in the measured motor velocity due to the Hall sensor.

This brief overview illustrates how the MathWorks workflow for the Zynq SoC enables model-based design for use in simulation and prototyping. To continue on into production, you can import the generated C and HDL code into the Vivado® Design Suite, where you can integrate them with executive routines, networking IP and other design components required for the complete system implementation.

To download the models shown in this article and learn more about how to use model-based design with the Zynq-7000 All Programmable SoC / Analog Devices Intelligent Drives Kit from Avnet, visit *mathworks.com/zidk*. From this page, you can also browse a Simulink model that implements a complete field-oriented control model on a Zynq SoC device and view videos showing this example in greater detail.

For information on how MathWorks products support the Xilinx Zynq-7000 All Programmable SoC family, visit *mathworks.com/zynq*.

FPGA
Boards & Modules

EFM-02
FPGA module with USB 3.0 interface. Ideal for Custom Cameras & ImageProcessing.

- Xilinx™ Spartan-6 FPGA XC6SLX45(150)-3FGG484I
- USB 3.0 Superspeed interface Cypress™ FX-3 controller
- On-board memory 2 Gb DDR2 SDRAM 64 Mb Dual SPI flash
- Samtec™ Q-strip connectors 191 (95 differential) user IO

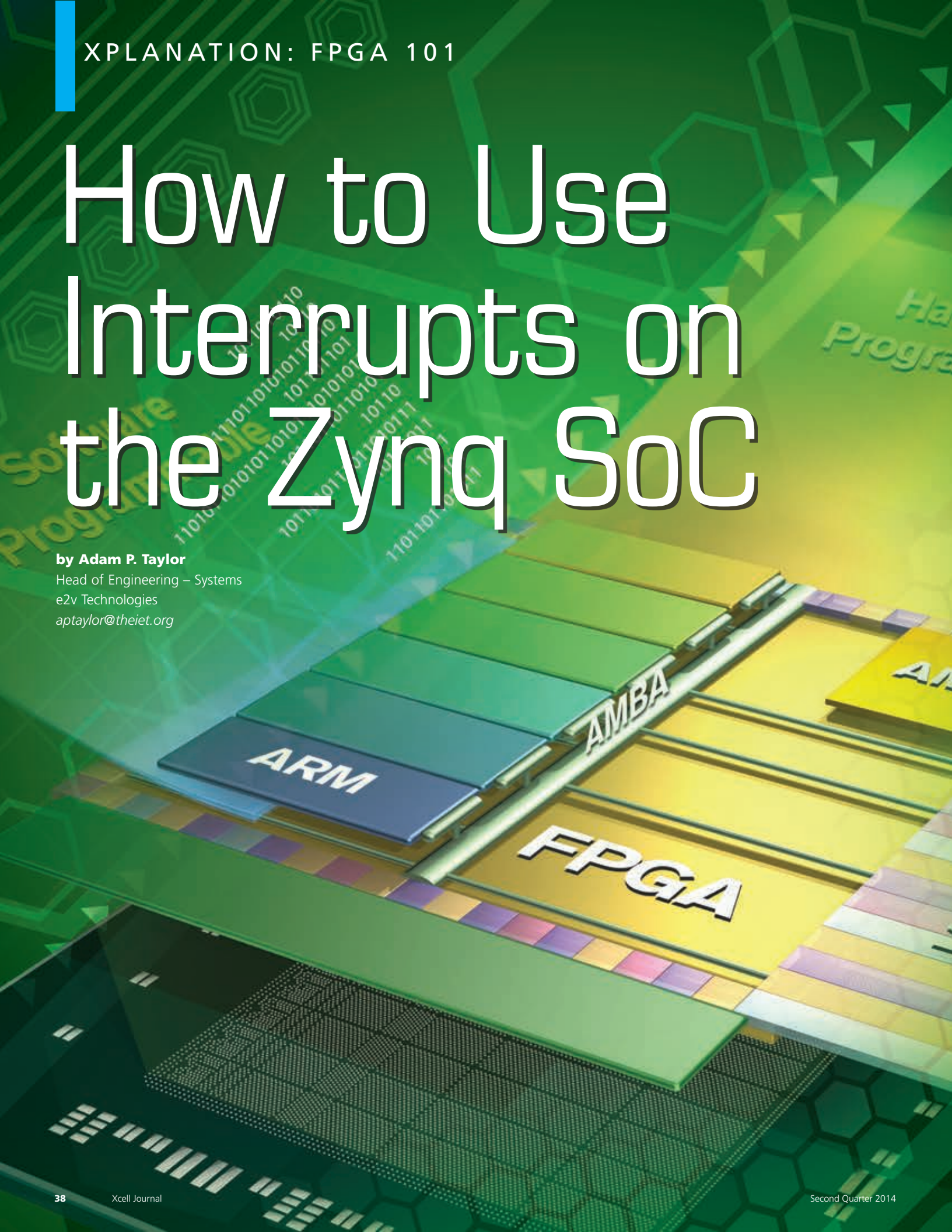
EFM-01
Low-cost FPGA module for general applications.

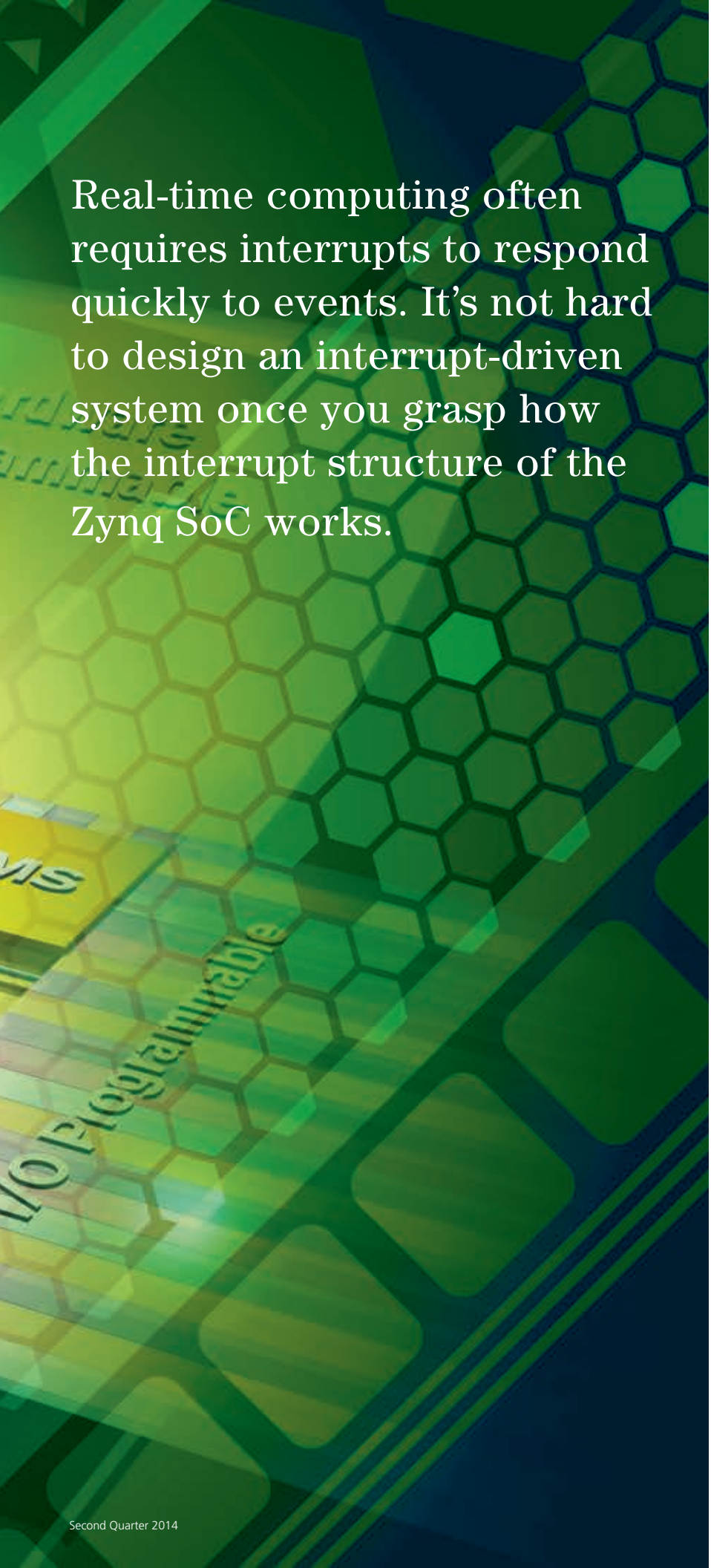
- Xilinx™ Spartan-3E FPGA XC3S500E-4CPG132C
- USB 2.0 Highspeed interface Cypress™ FX-2 controller
- On-board memory 4 Mb SPI flash
- Standard 0.1" pin header 50 user IO

CESYS
Hardware • Software • HDL-Design
www.cesys.com

How to Use Interrupts on the Zynq SoC

by Adam P. Taylor
Head of Engineering – Systems
e2v Technologies
aptaylor@theiet.org





Real-time computing often requires interrupts to respond quickly to events. It's not hard to design an interrupt-driven system once you grasp how the interrupt structure of the Zynq SoC works.

In embedded processing, an interrupt is a signal that temporarily halts the processor's current activities. The processor saves its current state and executes an interrupt service routine to address the reason for the interrupt. An interrupt can come from one of the three following places:

- Hardware – An electronic signal connected directly to the processor
- Software – A software instruction loaded by the processor
- Exception – An exception generated by the processor when an error or exceptional event occurs

Regardless of the source, interrupts can also be classified as either maskable or non-maskable. You can safely ignore a maskable interrupt by setting the appropriate bit in an interrupt mask register. But you cannot ignore a non-maskable interrupt, because these are the types typically used for timers and watchdogs.

Interrupts can be either edge triggered or level triggered. The Xilinx® Zynq®-7000 All Programmable SoC supports configuration of the interrupt either way, as we will see later.

WHY USE AN INTERRUPT-DRIVEN APPROACH?

Real-time designs often require an interrupt-driven approach simply because many systems will have a number of inputs (for example keyboards, mice, pushbuttons, sensors and the like) that will at times require processing. Inputs from these devices are generally asynchronous to the process or task currently executing, so you cannot always predict when the event will occur.

Using interrupts enables the processor to continue processing until an event occurs, at which time the processor can address the event. This interrupt-driven approach also enables a faster response time to events than a polled approach, in which a program actively samples the status of an external device in a synchronous manner.

THE ZYNQ SOC'S INTERRUPT STRUCTURE

As processors get more advanced, there are a number of sources interrupts can come from. The Zynq SoC uses a Generic Interrupt Con-

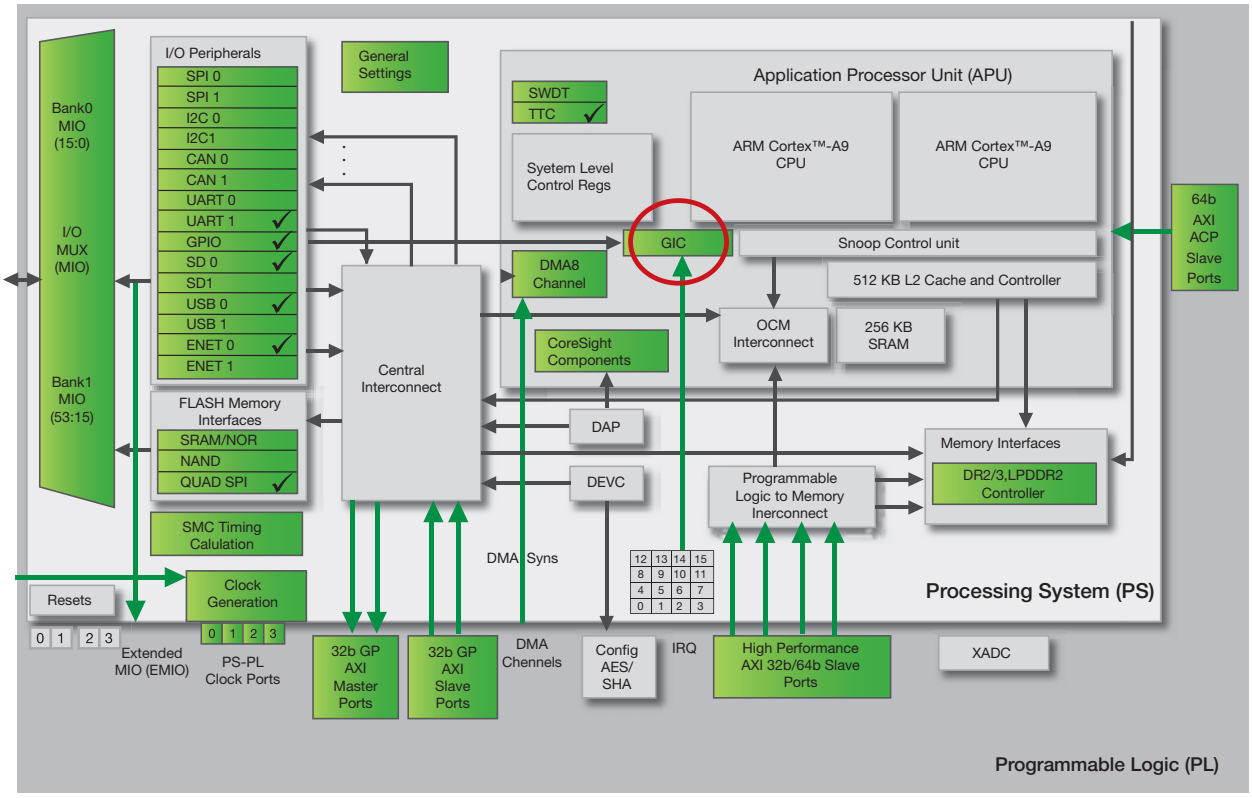


Figure 1 – The Generic Interrupt Controller is circled in red.

troller (GIC), as shown in Figure 1, to process interrupts. The GIC handles interrupts from the following sources:

- Software-generated interrupts – There are 16 such interrupts for each processor. They can interrupt one or both of the Zynq SoC's ARM® Cortex™-A9 processor cores.
- Shared peripheral interrupts – Numbering 60 in total, these interrupts can come from the I/O peripherals, or to and from the programmable logic (PL) side of the device. They are shared between the Zynq SoC's two CPUs.
- Private peripheral interrupts – The five interrupts in this category are private to each CPU—for example CPU timer, CPU watchdog timer and dedicated PL-to-CPU interrupt.

The shared peripheral interrupts are very interesting, as they are very flexible. They can be routed to either CPU from the I/O peripherals (44 interrupts in total) or from the FPGA logic (16 interrupts in total). However, it is also possible to route interrupts from the I/O peripherals to the programmable logic side of the device, as shown in Figure 2.

PROCESSING THE INTERRUPTS ON THE ZYNQ SOC

When an interrupt occurs within the Zynq SoC, the processor will take the following actions:

1. The interrupt is shown as pending.
2. The processor stops executing the current thread.
3. The processor saves the state of the thread in the stack to allow processing to continue once it has handled the interrupt.
4. The processor executes the interrupt service routine, which defines how the interrupt is to be handled.
5. The processor resumes operation of the interrupted thread after restoring it from the stack.

Because interrupts are asynchronous events, it is possible for multiple interrupts to occur at the same time. To address this issue, the processor prioritizes interrupts such that it can service the highest-priority interrupt pending first.

To implement this interrupt structure correctly, we will need to write two functions: an interrupt service routine to define the actions that will take place when the interrupt occurs, and an interrupt setup to configure the interrupt. The interrupt setup is a reusable routine that allows for constructing different interrupts. Generic for all interrupts within a system, the routine will set up and enable the interrupts for the general-purpose I/O (GPIO).

USING INTERRUPTS IN SDK

Interrupts are supported and can be implemented on a bare-metal system using the standalone board support package (BSP) within the Xilinx Software Development Kit

(SDK). The BSP contains a number of functions that greatly ease this task of creating an interrupt-driven system. They are provided within the following header files:

- Xparameters.h – This file contains the processor's address space and the device IDs.
- Xscugic.h – This file holds the drivers for the configuration and use of the GIC.
- Xil_exception.h – This file contains exception functions for the Cortex-A9.

To address a hardware peripheral, we need to know the address range and the device ID for the devices we wish to use—in other words, the GIC, which is provided mostly within the BSP header file xparameters. However, the interrupt ID is provided from xparameters_ps.h (there is no need to declare this header file within your source code as it is included in the xparameters.h file). We can use this interrupt labeled "ID" (it's the GPIO_Interrupt_ID) within our source file as shown below:

```
#define GPIO_DEVICE_ID    XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID  XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GPIO_INTERRUPT_ID    XPS_GPIO_INT_ID
```

For this simple example, we will be configuring the Zynq SoC's GPIO to generate an interrupt following a button push. To set up the interrupt, we will need two static global variables and the interrupt ID defined above to make the following:

```
static XScuGic Intc; // Interrupt Controller Driver
static XGpioPs Gpio; //GPIO Device
```

Within the interrupt setup function, we will need to initialize the Zynq SoC's exceptions; configure and initialize the GIC; and connect the GIC to the interrupt-handling hardware. The Xil_exception.h and Xscugic.h files provide the functions we need to accomplish this task. The result is the following code:

```
//GIC config
XScuGic_Config *IntcConfig;
Xil_ExceptionInit();

//initialize the GIC
IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);

XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
IntcConfig->CpuBaseAddress);

//connect to the hardware
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IN-
T,(Xil_ExceptionHandler)XScuGic_InterruptHandler,
      GicInstancePtr);
```

When it comes to configuring the GPIO to function as an interrupt within the same interrupt configuration routine,

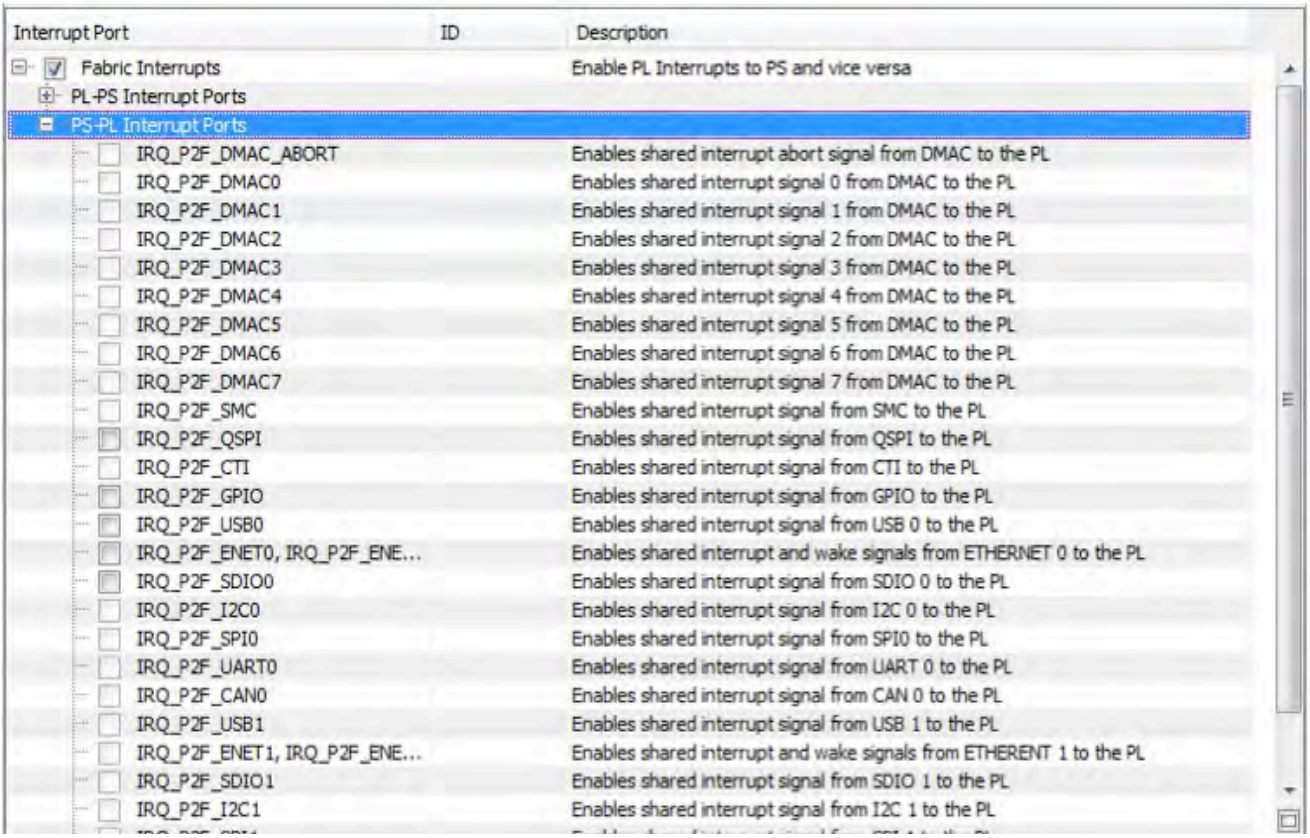


Figure 2 – These are the interrupts available between the processing system and the programmable logic.

we can configure either a bank or an individual pin. This task can be achieved using functions provided within xgpiops.h, for example:

```
void XGpioPs_IntrEnable(XGpioPs *InstancePtr, u8
   Bank, u32 Mask);
void XGpioPs_IntrEnablePin(XGpioPs *InstancePtr,
   int Pin);
```

Naturally, you will also need to configure the interrupt correctly. For instance, do you wish it to be edge triggered or level triggered? If so, which edge and level can be achieved using the function?

```
void XGpioPs_SetIntrTypePin(XGpioPs *InstancePtr,
int Pin, u8 IrqType);
```

where the IrqType is defined by one of the five definitions within xgpiops.h. They are:

```
#define XGPIOPS_IRQ_TYPE_EDGE_RISING 0 /**<
    Interrupt on Rising edge */
#define XGPIOPS_IRQ_TYPE_EDGE_FALLING 1 /**<
    Interrupt Falling edge */
#define XGPIOPS_IRQ_TYPE_EDGE_BOTH 2 /**<
    Interrupt on both edges */
#define XGPIOPS_IRQ_TYPE_LEVEL_HIGH 3 /**<
    Interrupt on high level */
#define XGPIOPS_IRQ_TYPE_LEVEL_LOW 4 /**<
    Interrupt on low level */
```

If you decide to use the bank enable, you need to know which bank the pin or pins you wish to enable interrupts are on. The Zynq SoC supports a maximum of 118 GPIOs. In this configuration, all of the MIOs (54 pins) are being used as GPIO along with the EMIOs (64 pins). We can break this configuration into four banks, with each bank containing up to 32 pins.

This setup function will also define the interrupt service routine, which is to be called when the interrupt occurs that uses the function:

```
XGpioPs_SetCallbackHandler(Gpio,
   (void *)Gpio, IntrHandler);
```

The interrupt service routine can be as simple or as complicated as the application defines. For this example, it will toggle the status of an LED on and off each time a button is pressed. The interrupt service routine will also print out a message to the console each time the button is pressed.

```
static void IntrHandler(void *CallBackRef, int
Bank, u32 Status)
{
       int delay;
         XGpioPs *Gpioint = (XGpioPs *)
         CallBackRef;
         XGpioPs_IntrClearPin(Gpioint, pbsw);
       printf("****button pressed****\n\r");
       toggle = !toggle;
       XGpioPs_WritePin(Gpioint, ledpin, toggle);
       for( delay = 0; delay < LED_DELAY; delay++)
       //wait
       {}
}
```

PRIVATE TIMER EXAMPLE

The Zynq SoC has a number of timers and watchdogs available. These are either private to a CPU or a shared resource available to both CPUs. Interrupts are required if you are to use these components efficiently in your design. The timers and watchdogs include the following:

- CPU 32-bit timer (SCUTIMER), clocked at half the CPU frequency
- CPU 32-bit watchdog (SCUWDT), clocked at half the CPU frequency
- Shared 64-bit global timer (GT), clocked at half the CPU frequency (each CPU has its own 64-bit comparator; it is used with the GT, which drives a private interrupt for each CPU)
- System watchdog timer (WDT), which can be clocked from the CPU clock or an external source
- A pair of triple timer counters (TTCs), each containing three independent timers. The TTCs can be clocked by the CPU clock or by means of an external source from the MIO or EMIO in the programmable logic.

To gain the maximum benefit from the available timers and watchdogs, we need to be able to make use of the Zynq SoC's interrupts. The simplest of these to configure is the private timer. Like most of the Zynq SoC's peripherals, this timer comes with a number of predefined functions and macros to help you use the resource efficiently. They are contained within the following:

```
#include "xscutimer.h"
```

This file contains functions (macros) that will provide a number of capabilities, including initialization and self-test. The functions within this file will also start and stop the timer, and manage the timer (restart it; check to see if it has expired; load the timer; enable/disable auto loading). Another of their jobs is to set up, enable, disable, clear and manage the timer interrupts. Finally, these functions also get and then set the prescaler.

The timer itself is controlled via the following four registers:

- Private Timer Load Register – This register is used in auto reload mode. It contains the value that is reloaded into the Private Timer Counter Register when auto reload is enabled.
- Private Timer Counter Register – This is the actual counter itself. When enabled, once this register reaches zero the interrupt event flag is set.
- Private Timer Control Register – The control register enables or disables the timer, auto reload mode and interrupt generation. It also contains the prescaler for the timer.

- Private Timer Interrupt Status Register – This register contains the private timer interrupt status event flag.

As for using the GPIO, the timer device ID and timer interrupt ID that are needed to set up the timer are contained within the XParameters.h file. Our example will use the pushbutton interrupt that we developed previously. When the button is pressed, the timer will load and start to run (not in auto reload mode). Upon expiration of the timer, an interrupt will be generated that will write a message out over the STDOUT. The interrupt will then be cleared to wait until the next time the button is pressed. This example will always load the same value into the counter; hence with the declarations at the top of the file, the timer count value is declared, as follows:

```
#define TIMER_LOAD_VALUE   0xFFFFFFFF
```

The next stage is to configure and initialize the private timer and load the timer count value into it.

```
 //timer initialisation
 TMRConfigPtr = XScuTimer_LookupConfig
   (TIMER_DEVICE_ID);
 XScuTimer_CfgInitialize(&Timer,
   TMRConfigPtr,TMRConfigPtr->BaseAddr);
//load the timer
 XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
```

We also need to update the interrupt setup subroutine to connect the timer interrupts to the GIC and enable the timer interrupt.

```
 //set up the timer interrupt
  XScuGic_Connect(GicInstancePtr, TimerIntrId,
(Xil_ExceptionHandler)TimerIntrHandler,
    (void *)TimerInstancePtr);
  //enable the interrupt for the Timer at GIC
  XScuGic_Enable(GicInstancePtr, TimerIntrId);
  //enable interrupt on the timer
  XScuTimer_EnableInterrupt(TimerInstancePtr);
```

Where TimerIntrHandler is the name of the function that is called when the interrupt occurs, the timer interrupt must be enabled on the GIC and within the timer itself.

The timer interrupt service routine is very simple. All it does is to clear the pending interrupt and write out a message over the STDOUT, as follows:

```
static void TimerIntrHandler(void *CallBackRef)
{

  XScuTimer *TimerInstancePtr =
     (XScuTimer *) CallBackRef;
  XScuTimer_ClearInterruptStatus(TimerInstancePtr);
  printf("****Timer Event!!!!!!!!!!!!!!****\n\r");
```

With this action complete, the final thing to do is to modify the GPIO interrupt service routine to start the timer each time the button is pushed, as such:

```
 //load timer
 XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
 //start timer
 XScuTimer_Start(&Timer);
```

To do this we first load the timer value into the timer and then call the timer start function. Now we can again clear the pushbutton interrupt and resume processing, as seen in Figure 3.

Many engineers initially approach an interrupt-driven system design with trepidation. However, the Zynq SoC's architecture, with the Generic Interrupt Controller coupled with the drivers provided with the SDK, enables you to get an interrupt-driven system up and running very quickly and efficiently.

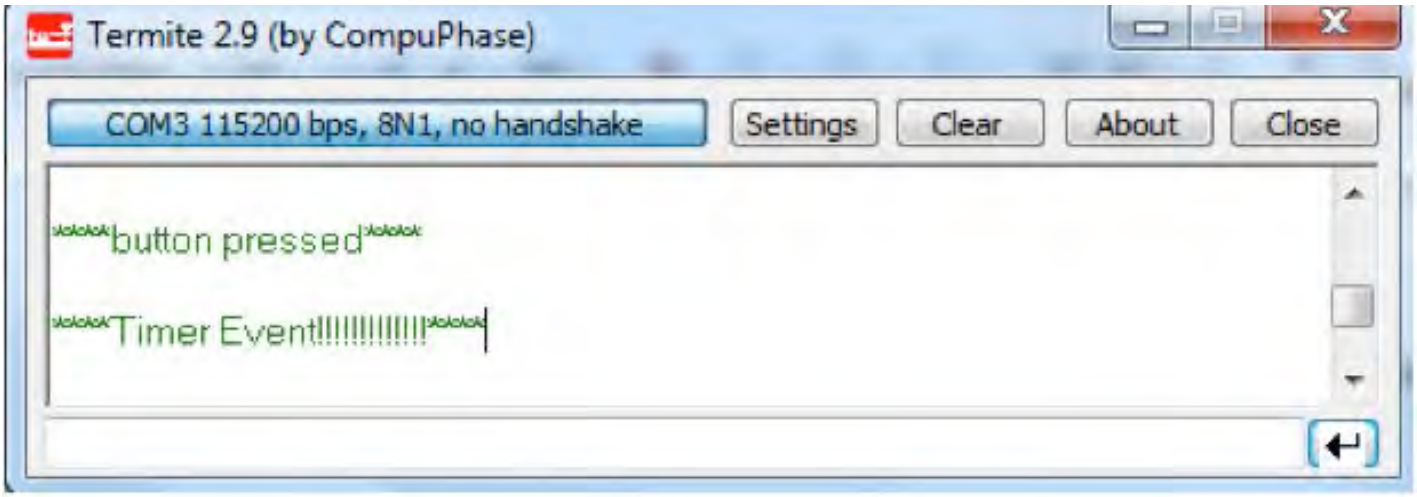
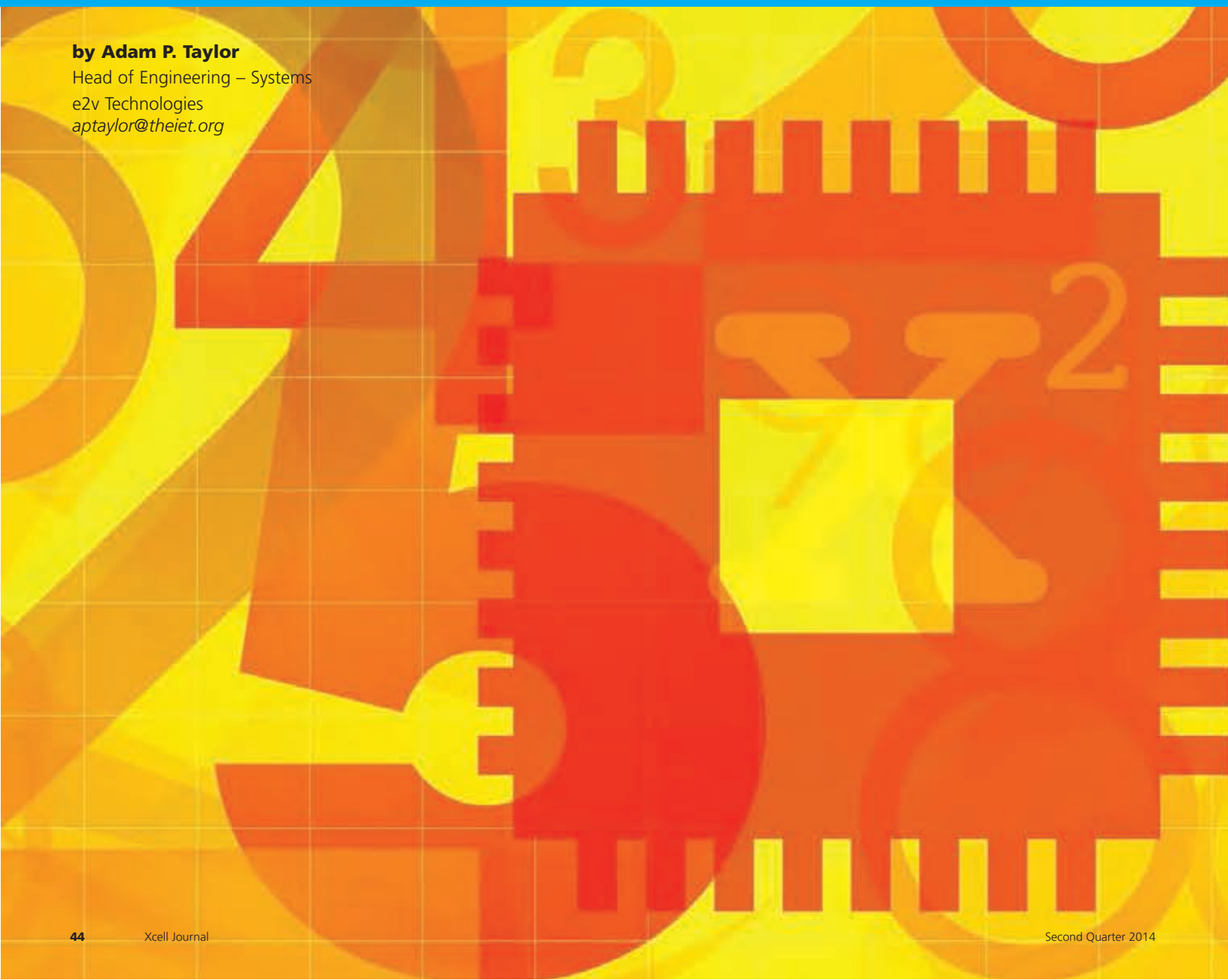


Figure 3 – This screen shows an example of the GPIO and timer interrupt event outputs.

Calculating Mathematically Complex Functions

by Adam P. Taylor
Head of Engineering – Systems
e2v Technologies
aptaylor@theiet.org



One of the great benefits of FPGAs is that you can use their embedded DSP blocks to tackle the knottiest mathematical transfer functions. Polynomial approximation is one good way to do it.

Thanks to their flexibility and performance, FPGAs have found their way into a number of industrial, science, military and other applications that require the calculation of complex mathematical problems or transfer functions. It is not uncommon to see tight accuracy and calculation latency times in the more critical applications.

When using an FPGA to implement mathematical functions, engineers normally choose fixed-point mathematics (see *Xcell Journal* issue 80, "The Basics of FPGA Mathematics," *http://issuu.com/xcelljournal/docs/xcell80/44?e=2232228/2002872)*. Also, there are many algorithms, such as CORDIC, that you can use to calculate transcendental functions (see *Xcell Journal* issue 79, "How to Use the CORDIC Algorithm in Your FPGA," *http://www.xilinx.com/publications/archives/xcel l/Xcell79.pdf)*.

However, when confronting functions that are very mathematically complex, there are more efficient ways of dealing with them than by implementing the exact demanding function within the FPGA. To understand these alternative approaches—especially one of them, polynomial approximation—let us first define the problem.

LAYING OUT THE PROBLEM

One such example of a complex mathematical transfer function would be within an FPGA that monitors a platinum resistance thermometer (PRT) and converts the resistance of the PRT into a temperature. This conversion typically occurs using a Callendar-Van Dusen equation. In its simplified form, shown below, this equation can determine temperatures between 0°C and 660°C.

$$R = R_0 \times (1 + a \, x \, t + b \, x \, t^2)$$

where R_0 is the resistance at 0°C and a and b are coefficents of the PRT and t is the temperature.

In reality, we want to go from a resistance to a temperature. To do so, we need to rearrange the equation so that the result is the temperature for a given resistance. Most systems that use a PRT will design electronics to measure the resistance of the PRT using an electronic circuit, leaving the FPGA to cal-

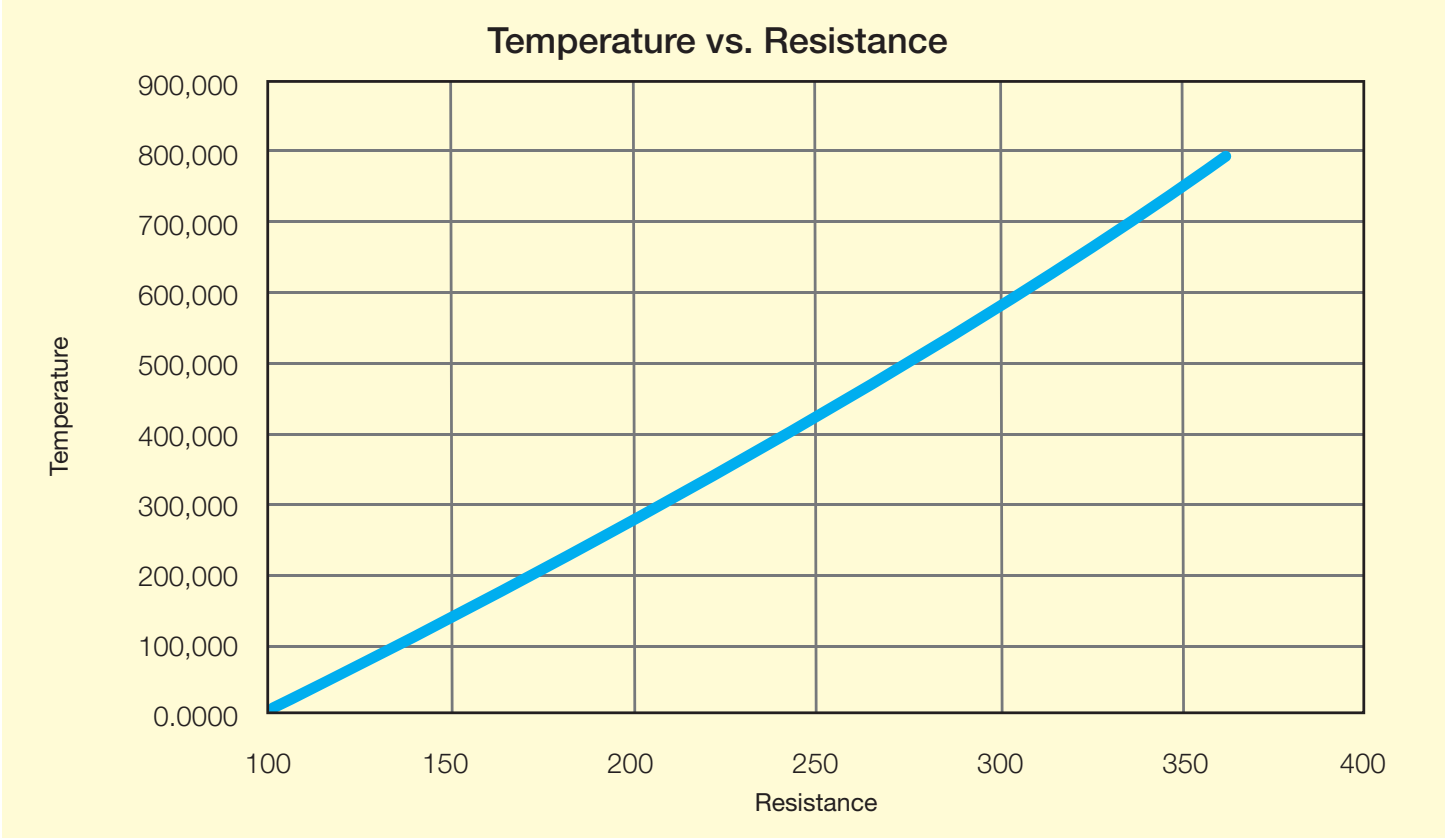


Figure 1 – The plotted transfer function

culate the temperature using the rearranged equation below.

$$t = \frac{-R_0 \times a + \sqrt{R_0^2 \times a^2 - 4 \times R_0 \times a \times (R_0 - R)}}{2 \times R_0 \times b}$$

Implementing this equation within an FPGA may be daunting for even a seasoned FPGA engineer. Plotting the obtained resistance against temperature results in a graph as shown in Figure 1. From the graph, you can clearly see the nonlinearity of the response.

Implementing the rearranged transfer function in an FPGA directly could be a significant challenge, both in terms of actual design effort required and then in validation (ensuring accuracy and function across boundary and corner-case conditions). Many engineers will look for different methods to implement the function in a way that will reduce design and validation effort so as to protect project time scales. One possible approach would be using a lookup table to store a number of points on the curve with linear interpolation between the points within the LUT.

This approach may fit the bill, depending upon the accuracy requirement and number of elements stored within the lookup table. However, you will still need to include a linear interpolator function within the design. This function can be mathematically sophisticated and will often include a non-power-of-two divide, which adds to the complexity.

CAPITALIZE ON FPGA RESOURCES

Instead, there is another method you can use to implement these types of transfer functions—one that capitalizes upon the very nature of the FPGA. Modern FPGAs like the Xilinx® Spartan®-6 and the 7 series Artix®, Kintex® and Virtex® lines contain much more than just the traditional lookup tables and flip-flops. They also come with built-in DSP slices, Block RAM and distributed RAM, along with many advanced hard IP cores such as PCIe® and Ethernet endpoints, high-speed serial links and so on.

Engineers often call the DSP slices DSP48s, due to the 48-bit accumulator they provide. However, these slices also supply 25 x 18-bit-wide multipliers and addition/subtraction capabilities, among many other faculties. It is these internal RAM structures and DSP slices that you can use to implement transfer functions with greater ease.

POLYNOMIAL APPROXIMATION

One method that utilizes the DSP- and RAM-rich architecture of the FPGA is polynomial approximation. To use this technique, you must first plot the mathematical function, covering the input value range in a mathematical program such as MATLAB® or Excel. You can then add a polynomial trend line to the data set in question, such that the equation for the trend line can then be implemented within the FPGA in place of the mathematically complex function, provided the trend-line equation meets the accuracy requirements.

Should one polynomial equation not provide sufficient accuracy over the entire transfer function input range, just add more. You can still rely on this approach so long as you generate a number of polynomial constants for use across the input range.

Most mathematical programs capable of adding a polynomial trend line allow you to select the order, or number of polynomial terms. The larger the order, the more accurate the fit should be—but the more terms you will need to implement within the FPGA. When performing this process for the transfer function example we are using in Microsoft Excel, we obtained the trend line and equation seen in Figure 2. This example used the polynomial order of four.

Having obtained the polynomial fit for the transfer function we want to implement, we can then double-check for accuracy against the original transfer function using the same analysis tool, in this case Excel. For the case in point where temperature is being monitored, it may be that the end measurement has to be accurate to +/-1°C, not a particularly demanding accuracy requirement. Still, it may prove difficult to achieve using just one polynomial equation, depending upon the range of measurements and the transfer function you are implementing. How can we address this problem?

MULTIPLE TREND LINES SELECTED BY INPUT VALUE

Should one polynomial equation not provide sufficient accuracy over the entire transfer function input range, just add more. It is still possible to

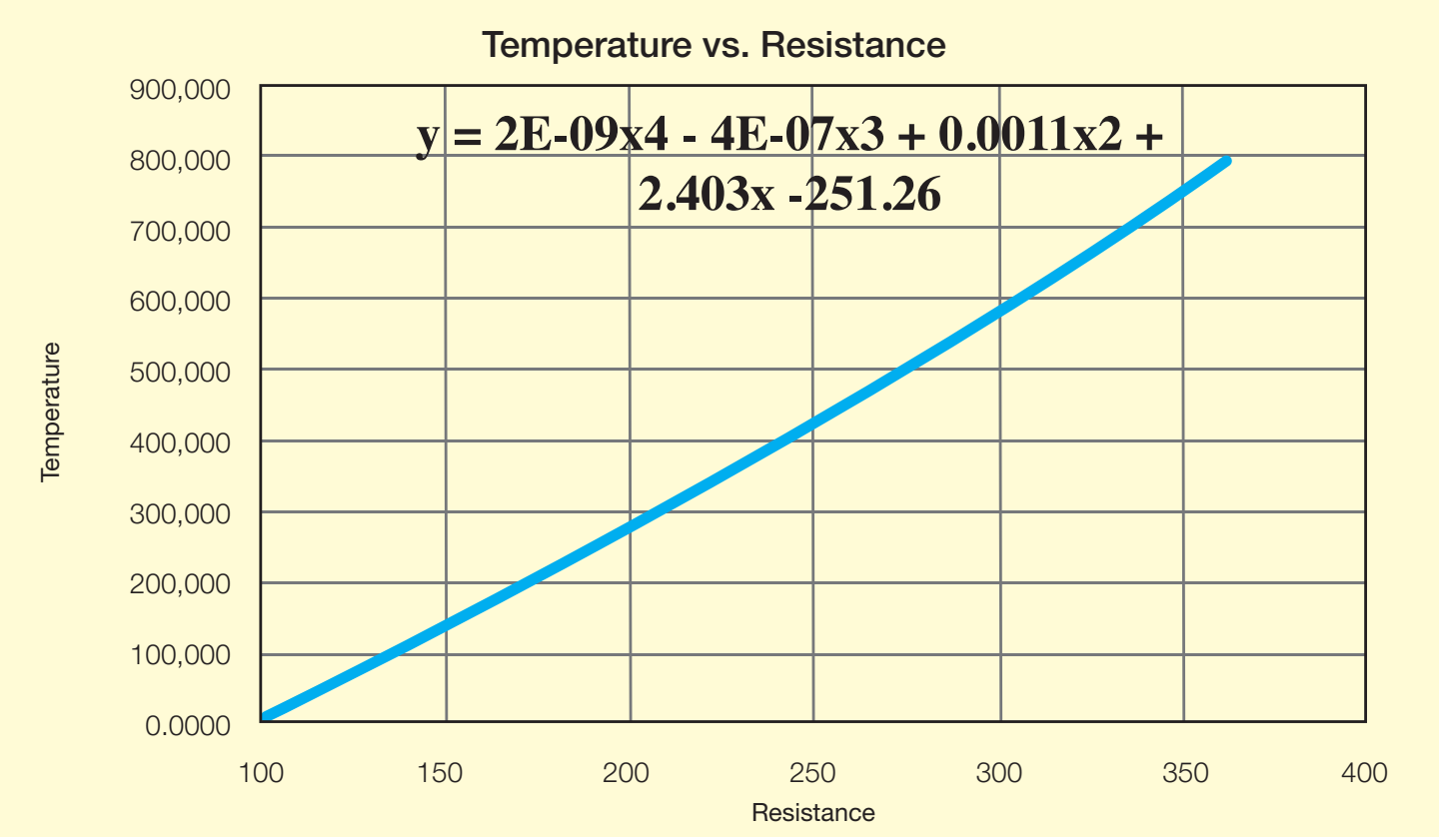


Figure 2 – Trend line and polynomial equation for the temperature transfer function

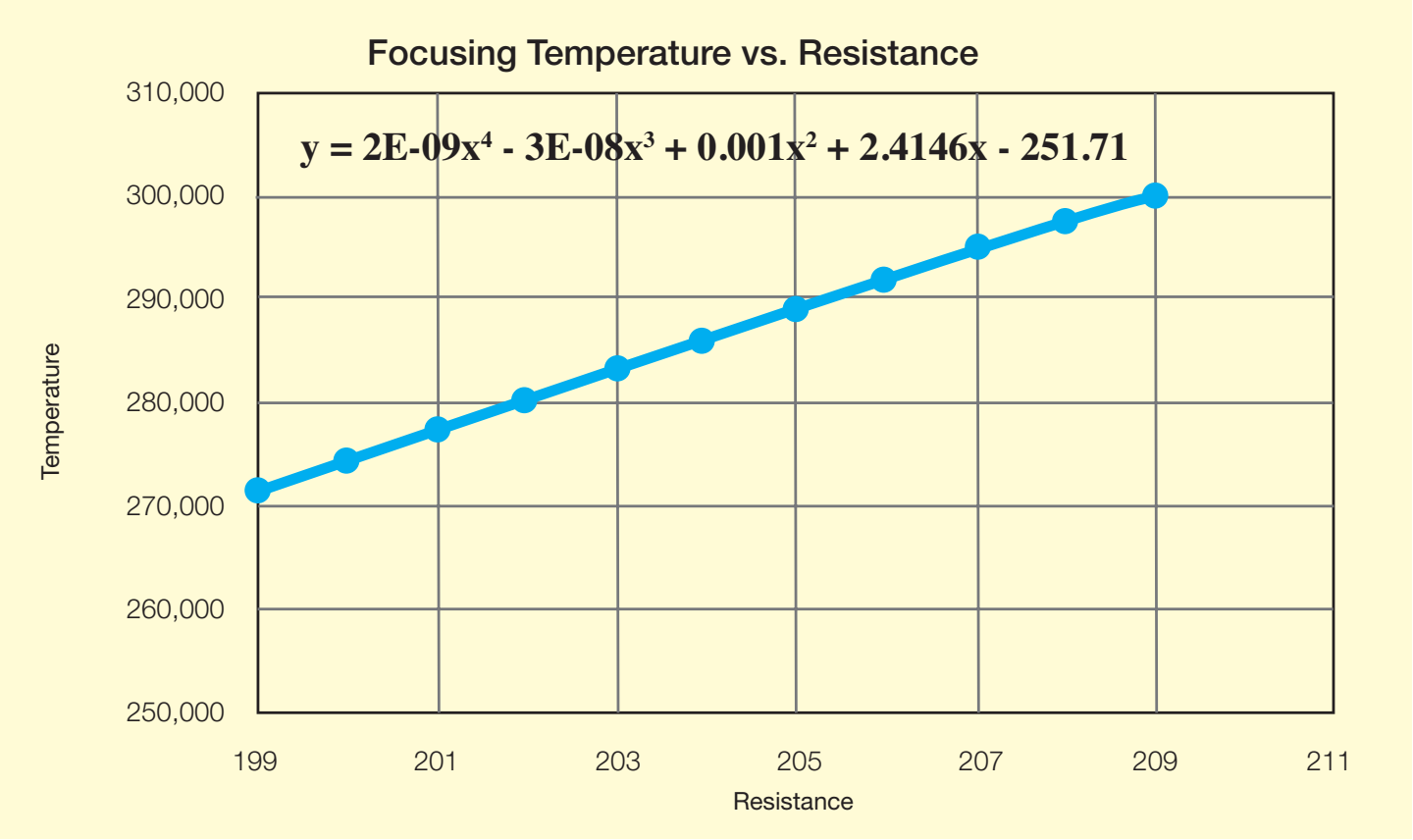


Figure 3 – Plotting between 269°C and 300°C to provide a more accurate result

use this approach so long as you generate a number of polynomial constants for use across the input range. Thus, once the input value goes outside specific bounds, a new set of constants is loaded.

Continuing with the temperature example, the first polynomial equation provides +/- 1°C of accuracy between 0° and 268°C. For many applications this will be more than sufficient. Suppose that we require an extended operating range and tolerance to 300°C, which would mean our initial approach did not meet the design requirements. Using a segmented approach, we can address this problem by plotting the range between 269°C and 300°C and obtaining a different polynomial equation that will provide more accuracy for this output range (see Figure 3).

In short, the implementation uses the first polynomial constants until the input value goes above a precalculated range that corresponds to 268°C. Above that range, the second set of constants is used to maintain the accuracy requirements.

In this way, you can break a transfer function into a number of segments to achieve the desired accuracy. You may opt to space these segments uniformly across the transfer function—that is, split them into 10 segments of equal value of X. Alternatively, you can make them nonuniform and segment them as required to achieve the desired accuracy, focused upon areas of the transfer function where accuracy is harder to obtain.

Among the trade-offs to consider when deciding upon your implementation, keep in mind that a uniform approach may require a larger memory footprint than a nonuniform approach. Depending upon the transfer function you are implementing, going with a nonuniform approach could result in a considerable saving.

HOW DOES IT COMPARE?

Of course, as I mentioned earlier there are other methods you can use to implement transfer functions. The four most commonly used methods aside from polynomial approximation are software routines, lookup tables, a lookup table with interpolation and CORDIC.

The use of software to calculate the transfer function complicates the system architecture due to the need to add a processor (with an associated increase in design complexity, BOM cost and so on). Even if the design team mitigates this drawback by using a system-on-chip such as the Xilinx Zynq®-7000 All Programmable SoC, challenges still remain. For starters, the time it takes to calculate the transfer function in software would be much longer than can be achieved in logic, reducing the system response time. In fact, calculation of transfer functions such the one used in our sample design is a classic exam-

Polynomial approximation presents a middle ground among the four alternative methods of implementing transfer functions, offering a good trade-off in performance, accuracy and footprint.

ple of where the processing should be offloaded to the programmable logic side of the Zynq SoC.

The efficacy of the second approach—using a lookup table containing precalculated values for the input—can vary depending upon the range and width of the input values. At times, the result will very quickly be a very large LUT that requires a lot of RAM within the FPGA. Depending upon the FPGA, this approach could require more resources than are available, or it might cause conflicts with requirements for other modules within the design. On the plus side, of course, this method will result in a very fast "calculation" of the result.

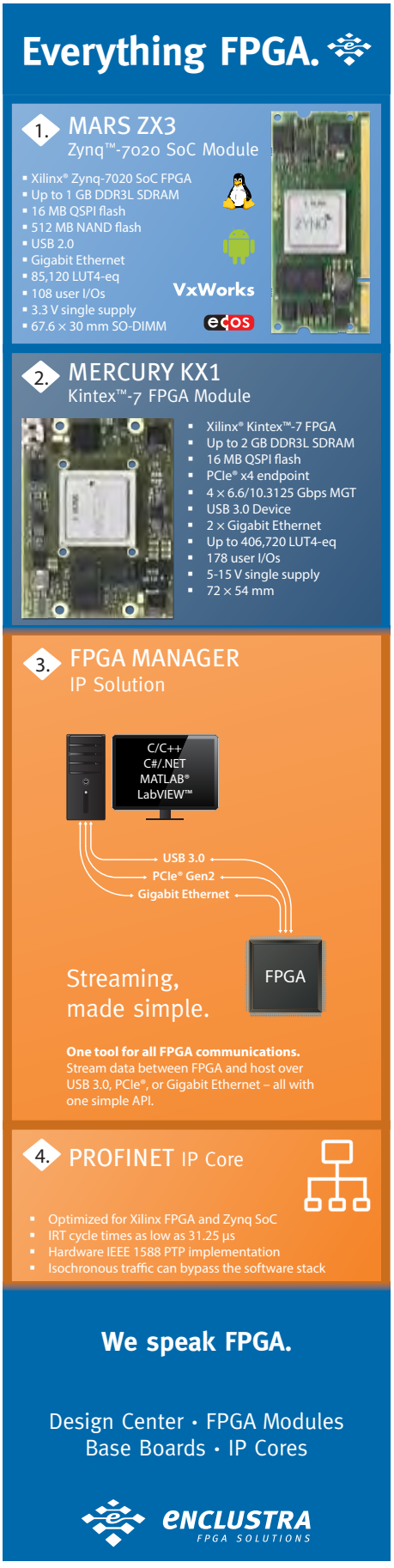
The third potential approach—a lookup table with interpolation—is one we explored earlier and is an attempt to reduce the number of memory locations needed with a full LUT approach. This technique does require the engineer to write a linear interpolation function within the FPGA, which can be a little involved. It is, however, still much simpler than the final option: CORDIC.

A CORDIC algorithm is capable of implementing transcendental functions such as sine, cosine, multiplication, division, square root and so on. Therefore, it is possible to implement transfer functions exactly using a combination of CORDIC algorithms and basic mathematical blocks. The technique can result in higher precision. However, for a complicated transfer function, this gain in precision will come at the cost of increased design and verification time. There will of course be an impact on the operating frequency of a device implemented in this way.

Polynomial approximation therefore presents a middle ground among the four alternative options, offering a good trade-off in performance, accuracy and implementation footprint.

EASE OF IMPLEMENTATION

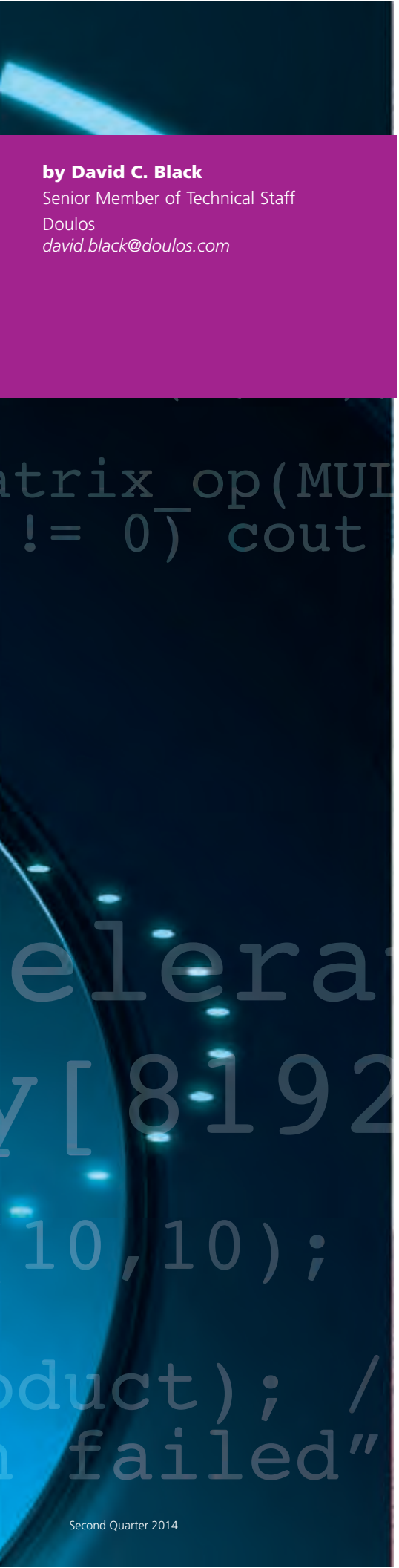
Every engineer wants to produce an FPGA that has an optimal utilization of device resources. Polynomial approximation allows you to benefit from the multiplier- and RAM-rich environment provided by the FPGAs, and to use these resources to easily implement what at first might appear to be a very complex mathematical transfer function.



Make Slow Software Run Fast with Vivado HLS

Anyone plagued by code bottlenecks should explore the one-two punch of high-level synthesis and the Zynq SoC.

by David C. Black
Senior Member of Technical Staff
Doulos
david.black@doulos.com



Have you ever written some software that, despite your best coding efforts, didn't run as fast as desired? I have. Have you thought, "If only there were an easy way to put some of the code into multiple custom processors or custom hardware that wasn't so expensive"? After all, your application is one of many, and custom hardware takes time and money to create. Or does it?

I began rethinking this proposition recently when I heard about the Xilinx® high-level synthesis tool, Vivado® HLS. In combination with the Zynq®-7000 All Programmable SoC, which combines a dual-core ARM® Cortex™-A9 processor with an FPGA fabric, high-level synthesis opens up new possibilities in design. This class of tools creates highly tuned RTL from C, C++ or SystemC source code. Many purveyors of this technology exist, and the rate of adoption has been increasing in recent years.

So, how hard would it be to migrate some of that slow code into hardware, if indeed I could simply use Vivado HLS to do the more demanding computations? After all, I usually wrote my code in C++, and Vivado HLS used C/C++ as an input. The ARM processor cores meant I could run the bulk of my software in a conventional environment. In fact, Xilinx has even made available a software development kit (SDK) and PetaLinux for this purpose.

ARCHITECTURAL CONCERNS

As I started to think about this transformation from a software perspective, I grew concerned about the software interface. After all, HLS creates hardware dedicated to processing hardware interfaces. I needed something easy to access, like a coprocessor or hardware accelerator, to make the software go faster. Also, I didn't want to write a new compiler. To make it easy to exchange data with the rest of the software, the interface needed to look like simple memory locations where we could place the inputs and later read back the results.

Then I made a discovery. Vivado HLS supports the idea of creating an AXI slave with relatively little effort. This capability started me thinking an accelerator might not be so difficult to create after all. Thus, I found myself coding up a simple example to explore the possibilities. I was pleasantly surprised with how it turned out.

Let's take a walk through the approach I took and consider the results.

For my example, I chose to model a set of simple matrix operations such as add and multiply. I didn't want it to be constrained to a fixed size, so I would have to provide both the input arrays and their respective sizes. An ideal interface would put all the values as simple arguments to a function, such as the code in Figure 1.

The interface to the hardware would need to have a simple way to map the function arguments to memory locations. Figure 2 shows a memory layout to support this mapping. The registers would hold information about how matrices were laid out and what the desired operations would be. The *command* register would indicate which operation to do. This would allow me to combine several simple operations into one piece of hardware. The *status* register would simply be a way to know if the operation was in progress or had finished successfully. Ideally, the device would also support an interrupt.

Going back to the hardware design, I learned that Vivado HLS allows for array arguments to specify small memories. Thus, the functionality would be described with a function such as Figure 3 shows.

Assuming the ability to synthesize the AXI slave, how would this fit with the software? My normal coding environment assumes Linux. Fortunately, Xilinx provides PetaLinux, and conveniently PetaLinux provides a mechanism known as the User I/O device. UIO allows a simple approach to mapping the new hardware into user memory space, and provides the ability to wait for an interrupt. This means you avoid the awkward time and process of writing a device driver. Figure 4 illustrates the system.

```
Matrix operand1(5,10), operand2(10,5), product(10,10);
int status;
status = matrix_op(MUL, operand1, operand2, product); // product = operand1 * operand2;
if (status != 0) cout << "ERROR: multiplication failed" << endl;
```

Figure 1 – Example call to accelerator

Addr	Register name	Dir	Bits	Contents	
0	Matrix0_ptr	RW	32	Address of matrix 0 data	
4	Matrix0_shape	RW	32	Rows matrix 0	Cols matrix 0
8	Matrix1_ptr	RW	32	Address of matrix 1 data	
12	Matrix1_shape	RW	32	Rows matrix 1	Cols matrix 1
16	Matrix2_ptr	RW	32	Address of matrix 2 data	
20	Matrix2_shape	RW	32	Rows matrix 2	Cols matrix 2
24	Matrix3_ptr	RW	32	Address of matrix 3 data	
28	Matrix3_shape	RW	32	Rows matrix 3	Cols matrix 3
32	-reserved-	-	32		
36	-reserved-	-	32		
40	Command	RW	32	0	enum
44	Status	RW	32	0	enum
				8192 x 32 memory	

Figure 2 – Register summary table

```
int Accelerator(int registers[16], int memory[8192]);
```

Figure 3 – Accelerator function API

There are of course a few drawbacks to this approach. For instance, the UIO device cannot be used with DMA, so you must construct matrices in the device memory and manually copy them out when done. A custom device driver in the future could address that issue if needed.

SYNTHESIZING THE HARDWARE WITH VIVADO HLS

Back to the topic of synthesizing the AXI slave. How difficult would this be? I found the coding restrictions to be quite reasonable. Most of the C++ language could be used with the exception of the dynamic allocation of memory.

After all, hardware doesn't manufacture itself during operation. This fact also restricts the use of the Standard Template Library (STL) functions, because they make heavy use of dynamic allocation. As long as the data remains static, most features are available. At first this task appeared onerous, but I realized it wasn't a huge deal. Also, Vivado HLS allows for C++ classes, templates, functions and operator overloading. My matrix operations could easily be wrapped in a custom matrix class.

Adding the I/O to create an AXI slave was easy. Simply add some pragmas to indicate which ports participate and what protocol they would use.

Running the synthesis tool was fairly easy as long as I didn't push all the knobs.

Running the synthesis tool was also fairly easy as long as I didn't push all the knobs. Figure 5 shows the overall steps involved, which I won't describe in detail here. Vivado HLS needs a bit of direction as to the target technology and clock speed. After that the process involved keeping an eye on the reports for violations of policy, and studying the analysis report to ensure Vivado HLS had done what I expected. Tool users need to have some appreciation for the hardware aspects, but technology classes exist to cover that issue. There is also the matter of running simulations both before and after synthesis to verify the expected behavior.

The Vivado IP Integrator made connecting the AXI slave into the Zynq SoC hardware a breeze, and removed concerns that signals would be hooked up incorrectly. Xilinx even has a profile for my development system, the ZedBoard, and IP Integrator exports data for the software development kit.

UNCLOGGING THE BOTTLENECKS

I am truly pleased with the results, and hope to do more with this chip-and-tool set combination. I have not explored all the possibilities. For instance, Vivado HLS also supports an AXI master interface. AXI would allow the accelerator to copy the matrices from external memory (although security issues might exist for this case). Nevertheless, I highly recommend that anyone looking at code bottlenecks in their software should look at this tool set. Ample training classes, resources and materials exist to enable a fast ramp, including those from Doulos. See *www.doulos.com* for more information.

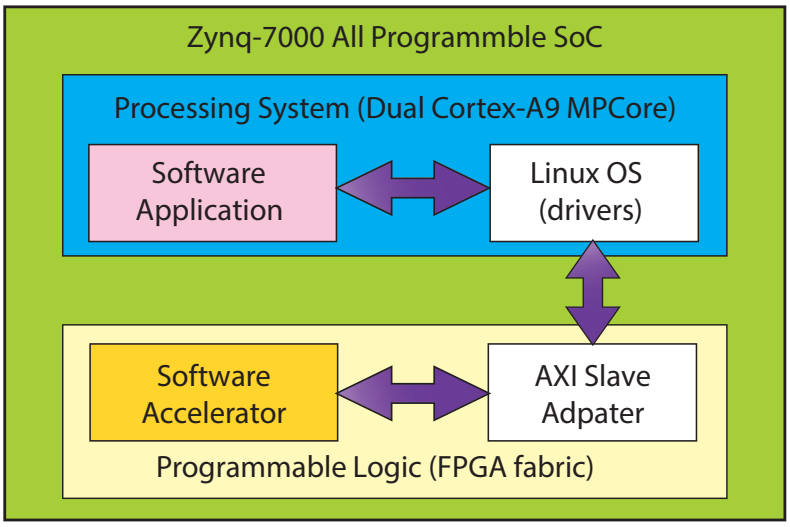


Figure 4 – System diagram

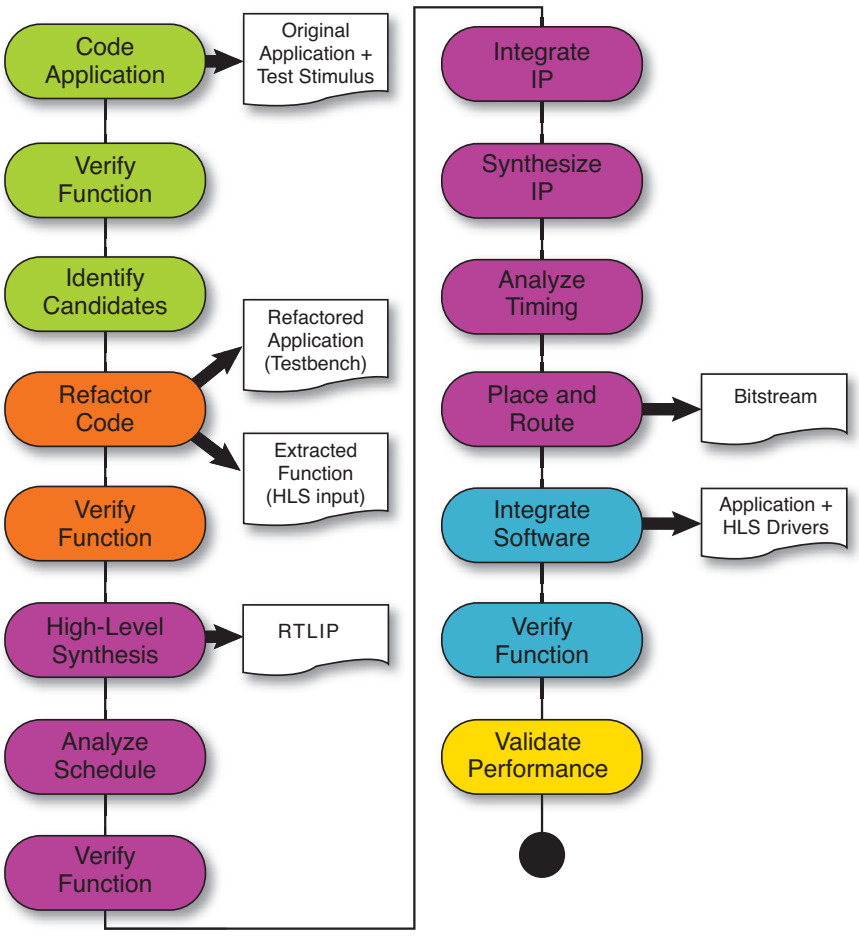


Figure 5 – Steps in design flow

Xilinx Opens a Tcl Store

by Greg Daughtry
Director of Product Marketing
Xilinx, Inc.
greg.daughtry@xilinx.com

Open-source repository for sharing Tool Command Language scripts is up and running at GitHub.com.

Over the last five years Xilinx has had a strategic focus on design methodology and tools to address productivity, to accelerate the design cycle and to help bring products to market faster by providing the industry's most advanced and comprehensive development environment.

Even with the productivity improvements of the next-generation Vivado® Design Suite combined with the comprehensive UltraFast™ Design Methodology, designing with today's All Programmable devices can be challenging. Designers must integrate hundreds of highly parameterized IP cores, hundreds of thousands of placeable objects and multiple millions of logic cells with Xilinx® All Programmable FPGAs, 3D ICs and SoCs. There are an infinite number of permutations to grapple with as designers push the boundaries with complex designs.

With the release of Vivado 2014.1 in April, Xilinx is taking another large step forward in designer productivity by hosting an open-source repository for sharing Tool Command Language (Tcl) code. This repository, called the Xilinx Tcl Store, will make it a lot easier to find and share Tcl (pronounced "tickle") scripts that other engineers have developed. With the power of Tcl, these scripts can extend the considerable core functionality of the Vivado Design Suite, enhancing productivity and ease of use. The Tcl Store is open to the user community to contribute to the greater good of all designers by publishing Tcl code that others might find useful.

DESIGNS GROWING MORE COMPLEX

The Vivado Design Suite was built on an open, scalable data model. As an open system, one of the keys to enabling productivity is making the tools smarter, and providing more customization choices and analysis capabilities so the designer can be better informed and drive the tools to provide optimal implementations.

Since the release of the Vivado Design Suite in 2012, there has been an explosion of Tcl scripting to perform tasks both small and large. It's increasingly important for designers to understand and utilize Tcl, since this is the basis for Vivado's XDC constraint language.

The Tcl commands allow you to develop and scrub timing constraints interactively, which saves compilation time and debug effort. The core commands allow object queries that can be used for custom reporting, and that can execute very elaborate tool control. The Vivado Design Suite makes it possible to also develop your own DRC and linting checks, along with highly customized flows to achieve better quality of results or faster run-times. Tcl also enables designers to make targeted design changes through engineering change order (ECO) operations.

The increased productivity provided with Tcl, ease of creation and readability make this language prime for the sharing of useful code. Up to now this sharing has largely occurred on an ad hoc basis, via e-mail and user forums. Some companies have established their own internal libraries of Tcl for use within their projects.

Now Xilinx is taking Tcl sharing to the next level with its new Xilinx Tcl Store.

WELCOME TO THE TCL STORE

The Xilinx Tcl Store provides examples of how to write custom reports, control specific tool behavior, make custom netlist changes and integrate with third-party electronic design automation (EDA) tools such as simulation, synthesis, timing and power analysis, and linting tools.

Natively accessed from the Vivado Integrated Design Environment (IDE), the Tcl Store enables users to select and install collections of Tcl scripts called "apps" directly from within the tool. Once installed, these apps have commands that appear just like built-

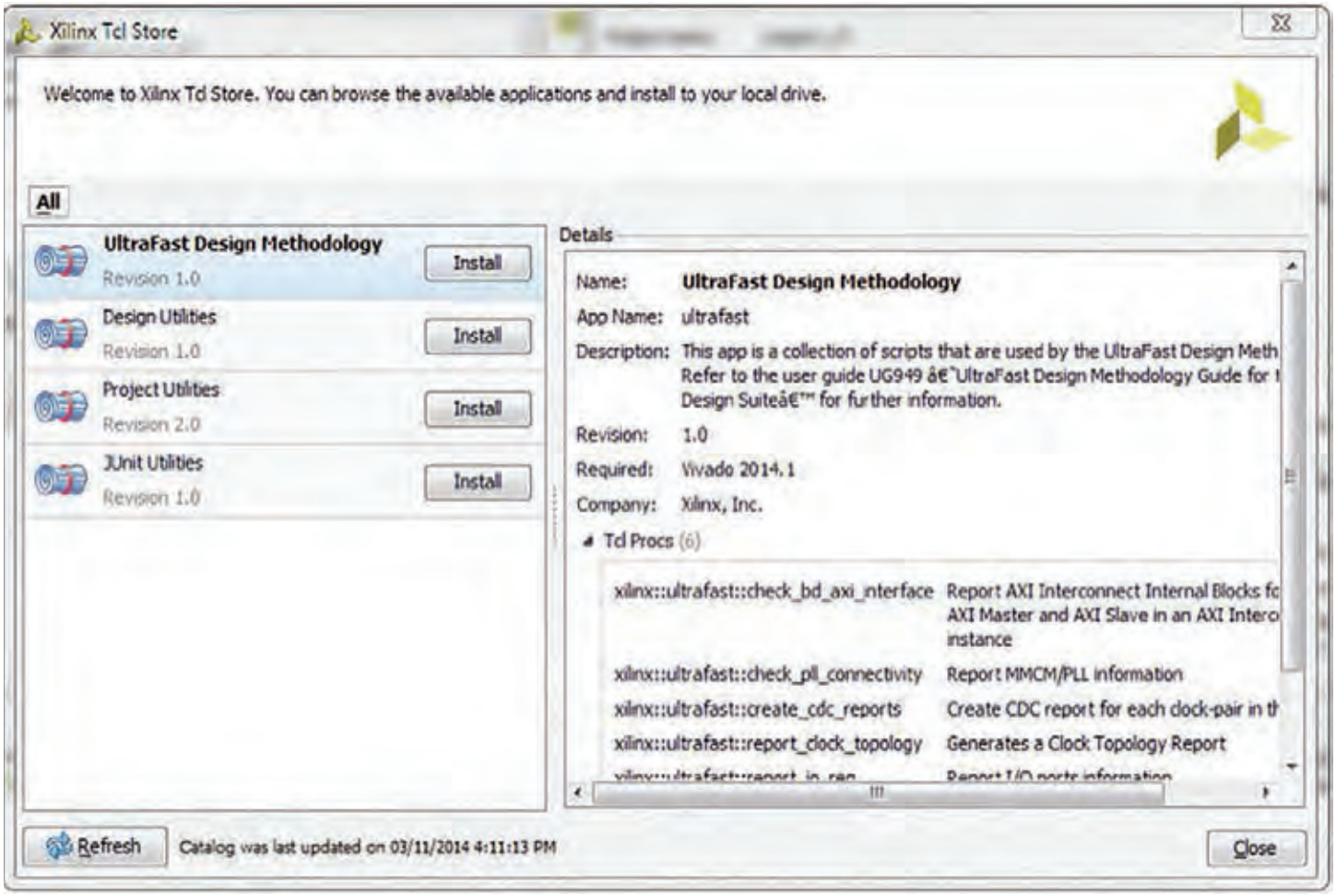


Figure 1 – The Tcl Store dialog box in the Vivado IDE allows installation of apps and browsing of the commands.

in Vivado Design Suite commands, right down to the help infrastructure. Vivado Design Suite supports different versions of apps using standard package facilities of Tcl, so if a newer version is released you can choose to upgrade with a single mouse click.

The Xilinx Tcl Store is intended to make it easier to find and use well-crafted Tcl scripts developed and supported by the user community, in the same manner as Linux development. Tcl scripting is a little more advanced than selecting IDE buttons. However, it is easy to learn. Documentation and user guides provide details on specific commands from the Tcl API and can be found on *xilinx.com/support*.

Let's take a closer look at the infrastructure for installing and using Tcl apps from the Xilinx Tcl Store.

INSTALLATION AND USE

Designers can access the Xilinx Tcl Store by means of an icon on the Getting Started page when you first launch the Vivado IDE. Alternatively, you may also go to the Tools Menu and select the "Xilinx Tcl Store" menu option. This will bring up the repository dialog box, which will give you a list of apps available to install (Figure 1).

As you browse the list of apps, within each app there is a list of commands (called "procs" in Tcl) that are available for execution. You will see a description of each app, and of each proc within the app, to get an idea of what it does. Click on the install button to install the app and register it so that it now shows up like native Vivado Design Suite commands. Once an app is installed, each time you start the Vivado Design Suite it loads automatically—there is no need to install the app each time you start a new session.

Procs have a naming convention that uses a facility in Tcl called namespaces. The names of the commands may seem a little more complex than normal Tcl commands, and have "::" characters embedded in them. For example, xilinx::ultrafast::check_pll_connectivity runs some connectivity checks on the clock-modifying blocks in Xilinx devices. The naming conventions serve to make sure the Tcl code is unique and that a proc in one app does not conflict with another proc by the same name in another app. Namespaces are a standard feature of Tcl.

To execute an app command, type in the fully qualified name of the proc in-

Usage of Tcl apps in the Xilinx Tcl Store is meant to be easy and simple. Xilinx's goal is to encourage use and sharing among development teams around the world to improve productivity.

cluding the namespace, and optionally pass in any required arguments, just like other Tcl commands. Since the commands are using standard namespaces, you can also choose to import the commands into the global namespace. The strategy will work fine if there are no conflicts between any other command names. This will allow you to omit the namespace qualifier and use the proc name alone. In the example above, if you imported the UltraFast app into the global namespace, you could call the check_pll_connectivity command directly without the namespace qualifers.

Designers can uninstall apps with a single click of the "Uninstall App" hyperlink within the details section of the app. There is also a "Refresh" button to update the catalog. The Tcl Store catalog is hosted on a third-party website that provides the ability to push out updates to app revisions independently of Vivado Design Suite releases. If the catalog is refreshed, the Vivado tools will perform a lightweight synchronization of the list of apps. If an updated version of an installed app is available, use the "Update" button to acquire it. The Vivado Design Suite will copy and sync the latest version of the app and install it. To avoid configuration control issues, upgrades are only installed at the designer's request. For those who are concerned about security and would prefer to keep the Vivado Design Suite from syncing outside of their network firewall, there is a parameter to disable the catalog synchronization.

Usage of Tcl apps in the Xilinx Tcl Store is meant to be easy and simple. Xilinx's goal is to encourage use and sharing among development teams around the world to improve productivity. Only the latest version of any given app is displayed and designers can only install or upgrade to the latest supported version. Of course, the best way to have good usage is to ensure that there is a rich library of useful code. Xilinx has seeded the repository with a collection of helpful utility and integration scripts that you can peruse as good examples of how to build your own reusable Tcl scripts.

CONTRIBUTING TO THE TCL STORE

There are two ways to contribute to the Tcl Store and make your script available to all Vivado Design Suite users. The first is to modify an app that already exists. The second is to develop and submit a request for a new app. To contribute code to the repository, you need to have some level of comfort with software development tools for revision control, or at least a willingness to learn.

Each app is controlled by a single person, usually the person who authored most of the code, referred to as the "app owner." The Xilinx repository as a whole is controlled by Xilinx, and the company maintains a process for releasing the apps into the public domain to enforce basic consistency across the apps. Xilinx employees will perform a "gatekeeper" role to ensure quality.

The "contributor" who wishes to modify an existing app or add a new one will work with the gatekeeper and app owner for the submission, consistent with the process on other open-source projects. A wiki on the site where the code is hosted documents this process.

Basic requirements will be enforced for all code submissions. Xilinx has attempted to keep this list—which is subject to change—as small as possible, while still ensuring a reasonable user experience. Here is the list of basic app requirements you need to adhere to:

- Follow basic coding-style guidelines by using procs with arguments that do not use or access global variables.
- Include basic documentation inside the proc that describes what it does, what the arguments are and what it returns.
- Make sure code passes a basic syntax check, and also passes a linting tool that is provided as a part of the Vivado Design Suite.
- Provide a minimum of one basic test for each proc that ensures the code at least runs and does what is expected.

THE TCL STORE ON GITHUB

The Xilinx Tcl Store is hosted on a third-party website called GitHub.com. The store uses revision-control tools to ensure distributed development happens in a controlled way. The key to this process is Git, a popular open-source, distributed revision-control tool that is commonly used for Linux. To access the repository for contribution and testing, you register for a free account on GitHub.com, and install and set up Git. GitHub provides an installation of Git tools for Windows

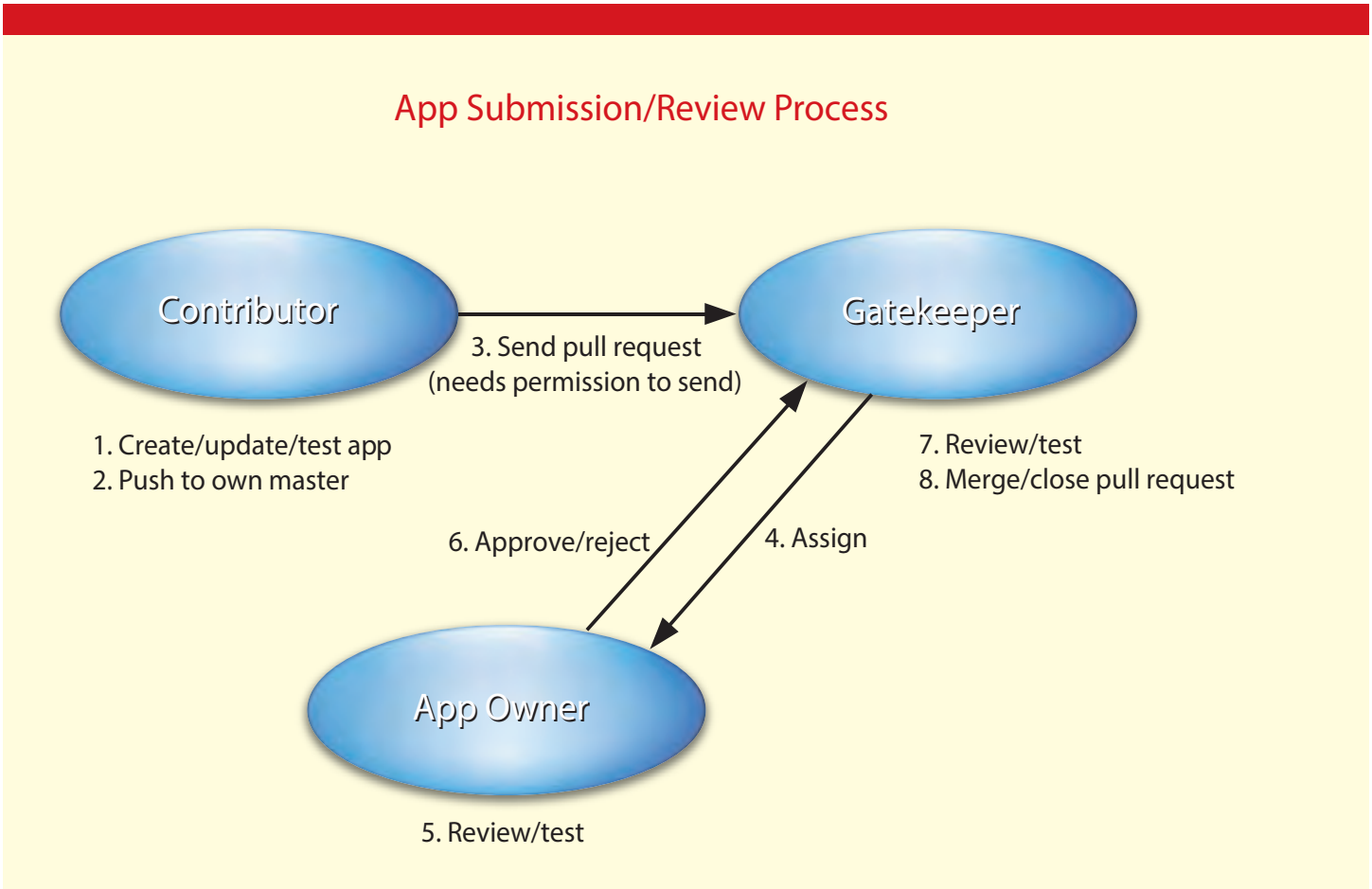


Figure 2 – Workflow of the Xilinx Tcl Store submission process passes through several discrete steps.

PCs. Linux machines typically already have it or can install it through standard packages. GitHub provides tutorials to help you get started with Git.

Once you have a GitHub account, here are the steps for contributing to the Tcl Store repository:

1. Clone the Xilinx Tcl Store master repository. This creates a local copy that is your sandbox and it allows you to develop locally and test without impacting others.
2. Place your new code in the correct directory, following the established guidelines based on app name and company or GitHub username. Use standard Git add commands.
3. Use Vivado Design Suite in the local repository, and call a few commands that are necessary for registering the code and generate a catalog.xml file. This is one of three files that you will need. The others are a package index and a Tcl index.
4. Open Vivado Design Suite in another location, point to the local repository and test your apps. Run the linter and your local tests until you are comfortable that all is working correctly.
5. Commit your changes and provide a message briefly documenting them.
6. Send an e-mail requesting permission to contribute to *tclstore@xilinx.com*. Indicate whether you'd like to create a new app and what you'd like to call it. If you'd like to modify or contribute to an existing app, please indicate that; you will need permission from the app owner.
7. Go to GitHub.com using a Web browser and issue a pull request. This formally initiates the process of merging your contributions into the repository. Work with the gatekeeper and app owner as appropriate to resolve any issues through GitHub and e-mail.
8. Congratulations! It feels good to help your fellow designers.

Figure 2 shows a basic diagram of the workflow showing the submission process.

THE FINE PRINT

The Xilinx Tcl Store is open source, and there is no facility to monetize or charge for contributions. Apps contributed to the Tcl Store are made freely available for derivative works through a BSD license commonly used in open-source projects.

Contributions to the repository will include a version of the BSD license with each app in order for it to be accepted and published publicly. If a company or user does not wish to release their intellectual property into the public domain, Vivado Design Suite does support local versions of the repository via the same mechanism that is used for testing prior to issuing a pull request.

Furthermore, since the project uses GitHub for hosting, submitters must agree to the GitHub terms of service when you register for an account, as this is a third-party service.

The apps in the app store are developed and supported by the user community. This means that Xilinx technical support has not received training on this functionality and will not be able to answer questions about Tcl code. Please direct support questions for these apps to the Xilinx user forums. If a bug or issue exists in a piece of code, you can file and track it directly in GitHub.com projects. Since this is an open-source development model, users are encouraged to fix these issues and improve the experience for the overall good of other users—just like Linux.

ROAD MAP

The Vivado 2014.1 introduction of the Tcl Store is just the beginning. Xilinx will be improving the Tcl Store this year by implementing the ability to search the descriptions of the apps and procs to make it easier to find functions. We will be providing a way to browse and view the source code without having to install the app. In addition, we intend to provide a review mechanism where users can specify a rating of one to five stars, and optionally provide a written review. This will give people feedback mechanisms on the more popular submissions.

We will also make use of better categorization with filtering capability based on the categories of apps, for example simulation, synthesis, implementation, project and netlist utilities. As the repository grows, we may institute more groupings and extend the taxonomy of apps to reflect the contributions. We want to make it as easy as possible to contribute apps, so we may look for ways to allow people to submit Tcl scripts by e-mail with a minimal support burden and no need to go through GitHub. The uncontrolled nature of such a process would not be coupled with the current installation scheme, and would perhaps be best suited for examples.

Thousands of designers around the world are using the Vivado Design Suite and hundreds of companies have adopted the UltraFast Design Methodology. The Xilinx Tcl Store will continue to increase designer productivity by providing a new open-source project between Xilinx, its partners and our customers aimed at sharing Tcl scripts.

INFORMATION AND RESOURCES

To request GitHub account access and to see tutorials on Git and GitHub, go here:

https://github.com/

The Xilinx Tcl Store code repository as well as a wiki that documents how to contribute are located here:

https://github.com/Xilinx/Xilinx-TclStore

The Xilinx Tcl Store wiki contains detailed information concerning the contribution process:

https://github.com/Xilinx/Xilinx-TclStore/wiki/Xilinx-Tcl-Store-Home

UG 894, the "Using Tcl Scripting Guide," contains information on the general scripting capabilities of the Vivado Design Suite:

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug894-vivado-tcl-scripting.pdf

UG 835, "Tcl Command Reference," contains information on all of the native Tcl commands available in Vivado Design Suite:

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug835-vivado-tcl-commands.pdf

What's New in the Vivado 2014.1 Release?

Xilinx is continually improving its products, IP and design tools as it strives to help designers work more effectively. Here, we report on the most current updates to Xilinx design tools including the Vivado® Design Suite, a revolutionary system- and IP-centric design environment built from the ground up to accelerate the design of Xilinx® All Programmable devices. For more information about the Vivado Design Suite, please visit www.xilinx.com/vivado.

Product updates offer significant enhancements and new features to the Xilinx design tools. Keeping your installation up to date is an easy way to ensure the best results for your design.

The Vivado Design Suite 2014.1 is available from the Xilinx Download Center at www.xilinx.com/download.

VIVADO DESIGN SUITE 2014.1 RELEASE HIGHLIGHTS

Vivado Design Suite 2014.1 increases your productivity with faster run-times, improved quality of results, automation of the UltraFast™ Design Methodology and hardware acceleration of OpenCL kernels through Vivado high-level synthesis (HLS).

DEVICE SUPPORT

Production Ready:

- Artix®-7 XC7A35T and XC7A50T
- XA Artix-7 XA7A50T, XA7A35T and XA7A75T
- Zynq®-7000 XC7Z015

General Access:

Kintex® UltraScale:

- XC KU035, XC KU040, XC KU060 and XC KU075

Early Access (contact local sales rep):

Kintex UltraScale SSI devices:

- XC KU100 and XC KU115

Virtex® UltraScale devices:

- XC VU065, XC VU080, XC VU095, XC VU125, XC VU145 and XC VU160

XILINX Tcl STORE

Xilinx is taking another large step forward in designer productivity by hosting an open-source repository for sharing Tool Command Language (Tcl) code. This repository, called the Xilinx Tcl Store, will make it a lot easier to find and share Tcl scripts that other engineers have developed. With the power of Tcl, these scripts can extend the considerable core functionality of the Vivado Design Suite, enhancing productivity and ease of use. The Tcl Store is open to the user community to contribute to the greater good of all designers by publishing Tcl code that others might find useful.

The Xilinx Tcl Store provides examples of how to write custom reports, control specific tool behavior, make custom netlist changes and integrate with third-party electronic design automation (EDA) tools such as simulation, synthesis, timing and power analysis, and linting tools. Natively accessed from the Vivado Integrated Design Environment (IDE), the Tcl Store enables users to select and install collections of Tcl scripts called "apps" directly from within the tool. Once installed, these apps have commands that appear just like built-in Vivado Design Suite commands.

To learn more about the new Xilinx Tcl Store, watch the QuickTake video at *http://www.xilinx.com/training/vivado/introduction-to-the-xilinx-tcl-store.htm*.

VIVADO DESIGN SUITE: DESIGN EDITION UPDATES

Vivado Implementation Tools

Performance and run-time improvements:

- Average 25 percent faster overall implementation run-time compared to 2013.4
- Average 2.5 percent better Fmax on 7 Series SSIT devices
- Average 5 percent improvement in Fmax across all devices.

Integrated Design Environment

Timing Constraints Wizard: An automated tool that guides users to create timing constraints for clocks, I/O and clock domain crossing constraints. Intelligence built into the wizard queries the Vivado Design Suite's design database to extract the clocking structure as well as existing constraints, often coming from IP reuse, then guides the user to correctly constrain the rest of the design.

To learn more about the new Timing Constaints Wizard, watch the QuickTake video at *http://www.xilinx.com/training/vivado/using-the-vivado-timing-constraint-wizard.htm*.

Vivado IP Integrator

- New "Signals" tab allows drag-and-drop connection, visualization and management of clock and reset domains in a design.
- New automated "Board Interface" tab allows quick connection to interfaces available on supported development boards.
- Designer Assistance now provides an option for users to specify a clock domain instead of assuming a default domain.

Tandem Configuration for Xilinx PCIe IP

- The Tandem Configuration IP core has been included within the IP Integrator. This core, which is specifically the AXI streaming variant, can be added to a design within IPI.
- Support has been added for Zynq®-7000 All Programmable SoC devices.
- For more information, see the PCI Express® IP Product Guides.

VIVADO DESIGN SUITE: SYSTEM EDITION UPDATES

Vivado High-Level Synthesis

Vivado HLS now offers early-access support of OpenCL kernels. OpenCL provides a framework and language for writing kernels that execute across heterogeneous platforms and can now be seamlessly converted to IP running on Xilinx All Programmable devices.

A new linear algebra library enables rapid IP generation of C/C++ algorithms that require functions such as Cholesky decomposition, singular-value decomposition (SVD), QR factorization and matrix multiplication.

Smoother integration of HLS designs into AXI4 systems occurs through new data-packing options that automate the alignment of data to 8-bit boundaries. Enhanced functionality is provided for AXI4 master interfaces as the USER ports can now be optionally included in the interface.

Improved resource usage is provided for designs using division operations. These operations now automatically benefit from smaller implementations.

System Generator for DSP

System integration of System Generator for DSP blocks is now faster and easier with the AXI4-Lite slave interface and corresponding software drivers for both Linux and bare-metal designs. Verification is improved thanks to the support for non-memory-mapped interfaces in hardware co-simulation.

Plug-and-Play IP Updates

Vivado 2014.1 provides increased quality and features for UltraScale™ GT-based IP:

- GT Wizard queries the device model at run-time for accurate physical resources and location.
- All GT-based IP cores call the GT Wizard at run-time
- Clocking and reset resources can be easily shared between GT instances.
- All GT ports can be enabled for debug.
- There is no need to edit any IP file.

Additional new key IP available for UltraScale devices in 2014.1 includes the HSSIO Wizard, System Management Wizard, SGMII over LVDS, Aurora 8B10B and 64B66B, CPRI and Serial RapidIO.

ULTRAFAST DESIGN METHODOLOGY

Second Edition of the UltraFast Design Methodology

Xilinx has delivered the first comprehensive design methodology in the programmable industry with its UltraFast technology. Xilinx hand-picked the best practices from experts and distilled them into this authoritative set of methodology guidelines for the Vivado Design Suite.

Now in its second edition, the UltraFast Design Methodology Guide extends support of the UltraScale architecture, adds a new Timing Constraints Wizard for rapid timing closure and includes new best practices, such as:

- Design methodology DRCs
- Revision control
- IP / IP Integrator methodology
- Simulation (including third-party flows)
- Verification
- Vivado HLS
- Partial reconfiguration

TAKE THE NEXT STEP

Vivado QuickTake Tutorials

For more information, watch the What's New in Vivado Design Suite video at *http://www.xilinx.com/training/vivado/whats-new-in-vivado.htm*.

Vivado Design Suite QuickTake video tutorials are how-to videos that take a look inside the features of the Vivado Design Suite. New topics include: Design Flow Overview, Using the Timing Constraints Wizard, Xilinx Tcl Store, Using Vivado with Xilinx Evaluation Boards and Packaging Custom IP for use with IP Integrator. See all QuickTake videos at *http://www.xilinx.com/training/vivado*.

Vivado Training

For instructor-led training on the Vivado Design Suite, please visit *www.xilinx.com/training*.

Download Vivado Design Suite 2014.1 today at *http://www.xilinx.com/download*.

Latest and Greatest from the Xilinx Alliance Program Partners

Xpedite highlights the latest technology updates from the Xilinx Alliance partner ecosystem.

The Xilinx® Alliance Program is a worldwide ecosystem of qualified companies that collaborate with Xilinx to further the development of All Programmable technologies. Xilinx has built this ecosystem, which leverages open platforms and standards, to meet customer needs and is committed to its long-term success. Alliance members—including IP providers, EDA vendors, embedded software providers, system integrators and hardware suppliers—help accelerate your design productivity while minimizing risk. Here are some highlights of recent alliance activities.

XYLON AND NORTHWEST LOGIC DELIVER MIPI CSI-2 CAMERA INTERFACE DEMO AT NAB 2014

https://www.youtube.com/watch?v=aKbkB7WN8CE

The MIPI Display Serial Interface (DSI) and Camera Serial Interface 2 (CSI-2) are becoming key, low-cost industry standards for connecting video displays and cameras to a wide variety of embedded systems. You can now leverage a MIPI-compatible interface implemented on the Xilinx platform. This low-cost way to interface multiple cameras and displays greatly enhances the use of Xilinx All Programmable devices at the heart of diverse Smarter Vision systems for all sorts of markets including automotive, industrial and medical. In early April at the NAB 2014 show in Las Vegas, visitors to the Xilinx booth saw the demo system based on a Xilinx ZC706 Zynq SoC Evaluation Kit developed by Premier Xilinx Alliance Program members Xylon and Northwest Logic. Please click on the link above to view a short video explaining the demo.

BARCO-SILEX DEMONSTRATES 4K VIDEO-OVER-IP WITH JPEG2000 ON A SINGLE KINTEX-7 AT NAB 2014

http://www.barco-silex.com/ip-cores/jpeg-2000

Barco-Silex upgraded to 4K its well-known video-over-Internet Protocol (VoIP) reference design that integrates JPEG2000 compression and Transport Stream solutions with Xilinx SMPTE 2022 cores on a single Kintex®-7 device. Fully compliant with the Video Services Forum (VSF) technical recommendation, this design ensures interoperability with major broadcast industry players as it was demonstrated at the VidTrans 2014 meeting in Arlington, Va., in February and again at NAB 2014 in April. This new version of the reference design combines JPEG2000 compression and flexible MPEG2-TS cores from Barco-Silex with the SMPTE2022 1-2 IP core from Xilinx on a Kintex-7 FPGA to carry a 4K video stream over a 1G network. The 4K video is captured via Quad-SDI, and is encoded by the high-quality Barco-Silex JPEG2000 encoder core. The integrated Barco-Silex Transport Stream solution combined with the Xilinx SMPTE2022 1-2 core provides interoperability thanks to its proven compliance with the VSF technical recommendations.

TOPIC SHOWCASES DYPLO SYSTEM AT EMBEDDED WORLD 2014

https://www.youtube.com/watch?v=8S-1GOcL-t4o

Premier Alliance member TOPIC Embedded Products has developed an operating system that will significantly reduce the

development time and cost of creating products based on FPGA-and-processor combinations, such as the Xilinx Zynq®-7000 All Programmable SoC. The Dyplo system's OS bridges the gap between hardware and software design, and provides a means to enable a fully software-driven development flow. Dyplo extensively uses partial reconfiguration, an advanced design technique in which the FPGA fabric can (partially and selectively) change its hardware configuration on the fly. Partial reconfiguration makes it possible to execute different functions by reusing the same FPGA fabric over time. Dyplo manages the reconfigurable blocks such that functions can be executed as desired, either in software or in hardware, depending on the execution context constraints, such as power consumption, performance and footprint. The link above will bring you to a short video explaining the demo.

DAVE EMBEDDED SYSTEMS FEATURES BORA SOM AT EMBEDDED WORLD 2014

http://www.xilinx.com/alliance/memberlocator/1-33H1QG.htm

The Italian company DAVE Srl showcased BORA, a system-on-module (SoM) equipped with Xilinx Zynq XC7Z010 and XC7Z020 devices, at Embedded World 2014 in Nuremburg, Germany, in February. The key benefits for customers are the shortened development time, lower costs and reduced engineering resources that come with using a compact and integrated one-chip solution that includes both a CPU (the onboard ARM® Cortex™-A9 processor) and an FPGA. Customers also avoid manufacturing complexities with DAVE's SoM solution. Additionally, BORA has both Linux and a real-time operating system (RTOS) running simultaneously on the same SoC. See this application note for more details: *http://www.dave.eu/sites/default/files/files/an-belk-001-amp-li-nux-freertos.pdf*.

INTERFACE CONCEPT'S MTCA.4 VIRTEX-7 AMC MODULE SELECTED BY SLAC FOR ACCELERATOR DIAGNOSTICS

http://www.gomaelettronica.it/en.html

Diagnostics and beam control in high-energy physics require ever higher signal-processing power. The high-energy physics community has defined a new MTCA.4 standard that specifies extensions to MicroTCA to support uRTM applications. The SLAC National Accelerator Laboratory in Menlo Park, Calif., has selected a module from Alliance Program member Interface Concept for assisting in the diagnostics of high-speed particles. In partnership with Deutsches Elektronen-Synchrotron (DESY), Interface Concept has developed a new Virtex®-7 MTCA.4 and a four-channel, 1,300-Msps 12-bit resolution ADC FPGA mezzanine card (FMC) with a sophisticated clock system that allows for synchronization (up to eight channels using two FMCs).

The MTCA.4 carrier features a Virtex-7 VX690T (Speed Grade -2) with 3.25 Gbytes of DDR3 memory running at 1,600 MT/s (spread on two banks) and a QSPI flash. Each HPC FMC interface, compliant with VITA 57, provides eight GTH transceivers and 80 LVDS lanes. The fabric links are composed of 16 GTH transceivers and the uRTM interface provides four GTH transceivers and 38 LVDS lanes. Onboard memories allow you to store up to three selectable FPGA images. A module Management Controller Unit carries out the onboard power and temperature monitoring, Virtex-7 configuration and IPMI interface (MMC v1.0 compliant). Additionally, Interface Concept provides VHDL code for system services and reference designs such as PCIe DMA engines, signal capture and processing. Example code is available for each FPGA interface, facilitating design and integration for customers.

GET PUBLISHED

WOULD YOU LIKE TO BE PUBLISHED IN XCELL PUBLICATIONS?

It's easier than you think!

Submit an article draft for our Web-based or printed Xcell Publications and we will assign an editor and a graphic artist to work with you to make your work look as good as possible.

For more information on this exciting and highly rewarding program, please contact:

Mike Santarini
Publisher, Xcell Publications
xcell@xilinx.com

Xcell PUBLICATIONS

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.

XAPP1177: DESIGNING WITH SR-IOV CAPABILITY OF XILINX VIRTEX-7 PCI EXPRESS GEN3 INTEGRATED BLOCK

http://www.xilinx.com/support/documentation/application_notes/xapp1177-pcie-gen3-sriov.pdf

Evaluating single-root I/O virtualization (SR-IOV) capability can be a complex process, with many variations seen among different operating systems and system platforms. In demonstrating the SR-IOV capability of the Xilinx® Virtex®-7 FPGA PCI Express® Gen3 integrated block, this application note establishes a baseline system configuration and provides the necessary software to quickly bring up and evaluate the SR-IOV features of the Virtex-7 FPGA PCIe® Gen3 integrated block.

Author Vivek Surabhi explains the key concepts of SR-IOV and details how to configure the SR-IOV capability. The document shows how to create a PCI Express x8 Gen3 endpoint design configured for two physical functions and six virtual functions. The reference design, which targets a Virtex-7 FPGA VC709 Connectivity Kit, has been hardware-validated on a system with SR-IOV capability.

XAPP1171: PCI EXPRESS ENDPOINT-DMA INITIATOR SUBSYSTEM

http://www.xilinx.com/support/documentation/application_notes/xapp1171-pcie-central-dma-subsystem.pdf

This application note by Brian Martin demonstrates a Vivado® Design Suite subsystem for endpoint-initiated direct memory access (DMA) data transfers through PCI Express. The provided subsystems target the Zynq®-7000 All Programmable SoC ZC706 and Kintex®-7 KC705 device to initiate data transfers between DDR3 memory and an externally connected PCI Express Root Complex. You could modify the subsystem for use in other devices or applications that require data transactions to be initiated from within the FPGA logic.

The application note demonstrates several key features of the Vivado Design Suite and the IP cores used in the design, starting with generating a block diagram subsystem using Vivado's Tcl commands and scripting. Other areas covered include PCI Express endpoint configuration; DMA-initiated data transfers over PCI Express; and achieving high throughput into the Zynq SoC processing system through the high-performance AXI interface. The author also explores dynamic address translation between a 64-bit root complex (host) address space and a 32-bit FPGA (AXI) address space, and outlines a methodology to perform DMA scatter-gather operations using dynamic address translation.

XAPP1180: REFERENCE SYSTEM: KINTEX-7 MICROBLAZE SYSTEM SIMULATION USING IP INTEGRATOR

http://www.xilinx.com/support/documentation/application_notes/xapp1180.pdf

This application note and reference system demonstrate the functionality of a MicroBlaze™ processor system on the Kintex-7 device architecture using the Xilinx IP Integrator tool in simulation and in hardware. The system includes common peripherals such as main memory as well as RS232 communications. Author James Lucero provides several standalone software applications to verify the functionality of the peripherals. Applications include hello_uart and hello_mem.

Lucero also explains how to set up the simulation environment for the system, execute the simulation using either the Vivado simulator or Mentor Graphics' ModelSim® environments, and run the design on hardware. The document describes running the design in simulation and hardware, targeting the KC705 board that contains the Kintex-7 XC7K410TFFG900-2 FPGA.

XAPP1158: USING VXWORKS BSP WITH ZYNQ-7000

http://www.xilinx.com/support/documentation/application_notes/xapp1158-zynq-7000-vxworks-bsp.pdf

Here is a startup guide for new users of VxWorks, the real-time operating system (RTOS) from Wind River, on the Zynq-7000 All Programmable SoC. Authors Uwe Gertheinrich, Simon George and Kester Aernoudt provide step-by-step instructions for running the VxWorks 6.9.3.1 board support package (BSP) on the Zynq SoC, and additionally provide an overview of the boot process. The application note starts by explaining the important elements of the Zynq SoC software environment to provide a better understanding of BSP and application generation. The authors explain the Zynq SoC processor subsystem boot process and describe how to add VxWorks, including building and debugging the application as well as remotely running a custom application of VxWorks on the Zynq SoC.

XAPP742: AXI VDMA REFERENCE DESIGN

http://www.xilinx.com/support/documentation/application_notes/xapp742-axi-vdma-reference-design.pdf

If you've ever contemplated building a video system using Xilinx native video IP cores to process configurable frame rates and resolutions in Kintex-7 FPGAs, this application note will show you how. The reference design focuses on run-time configuration of an onboard clock generator for a video pixel clock, and on using video IP cores such as AXI Video Direct Memory Access (VDMA), Video Timing Controller (VTC), test pattern generator (TPG) and the DDR3 memory controller for running selected combinations of video resolution and frame rate. Authors Pankaj Kumbhare and Vamsi Krishna discuss the configuration of each video IP in detail, helping designers make effective use of these cores. Each video IP block is configured dynamically to process various combinations of frame rate and resolution.

XAPP1200: KINTEX-7 FPGA TRANSCEIVER WIZARD EXAMPLE DESIGN

http://www.xilinx.com/support/documentation/application_notes/xapp1200-k7-xcvr-wiz-example-design.pdf

The KC705 Evaluation Kit provides a comprehensive, high-performance development and demonstration platform using the Kintex-7 FPGA family for high-bandwidth and high-performance applications in multiple market segments. This application note by Dinesh Kumar and Thupalli Ramachandra uses the KC705 kit and the GTX Transceiver Wizard to demonstrate a transceiver example design running on Kintex-7 FPGA hardware. The authors have fully verified the reference design and tested it on hardware.

XAPP1202: SYSTEM TRAFFIC GENERATION AND PERFORMANCE MEASUREMENT

http://www.xilinx.com/support/documentation/application_notes/xapp1202-sys-tg-pm.pdf

In this application note, authors Kondalarao Polisetti and Pankaj Kumbhare demonstrate AXI4 system traffic generation and performance measurement using two Xilinx cores: the AXI Traffic Generator (ATG) and the AXI Performance Monitor (APM). The accompanying reference design focuses on run-time configuration for different instances of ATG and APM, and shows how to configure and program these IP cores to get the system performance metrics. The reference design also shows the run-time system throughput and latency of the system for different configurations using a Web server application. The authors used the Vivado Design Suite 2013.4 to successfully place and route the interface at 100 MHz on the main AXI4 interfaces to the memory controller.

XAPP1199: SMPTE 2022-5/6 HIGH-BIT-RATE MEDIA TRANSPORT OVER IP NETWORKS WITH FORWARD ERROR CORRECTION

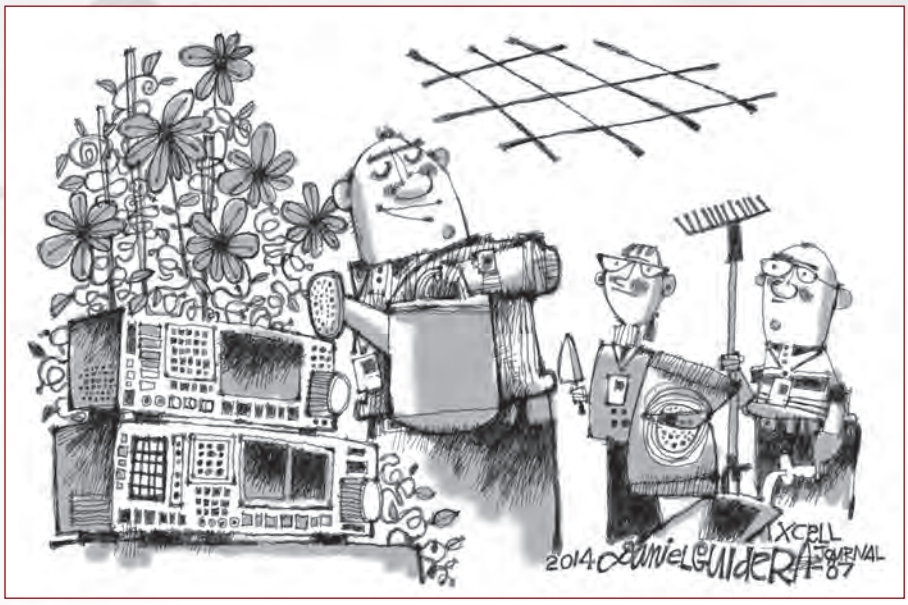
http://www.xilinx.com/support/documentation/application_notes/xapp1199-smpte2022-56-over-ip.pdf

Here's a video-over-IP network system that leverages the performance features of the LogiCORE™ IP SMPTE 2022-5/6 video-over-IP transmitter and receiver cores. The reference design by authors Gilbert Magnaye, Josh Poh, Myo Tun Aung and Tom Sun focuses on high-bit-rate, native media transport over 10-Gbps Ethernet with a built-in forward error correction (FEC) engine. The design is able to support up to three SD/HD/3G-SDI streams.

The transmitter platform uses three SMPTE SDI cores to receive the incoming SDI video streams. The received SDI streams are multiplexed and encapsulated into fixed-size datagram packets by the SMPTE 2022-5/6 video-over-IP transmitter core and sent using the 10-Gigabit Ethernet MAC core. The 10-Gbit link is supported by a 10-Gigabit Ethernet PCS/PMA core using an optical cable connected to the receiver end. On the receiver platform, the 10-Gigabit Ethernet MAC collects the Ethernet datagram packets. The SMPTE 2022-5/6 video-over-IP receiver core filters the datagram packets, and de-encapsulates and demultiplexes the datagrams into individual streams, then outputs those streams through the SMPTE SDI cores. The Ethernet datagram packets are buffered in DDR3 SDRAM for both the transmitter and receiver.

The DDR traffic passes through the AXI4 interconnect to the 7 series AXI memory controller. A MicroBlaze processor initializes the cores and reads the status. The reference design targets the Xilinx Kintex-7 FPGA KC705 Evaluation Kit.

Xpress Yourself in Our Caption Contest



DANIEL GUIDERA

Spring is in the air, even if your garden grows indoors. Xercise your funny bone as you tend your posies by submitting an engineering- or technology-related caption for this cartoon showing a couple of engineers watering their plants. The image might inspire a caption like "Once they cut static power and dynamic power in their design, Ernie and Frank decided to experiment with Flower Power."

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 will receive a Digilent Zynq Zybo board, featuring the Xilinx® Zynq®-7000 All Programmable SoC (*http://www.xilinx.com/products/boards-and-kits/1-4AZFTE.htm*). Two runners-up will gain notoriety, fame and a cool, Xilinx-branded gift from our swag closet.

The contest begins at 12:01 a.m. Pacific Time on April 16, 2014. All entries must be received by the sponsor by 5 p.m. PT on June 30, 2014.

So, put down your trowel and get writing!

PAUL McFARTHING, development engineer at Smith & Nephew (Andover, Mass.), won a shiny new Digilent Zynq Zybo board with this caption for the tattooed engineer in Issue 86 of *Xcell Journal*:

"I told you we didn't have to worry about the skin effect."

Congratulations as well to our two runners-up:

"Wearable computing is so passé. This is the real future of embedded computing."

— Chris Lee, technical leader, Cisco Systems (San Jose, Calif.)

"Looks great, huh? Only three ECOs, too. New personal record."

— David Riley, principal hardware engineer, Mantaro Networks (Philadelphia)

NO PURCHASE NECESSARY. You must be 18 or older and a resident of the fifty United States, the District of Columbia, or Canada (excluding Quebec) to enter. Entries must be entirely original. Contest begins on April 16, 2014. Entries must be received by 5:00 pm Pacific Time (PT) on June 30, 2014. Official rules are available online at www.xilinx.com/xcellcontest. Sponsored by Xilinx, Inc. 2100 Logic Drive, San Jose, CA 95124.

Synplify Premier
Put yourself in the driver's seat for Xilinx FPGA Design

- Unmatched quality of results for Xilinx® FPGAs
- Fastest turnaround time, without timing performance compromise
- Results preservation from one run to the next
- Implementation control when you need it

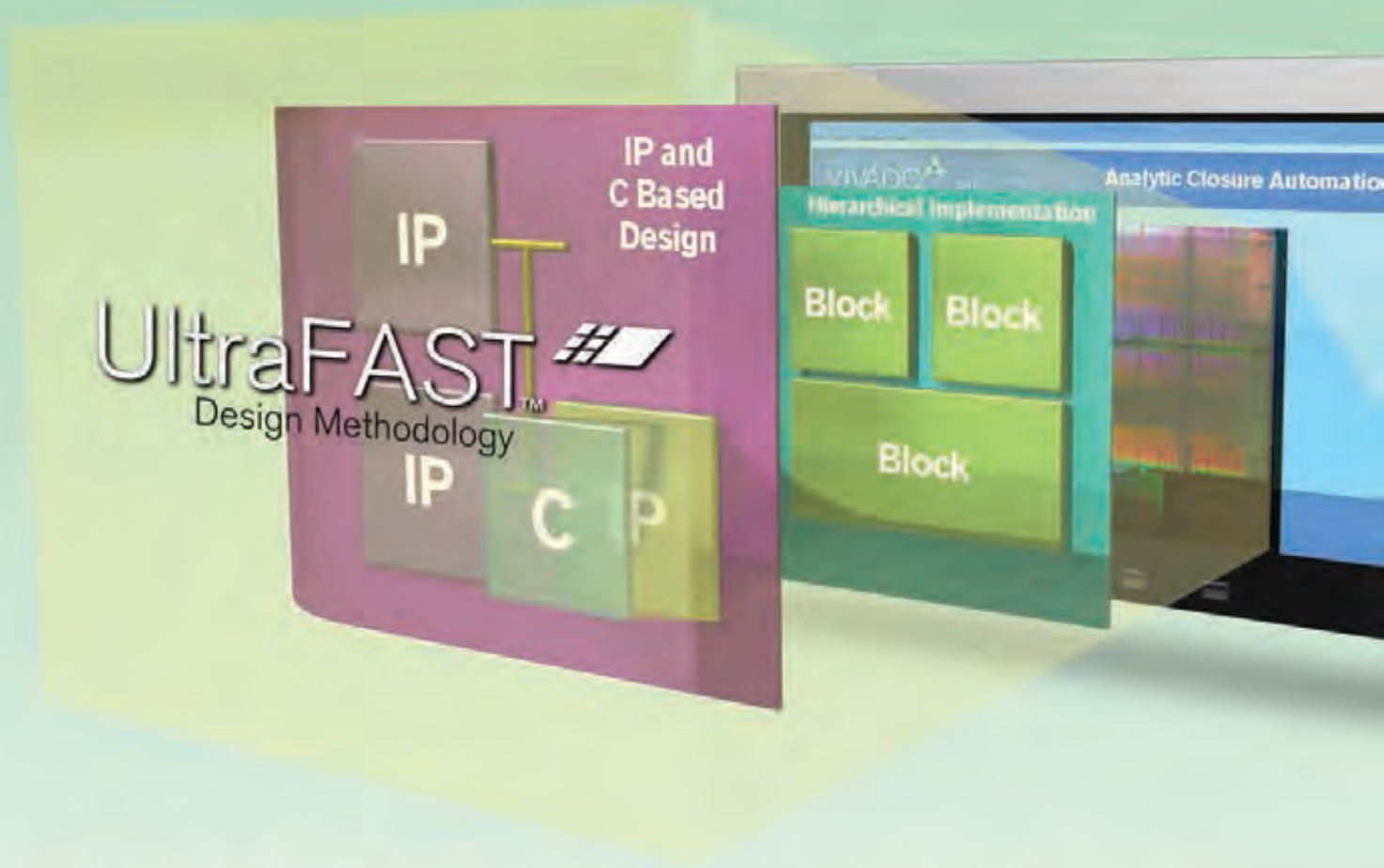
To find out more and test drive Synplify Premier® today, visit

www.synopsys.com/fpgaqualityofresults

SYNOPSYS®
Accelerating Innovation

Xilinx Introduces

The **UltraFast™ Design Methodology** for the Vivado® Design Suite



The UltraFast Design Methodology from Xilinx enables accelerated and predictable design cycles.

VIVADO

XILINX
ALL PROGRAMMABLE.

Learn More: www.xilinx.com/ultrafast