ISSUE 93, FOURTH QUARTER 2015

Xcell journal

SOLUTIONS FOR A PROGRAMMABLE WORLD

Xilinx Speeds Customer Medical Innovations to Market

FPGA-Based Fuzzy Controller Manages Sugarcane Extraction

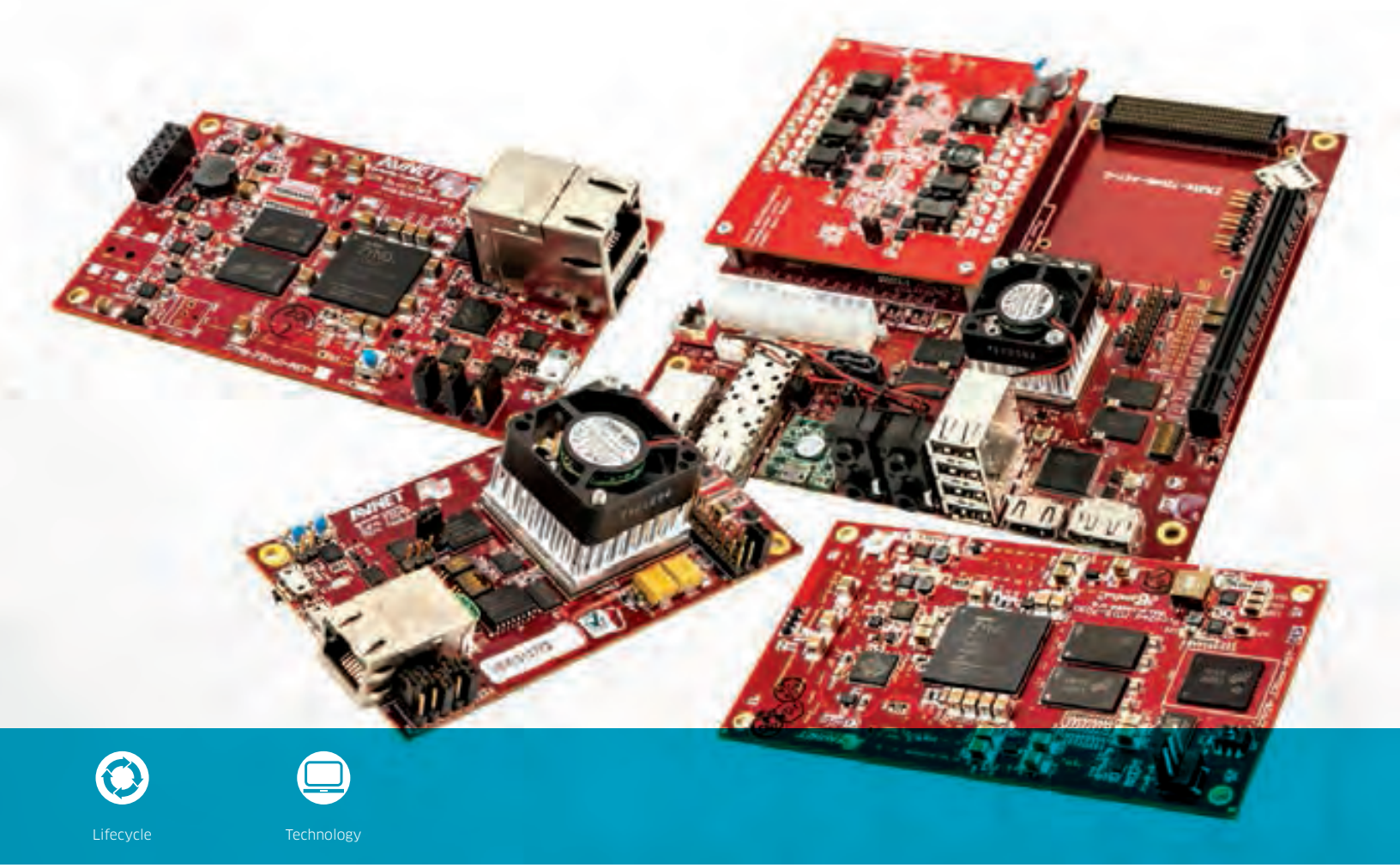
Zynq MPSoC Gets Xen Hypervisor Support

Making XDC Timing Constraints Work for You

The latest Xilinx tool updates and patches, as of October 2015

XILINX
ALL PROGRAMMABLE™

www.xilinx.com/xcell



Lifecycle

Technology

Design it or Buy it?

Shorten your development cycle with Avnet's SoC Modules

Quick time-to-market demands are forcing you to rethink how you design, build and deploy your products. Sometimes it's faster, less costly and lower risk to incorporate an off-the-shelf solution instead of designing from the beginning. Avnet's system-on module and motherboard solutions for the Xilinx Zynq®-7000 All Programmable SoC can reduce development times by more than four months, allowing you to focus your efforts on adding differentiating features and unique capabilities.

Find out which Zynq SOM is right for you http://zedboard.org/content/design-it-or-buy-it

DESIGNED BY AVNET

MMP

picoZed™

microZed™ FLEXIBILITY REDEFINED MICROZED.ORG

HAPS-80 FPGA-Based Prototyping Solution Delivers Up to 100 MHz System Performance

- New automated high-speed pin-multiplexing delivers the highest system performance
- Integrated ProtoCompiler software automates partitioning to reduce time to first prototype to less than two weeks
- Built-in debug enables the capture of thousands of RTL signals
- Scalable architecture supports up to 1.6 billion ASIC gates based on the Xilinx Virtex UltraScale FPGA

To learn more visit: **www.synopsys.com/haps-80**

SYNOPSYS®
Silicon to Software™

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 Judy Gelwicks
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/

Zynq UltraScale+ Says 'Hello World'

Congratulations to Xilinx and especially to the Zynq® UltraScale+™ MPSoC design team for shipping the first Zynq MPSoC XCZU9EG one quarter ahead of schedule. This accomplishment makes it a "three peat"—the third time in a row Xilinx has beat the competition to market by being the first programmable-logic company to ship devices on a leading process node. Xilinx was the first to 28 nanometers with the 7 series, first to 20nm with the UltraScale™ family and now first to 16/14nm FinFET with the UltraScale+.

The news broke on Sept. 30, when Xilinx announced it had sent the first sampling shipments of the new device to two customers. The design went from tapeout to shipment in two and half months, which is a true testament to the engineering skills of Xilinx®'s Zynq MPSoC design and quality teams as well as the good folks at foundry TSMC.

Xilinx received the first silicon in late September for characterization and test, and within hours was able to boot an upstream Linux kernel on the new device (see video demo).

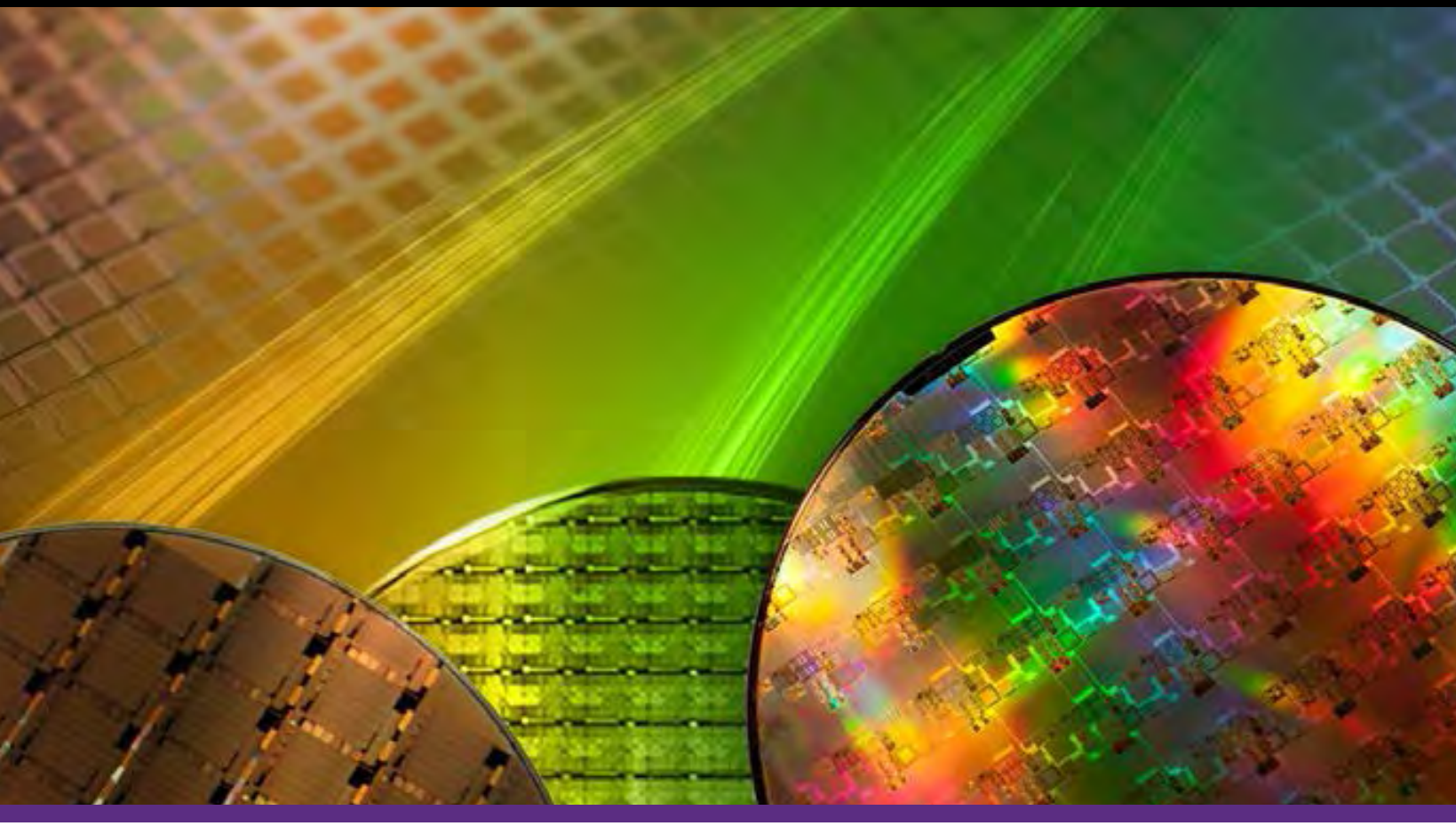
The video demonstrates an "upstream" Linux kernel booting on the new Zynq UltraScale+ MPSoC, the first member of the Xilinx UltraScale+ portfolio to ship.

We highlighted the architectural features as well as the performance-per-watt advantages of this exciting new device in the cover story of *Xcell Journal* issue 90. Implemented in TSMC's 16nm FinFET+ process technology, the Zynq MPSoC features on a single device a quad-core 64-bit ARM® Cortex™-A53 application processor, a 32-bit ARM Cortex-R5 real-time processer and an ARM Mali-400MP graphics processor, along with 16nm FPGA logic (with UltraRAM), a host of peripherals, security and reliability features, and an innovative power control technology. The new Zynq UltraScale+ MPSoC gives users what they need to create systems with a 5x performance/watt advantage over systems designed with the 28nm Zynq SoC.

In my tenure here as the publisher of Xcell Publications, it's been truly remarkable to read about the amazing innovations our readers have been able to create with our 28nm Zynq-7000 All Programmable SoC. So, I'll be eagerly waiting to see what you will do with the new Zynq MPSoC, with its many multiprocessing features. As the device becomes more broadly available over the next few quarters, I hope you will share your Zynq MPSoC design experiences with your peers by contributing articles to *Xcell Journal* and our new magazine for software developers, *Xcell Software Journal.* I look forward to reading about your remarkable designs.

Mike Santarini
Publisher

Accelerate Synthesis Runtime by 3X with Synplify Premier



Achieve fast runtimes with:

- Incremental and fast synthesis flows
- Distributed synthesis technology
- Fast compile with multiprocessing using single or multiple machines

To learn more about how Synopsys FPGA design tools accelerate synthesis runtimes, visit: **www.synopsys.com/fpga**

SYNOPSYS®
Silicon to Software™

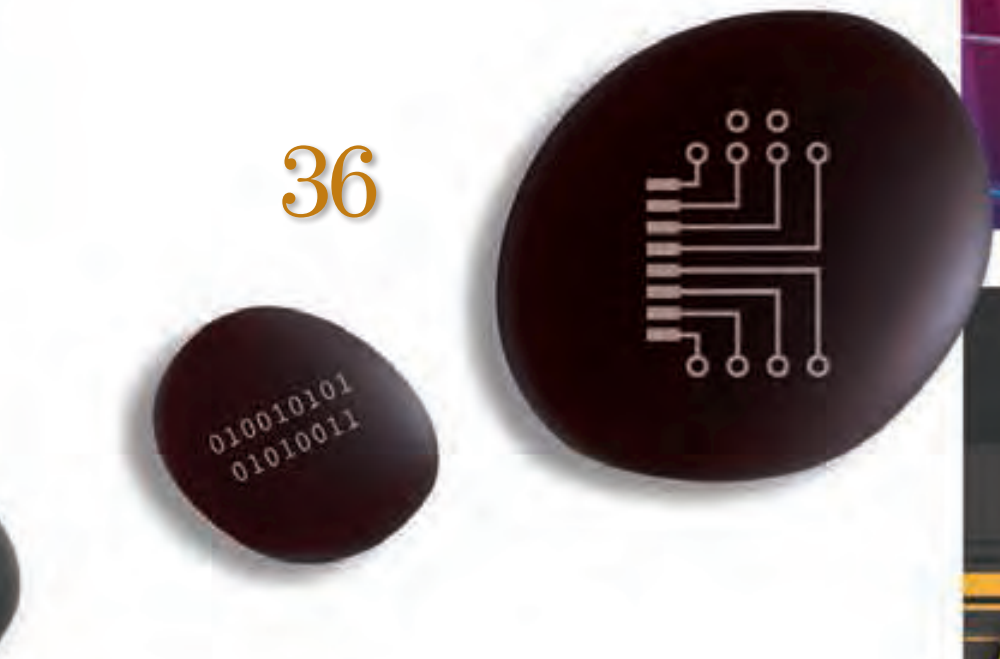
VIEWPOINTS

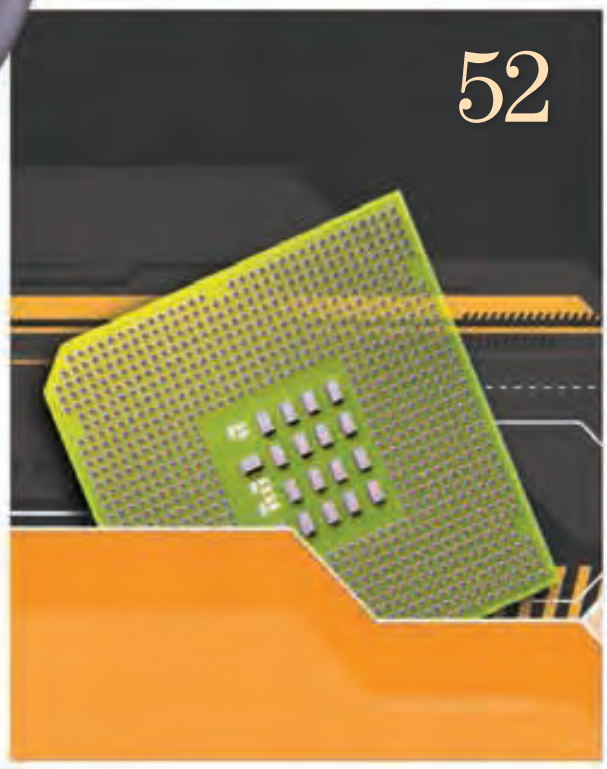
XCELLENCE BY DESIGN APPLICATION FEATURES

Cover Story

Xilinx Speeds
Customer Medical
Innovations to Market

THE XILINX XPERIENCE FEATURES





XTRA READING

Xilinx Speeds Customer Medical Innovations to Market

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

Electronics affects our lives in so many ways, but one area where it is literally having a life-or-death impact is in the healthcare industry. People are living longer thanks in large part to continual advances in healthcare fueled by rapid improvements in medical electronics. Customers have created these innovations with Xilinx® All Programmable devices over the last three decades.

Today, Xilinx FPGAs (and increasingly, systems-on-chips like the Zynq®-7000 All Programmable SoC) are at the heart of a growing number of medical systems. End products run the gamut from surgical robots, patient monitors, ventilators, medical imaging (CT, MRI and ultrasound) and X-ray machines to defibrillators, endoscopy machines, infusion pumps and analyzers. FPGAs are also central to the high-performance computing systems that enable researchers to perform genomic sequencing at lightning speed and those that enable scientists and pharmaceutical companies to more quickly develop and refine medicines to treat the symptoms of diseases.

The reason FPGAs have assumed such a key role in the medical market is simple. FPGAs—and more recently, Zynq SoCs—enable medical equipment developers to lower the risk of failures and speed their equipment through the regulatory process. With All Programmable devices, designers can take the functions of their systems that must be highly reliable and implement them in the logic of the Xilinx devices while running other, less-critical functions in software (Figure 1).

"Medical technologies are evolving rapidly," said Kamran Khan, senior product marketing engineer for Xilinx's medical group. "They are getting much smarter, more connected, more integrated, more compact and less invasive for faster patient recovery times. And, more important, they are better at what they do." Demand for medical devices with advanced features is growing as the population worldwide increases and as people live longer, thanks in part to the rapidly evolving healthcare advances of the last half century.

LIVING LONGER, HEALTHIER LIVES

The biggest growth driver of the healthcare industry and thus the medical equipment market is the expected increase in the world's population. Today, the global population stands at roughly 7.3 billion; by 2050, it is expected to grow to 9.7 billion. Where-

as today, people 65 and older represent roughly 23 percent of the world's population, by 2050 that number is expected to be 32 percent, or 3.1 billion people. Those senior citizens are expected to be the segment of the population most likely in need of regular medical care.

"People are living longer than before, because our quality of life and understanding of medicine are better," said Khan. "Because people are living longer, we need better healthcare to support them in their older age, and we need to be able to supply this with less-invasive techniques that have fewer side effects."

Population growth along with the aging of the populace represents a great challenge for the world's medical community as well as governments, which are increasingly turning to state-regulated healthcare. This population growth also serves as a grand opportunity for innovation in medical electronics and other medical fields. As such, the medical market is expected to grow to $212 billion by 2019, with the semiconductor spend expected to reach $6 billion, said Khan.

"The medical electronics industry used to be massively consolidated, and so companies like Siemens and General Electric could develop new generations of their systems at their own pace," said Khan. "Nowadays, X-ray and ultrasound have become a commodity. Today, people realize that it is not very difficult to get into these mature, low-risk applications, even with the regulatory burden. Now we are seeing a boom in medical equipment startups, and it's a global boom, with a large number of new companies from China and South America entering the market. Some countries, like China and Brazil, favor domestic products, so there are new companies emerging to serve these new markets."

Khan said this new competition is of course increasing both time-to-market pressure and pricing pressure for all players. At the same time, it is also driving greater value and innovation into the medical world faster, for the benefit of us all.

"The market is demanding more integration of features and more portability," said Khan. "Medical facilities used to have one machine that specialized in one particular task. Each of those machines was quite large, and if many different machines were needed, they occupied quite a bit of space in a hospital room."

What's more, Khan said, the systems wouldn't necessarily communicate or be compatible with one another, which could cause other complications. So today there is a demand to have a single piece of equipment perform multiple tasks. Similarly, users prefer a smaller form factor so that the equipment takes up less space and is easier to move from room to room. It's even better if the equipment is battery powered so that it can be used in areas with no electricity or in ambulances. Simultaneously, there is a growing need for each system to be able to interact with other pieces of equipment in real time to, for example, change dosages of a medicine in response to a patient's changing vital signs.

"Patient monitors, for example, used to be quite simplistic, but not anymore," said Khan. "They used to take in a few channels of analog biotelem-

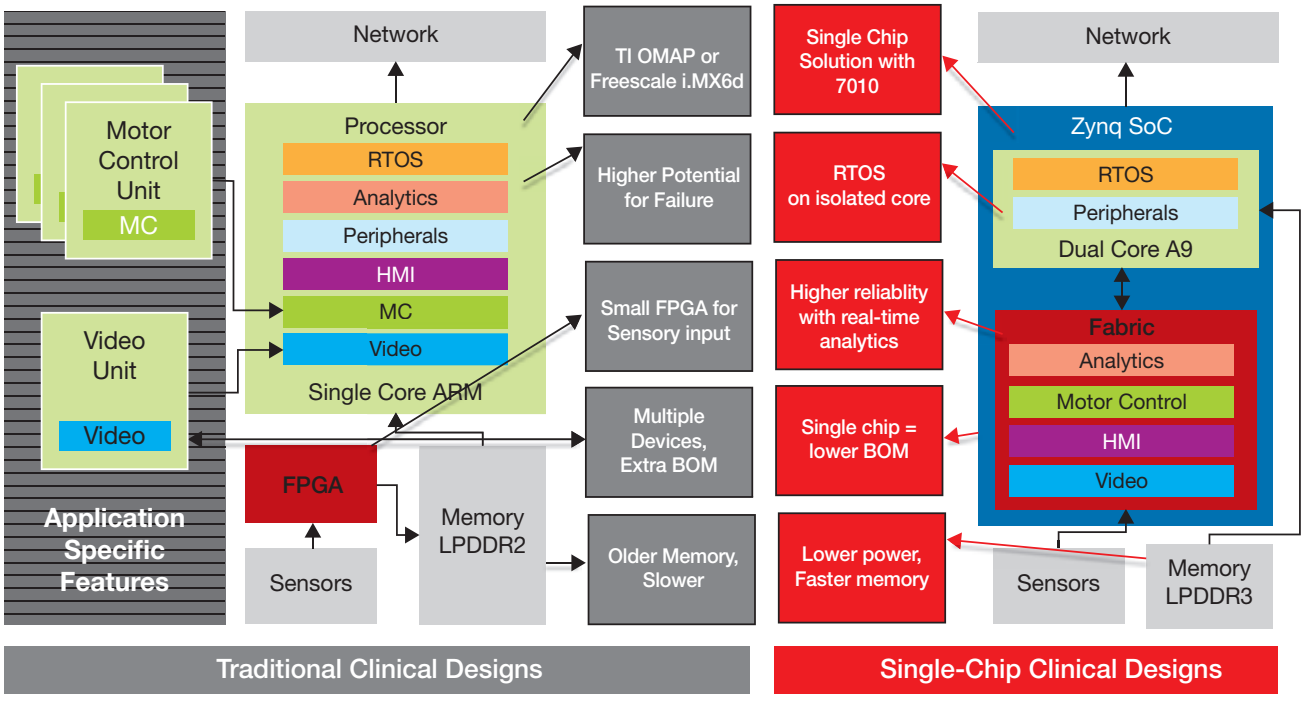


Figure 1 – Zynq SoC platforms enable medical equipment companies to quickly create innovative, optimized systems and bring them to market.

'Nowadays, medical equipment must be more integrated and smarter. Systems must be able to communicate and work in concert.'

etry, apply some very simple processing to it and display the information on a monitor. It was quite simple, so a lot of this was done with embedded processors from companies like TI or Freescale. But nowadays, medical equipment must be more integrated and smarter. Systems must be able to communicate and work in concert."

For example, he said, a patient monitor must be able to talk to the ventilator and infusion pump. If a patient's vital signs begin to spike, both systems must respond appropriately, with the ventilation pump adjusting the oxygen mix and the infusion pump adjusting the medicine dosage. "They also have to communicate to the hospital's main network to keep staff apprised of emergencies," Khan said. Moreover, the information must be kept as part of the patient's long-term medical record.

Khan said that the healthcare industry is also embracing the Internet of Things, cloud computing and the modern network infrastructure. All of these developments are making it possible to monitor the health of outpatients and record health patterns remotely, as well as being able to react in real time to emergency situations. There is a burgeoning industry to monitor patient health in real time much in the same way as home security companies monitor property.

THE EVER-GROWING REGULATORY CHALLENGE

In the midst of this industry growth and drive toward smarter, more connected and more integrated medical systems, vendors are faced with ensuring that their equipment conforms to increasingly more stringent safety and reliability regulations from a growing number of regulatory bodies worldwide.

A changing regulatory environment is often cited as the single biggest challenge medical equipment companies face today. Companies can't legally sell their products until their systems clear relatively strict regulatory guidelines and testing. Medical equipment failure can lead to medical liability lawsuits.

The amount of regulation depends largely on the type of equipment a company wants to bring to market and scales accordingly. Less-critical equipment that doesn't touch the patient's body typically requires a regulatory cycle that's roughly half a year, while equipment performing more-critical functions that do touch a patient's body requires roughly two years' worth of approval cycles before release to the market.

"Even in established markets like patient monitoring or ultrasound, they still require a regulatory cycle of a year to a year and half, typically because there is so much documentation and testing required," said Khan. "They have to create a technical file that supports their product, submit the documentation to the regulator which, after a long review cycle, passes or fails the product. The goal is to pass on the first try, because if they don't pass, the second review is even more detailed and can involve an even longer cycle. It's like being audited by the IRS."

Khan said the real goal is to understand and quantify the risk a medical system poses. "The perception is that medical devices can't fail, but regulatory bodies know that everything fails," said Khan. "What they want to do is have OEMs understand and lower the failure risk, at the same time knowing all possible cases in which a device could fail and what will happen if it does."

Khan said that while the various regulatory bodies worldwide scrutinize both hardware and software, they tend to look hardest at software, since failures in software are believed to be more likely to cause the system to fail into an unknown state. "Most people have encountered software bugs on their PCs or mobile devices and not so commonly on the actual hardware, so software typically comes under greater regulatory scrutiny in general in medical devices," Khan said.

"A lot of medical devices coming to market typically require embedded processors," he went on. "They may not require a lot of compute horsepower but they have some amount of embedded processing and thus software. The challenge is, how do I show to regulatory bodies and to customers that the device is safe and won't hurt patients? For example, if it's an infusion pump that's delivering my daily medicine, how do I know that it is delivering the right amount of medicine on time and that it won't stop in the middle of the night?"

And as devices become more complex to perform multiple tasks, the code also becomes more complex and much larger. "'Reliable' software is still something that isn't well understood," said Khan. "Software is very complex; it's hard to debug and understand the risk of it. This high potential for failure results in high risk, even for mundane medical systems that aren't life-critical."

This is one of the main reasons why FPGAs and more recently, Zynq SoCs have become so popular with medical equipment manufacturers. With these All Programmable devices, companies can lower the risk of failures and speed up the regulatory process. They can essentially take functions of their

system that must be reliable and implement them in the logic of the Xilinx devices while placing other, less-critical functions in software.

XILINX DEVICES SPEED MEDICAL INNOVATION

Khan said that with its decades of experience serving the medical electronics sector, Xilinx has developed a comprehensive medical toolbox consisting first and foremost of Xilinx's All Programmable FPGAs and SoCs. The toolbox also contains certified design tools and methodologies to ensure quality, reliability and redundancy, along with tried-and-true silicon IP and software stacks from Xilinx and members of the Xilinx Alliance Program. The company's new SDSoC™ development environment will enable medical customers to even more quickly create optimized systems with critical functions implemented in the Zynq SoC's logic while less-critical functions run on the Zynq SoC's ARM® processing system (Figure 2).

For the last 10 of the 30 years that Xilinx has been serving the medical market, FPGAs have been rapidly displacing ASICs and ASSPs in medical gear. Medical equipment is sold in relatively low volumes worldwide, and so the cost, coupled with the strict and time-consuming testing-and-regulatory process, makes ASICs and ASSPs prohibitive. As a result, a vast majority of medical equipment today employs Xilinx devices in some capacity.

Starting in the late 1980s and early 1990s, said Khan, customers began using Xilinx's smaller FPGAs as sensor interfaces in medical equipment. But over time, companies started adding more-critical functions to the devices as FPGAs began displacing ASICs and ASSPs. In the newest equipment, Xilinx devices are playing a pivotal role at the heart of these systems, especially with the Zynq SoC and the recently shipped Zynq UltraScale+™ MPSoC, which has additional safety and security features beyond those offered in the Zynq SoC.

"With our Zynq SoC portfolio, we are able to reduce risk and speed medical innovations to market," said Khan. "We are able to take in the design's software elements with our new SDSoC tools, implement them in programmable-logic fabric instead of in software and then add in layers of redundancy to provide more layers of reliability in these systems."

For example, said Khan, if a company is designing an infusion pump, part of the system will be controlling the motors so as to deliver the medicine in exact quantities at the exact time specified, with the metrics staying exactly where a doctor set them. Meanwhile, another part of the infusion pump is biotelemetry—monitoring the patient and ensuring he or she is OK.

"Using our isolation design flow, customers can partition their system into critical and noncritical functions, implement critical functions in logic and create physical barriers between critical functions in the system," said Khan. "They can build in extra safety measures so that if a failure condition may occur, it shuts

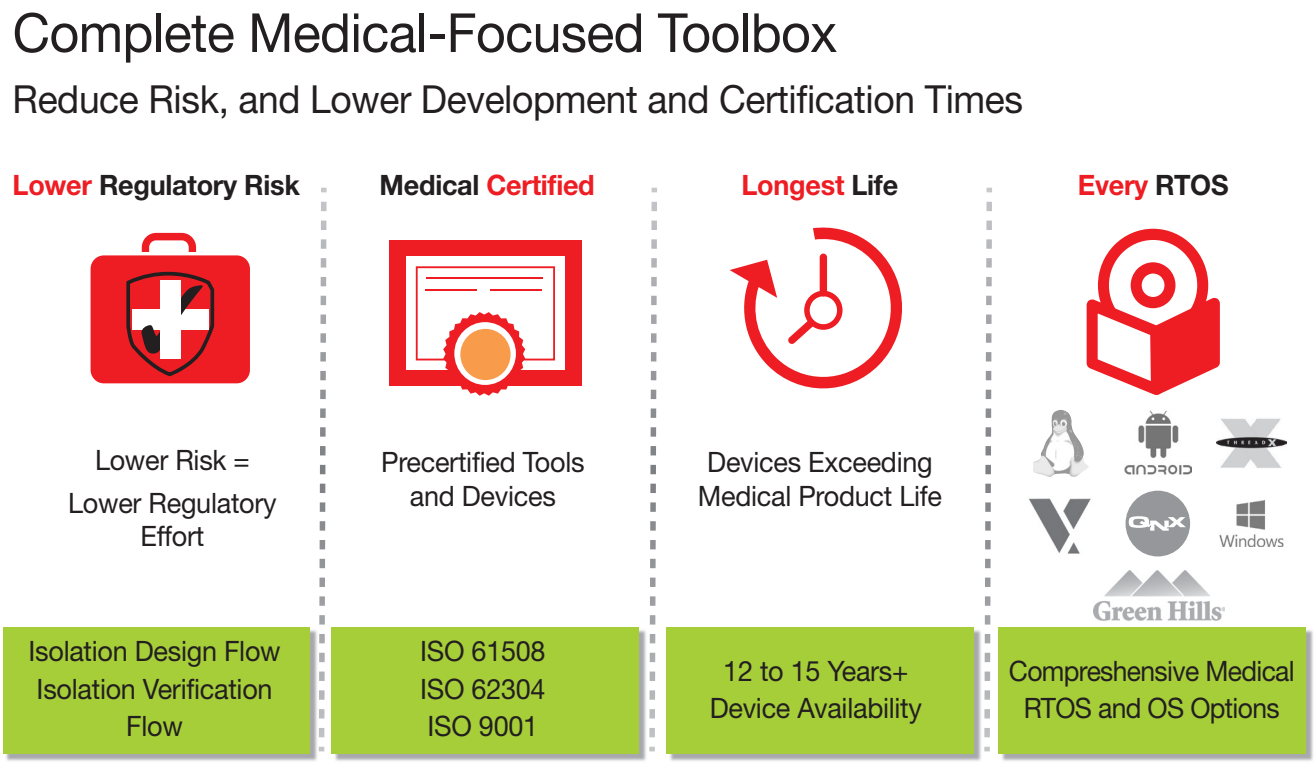


Figure 2 – Xilinx has proven design tools that lower risk and help speed designs through the regulatory process.

Figure 3 – TOPIC Embedded Products' Dyplo IP helps speed medical product design and development.

down in a safe and predictable manner. Furthermore, they can show regulatory bodies that they are building their design in trusted Xilinx fabric. And then, using reports from our IDT tool, they are also able to show regulators the signal paths, predictable outcomes and fail-safes."

With Xilinx's new SDx development environments (SDAccel™ for C, C++ and OpenCL™ design entry into FPGAs; and SDSoC for C/C++ design entry in Zynq SoCs), medical equipment companies can now develop the prototype of their systems in C; determine what functions, both critical and noncritical, would best be suited to running in hardware and software; and then use the isolation design flow to implement the hardware functions in greater detail and add layers of redundancy for further reliability. "Using a system-level methodology can cut months out of a design cycle at the back end," said Khan.

Khan noted that with Xilinx's rich history serving the military and space community, Xilinx's commercial devices used in medical applications already exceed any requirements for radiation tolerance. Xilinx also diligently ensures that its devices, its Vivado® Design Suite and IP (its own and cores from alliance members serving the medical device market) adhere to strict quality and safety standards. Among them are the ISO 60601 3rd edition and ISO 13485 medical device design standards, and international standards including ICE 61508 (functional safety) and ICE 62304 (RTOS integration). The result is to speed customer end-product designs through the regulatory processes.

"Customers have to get certification on their end products, rather than the individual devices, but what we do is ensure our devices, tools and IP conform to these standards," said Khan. "For example, ICE 62304 is a standard for RTOS integration. Alliance member QNX's RTOS has already been precertified for 62304, so using it on the Zynq SoC can cut six months off the certification process. Likewise, alliance member TOPIC Embedded Products offers remarkable IP to further speed prototyping and design. Their IP, design flow and SOM board are precertified to be compliant with the ISO 13485 quality-management standard. This enables customers to cut even more time off the regulatory process" (see video, Figure 3).

A PLATFORM PLAY

With the increased regulatory burden and mounting time-to-market pressures, many medical companies today are employing platform design business strategies built around the Zynq SoC.

"The market is quickly coming to the realization that it can't build each product from scratch and that it needs to take a scalable approach to its product lines," said Khan. "Platform-based design and cost-sizing are huge, for example, in the medical imaging space, where companies offer a portable version, low end, midrange and high end in a product line. By designing one platform based on the Zynq SoC for the high end, they can use the same hardware at each level and scale the functions to suit the needs of each end market by reducing features."

A platform approach built around the Zynq SoC has many advantages over a platform composed of multiple discrete parts, said Khan. "Medical devices are typically on the market from 10 to 15 years, much longer than consumer products, which typically have a two- to three-year lifespan," he said. "Medical devices are typically in design for three years, can go through regulatory approval for another one to three years and then have to be in the market for 10 more years. But most embedded processors today have a lifespan of about five years and then they are end-of-lifed for a newer version of the device. That's because most are designed mainly for the consumer market. But in the medical device industry, if a chip needs to be changed for a newer one because the older version isn't available any longer, the product needs to go through the regulatory process again."

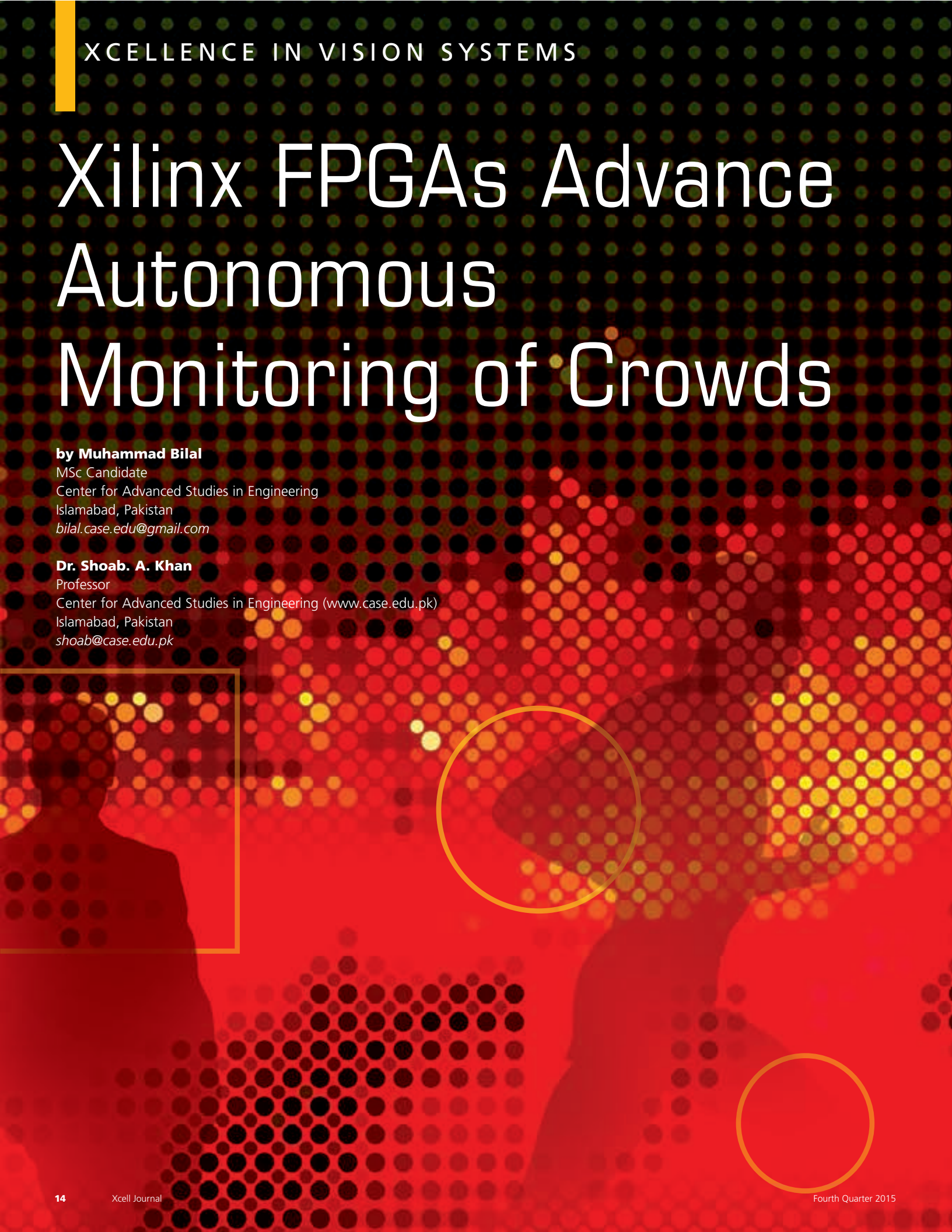
Khan said the Zynq SoC and MPSoC families give customer designs the performance advantages of an embedded processor or multiple processors plus the flexibility, product differentiation and safety of programmability. On top of all that is I/O flexibility to accommodate a vast range protocols, sensors and video configurations. "Integrating multiple system functions in the Zynq SoC and MPSoC families saves space, lowers BOM cost and lowers power drastically compared with multichip platforms and speeds medical innovations to market," said Khan.

For more information on Xilinx in medical applications, visit *http://www.xilinx.com/applications/medical.html*.

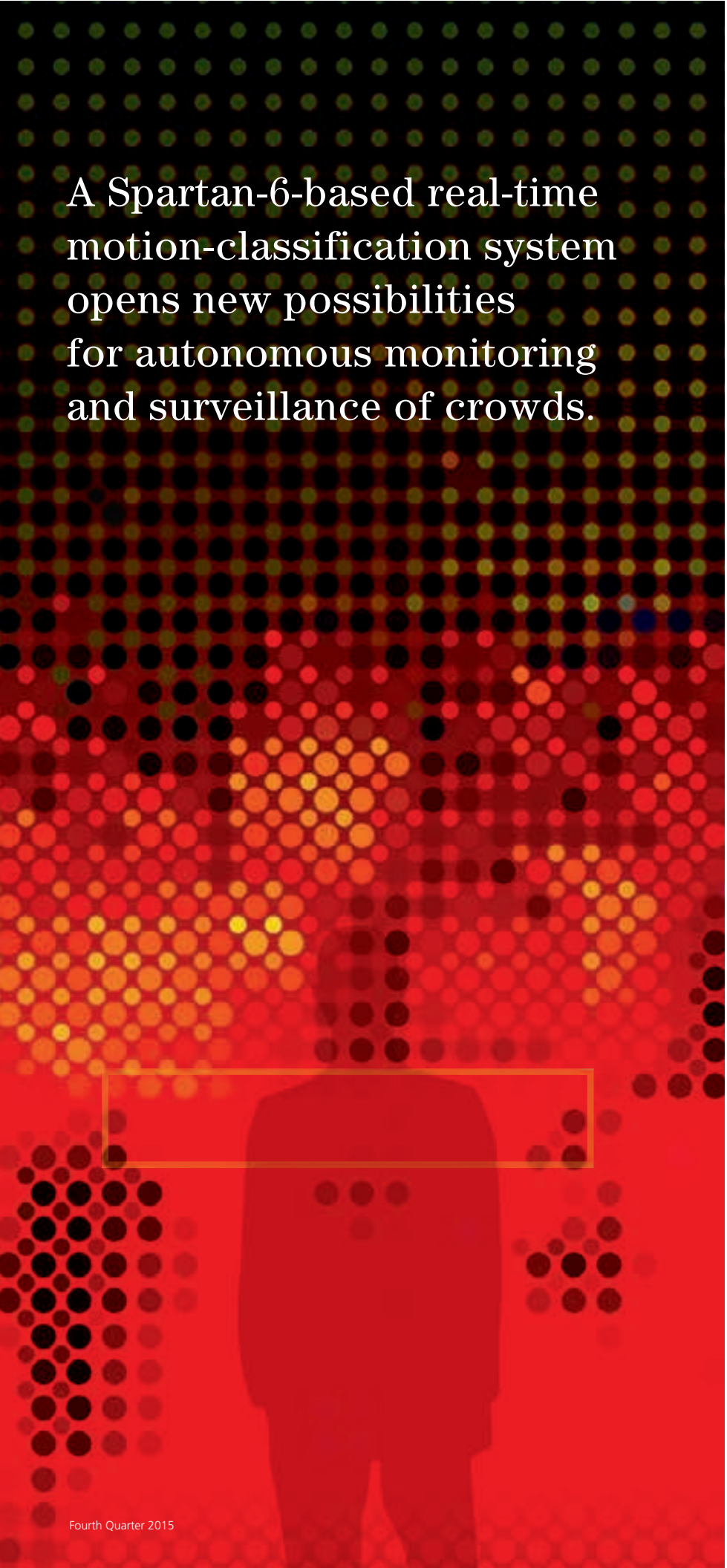
Xilinx FPGAs Advance Autonomous Monitoring of Crowds

by Muhammad Bilal
MSc Candidate
Center for Advanced Studies in Engineering
Islamabad, Pakistan
bilal.case.edu@gmail.com

Dr. Shoab. A. Khan
Professor
Center for Advanced Studies in Engineering (www.case.edu.pk)
Islamabad, Pakistan
shoab@case.edu.pk



A Spartan-6-based real-time motion-classification system opens new possibilities for autonomous monitoring and surveillance of crowds.



Crowd surveillance and monitoring have become an area of great importance in current times. Governments and security departments have started looking for more advanced ways to intelligently monitor human crowds across public places to detect any unusual activity before it is too late to react. However, there are still some barriers to cross before achieving this goal effectively. For example, if it is desired to monitor all possible crowd activities across a whole city at once, 24 hours a day, then it is certainly not possible through all-manual monitoring only, especially when there are thousands of CCTV cameras installed.

The solution to this problem lies in developing new, intelligent cameras or vision systems that could autonomously monitor the activities of human crowds through advanced video analytics techniques, and therefore could immediately report any unusual event to a central control station.

The design of such an intelligent camera/vision system would require not only the typical imaging sensors and optics, but also a high-performance video processor to perform video analytics. The reason for having such a powerful video processor onboard is the high processing requirements of sophisticated video analytics techniques, most of which use computationally intensive video-processing algorithms.

FPGAs are ideally suited for such performance-hungry applications. And thanks to the UltraFast™ design methodology enabled by high-level synthesis (HLS) in Xilinx®'s Vivado® Design Suite, it's now possible to create an optimal and high-performance design for FPGAs with great ease. Further, the fusion of an embedded processor such as the Xilinx MicroBlaze™ inside the FPGA's reconfigurable logic means that applications with a complex control flow can now be easily ported to FPGAs.

Keeping this theme in view, we designed a prototype for a human crowd motion-classification and monitoring

Using the sum of weighted absolute differences (SWAD), we computed more than 900 motion vectors across the image.

system using Vivado HLS, Xilinx's Embedded Development Kit (EDK) and software-based EDA tools from the ISE® Design Suite. Our design methodology was based on what we think of as a software-controlled, hardware-accelerated architecture. We targeted our design for the low-cost Xilinx Spartan®-6 LX45 FPGA. The overall system design, which we completed in a short span of time, has demonstrated promising results in terms of real-time performance, low cost and a great flexibility of design.

SYSTEM DESIGN

We accomplished the overall system design in two phases. In the first phase, we developed a human crowd motion-classification algorithm. After the verification of this algorithm, our next step was to implement it in an FPGA. In this second phase of development, our main focus was on the architectural design aspects of an FPGA-based real-time video-processing application. Tasks included developing a real-time video pipeline, developing hardware accelerators and, finally, integrating them and implementing algorithmic control and data flow to complete the system design.

Let's take a walk through each of these development stages, starting with a brief description of the algorithm design followed by a detailed look at its subsequent implementation on the FPGA platform.

ALGORITHM DESIGN

Various algorithms have been proposed in the literature regarding crowd surveillance and monitoring. Most of these algorithms start with detecting (or placing) interest points in the human crowd scenes and tracking them over time to collect motion statistics. These motion statistics are then projected to some precomputed motion models to predict any unusual activity [1]. Further modifications include the clustering of interest points and tracking of these clusters instead of individual interest points [2].

Our algorithm for crowd-motion classification is based on the same concept, except we preferred using a template-matching scheme for performing motion estimation instead of the former approaches like the Kanade-Lucas-Tomasi (KLT) feature tracker. This template-matching scheme proved much better for motion estimation in cases of low or varying contrast at the cost of a few more computations.

In order to perform motion estimation using this scheme, we divided our video frame into a grid of smaller rectangular patches, and performed motion computation on the current and previous images in each patch using a method based on the sum of weighted absolute differences (SWAD). Each patch in turn contributed one motion vector that depicts the extent and direction of motion in between two frames at that particular position. The result is computation of more than 900 motion vectors across the whole image. The steps involved in computation of these motion vectors are shown in Figure 1.

We also utilized a weighted Gaussian kernel to achieve robustness against occlusion and zero contrast areas in the image. Furthermore, the processing of one patch for computation of a motion vector is independent of the other patch's processing, making this approach ideal for parallel implementation on FPGAs.

After computing motion vectors across the entire image, the algorithm computes their statistical properties. These properties include average motion vector length, number of motion vectors, dominant direction of motion and similar metrics.

We also compute a 360-degree histogram of the direction of motion vectors and further analyze its properties, such as standard deviation, mean and coefficient of variation. These statistical properties are then projected to a precomputed motion model to classify the current motion in one of several categories. We account for these statistical properties across multiple frames to ascertain the classification results.

The precomputed motion model is built in the form of a weighted decision tree classifier that takes into account these statistical properties to classify the motion under observation. For example, if the motion is observed to be high and there is a sudden change in momentum in the scene while the direction of motion is random or out of the image plane, then it will be classified as a possible panic condition. The algorithm development was done using Microsoft Visual C++ with the OpenCV library. A complete demonstration of this algorithm can be found at the Web links provided at the end of this article.

FPGA IMPLEMENTATION

The next phase in our system design was the FPGA implementation of our algorithm. Such an implementation

comes with its own design challenges, such as the fact that video input/output and frame buffering are now a part of FPGA-based design. Also, limited resources and available performance may require necessary design optimizations.

Keeping these design aspects and other architectural considerations in view, we divided our overall FPGA-based implementation in three parts. In the first part, we developed a generic real-time video pipeline into the FPGA to take care of necessary video input/output and frame buffering. Then, in the next part, we developed algorithm-specific hardware accelerators. Finally, in the third phase of design, we integrated them together and implemented algorithmic control and data flow. This completed our overall FPGA-based system design.

It's worth taking a closer look at each stage of the process.

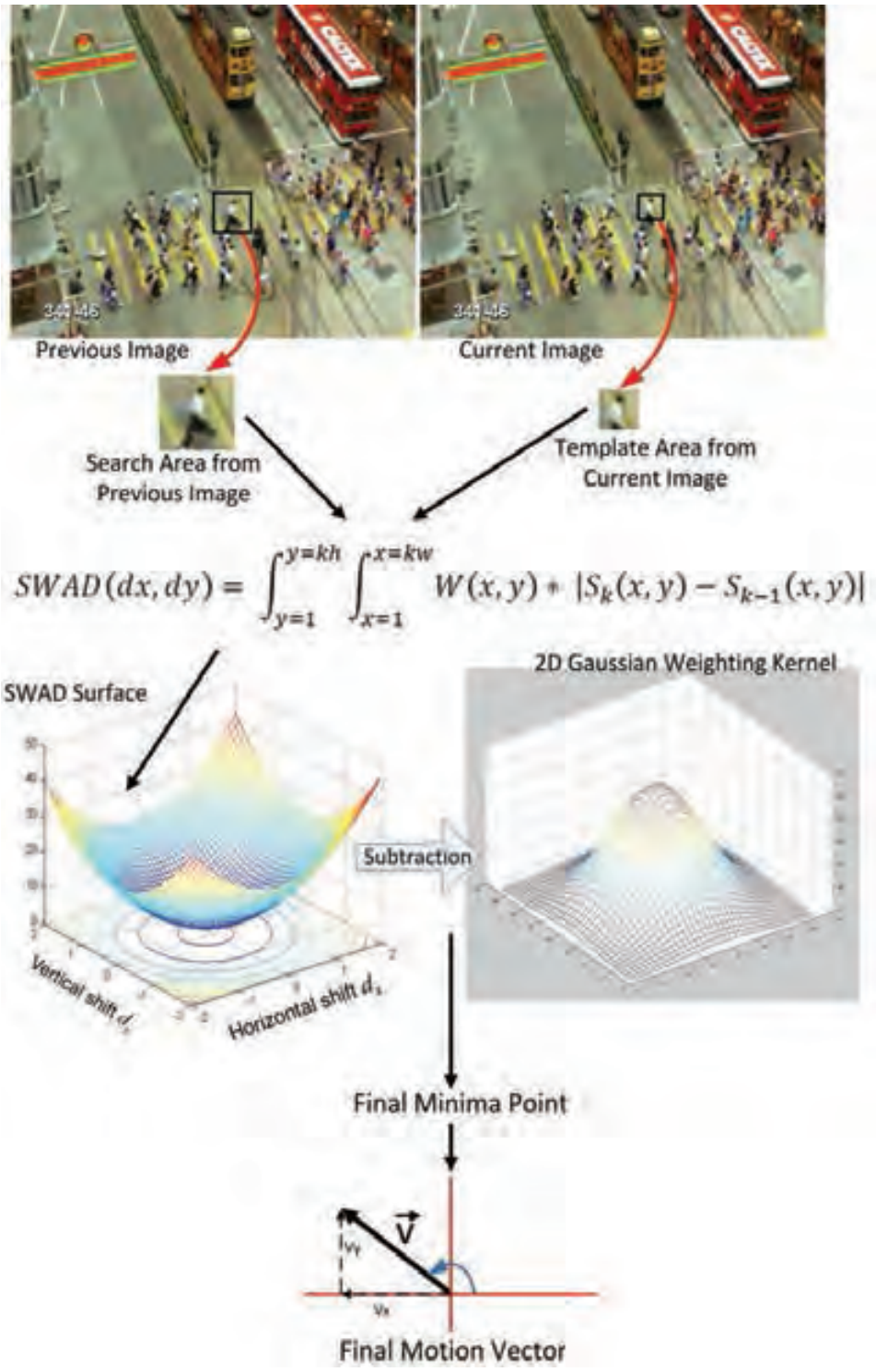


Figure 1 – Steps in computing motion vectors, starting with image capture (top)

REAL-TIME VIDEO PIPELINE

A real-time video pipeline is the most important building block in developing any video-processing application for FPGA platforms. Such a pipeline hides the complex memory management related to video input/output and frame buffering from the user, and provides a simple interface for accessing video frame data for processing.

Although there are several advanced and commercially licensed video pipelines [3] available in this regard, we opted to build a customized video pipeline for this purpose. We built this pipeline around the Xilinx EDK, with custom video capture/display ports for handling video input/output data. The pipeline is easily configurable for other Xilinx FPGA families as well.

The video capture port decodes the incoming video stream data from the video ADC and buffers it locally. It is then forwarded to the main memory for constructing a video frame. Similarly, the video display port encodes the video frame data present in a local buffer and forwards it to the video DAC for display purposes. The video input and output ports are connected to the main peripheral bus of a MicroBlaze host processor, which handles this video data traffic to and from the main memory.

The video ports are capable of generating interrupts to inform the MicroBlaze processor that new data is available at the video input port or required by the video output port. The video ports carry out a "ping-pong" buffer-management scheme such that if the MicroBlaze is unable to immediately service a video

port, then buffer overflow or underrun doesn't happen. Figure 2 shows the interconnection between the video ports and the MicroBlaze processor.

The video ports are designed to detect and produce a video line number, field ID (in case of interlaced video) and other control information in the video input/output stream. This information is passed to the MicroBlaze processor through the video ports' interrupt service routines (ISRs) when enough video data is buffered by the video input port or required by the video display port. These service routines in turn perform the video data transfer between video port local memories and main memory via DMA.

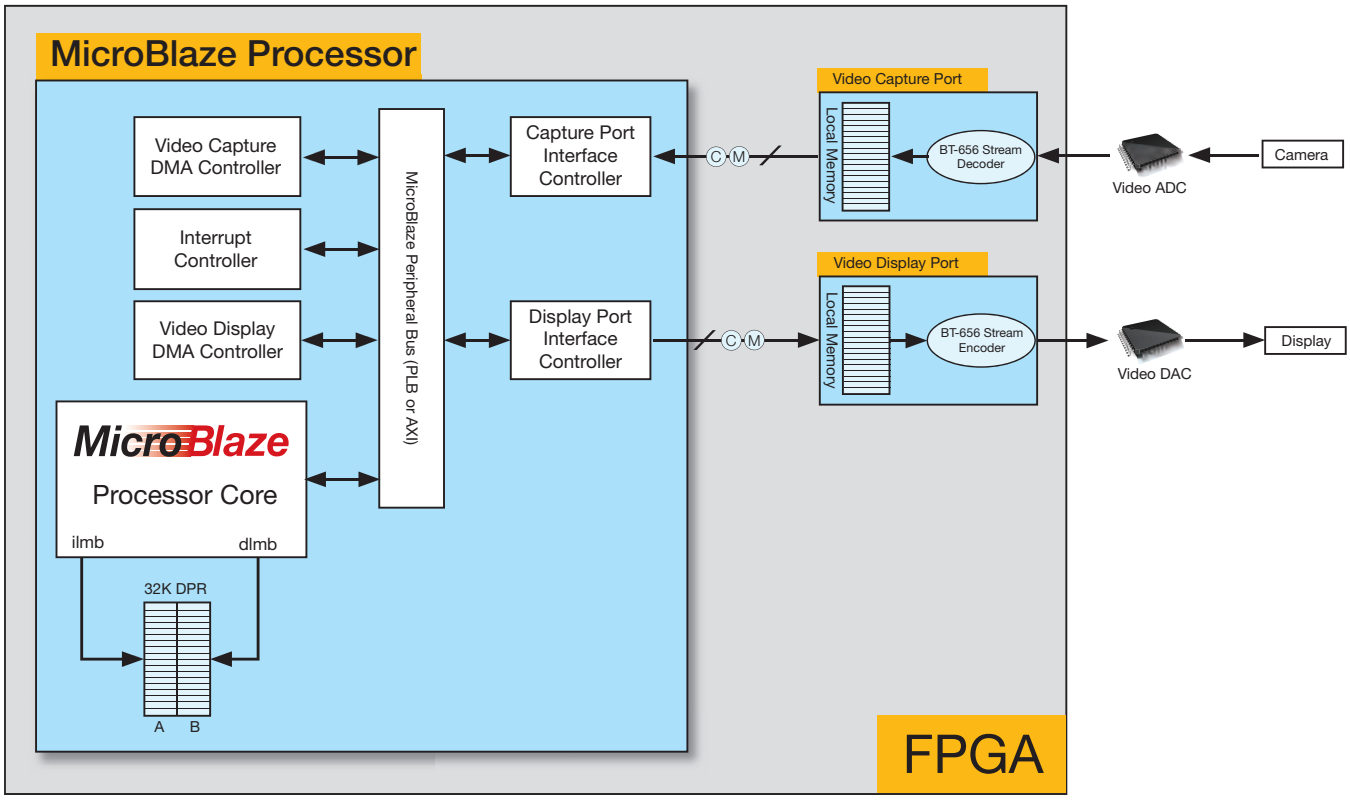
In addition to video port ISRs, a set of higher-level video frame queue-management functions that we named the Video Frame Queue API operates in between these ISRs and a user-level application. This API maintains a queue of multiple capture and display frames to support double or triple frame-buffering schemes. A user application running in MicroBlaze can easily acquire a video capture frame or can submit a video display frame using Video Frame Queue API functions. Figure 3 shows these functions in their respective levels of hierarchy.

There are several benefits of using a MicroBlaze as a host processor to interconnect various building blocks in the system. For example, we can easily interface with a wide range of external memories (SRAM, SDRAM, etc.) with MicroBlaze for loading or storing the video frame data from video ports. Similarly, we can use DMA controllers in the EDK for transporting video data between video ports and main memory. We also interfaced custom hardware accelerators in the same fashion with the MicroBlaze processor.

These Video Frame Queue API functions together with video port ISRs and video input/output ports complete the construction of the video-processing pipeline in our design. Figure 4 shows an actual video frame captured, processed and then displayed using this video pipeline on our FPGA. It also shows a picture-in-picture feature with a zoomed-out view of computed motion vectors.

VIVADO HLS-BASED HARDWARE ACCELERATOR

In the crowd motion-classification algorithm we have described, the most time-consuming and computationally intensive task was to compute motion vectors. The other system task—that is, doing classification—did not involve pixel-level processing and was pretty simple and easy to accomplish. Keeping this aspect of the design in mind, we built a hardware accelerator for computing motion vectors. We designed, tested and later synthesized the accelerator in RTL using Xilinx Vivado HLS in the C/C++ language.



Note: (C) (M) Donate 'Control' and 'Memory' interfaces respectively.

Figure 2 – Video ports and their interconnection

One of the key features of Vivado-generated RTL code is that it is optimal to a great extent. Vivado HLS synthesizes array accesses (such as pixel data stored in an array) into memory interfaces and auto-generates required addresses by analyzing the code. It also analyzes precomputable offsets and constants to perform so-called "strided" memory accesses very fast. The strided memory accesses originate from accessing data from multiple rows of an image (such as in 2D convolution).

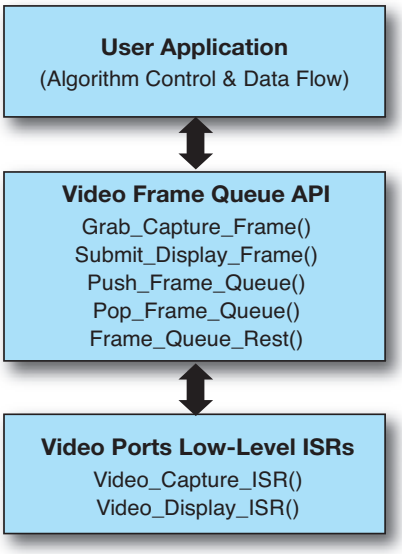


Figure 3 – Video port ISRs and Video Frame Queue API functions

The key considerations in designing a Vivado-based accelerator were parallelizing the computation of motion vectors and maximizing the data read from the main memory. For this purpose we used eight Block RAMs to load and store video frame data in parallel. The hardware accelerator core is capable of computing four motion vectors in parallel and for this purpose it utilizes all eight of these Block RAMs. The data pump to these Block RAMs from the main memory is controlled by the MicroBlaze through DMA.

The hardware accelerator generated by Vivado HLS comes with some auto-generated handshake signals that are necessary to start and stop the hardware accelerator. These handshake signals include "start," "busy," "idle" and "done" flags. These flags are routed to the MicroBlaze processor through GPIOs to perform handshaking. Figure 5 shows the interconnection between this hardware accelerator, the eight Block RAMs and the MicroBlaze processor's main peripheral bus.

The Block RAMs—named SA1, TA1 to SA4, TA4 in Figure 5—are each 16 kbytes in size. Each pair of SA1, TA1 to SA4, TA4 holds enough data for the computation of one complete row of motion vectors. Thus, upon completion of its run, the hardware accelerator outputs four rows of motion vectors written back in the same Block RAM memories. These computed motion vectors are then read back by the MicroBlaze processor, which copies the result in its main memory as a grid of motion vectors. (Figure 4 shows an actual frame overlaid with a grid of motion vectors computed through this hardware accelerator.)

The hardware accelerator operates at 200 MHz and all the processing needed to compute motion vectors across the whole image completes in less than 10 milliseconds, including all data transfers to and from memory.

Figure 4 – An actual FPGA-processed frame with motion vector grid overlaid at bottom right

ALGORITHM CONTROL AND DATA FLOW

With the video pipeline and hardware accelerator in place, the last step in completing the system was to integrate these two elements with the MicroBlaze host processor and to implement algorithm control and data flow at a user-level application in C/C++ using Xilinx's Software Development Kit (SDK). Implementing algorithm control and data flow in the Xilinx SDK brings a great deal of flexibility in design. That's because you can design and integrate new hardware accelerators in the same fashion, and modify necessary control and data flow to incorporate new hardware accelerators. The result is a kind of software-controlled, hardware-accelerated design

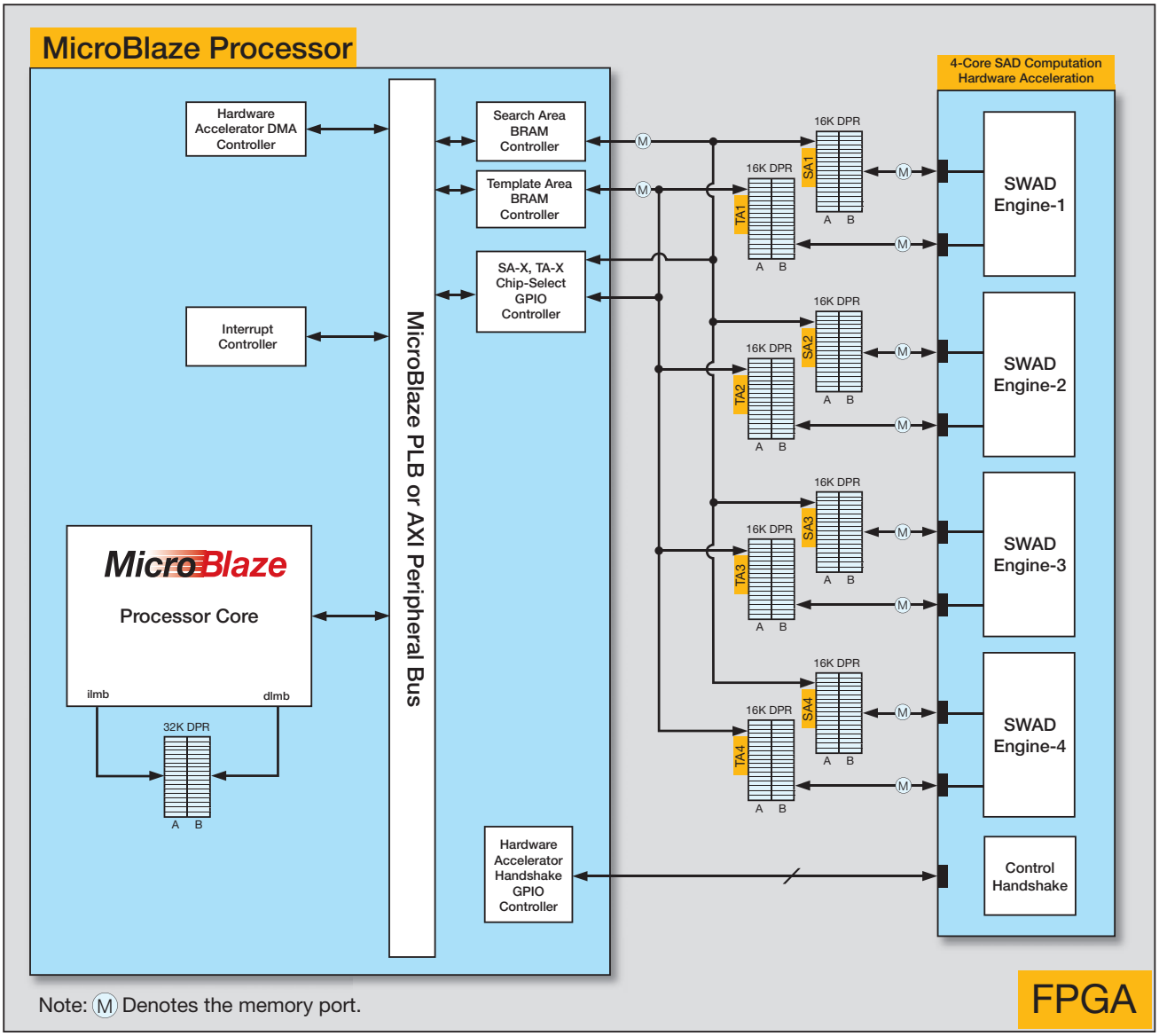


Figure 5 – Vivado HLS-based hardware accelerator and its interconnections

that is as flexible as an all-software implementation and as high in performance as an all-hardware implementation.

The control and data flow of our crowd motion-classification algorithm starts by capturing a video frame through the Video Frame Queue API functions. When a frame is acquired, the user application transfers the current and previous video frame data to the hardware accelerator and gets the motion vectors computed.

At this point, the system computes the motion vectors' statistical properties and classification results in software. The reason for doing so is that these steps do not involve any pixel-level processing and add very little processing overhead. When the classification results are computed, their results and motion vectors are displayed on the processed frame using on-screen display functions. These OSD functions are also implemented in C/C++ in the Xilinx SDK.

With all these building blocks (real-time video pipeline, hardware accelerator and algorithm control/data flow) in place, our overall system design was completed. We tested our FPGA-based implementation in comparison with an earlier desktop PC-based implementation for accuracy of results. The two results were found to be identical. We used various test videos from the University of Minnesota database (*http://mha.cs.umn.edu/proj_recognition.shtml*) and from *www.gettyimages.com* for testing our system.

IMPLEMENTATION RESULTS

The overall design used only 30 percent of slice LUTs, 60 percent of BRAM and 12 percent of DSP48E multiplier resources on our Spartan-6-LX45 FPGA.

Figure 6 shows the hardware setup (top) and actual system output. The hardware setup consists of a Digilent Atlys Spartan-6 FPGA board and a custom video interface card to provide video input/output functionality to the FPGA using a video ADC and DAC. You can watch a detailed demonstration of this system on the following Web links:

http://www.dailymotion.com/video/x2av1wo_fpga-based-real-time-human-crowd-motion-classification-demo_school

http://www.dailymotion.com/video/x23icxj_real-time-motion-vectors-computation-on-fpga_news

http://www.dailymotion.com/video/x28sq1c_crowd-motion-classification-using-motion-vectors-statistical-features_school

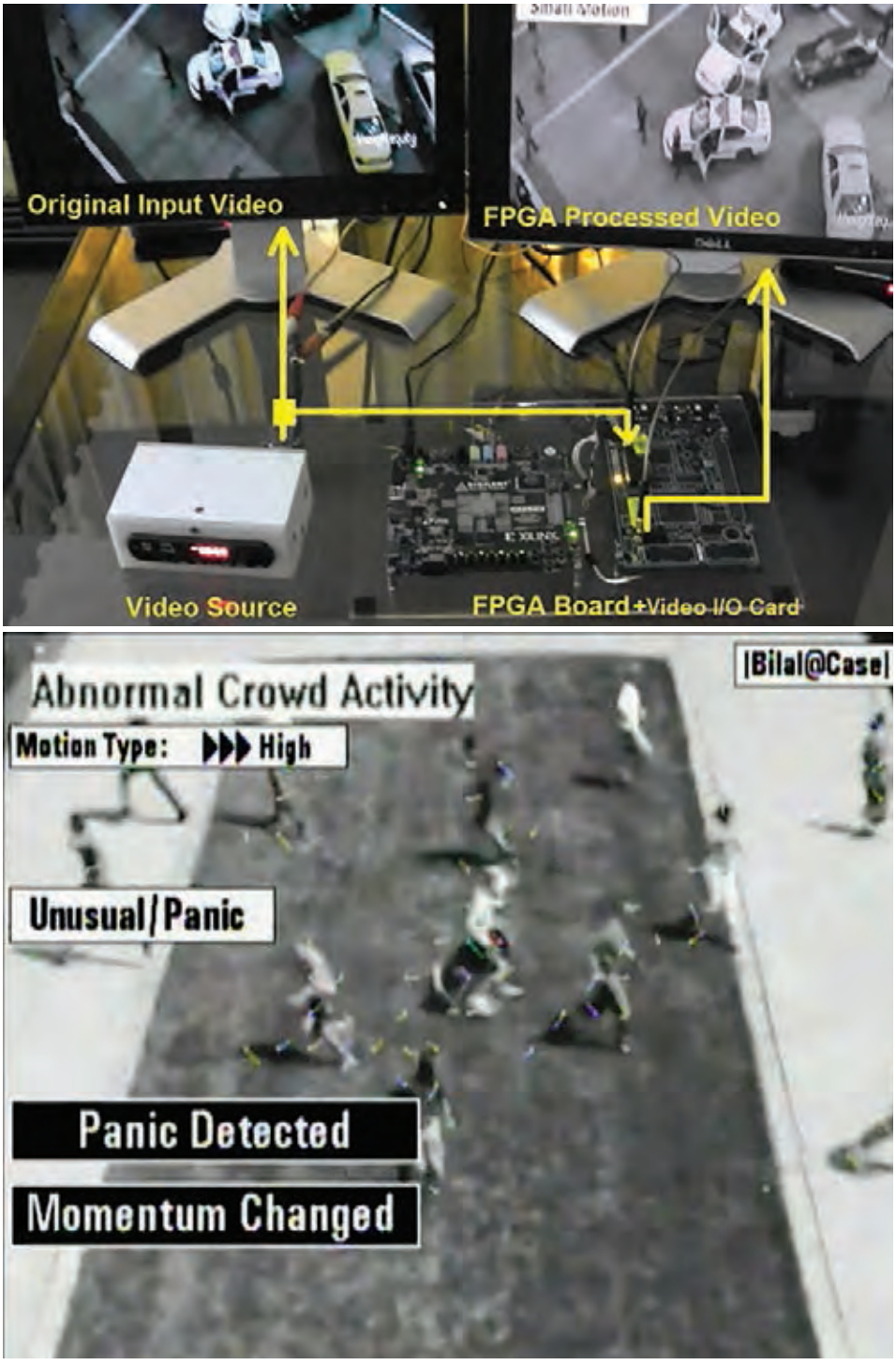


Figure 6 – Hardware setup (top) and an actual FPGA-processed frame classifying a scene as one of panic

GREAT FUTURE POTENTIAL

FPGAs are the ideal platform for performance-hungry applications like real-time video processing. Developing such an application requires a few architectural considerations to fully leverage the performance of the chosen FPGAs. Furthermore, utilizing advanced tools like the EDK and Vivado HLS, it's possible to achieve an overall system design with much more efficiency and in a much reduced development time than in the past.

Thus, there is great potential in implementing performance-critical applications over FPGAs using these tools, as we have demonstrated in this project. With a working platform in place, we hope to extend this work to address more technical problems, such as automatic traffic monitoring, automatic patient observation in hospitals and many more applications.

REFERENCES

1. Ramin Mehran, Mubarak Shah, "Abnormal Crowd Behavior Detection Using Social Force Model," IEEE International Conference on Computer Vision and Pattern Recognition (CVPR), Miami, 2009
2. Duan-Yu Chen, Po-Chung Huang, "Motion-based unusual-event detection in human crowds," *Journal of Visual Computation and Image Representation*, Vol. 22 Issue 2 (2011), pages 178–186
3. OmniTek OSVP: *http://omnitek.tv/sites/default/files/OSVP.pdf*

ACKNOWLEDGEMENTS

The author wishes to thank his co-author, Professor Shoab A. Khan, for his great ideas, and to acknowledge Dr. Darshika G. Parera and Dr. Umair Ahsun for outstanding inspiration.

Test New Memory Technology Chips Using the Zynq SoC

A platform based on Xilinx's ZC706 Evaluation Kit proved fast and flexible enough for MRAM testing at Qualcomm.

by Botao Lee
Senior Staff Engineer
Qualcomm Technologies, Inc.
blee@qti.qualcomm.com

Baodong Liu
Staff Engineer
Qualcomm Technologies, Inc.
baodongl@qti.qualcomm.com

Wah Hsu
Senior Staff Engineer
Qualcomm Technologies, Inc.
wahh@qti.qualcomm.com

Bill Lu
Staff Engineer
Qualcomm Technologies, Inc.
xiaolu@qti.qualcomm.com

The electronics industry is heavily invested in the development of new memory technologies such as PRAM, MRAM and RRAM. The performance of new memory technology test chips is improving rapidly, but work still needs to be done before these devices can go full-scale to compete with or replace conventional memories.

Generally speaking, when a test chip for a new memory technology becomes available, basic tests have already been carried out to check for manufacture-related problems such as stuck-at faults, transition faults and address-decoding faults. But another type of testing is necessary as well in the form of performance-related tests that will disclose how fast the chip can be reliably accessed, as well as how much the chip access speed impacts the performance of the whole computing system.

To successfully carry out the planned performance tests, the test environment must be able to generate configurable digital waveforms to access the chip. It must also be able to construct an entire computing environment to measure the impact of chip access speed. There are many ways to create or purchase a test environment to satisfy these needs. But our team at Qualcomm decided to make our own environment based on Xilinx®'s Zynq®-7000 All Programmable SoC ZC706 Evaluation Kit.

INS AND OUTS OF MEMORIES

Conventional memory technologies like DRAM, SRAM and flash store ones and zeros using an electrical charge in each memory cell. DRAM is widely used in PCs and mobile computing devices to run programs and to store temporary data. SRAM is commonly used as cache memory and register files in microprocessors. It is also frequently found in embedded systems when power consumption is a big concern. Unlike DRAM or SRAM, flash memory offers persistent storage after power is removed from the system. Flash memory runs more slowly than the others, and might wear out with excessively high numbers of programming cycles.

In comparison to conventional charge-based memory technologies, new memory technologies are based on other physical properties of their storage elements. As an example, a memory element of magnetoresistive RAM (MRAM) is formed from two ferromagnetic plates separated by a thin layer of insulator. Each plate can hold a magnetization. One of them is permanent, the other can be changed by

Software runs on the Zynq SoC's ARM A9 processors, while the memory controller core is created using the programmable logic.

an external field to store data. The stored data is read by measuring the electrical resistance of the element. MRAM is similar in speed to SRAM and similar in density to DRAM. Compared with flash memory, MRAM runs much faster and suffers no degradation from programming.

REQUIREMENT ANALYSIS

When devising a scheme for evaluating our MRAM test chip, we settled on a Zynq SoC approach because of the following considerations:

- The FPGA Mezzanine Card (FMC) interface on the ZC706 board provides high-speed signaling capability to and from the memory test chip through an FMC daughtercard.
- The programmable logic (PL) portion of the Zynq SoC provides the ability to construct parameterizable memory controller cores. This is essential to meet the requirement that the test chip access speed can be varied.
- The Zynq SoC's processing system (PS), which consists of two ARM® A9 cores, provides the ability to modify test chip access speed through software.
- The PS also makes it possible to construct a complete computing system. This is essential to meet the requirement that the test system measure the impact of chip access speed on a full computing environment.

HARDWARE AND SYSTEM ARCHITECTURE

The hardware architecture of our chip test environment is illustrated in Figure 1. Software runs on the Zynq SoC's ARM A9 processors, while the memory controller core is created using the programmable logic. We established a DMA channel between the PS and the controller core to move large blocks of data between them easily. The memory test chip resides on the FMC daughtercard, and it talks with the memory controller core through the FMC interface.

The system architecture is illustrated in Figure 2. The three layers on the bottom are hardware layers and the three layers on the top are software layers. We selected Linux as the oper-

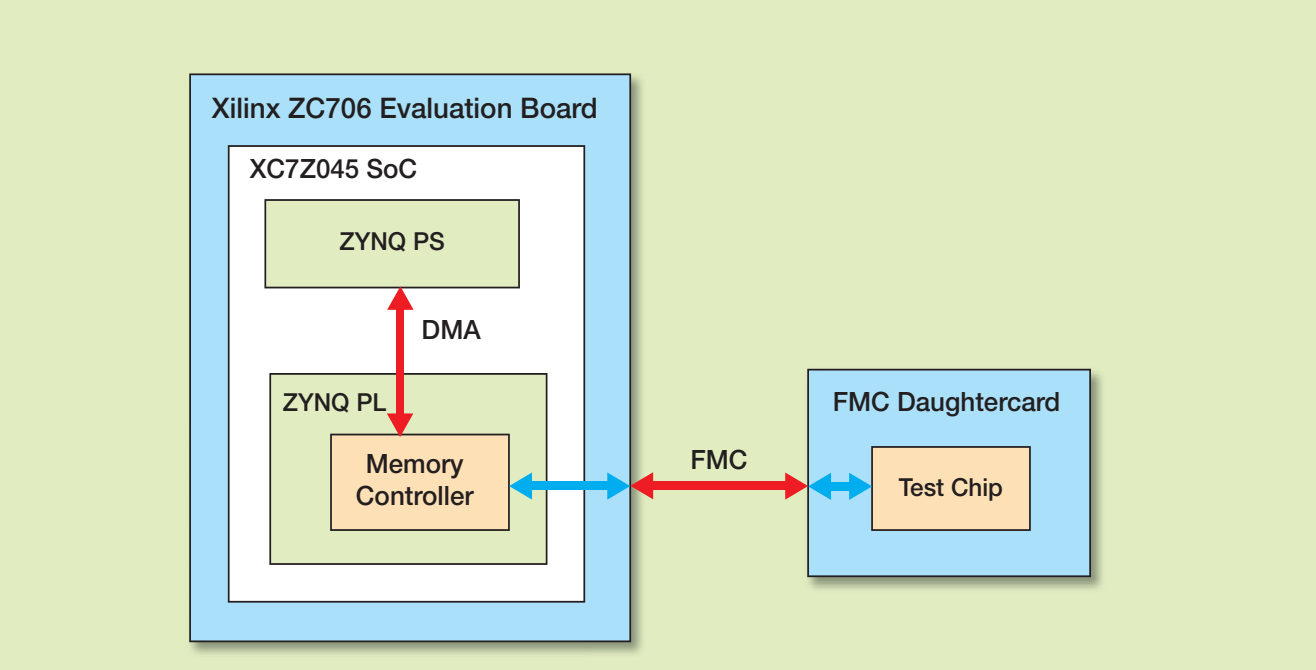


Figure 1 – Hardware architecture of the test environment

Application	Test Configuration Performance Profiling		
OS	Linaro Linux		
Device Driver	Memory Controller Device Driver		
Core	2x ARM A9	Memory Controller	
Device	XC7Z045 SoC		Memory Test Chip
Board	ZC706 Evaluation Board		FMC Daughtercard

Figure 2 – Test environment's system architecture

ating system because it is open source, so the source code can be tweaked if needed. Although no tweaking was done in the current stage of development, it might be necessary to take advantage of some unique properties of new memory chips down the road.

The software we wrote at the application layer fell into two categories. One category was for configuring the memory controller core, and the other one involved profiling the performance of the memory chip and the performance of the whole system.

EASY MIGRATION OF HARDWARE AND SOFTWARE

With help from the local Xilinx FAE, we brought up the test environment within a month. Most of our effort was spent on designing and implementing the interface between software and hardware layers. This is actually one of the reasons that we like the Zynq SoC: It contains both microprocessors and programmable logic in one device, which makes migrating functions between hardware and software fairly easy. In our design, we fine-tuned the software/hardware partition a couple of times and eventually settled on the one we liked. To comfortably work on a Zynq SoC-based system, one needs to understand both hardware and software reasonably well.

Another thing we liked was the Vivado® Design Suite tool chain. The Vivado environment intuitively shows the design blocks, automatically assigns register addresses and checks for errors before exporting hardware information to the software development process. The Vivado Design Suite also provides in-system signal-level debugging ability, which is a must-have to pinpoint the root cause of any RTL issue.

The final thing we want to mention here is the Linux OS. Our software at the application level is heavily GUI based. The popularity of the Linux OS allowed us to leverage our previous experience on Linux GUI development so that we could get the test programs up and running quickly.

QUICK AND COST-EFFECTIVE

Using the Zynq-7000 All Programmable SoC ZC706 Evaluation Kit, our team quickly constructed a complete computing environment for testing new memory technology chips at minimal cost. We expect to one day use the same design methodology to build similar systems for other purposes as well.

Unleashing High-Performance USB Devices with Artix-7 FPGA

The low-power Xilinx FPGA family puts bus-powered USB device designs within easy reach.

by Tom Myers
Senior Hardware Engineer
Anritsu Company
tom.myers@anritsu.com

With several billion ports in the marketplace, Universal Serial Bus (USB) is the go-to interface for subgigabit connections between hosts and peripheral devices. However, due to the USB specification's stringent in-rush and steady-state operating-current limitations, FPGAs were often overlooked for bus-powered device applications in favor of lower-performance, less flexible microcontroller solutions.

With the arrival of the Artix®-7, the latest addition to the Xilinx® low-power device portfolio, this is no longer the case. Paying careful attention to system-level power conversion efficiency and sequencing, and using the power estimation and optimization tools available in the Vivado® Design Suite, designers can overcome these challenging limits to provide a tightly integrated, tailored bus-powered device with high performance.

Let's take a look at how to build a USB 2.0 high-speed bus-powered device around an Artix-7 MicroBlaze™-based platform. We successfully used this approach at Anritsu in the development of a recent microwave power measurement product. The new design's USB 2.0 High-Speed interface provides a substantial increase in measurement throughput compared with the USB Full-Speed microcontroller-based solution used in the previous-generation product. Increased measurement throughput reduces test time in a manufacturing production test application. The result is a cost savings for our customers.

SYSTEM DESIGN

In our project at Anritsu, we knew the major obstacle we had to overcome was going to be the 500-milliamp (5-volt nominal) steady-state current draw limit. Therefore, our approach to system design centered

Lowering the device's die temperature reduces leakage power consumption. Strategies include minimizing device die size and selecting the largest package possible.

around the power budget. We tabulated typical and maximum current draws from datasheet numbers on a power budget spreadsheet.

A large part of the power budget was driven by the minimum off-chip memory requirement of 200 Mbytes. The best fit for this requirement was found to be a standard 4-Gbit LPDDR2 device. We generated current-draw estimates for this device with the detailed methodology provided by vendor application notes, applying the estimated data flow profile. We also evaluated various programmable devices and other solutions with tools such as Xilinx Power Estimator with assumptions as to the functionality, clock speed and toggling rates.

We identified a few candidate devices and refined the power, size and I/O estimates by building out a subset of the complete system with a MicroBlaze, the memory controller (using Memory Interface Generator, or MIG) and adding in the interface blocks for the various peripherals using Vivado's IP Integrator tool. We quickly obtained a synthesizable target and refined the power consumption numbers with Vivado power reports.

Since MIG doesn't provide a native AXI connection to LPDDR2 devices, we developed this link later in-house. Until our shim was available, we used the MIG-generated LPDDR2 example design for these preliminary power estimate and sizing builds. Figure 1 shows the resulting system architecture.

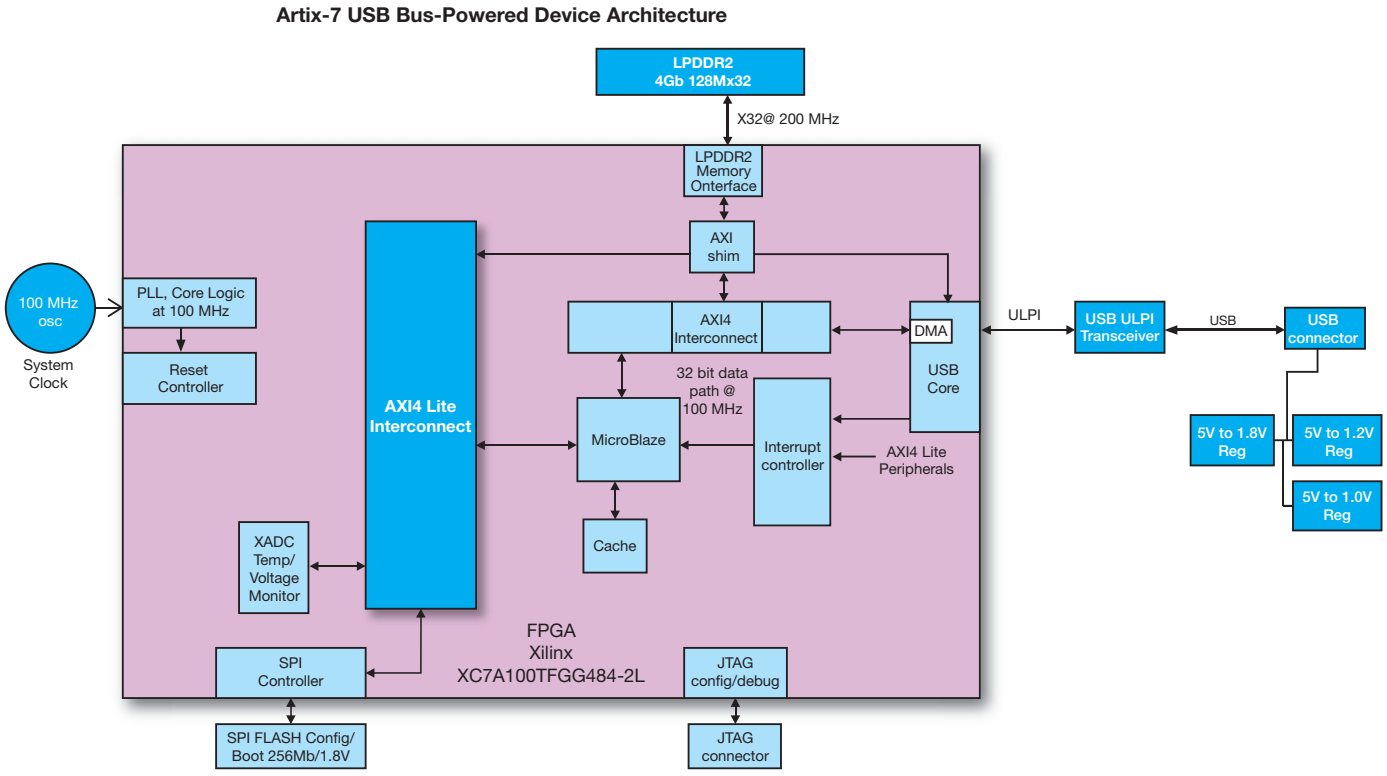


Figure 1 – System architecture for our Artix-7-based design

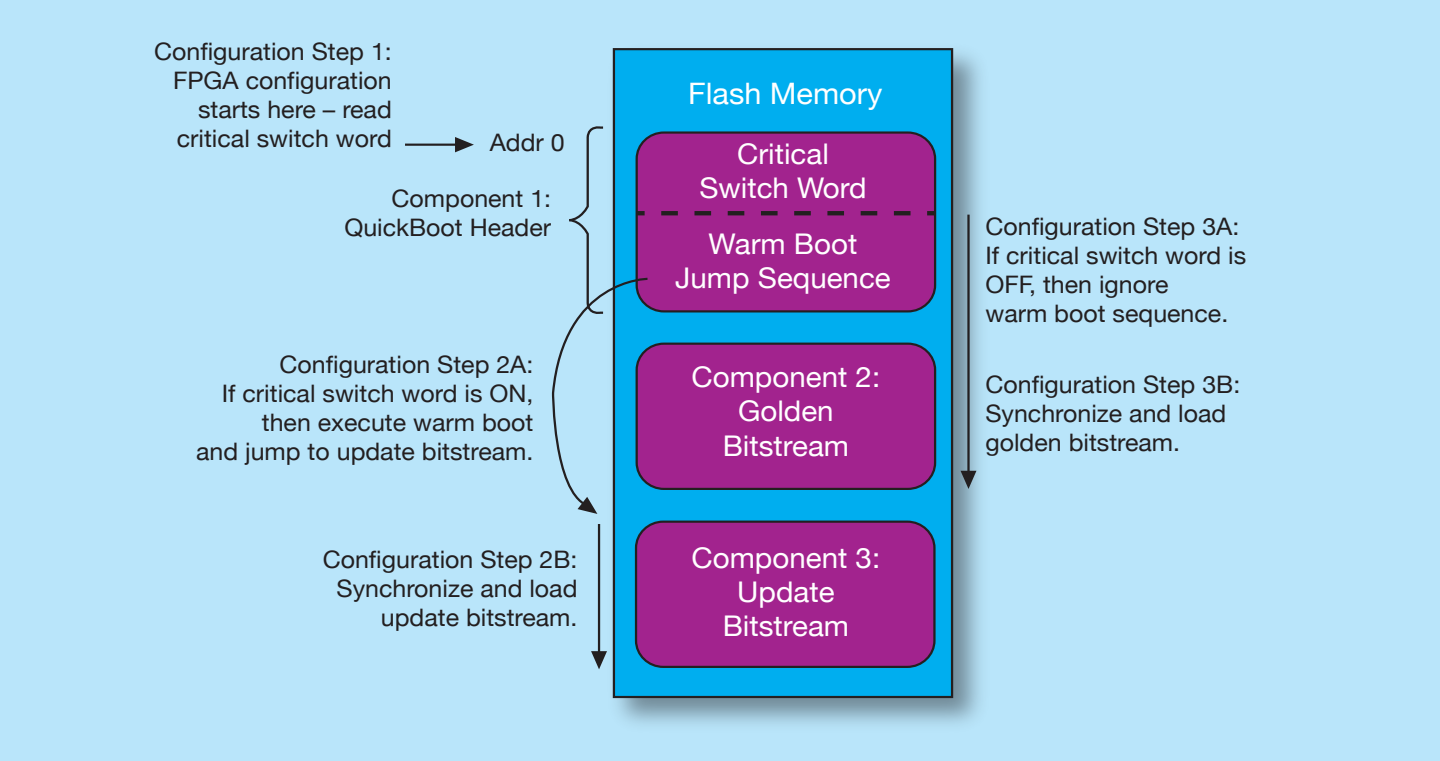


Figure 2 – QuickBoot flash memory components and configuration method

As described in the "Vivado Design Suite User Guide: Xilinx Power Analysis and Optimization" (UG907), lowering the device's die temperature reduces leakage power consumption. Strategies we used included minimizing the device die size and selecting the largest physical device package possible, given the application's tight board real estate constraints.

We minimized conversion losses and regulator circuitry costs by reducing the number of voltage rails. After locking down the device power requirements, we designed the voltage-conversion circuits to step down from the nominal USB 5-V bus voltage to these rails.

Up to this point, we have been focusing on steady-state current draw. However, one must also keep in mind inrush current draw. One way to minimize inrush current is to select regulators with soft-start capability and sequence them. You must carefully balance the FPGA's sequencing and ramp time requirements with the USB requirements.

EXPECT THE UNEXPECTED

Although various mechanisms are provided to gracefully shut down and remove USB devices, in reality many users will abruptly unplug the device without warning. This can be a problem if the firmware update process is not sufficiently robust, leading to nonresponsive, "bricked" devices, unhappy customers and expensive device returns for firmware recovery. Anritsu competes on the basis of reliability and speed for high-volume manufacturing test. Therefore, our main requirements included fast boot time and fast firmware update time.

We solved this problem by implementing the QuickBoot golden-image firmware update architecture and processes described in Xilinx application note XAPP1081 and summarized in Figure 2. The traditional 7 series fallback multiboot solution provides a boot process that maintains a known-good "golden" image including bitstream in the configuration flash memory. During the update process, an updated "working" image is loaded in memory after the golden image. If the update process fails, or the working image is somehow corrupted, the FPGA automatically detects the error and falls back to the golden image. The XAPP1081 QuickBoot method extends this procedure with improved configuration time and golden-image update features.

Based on the success of this project, we are looking ahead to how the next generation of devices from Xilinx might enable additional functionality for Anritsu products. For example, a large amount of the power budget is consumed by the off-chip SDRAM interconnect. We look forward to investigating how we might use the newer 16-nanometer UltraScale+ lineup's UltraRAM to reduce or eliminate this load and perhaps put the ARM7-enabled Zynq®-7000 All Programmable SoC product line in reach for our application.

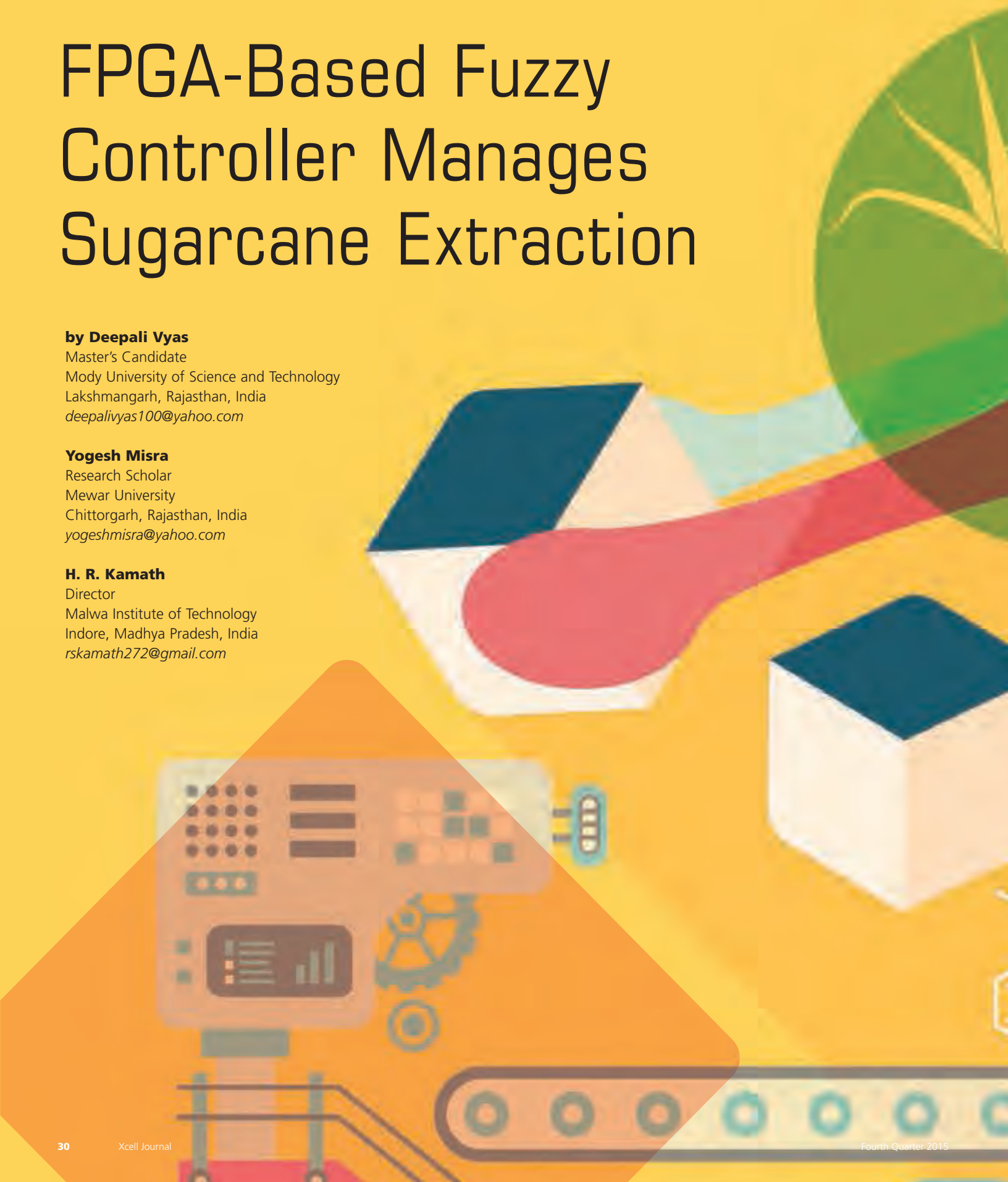
For further information, contact *tom.myers@anritsu.com*.

FPGA-Based Fuzzy Controller Manages Sugarcane Extraction

by Deepali Vyas
Master's Candidate
Mody University of Science and Technology
Lakshmangarh, Rajasthan, India
deepalivyas100@yahoo.com

Yogesh Misra
Research Scholar
Mewar University
Chittorgarh, Rajasthan, India
yogeshmisra@yahoo.com

H. R. Kamath
Director
Malwa Institute of Technology
Indore, Madhya Pradesh, India
rskamath272@gmail.com



A three-input fuzzy controller implemented in a Xilinx Virtex-6 FPGA maintains the cane level during the sugar-manufacturing process.

Sugar is an important food ingredient that's widely used in day-to-day life. More than half of the total global supply of raw sugar is produced from sugarcane. India is the second largest sugar manufacturer in the world after Brazil, with 60 million cane farmers and their dependents involved in cane cultivation, making for a $12 billion business.

Because the extraction of cane juice is a nonlinear process, our team looked to fuzzy logic for a way to improve the flow. Analysis by researchers at the Mody Institute of Science and Technology (MITS) reveals that the performance of our fuzzy-based controller, designed and implemented in a Xilinx® FPGA, has proven better than that of a conventional controller. With the specific parameters for crushing 2,500 tons of cane per day, a flow rate of 26.6 kg/second is required.

Before looking into the details of how we implemented our three-input fuzzy controller, it's helpful to understand the basics of sugar manufacturing.

HOW CANE IS EXTRACTED

A schematic of the cane juice extraction process is shown in Figure 1. The cane billets that the sugar mill receives from cane growers are weighed and dumped in the cane yard. From there, a crane lifts them to the cane carrier. The cane carrier moves continuously and is responsible for bringing the cane to the factory floor for sugar production.

The cane first passes through two sets of rotating knives. Cane knives cut the cane into pieces, while shredder knives prepare the fiber. A rake carrier feeds these small fibers, which measure roughly 1 to 2 cm, to a Donnelly chute. Cane juice is extracted by crushing the fibers in two or three rolls of the mill. This process is repeated through sets of five or six mills. Residual cane known as bagasse is sent to the boiler, where it is burnt as fuel, while the extracted juice is sent for clarification and then to the pan section, where sugar is made from the juice.

The supply of cane for processing is very uneven, and this uneven supply of cane during juice extraction adversely affects the efficiency of the sugar mill and can cause mill breakdown, stoppage and jamming of the equipment. For optimum juice extraction, it's necessary to maintain the cane level in the Donnelly chute at a desired height.

We expected that fuzzy logic would nullify the uneven supply of cane and maintain the cane level at the desired height by varying the speed of the rake carrier better than conventional controllers. This is why we made an attempt to introduce the concept of fuzzy logic to the sugar world.

Our first step in 2014 was to design a two-input fuzzy controller [1] that precisely monitored the variation in two parameters: weight of cane on the rake carrier and height of cane in the Donnelly chute.The controller aims at maintaining a constant level in the chute so as to maintain the needed flow rate of 26.6 kg/s. When we compared the results with those of the conventional controller, it was clear that the two-input fuzzy controller performed much better. Since the cane is crushed between the rolls, we decided as an experiment to introduce a third parameter—roll speed—on the same algorithm. The addition of this third parameter revealed that roll speed is an equally important variable as the other two.

So, later in 2014 we introduced roll speed as our third parameter [2]. We redesigned the algorithm with this additional parameter and implemented it using MATLAB®. When the software implementation of our new

three-input controller was completed [3], the next step was to implement the algorithm and develop an entire fuzzy system using a Xilinx FPGA. FPGAs are reprogrammable silicon chips that provide real-time hardware implementation of electronic circuits. They are highly reliable, cost-effective and provide a medium to check the performance of a circuit before fabrication. Thus, the Xilinx Virtex®-6 FPGA turned out to be a perfect solution for our hardware implementation.

HARDWARE DESIGN

Figure 2 shows the algorithm for a three-input fuzzy controller. The control philosophy remains the same as in the two-input version, modified according to the three inputs and implemented on MATLAB. The control philosophy is such that it governs weight, height and the three conditions of roll speed: that is,when the speed is Roll Low (RL; 12 cm/s), Roll Medium (RM; 14.3 cm/s (RM) and Roll High (RR; 16.6 cm/s).

After designing the controller on MATLAB, our next step was to design the hardware required to measure the input parameters. Load cell measures the amount of cane on the rake carrier. To measure the level of cane in the chute, we added height sensors to the design. A tachogenerator sensor measures the rotational speed of rolls.

The output of the load cell, height sensor and tachogenerator is in microvolts. In order to use these metrics in the next steps of the process, we had to amplify the values to a measurable level, namely, from microvolts to millivolts. We accomplished this amplification using a signal-conditioning system on PSpice. Next, we converted the results to a digital value using an analog-to-digital converter (ADC) that we connected in series with the conditioning system. In this way the amplified input is fed to the controller.

FIVE-STEP PROCESS

The VHDL implementation of a fuzzy controller using Xilinx hardware is divided into five steps: fuzzification of inputs, rule evaluation, implication, aggregations and defuzzification.

There are two methods of designing a fuzzy-logic controller, Mamdani and Sugeno. The Mamdani method is difficult and very complex. According to the research, the Mamdani method requires the centroid of a two-dimensional shape by integrating across a continuously varying function. Hence, this method is not computationally efficient. On the other hand, the Sugeno method of design is much simpler. Therefore, we adopted the Sugeno method of implementation.

The first and foremost step, fuzzification, includes conversion of crisp values to fuzzy values, which are then represented by membership functions. Crisp values are those that belong to a particular set; fuzzy values lie in a particular range and are not confined to a particular set.

The three input variables are weight, height and roll speed. A triangular membership function is used to represent these input variables. The universe of discourse of input parameter "WEIGHT" is in the range of 500 kg to 1,000 kg, and it is fuzzified into 11 triangular linguistic variables (LV). The universe of discourse of input parameter "HEIGHT" is in the range of 0 to 180 cm and it is fuzzified into seven triangular LVs. The universe of discourse of input parameter "ROLLSPEED" is in the range of 12 cm/s to 16.6 cm/s and it is fuzzified into three triangular LVs.

The fuzzification in terms of VHDL code is done as follows. We defined each membership function by three points and two slopes, as shown in Figure 3. The upward slope (Slope 1) and downward slope (Slope 2) can be evaluated using the following formula:

```
S1= (y2-y1/Point2-x1)
S2= (y2-y1/x2- Point 2)
```

The degree-of-membership (DOM) function (μ) forms the next step of fuzzification. Our algorithm divides a membership function into four segments, namely Segment-1(μ=0), Segment-2{(Input - point 1)* slope 1}, Segment-3{(Input-point 2)* slope 2} and Segment-4 (μ=0). The value of DOM is calculated as follows:

- If input value < Point 1 (Segment 1), then DOM =0.
- If input value ≤ Point 2 and ≥ Point 1 (Segment 2), then DOM = (Input Value –Point 1) * Slope 1.
- If input value ≤ Point 3 and ≥ Point 1 (Segment 3), then DOM = FF- (Input Value –Point 2) * Slope 2.
- If input value ≥ Point 3 (Segment 4), then DOM = 0.

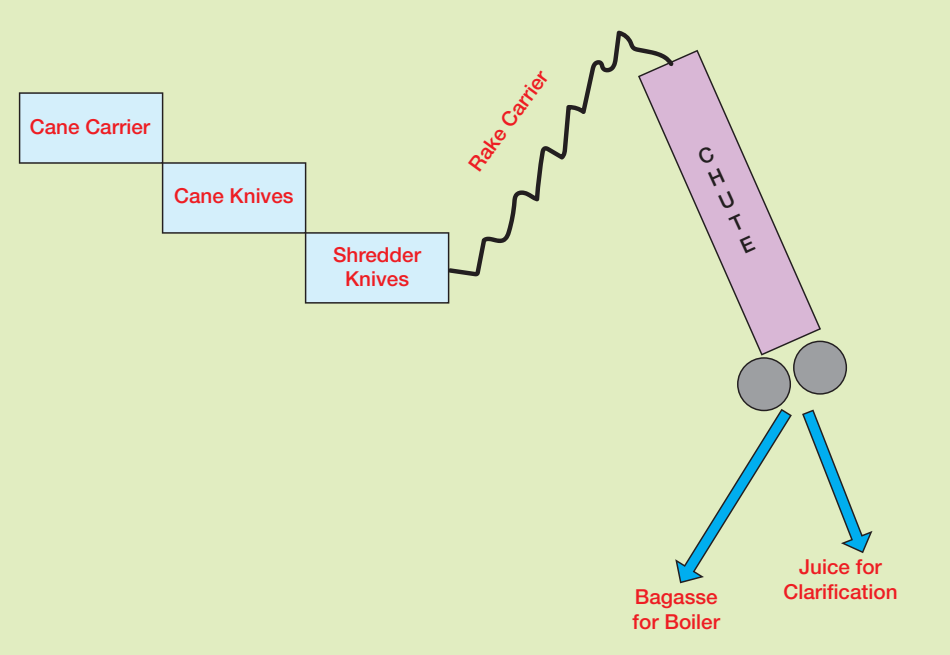


Figure 1 – Schematic of the cane juice extraction process

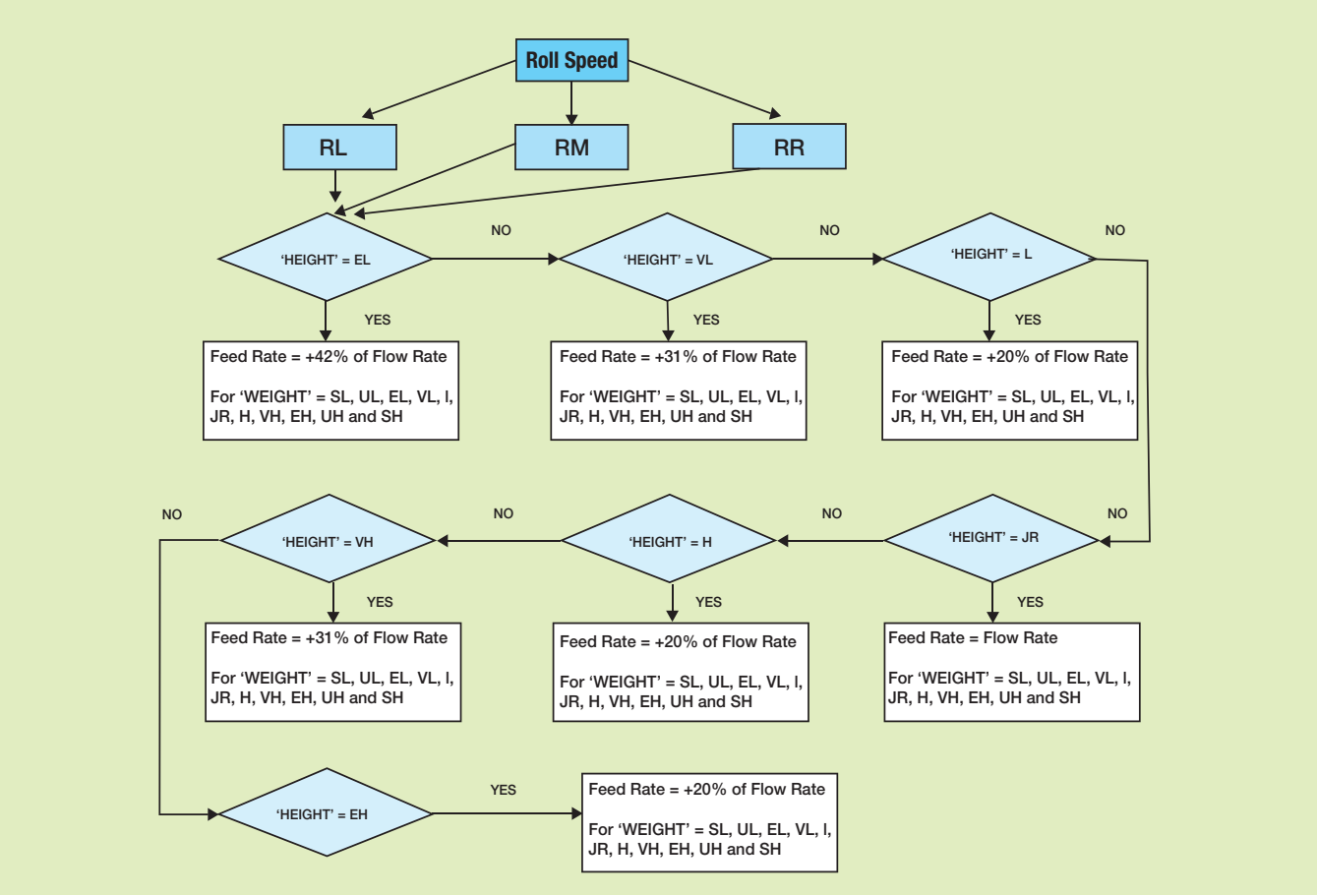


Figure 2 – Development algorithm for a three-input fuzzy controller

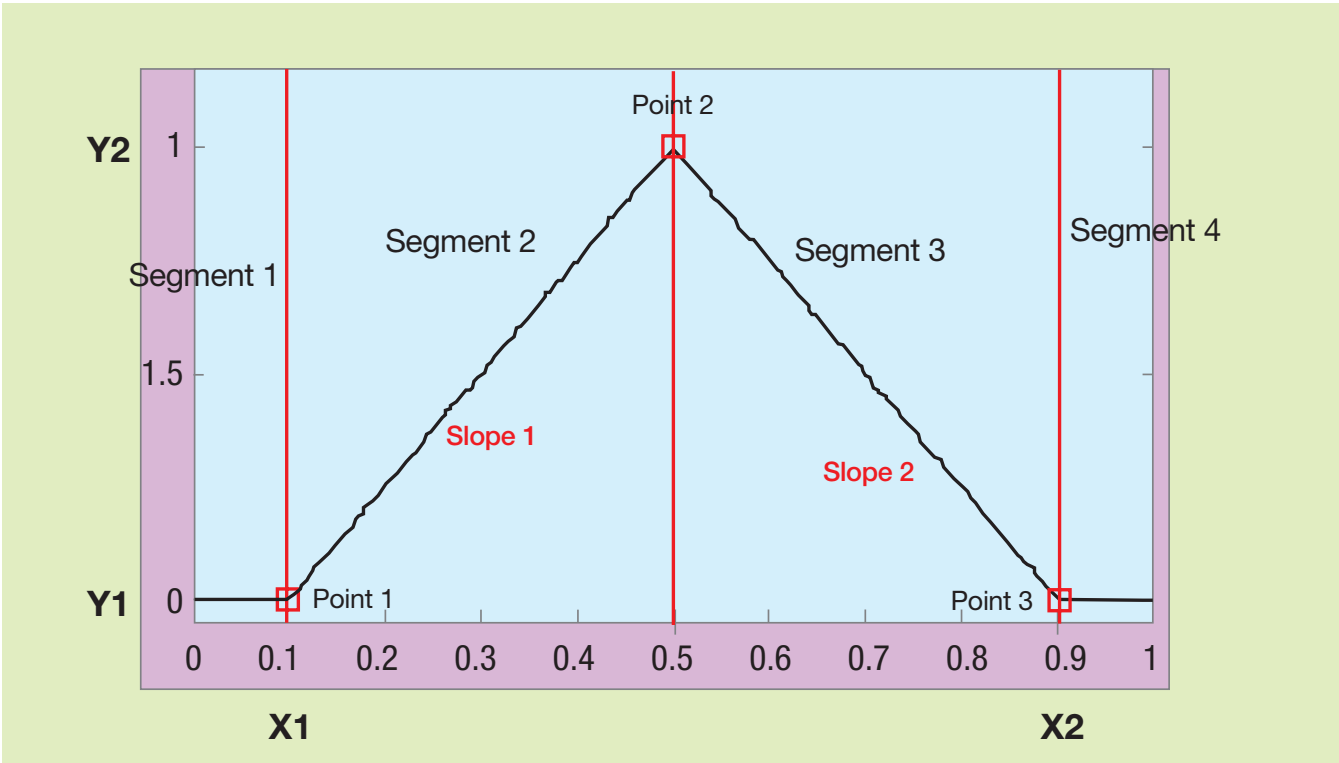


Figure 3 – Membership function, defined by three points and two slopes

DIFFERENT DEGREES OF MEMBERSHIP

The next step is the designing of rules so as to determine the action to be taken in response to the different degree-of-membership functions. A fuzzy rule is formed using a simple If-Then condition where we have a consequence to every antecedent. The Fuzzy Logic Toolbox in MATLAB provides different operators to combine multiple antecedents. We selected the AND operator to combine three antecedents, since it represents a minimum operation among multiple antecedents. For a three-input controller, a total of 231 rules are generated. We designed a rule matrix for these rules. A minimum function finds the minimum of three values—that is, the minimum of DOM among three input variables is calculated.

We also found that the consequent of many rules is the same. All such rules having the same consequent are collected and the maximum of these values is calculated using a maximum function. At the next step, we collected all rules having the same consequents. Different maximum functions are encoded to evaluate a maximum value for the entire LV.

After the output for each rule has been identified, the last step is to combine all the output into a single value—in other words, these values are to be converted to crisp values. This is done using defuzzification.

Defuzzification forms the last and an important step in designing a fuzzy system. Defuzzified values generate a single crisp value, which is the speed of the rake motor. The Sugeno method of defuzzification that we used is the weighted-average method. In this method, we multiply the fuzzy output obtained from aggregation with its corresponding singleton value, then divide the sum of these values by the sum of all the fuzzy output obtained from the rule evaluation—that is, the values obtained after aggregation.

VIRTEX-6 IMPLEMENTATION

After implementing the above steps, we successfully designed a three-input fuzzy controller using a triangular membership function and centroid defuzzification. The program code is available with the authors. We simulated our three-input fuzzy controller using the Fuzzy Toolbox of MATLAB version 7.11.0.584 (R2020b) and implemented it on a Xilinx Virtex-6 FPGA with Xilinx's ISE® Design Suite

Parameters		Cane level (cm)	Cane weight (kg)	Motor speed (rpm)	Carrier speed (cm/s)	Cane in carrier (kg/cm)	Feed rate (kg/s)	Data for next sampling		Cane level *(VHDL)	Cane (MATLAB) ** (cm)
Time (sec)	Roll speed (cm/s)							kg	cm		
0	15.4	90.0	750	47.0	24.6	0.938	23.1	-16.0	-6.4	83.6	85.7
10	15.8	83.6	729	52.0	27.2	0.911	24.8	-5.0	-2.0	81.6	84.3
20	15.0	81.6	792	50.0	26.2	0.990	25.9	+19.0	+7.6	89.2	90.4
30	16.2	89.2	908	42.0	22.0	1.135	25.0	-9.0	-3.6	85.6	86.5
40	16.6	85.6	965	44.0	23.0	1.206	27.7	+11.0	+3.9	89.5	90.5
50	13.4	89.5	720	49.0	25.7	0.900	23.1	+16.0	+6.4	95.9	95.9
60	13.8	95.9	760	39.0	20.4	0.950	19.4	-27.0	-9.6	86.3	86.3
70	13.4	86.3	790	44.0	23.0	0.988	22.7	+12.0	+4.8	91.1	91.3
80	15.4	91.1	820	46.0	24.1	1.025	24.7	0.0	0.0	91.1	93.4
90	16.2	91.1	555	73.0	38.2	0.694	26.5	-4.0	-1.6	89.5	93.4
100	13.0	89.5	609	51.0	26.7	0.761	20.3	-5.0	-2.0	87.5	92.3
110	14.3	87.5	578	62.0	32.5	0.723	23.5	+6.0	+2.4	89.9	90.2
120	14.6	89.9	598	57.0	29.8	0.748	22.3	-11.0	-4.4	85.5	87.0
130	12.3	85.5	700	44.0	23.0	0.875	20.1	+4.0	+1.6	87.1	88.8
140	12.6	87.1	679	48.0	25.1	0.849	21.3	+11.0	+4.4	91.5	91.7
150	15.4	91.5	800	46.0	24.1	1.000	24.1	-6.0	-2.4	89.1	91.3
160	12.0	89.1	845	32.0	16.8	1.056	17.7	-15.0	-6.0	83.1	84.2
170	14.3	83.1	835	45.0	23.6	1.044	24.6	+17.0	+6.1	89.2	90.3
180	14.6	89.2	874	42.0	22.0	1.093	24.0	+6.0	+2.4	91.6	92.1
190	15.0	91.6	900	41.0	21.5	1.125	24.2	+2.0	+0.8	92.4	92.1
200	15.4	92.4	924	40.0	20.9	1.155	24.1	-6.0	-2.4	90.0	91.4

* Cane level of FPGA-implemented system after each sampling ** Cane level of MATLAB-implemented system after each sampling

Table 1 – Cane level at a 90-cm roll speed varies during each sample.

14.5 using VHDL. The sampling period is 10 seconds and total simulation duration is 200 seconds.

We have examined in total 756 different conditions of input parameters under six different cases, but here we have focused on the case when the cane level and cane weight on the carrier at the initial stage of simulation are 90 cm and 750 kg respectively. The roll speed varies at the time of each sampling. The simulation result is given in Table 1.

The steps of hardware implementation included VHDL modeling, simulation, synthesis and FPGA implementation in our lab on the MITS campus. We used a mixed type of modeling for designing the VHDL model of our three-input fuzzy controller, which includes behavioral and structural modeling. The design is simulated on Xilinx's ISim simulator. The waveform that ISim generated verified the functionality of the controller. Figure 4 shows the simulated waveform for the case when the weight of cane in the rake carrier is 750 kg, the height of the cane level in the Donnelly chute is 90 cm and the roll speed is 16.6 cm/s. Under these conditions, the expected rake motor speed is 54.2 rpm (MATLAB). The defuzzified simulated results are 36H, or 54 rpm, which matches with the MATLAB results and verifies the design.

After simulation, we synthesized the design to generate the technology schematic and the approximate device utilization report. We found that our design used more than 78 percent of the Virtex-6's slice lookup tables (LUTs), 93 percent of occupied slices, 1 percent of slice registers and 1 percent of LUT flip-flops.

We then compared the VHDL results with results of a conventional controller (see Table 2). The comparison proves that the fuzzy-logic system is more efficient than a conventional controller.

The lab at MITS provides a Spartan®-6 FPGA for research use. However, we found that the number of LUT blocks required exceeded the capacity of the target device. This is why we implemented the design on a Virtex-6 instead. But due to a lack of resources, we couldn't perform real-time implementation in the lab.

At the next step we are looking forward to linking up with the government of India's National Sugar Institute to develop the whole system and verify the results in a real-world environment. We have already delivered a presentation to the institute and received a positive response. It is our belief that the concept of fuzzy logic stands poised to change the future of the sugar industry.

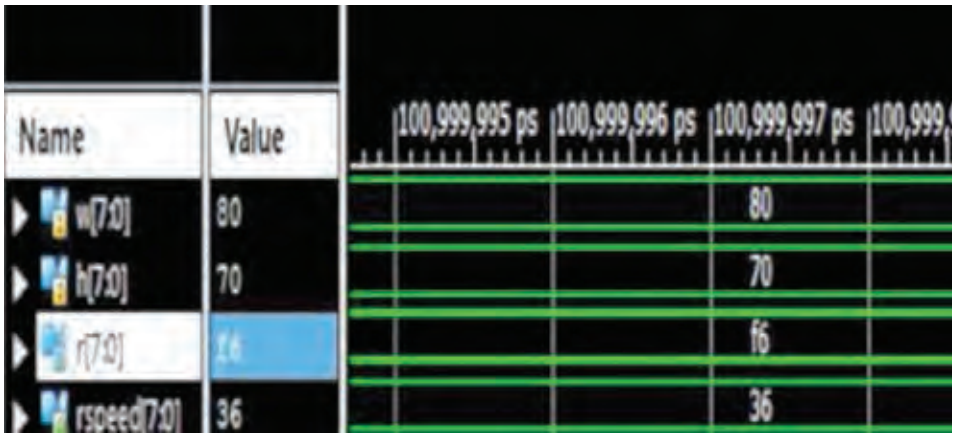


Figure 4 – Simulated waveform for the case when weight = 750 kg, height = 90 cm and roll rate = 16.6 cm/s

REFERENCES

1. Y. Misra and H.R. Kamath, "Design Algorithm of Conventional and Fuzzy Controller for Maintaining the Cane Level During Sugar Making Process," *International Journal of Intelligent Systems and Applications*, ISSN: 2074-9058,Vol.7 No.1, December 2014

2. Y. Misra and H.R. Kamath, "Implementation and Performance Analysis of a Three Inputs Conventional Controller to Maintain the Cane Level During Cane Crushing in FPGA using VHDL," *International Journal of Engineering Research & Technology* (IJERT), ISSN: 2278-0181, Vol. 3 Issue 9, September 2014

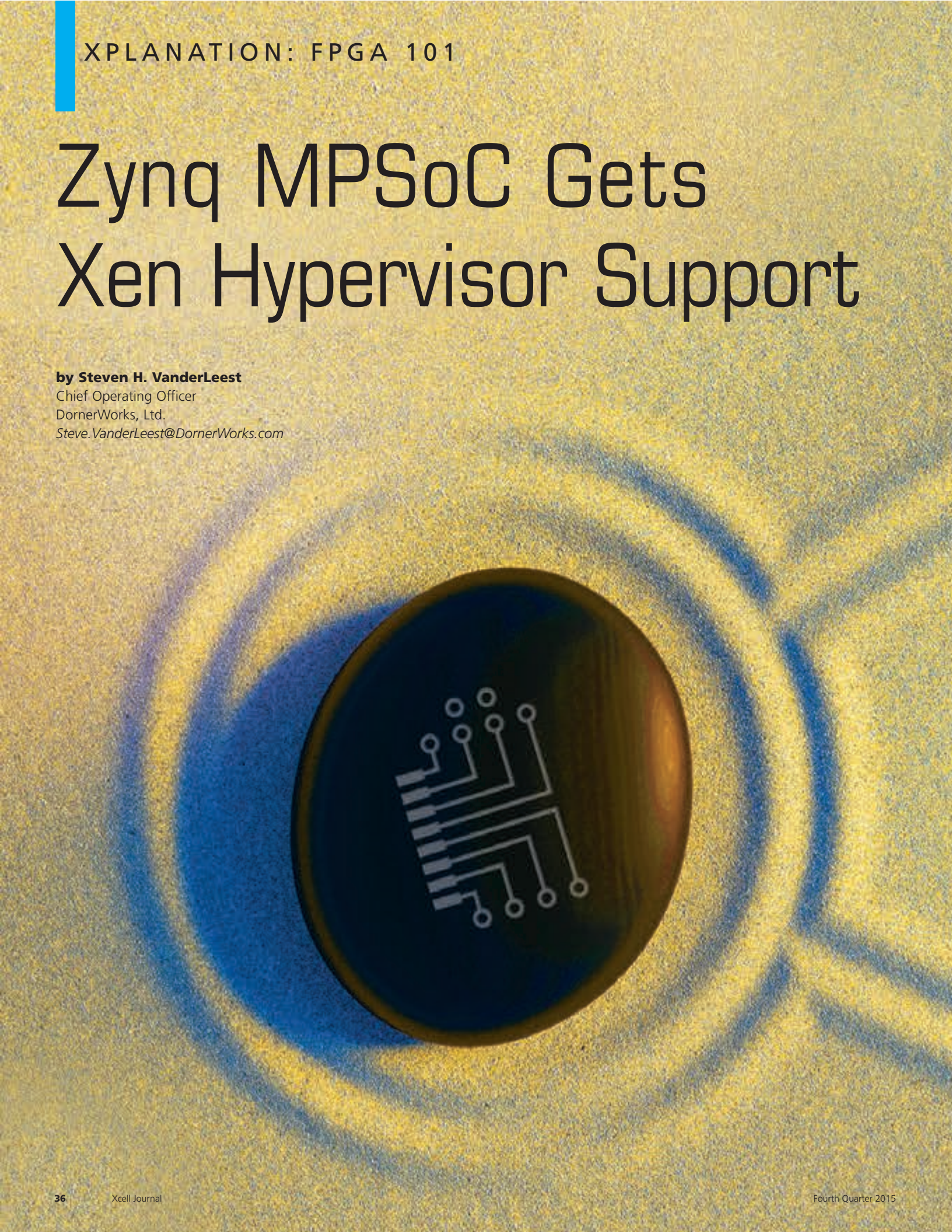
3. Y. Misra and H.R. Kamath, "Analysis and design of a three input fuzzy system for maintaining the cane level during sugar manufacturing," *Journal of Automation and Control* (accepted), ISSN: 2372-3041

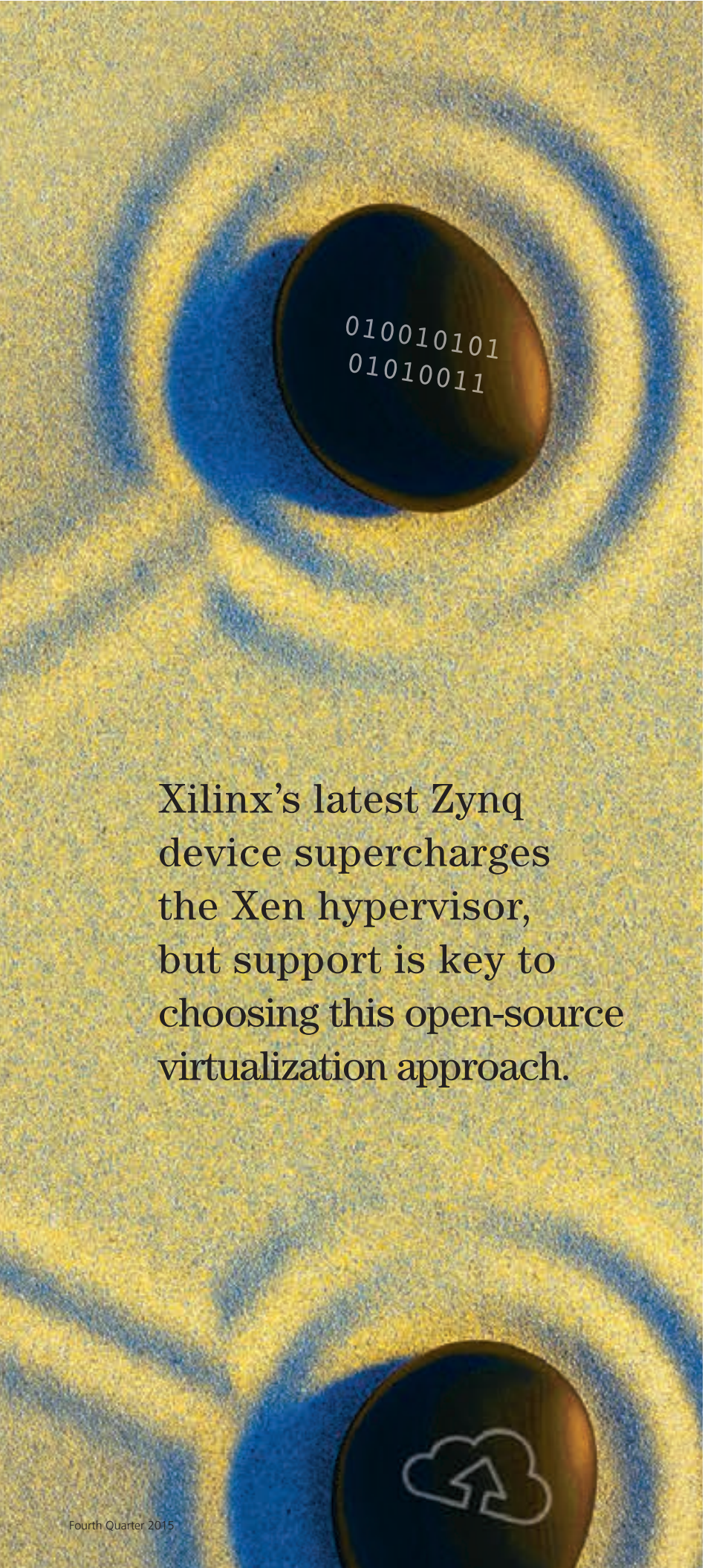
	Three-input conventional controller	Three-input fuzzy controller (MATLAB)	Three-input fuzzy controller (Xilinx VHDL)
Percentage of time cane is in between 85 cm-95 cm (% Time)	45.8	94.7	88.0
Lowest level of cane in chute (cm)	61.7	84.2	81.6
Highest level of cane in chute (cm)	103.5	95.9	95.9

Table 2 – Comparison of results

Zynq MPSoC Gets Xen Hypervisor Support

by Steven H. VanderLeest
Chief Operating Officer
DornerWorks, Ltd.
Steve.VanderLeest@DornerWorks.com





Xilinx's latest Zynq device supercharges the Xen hypervisor, but support is key to choosing this open-source virtualization approach.

The Xen open-source hypervisor is a full-featured virtualization technology traditionally used in cloud computing and more recently finding its way into embedded systems. DornerWorks is providing Xen support on the new Zynq® UltraScale+ MPSoC device, bringing multiple benefits to Xilinx users. Not only does the Xen Zynq hypervisor enable fast software integration and heightened system safety and security, it also brings the power of enterprise-style cloud computing to the embedded world.

The rigorous design compartmentalization that the hypervisor provides makes for rapid integration of new software (including entire operating systems) on the same computing device. At the same time, the isolation reduces or even eliminates unexpected interference between independent software functionality.

Moreover, isolation greatly enhances the level of system safety and security by reducing unexpected interaction between functions and presenting a smaller attack surface exposed to threats, thus making it easier to prove safety or security properties. The arrival of enterprise-style cloud computing in the embedded world offers many of the same advantages, such as deployment of legacy software on new hardware with few (if any) changes to the software.

Let's briefly review what hypervisors are before getting into the specifics of Xen Zynq, the open-source Xen hypervisor on the Zynq MPSoC.

WHAT IS A HYPERVISOR?

A hypervisor is the foundational software layer enabling virtualization. Just as an operating system (OS) manages simultaneously running applications, each contained within a process with access to machine resources managed by the OS, so the hypervisor manages simultaneously running operating systems, each contained within a virtual machine with access to machine resources managed by the hypervisor.

Virtualization is an idea dating back to the 1960s. Popek and Goldberg formalized the idea of a virtual machine monitor (VMM) in 1974 with three defining characteristics:

- The VMM provides programs with a runtime (virtual) environment essentially the same as the original (physical) machine.
- The VMM has a negligible impact on performance.
- The VMM manages the system resources.

A hypervisor is a VMM focusing almost exclusively on the basic machine management tasks. That means some commonly expected services like file systems, graphical user interfaces and network protocol stacks are not implemented at this level, but rather are delegated to a higher layer, such as a guest operating system running in one of the virtualized machines the hypervisor is hosting.

A hypervisor running natively on hardware, as described above, is considered a Type 1 hypervisor. By contrast, a Type 2 hypervisor is not the lowest layer of software, but is hosted on an operating system. This type is typically used to allow one OS to run on another, such as when a Mac user runs Windows on their MacBook using Parallels or when a Windows user boots up Linux within a virtual machine using VirtualBox.

There are also important differences between enterprise and embedded hypervisors. Cloud computing and big data are typical enterprise use cases for hypervisors. Hypervisors in the embedded space are a more recent development, with adoption occurring as processors appeared with sufficient performance and acceptable power consumption.

Use cases for embedded hypervisors have a common theme: consolidation of multiple complex functions into a single computing platform while maintaining some separation between them. For aerospace applications, a hypervisor is often used to support integrated modular avionics, where the software formerly executing on federated (independent) avionics hardware is consolidated into a single computing platform. The functions might include flight control, navigation, flight management systems, collision avoidance and more. The FAA requires that combined software functions that were formerly running on separated hardware cannot affect one another. This isolation is accomplished via rigorous partitioning defined by standards such as DO-248C.

While the FAA is concerned with the safety of commercial flights when consolidating functionality, military avionics has a parallel need to provide separation in order to support security. Approaches that support multiple classification levels on one system with strict separation use an architecture called Multiple Independent Levels of Security (MILS).

For health care applications, the industry is considering similar consolidation using hypervisors for high-end medical devices such as MRI scanners, robotic (or robotically assisted) surgery devices and CT scan machines, all of which currently incorporate multiple independent processing systems. The combined functions might include graphical user interfaces for physicians, image processing, real-time motor control, patient information databases and system management functions.

For automotive applications, hypervisors are an attractive way of combining the dozens of separate microprocessors and microcontrollers embedded in a car. Virtually all automotive OEMs are considering the move to hypervisors to combine functions such as infotainment, driver and passenger controls, advanced driver assistance systems (ADAS), instrument cluster, navigation systems, Internet connectivity and, eventually, real-time controls.

When considering virtualization solutions, it is important to evaluate the VMM characteristic regarding negligible impact on performance. The hypervisor controls all hardware resources (CPU, memory, I/O) and thus may impact the performance of any of them. For the CPU, one important metric is the time it takes to switch a core from running one virtual machine to another. This is sometimes called the context switch time, but may also be called the partition or domain switch time to differentiate it from a similar concept of operating systems switching between processes. An associated metric is the jitter, which is a measure of how much this switch time varies, thus affecting determinism and predictability.

Real-time designers are also interested in measuring the minimum time slice that can be scheduled, which constrains the maximum frequency of the CPU schedule, or put another way, the maximum number of virtual machines that can be executed in a given period. When measuring impact on memory, the footprint of the hypervisor kernel consists of a constant base portion along with an incremental portion for each added guest (virtual machine). The cumulative footprint then constrains the maximum number of virtual machines possible. For I/O, bandwidth and latency are the key measurements to make for each device of interest, although you could also make estimates based on some generic metrics such as the overall interrupt latency or the raw communication bandwidth.

Many hypervisors support two approaches to I/O: exclusive and shared. Exclusive I/O typically incurs a somewhat lower overhead, where the hypervisor provides a virtual machine with direct and sole access to a particular I/O device, often referred to as a "pass-through" device. Shared I/O entails a somewhat higher overhead, because the hypervisor must impose mechanisms to enforce a sharing scheme.

ASPECTS OF OPEN SOURCE

The term "open source" is used to describe software that is free, as in speech, but not necessarily free as in beer. Open-source software provides the freedom to modify and share source code under carefully developed licensing to guarantee that freedom is preserved. Some of the most commonly recognized open-source license agreements are the GNU General Public License (the active versions are GPLv2 and GPLv3), the GNU Lesser General Public License, the Apache license and the BSD license (in several variations).

Open source is not necessarily free of charge. Businesses built around open-source products typically use a different kind of revenue model than traditional software vendors, instead selling product support, accessories (like printed user manuals), training or custom design services. Red Hat is one of the best-known examples, building a billion-dollar business around the open-source Linux operating system.

MAPPING XEN TO THE NEW ZYNQ

The new Zynq UltraScale+ MPSoC from Xilinx offers a powerful platform for running the Xen hypervisor. This device provides a quad-core ARM® Cortex™-A53 with hardware virtualization extensions and 64-bit capability in the ARMv8 instruction set. Powerful hardware requires a rich ecosystem of software in order to take full advantage of its features and performance. While developing the new Zynq MPSoC, Xilinx surveyed key customers in a variety of industries, including aerospace and defense, health care, telecommunications and automotive. The message: Most customers expected to have hypervisor options for the new device, and half of them desired an open-source hypervisor. Xilinx chose Xen as the open-source hypervisor and chose DornerWorks to offer the support service for the new Xen Zynq.

The Xen hypervisor hosts guest operating systems within virtual machines, providing them with a virtualized view of the underlying machine. The guest OS and its applications then utilize the virtualized CPU, memory and I/O, while Xen manages how the virtualized resources are mapped to physical resources.

In Xen, each virtual machine is called a domain. In order to keep the hypervisor kernel as small as possible, Xen gives one domain special privileges. This system domain is called dom0. It starts up other guest domains (each called a domU), configures the schedule and memory mappings that the kernel enforces, and manages I/O access permissions. To provide a little more detail, let's consider several views of the hypervisor environment: the boot sequence, ARM exception levels, running schedule and resource management.

Starting from power-on, the boot sequence on the new Zynq MPSoC can be configured in a variety of ways, including variations on which processor (Cortex-A53 or Cortex-R5) starts first. Most use cases will keep the two processors quite independent, so the standard Xen Zynq hypervisor distribution will run only on the Cortex-A53. Figure 1 illustrates a typical boot sequence. If

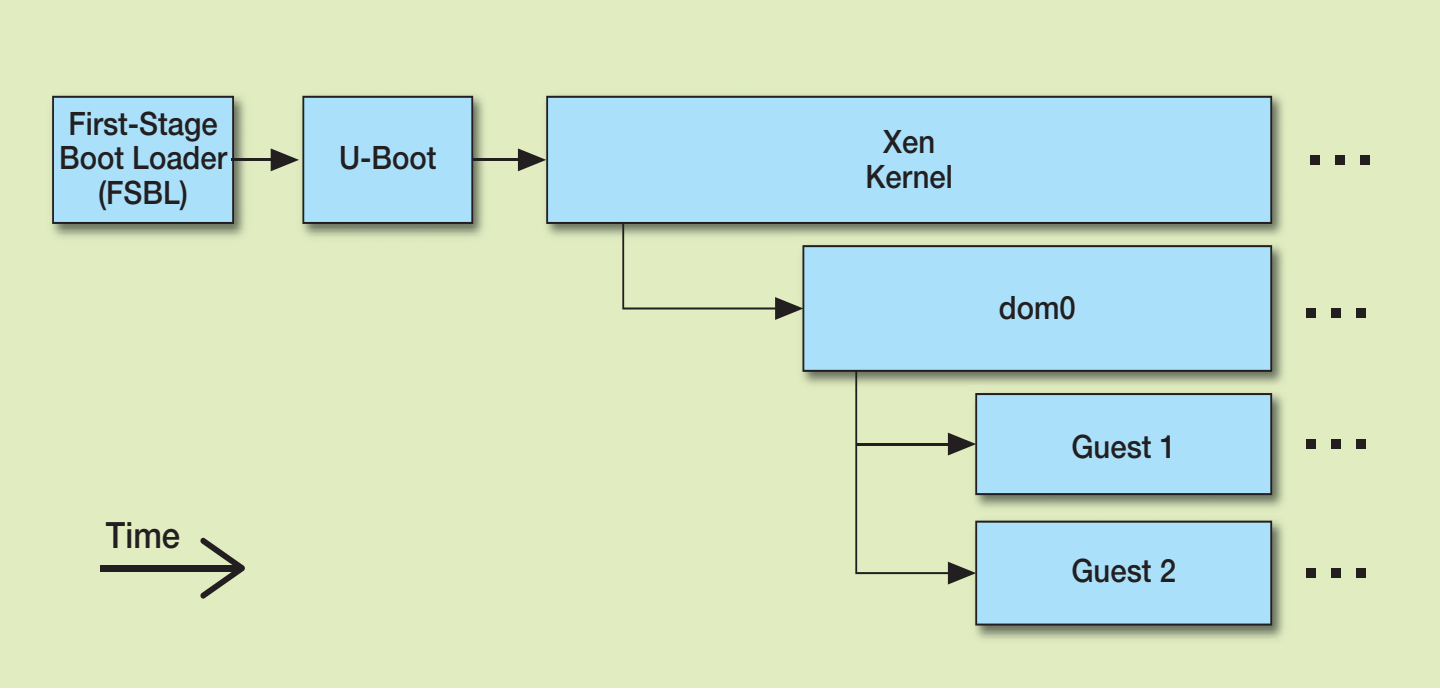


Figure 1 – Typical boot sequence shows stages until the guest OS is running.

Hypercalls are analogous to system calls that allow an application to invoke an operating system service, but in this case, they invoke hypervisor services. By default, dom0 can make any hypervisor call.

the Cortex-R5 were used to host an independent, nonvirtualized secure OS, this would typically boot first, from a simple first-stage boot loader (FSBL). Once the R5 had booted, it would then initiate the A53, starting with its own FSBL. A second-stage loader, such as U-Boot, would typically be used to provide broader booting functionality, perhaps including some integrity checking of the hypervisor kernel image.

At this stage, the Xen hypervisor kernel is invoked. The kernel initiation includes checking for a valid dom0. In turn, dom0 checks for valid images for the guest domains and then initiates and schedules them on one or more cores. In most cases, dom0 continues to run in order to monitor the system, provide management of shared resources and handle certain system faults. The hypervisor kernel runs during each domain context switch and is also invoked through hypercalls. Hypercalls are analogous to system calls that allow an application to invoke an operating system service, but in this case, they invoke hypervisor services. By default, dom0 can make any hypervisor call, while a domU is restricted to certain ones. However, developers can use the Xen module XSM-FLASK to implement finer-grained control of hypercall access.

The processor hardware enforces privileges within categories defined by the ARM exception-levels model. The Cortex-A53 uses the ARMv8 architecture, which defines four exception levels, as illustrated in Figure 2, with the highest privileges for the bottom level in the diagram, and decreasing as one moves upward. Complete access privileges are granted at exception level EL3, which is used for the ARM TrustZone monitor. Hypervisors run at EL2 to provide virtualization of guest domains. Within each hosted virtual machine, the hosted operating system runs at EL1. Finally, user applications run with the least privilege at EL0. When changing to an exception level with lower privilege, the virtualized machine registers must be the same width or narrower. That means you can have a 64-bit hypervisor and a 32-bit guest, but not vice versa. Xen Zynq uses the AArch64 execution mode of the ARMv8 architecture, and thus supports 64-bit or 32-bit guests.

The privileged domain, dom0, establishes the schedule, thus determining when domains run and on which core or cores. The hypervisor kernel then executes the configured schedule. To achieve certain types of determinism, you might configure a schedule where a guest domain has sole access to the machine during its time slot. Figure 3 provides an example, where Guest 1 runs on several cores (along with dom0) in a single time slot, while Guests 2 and 3 do not need this restriction, so they can be scheduled in a more mix-and-match load-balancing scheme during other time slots.

The hypervisor manages all resources of the machine. The CPU cores are managed primarily by time-sharing, as discussed above. The hypervisor uses hardware timers to enforce the schedule. Memory is shared not by dividing time, but by dividing space, allocating a portion of the memory to each guest domain. The hypervisor uses the hardware memory management unit (MMU) to enforce the memory layout. Management of I/O varies widely, depending on the type of device. Some I/O devices are mapped directly to the Cortex-A53, while others must be configured to connect through the FPGA programmable fabric.

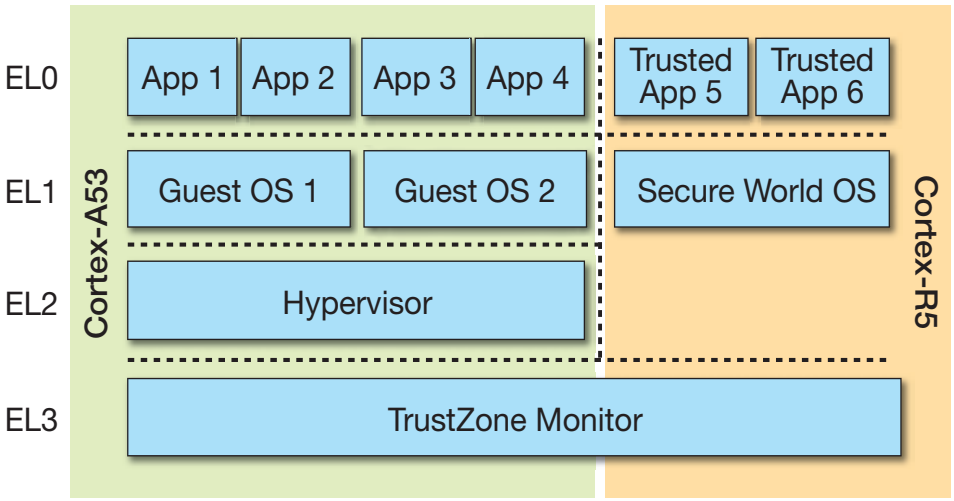


Figure 2 – This diagram of ARM Exception Levels shows the hypervisor mapped to EL2.

Guest access to I/O devices is configured and managed by dom0, with appropriate hypercalls to the Xen kernel to establish memory mappings to the devices. Dom0 can grant a guest domain access to specific I/O devices as needed, or it may manage shared I/O itself, acting as the gateway to enforce a sharing mechanism. Interdomain communication in Xen (including I/O) typically uses Xen event channels for notifications and shared memory for passing data. Shared I/O device drivers in Xen use a split-driver model, where the top half in the guest domains provides the API to the guest OS and the functionality to pass data back and forth to dom0. The bottom half of the driver inside dom0 then performs the actual I/O operations to the device.

CREATING SUPPORT FOR XEN ZYNQ

As Xilinx sought customer feedback about the anticipated next-generation Zynq SoC device, it heard that many customers expected strong hypervisor support and half of them wanted an open-source option. This support had to be more than a simple help-desk-style service. Rather, the support options would need to be more extensive, to provide help designing embedded systems that balance demanding needs (such as high bandwidth, low latency, low power, high reliability) and that connect to a wide variety of system devices in an embedded environment. Xilinx selected DornerWorks because of our expertise with the Xen hypervisor, our embedded-engineering design experience and our role as a premier member of the Xilinx Alliance Program, providing additional options for customers that also seek support for the FPGA design portions of their systems.

DornerWorks collaborated with Xilinx on finishing the port of Xen to the

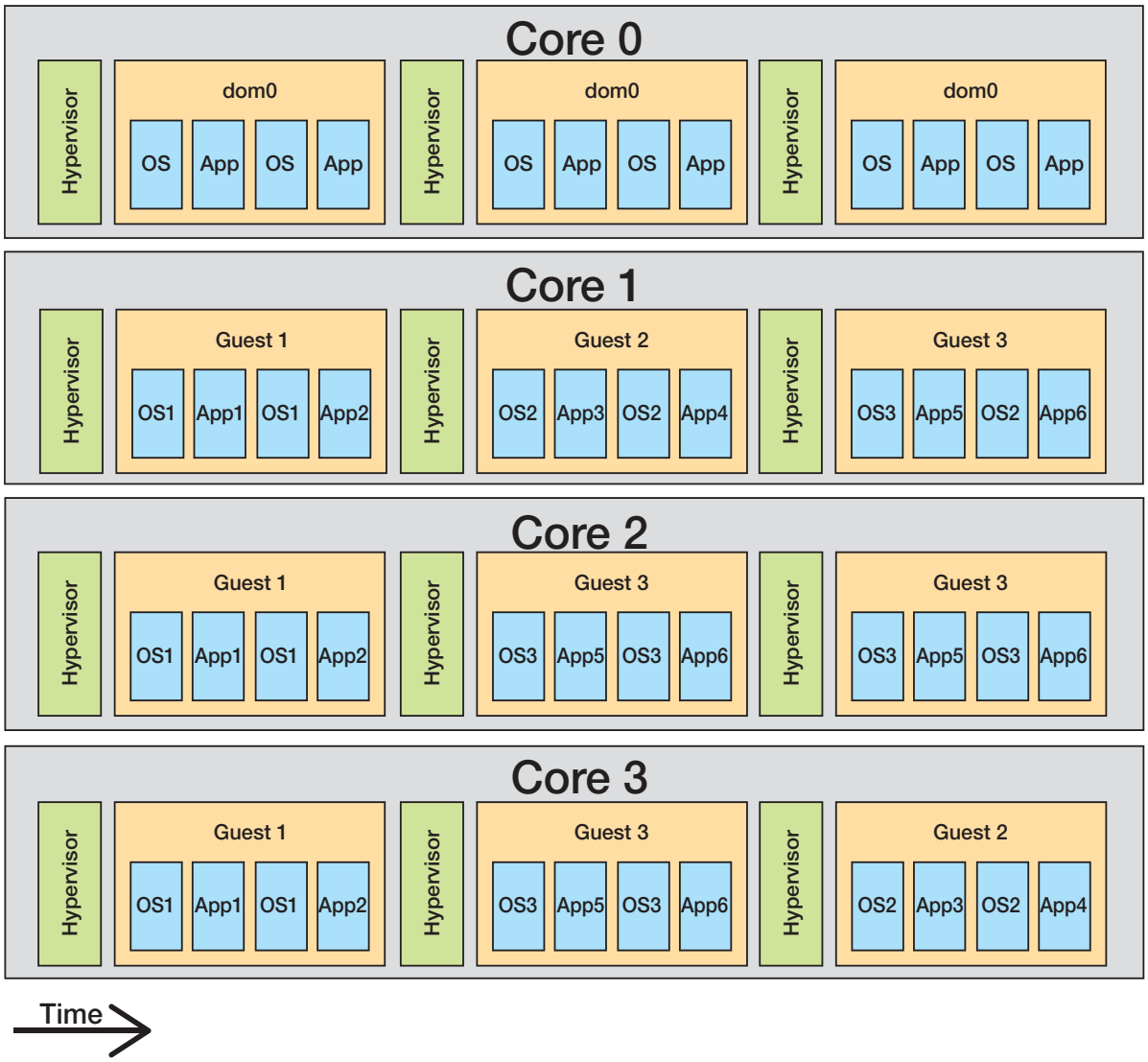


Figure 3 – Multicore scheduling places Guest 1 in an exclusive time slot and mixes Guests 2 and 3.

new Zynq MPSoC and then confirmed correctness by verification and validation testing. Our testing needed to cover not only the Xen hypervisor kernel running correctly on the hardware, but also the dom0 privileged domain (running Linux) and guest domains with a variety of supported guest operating systems. We named this package of software the Xen Zynq Distribution.

We faced the additional challenge of testing before we had the actual hardware. Our stand-in model of the hardware was the QEMU open-source machine emulator software running on an x86 developer system for individual debugging and testing or on our team's build server for continuous integration testing. Additionally, we developed against an emulation board named Remus (not to be confused with the Xen live migration tool of the same name), which uses six Xilinx Virtex®-7 FPGAs to emulate the Zynq MPSoC.

Figure 4 shows our continuous-integration approach, centered on a build-and-test server. On a periodic basis, the server queries the repository of source code. If it detects any changes, the server performs an incremental build on the dependent portions of the build image. It then loads the images necessary for each test onto each device of our target farm and invokes the test script. In some test cases, external stimuli are applied to the targets. The test server gathers results, collates them and presents a summary dashboard that provides a view of the overall health of the test suite or pinpoints where there are issues to resolve.

DornerWorks has also developed the infrastructure to provide comprehensive support for Xilinx customers using the Xen hypervisor on the new Zynq MPSoC. The base level of support is driven by open-source community activism, allowing users to compare notes and share information. DornerWorks will host forums and gather issues from the community. We use Jira as our tracking tool for issues that Xilinx uncovers, for internally detected issues and for customer-identified issues (from the community or from paid subscribers). In order to sustain our work on Xen, we also offer paid subscription and custom design support services, which provide the business-critical support by contract that many customers demand in order to reduce their business risk and ensure timely response to their needs. You can find more details about the support options at *http:// http://xen.world*.

TEST-DRIVE XEN YOURSELF

While you are waiting for the new Zynq MPSoC to ship early next year, you can already start investigating Xen. Xen runs on an ordinary x86 PC, either natively as a Type-1 hypervisor or hosted inside VirtualBox on top of Windows for development purposes. To try out embedded Xen, you'll need emulated or actual ARM hardware. Choose an ARM processor that has the virtualization extensions—ideally the

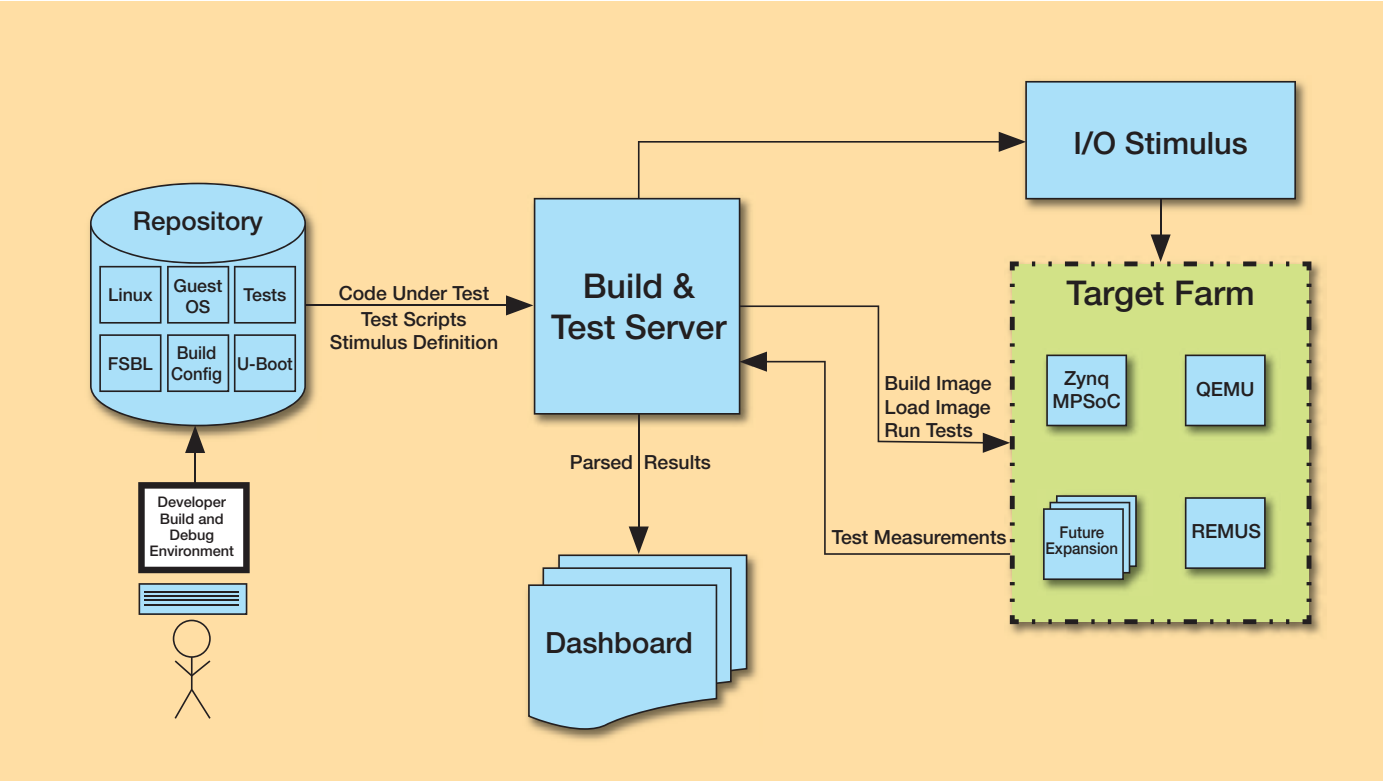


Figure 4 – Continuous integration automates build and test of Xen Zynq.

Cortex-A53, but others, such as the Cortex-A15, can also provide a fairly representative environment. Figure 5 depicts the workflow for building a complete hypervisor-based system for an embedded target. You can find Xen at *http://www.xenproject.org/*, along with information on building a Linux image to serve as dom0 and building a variety of guest OS images.

DornerWorks has published the Xen Zynq Distribution for the new Zynq MPSoC, ready for download from our website: *http://dornerworks.com/services/XilinxXen*. Simply add guest OS images and you have your own embedded, virtualized system.

With Xen on the new Zynq MPSoC, you've got cloud computing in the palm of your hand.

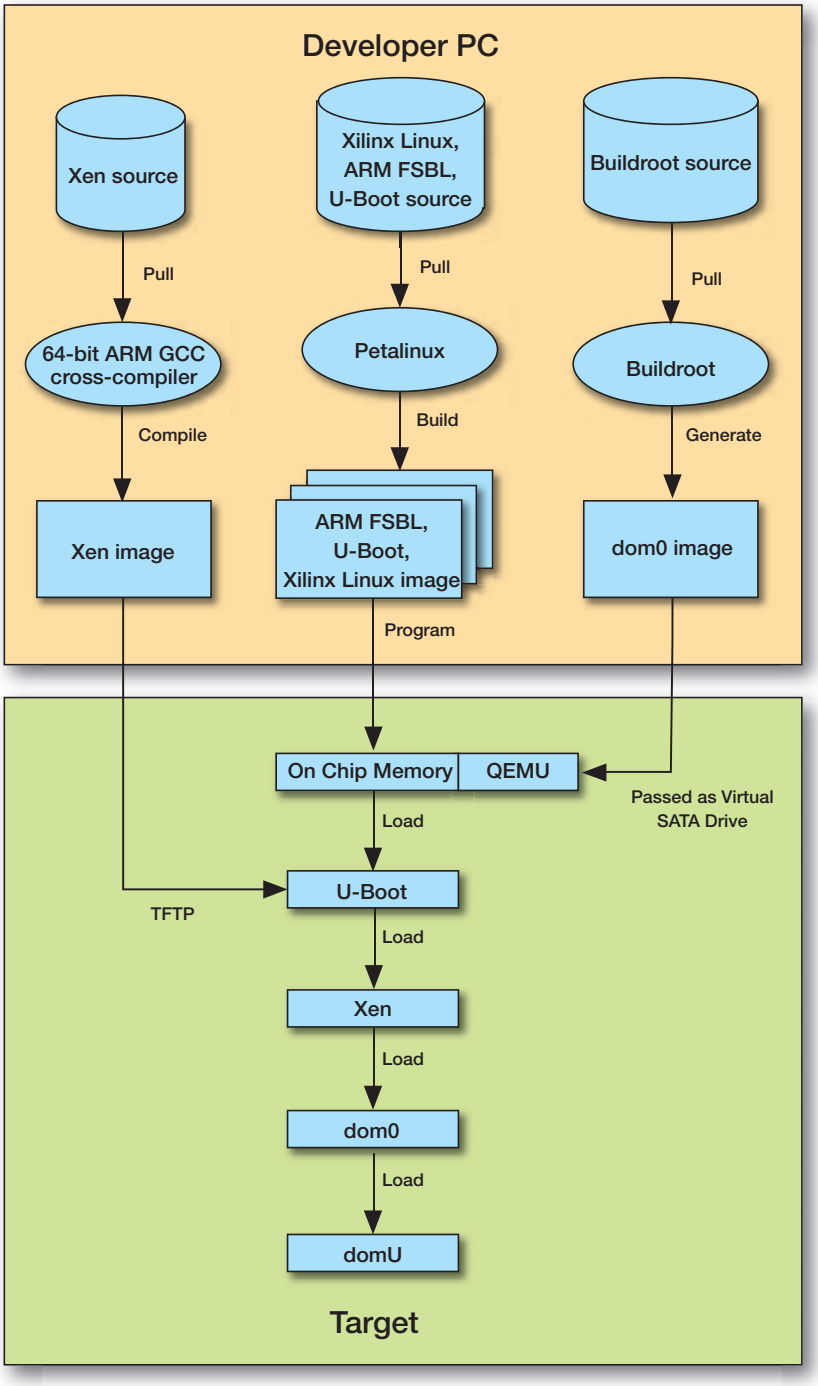


Figure 5 – The Xen development work flow

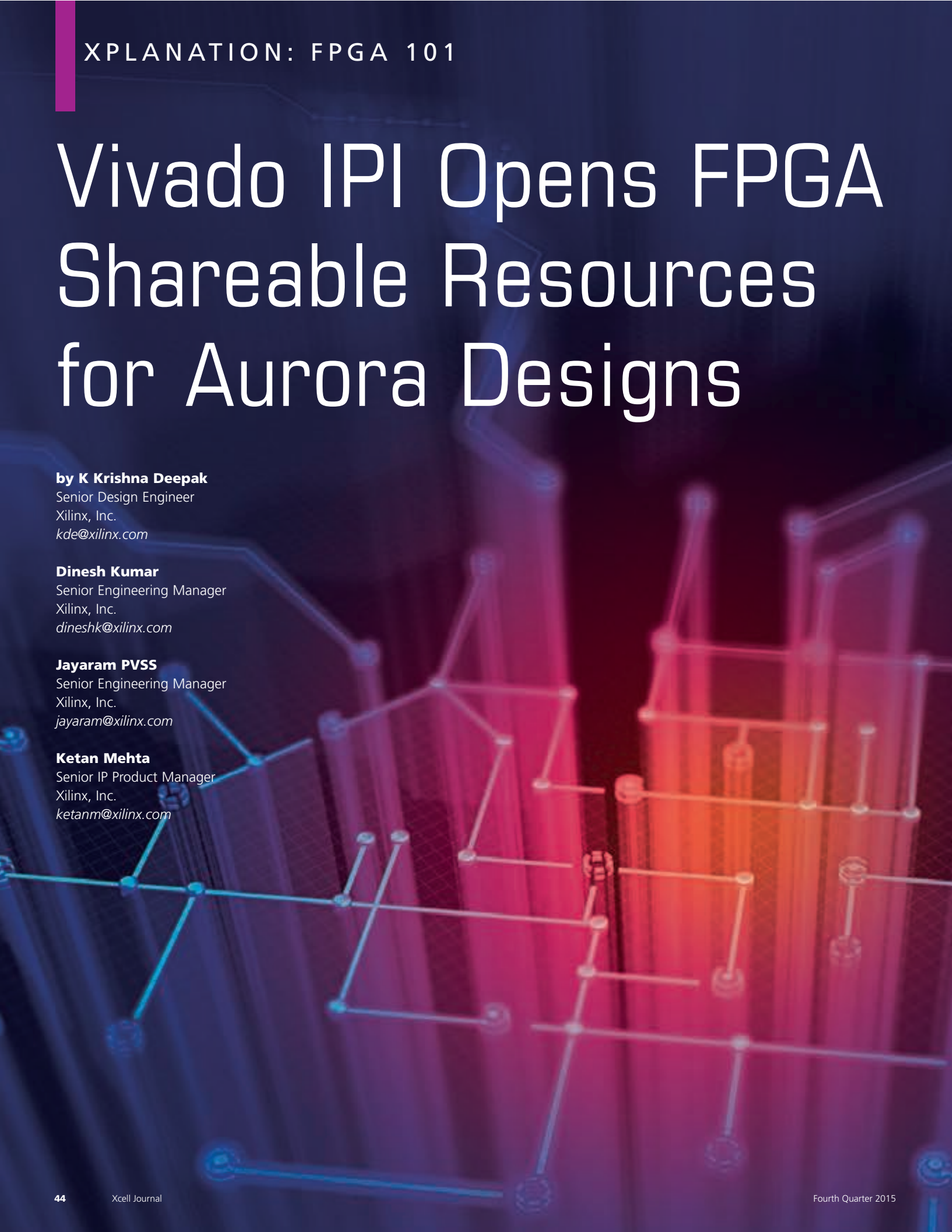
Vivado IPI Opens FPGA Shareable Resources for Aurora Designs

by K Krishna Deepak
Senior Design Engineer
Xilinx, Inc.
kde@xilinx.com

Dinesh Kumar
Senior Engineering Manager
Xilinx, Inc.
dineshk@xilinx.com

Jayaram PVSS
Senior Engineering Manager
Xilinx, Inc.
jayaram@xilinx.com

Ketan Mehta
Senior IP Product Manager
Xilinx, Inc.
ketanm@xilinx.com



Xilinx's IP Integrator tool will help you improve design-entry productivity and resource optimization in multicore Aurora designs.

One of the major challenges that customers encounter when using multiple instances of intellectual property (IP) in a big design that must fit into a single FPGA is how to share the resources effectively across the system. The shared-logic feature of Xilinx®'s Aurora serial communications core provides users with shared resources across multiple instances. The IP Integrator tool within the Vivado® Design Suite is the key to making the most out of these shared resources.

The electronics industry is rapidly shifting toward high-speed serial connectivity solutions while moving away from parallel communication standards. Industry-standard serial protocols have fixed line rates and defined lane widths, sometimes underutilizing the capabilities of gigabit serial transceivers.

Aurora, a high-speed serial communication protocol from Xilinx, has been very popular in the industry and is typically used in applications where competing industry protocols are either too complex or too resource-intensive to implement. By delivering a low-cost, high-data-rate, scalable IP solution, Aurora provides a flexible means to build a high-speed serial data channel.

High-performance systems and applications that need to be scaled, both in line rate and channel width, are looking to Aurora as a solution. Aurora is also establishing a presence in ASIC designs as well as in systems built of multiple FPGAs

IPI visualizes cores as top-level blocks. Connections across standard interface ports are now more intuitive, intelligent and in some cases automatic.

with backplanes transporting gigabits of data. Aurora's simple framing structure coupled with protocol-extending flow control capability can be used to encapsulate data from existing protocols. Its electrical requirements are compatible with commodity equipment. Xilinx delivers Aurora 64b66b and Aurora 8b10b cores as part of the Vivado Design Suite's IP Catalog.

The Vivado IP Integrator (IPI) is a key tool for resource optimization in complex multicore systems. In this regard, IPI will help you make the best use of the shareable resources in the Aurora 64b66b and Aurora 8b10b cores, especially the "shared-logic" feature. For convenience, let's focus on the Aurora 64b66b IP, with the understanding that similar techniques are applicable to the Aurora 8b10b core as well.

AURORA'S SHAREABLE RESOURCES AT A GLANCE

Figure 1 is the representative block diagram of the Aurora 64b66b core. Highlighted are the clocking resources such as mixed-mode clock manager (MMCM), BUFG and IBUFDS, along with gigabit transceiver (GT) resources such as GT common and GT channel, illustrated as GT1 and GT2 for a

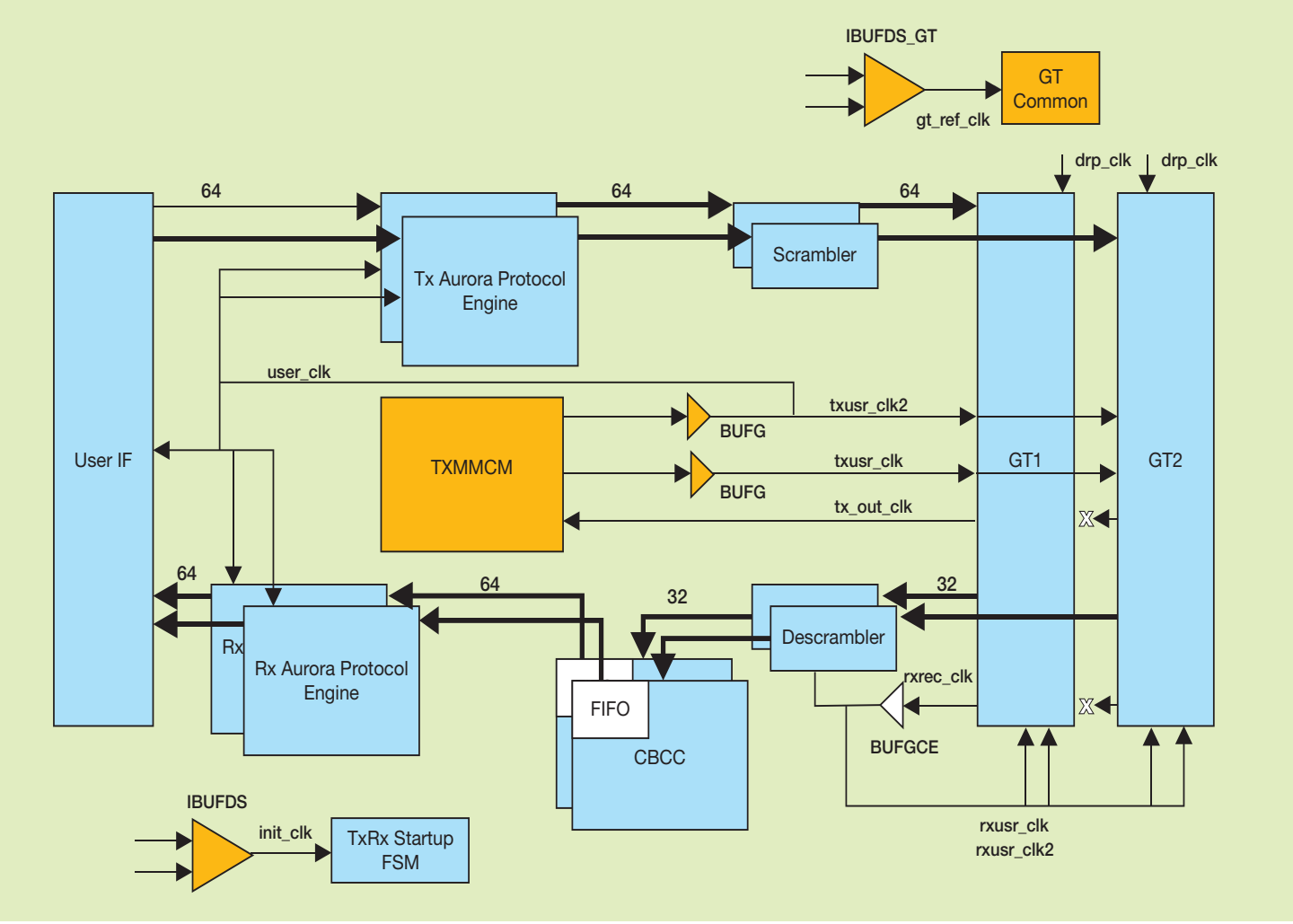


Figure1 – Shareable resources, highlighted in orange, in the Aurora 64b66b core

two-lane design based on a Xilinx 7 series device.

For a typical16-lane Aurora 64b66b core, the clocking and GT resource requirements, as used in a Kintex®-7 FPGA KC705 Evaluation Kit, are tabulated in Table 1.

Clocking and GT resources in an FPGA are specific to the device and the package selected. Often, multiple IP cores will demand resources for use at the system level. Hence, it becomes imperative to optimize the utilization of these precious resources to reduce the system cost as well as the power consumption.

The IPI tool visualizes cores as top-level blocks, and connections across standard interface ports are now more intuitive, intelligent and in some cases automatic. Appropriate design rule checks are built into the tool and around the IP to ensure the wrong connections are highlighted so the designer will spot them at the time of design entry itself. Top-level wrapper files and inference of appropriate pin-level I/O requirements are automatic, making the tool productive for system designers. If you have designed custom sub-blocks, you could consider packaging your design by following Xilinx application note 1168, "Packaging Custom AXI IP for Vivado IP Integrator" (XAPP1168), and use the sub-blocks in IPI.

16-Lane Aurora Design		
Resource	Availability in Device	Used by Aurora
MMCME2_ADV	10	1
IBUFDS_GTE2	8	2
GTE2_COMMON	4	4
GTXE2_CHANNEL	16	16

Table 1 – Clocking and GT resource utilization on a Kintex-7 FPGA KC705 Evaluation Kit

AURORA RESOURCE SHARING

As part of the shared-logic feature supported across multiple GT-based Xilinx cores, the Aurora core can be configured either as "Shared logic in core (Master)" or "Shared logic in example design (Slave)." A combination of the two configurations makes it possible to share the clocking and GT resources across master and slave when instantiated at the system level.

For applications in which the shared-logic feature is to be used, handcrafting the connections across multiple pieces of IP could create errors and will increase the overall design-entry time. Tool-assisted design entry is the way to solve this problem, and Xilinx's IP Integrator has elegantly addressed it.

Not only does the shared-logic feature of Aurora provide users with shared resources across multiple instances, it also allows an out-of-the-box experience for utilizing the GT channels in the same GT quad without the pain of editing the GT common, PLL, clocking and related modules. The only constraint is that the line rates of the "shared" cores should be the same (harmonics are also allowed if you can take a penalty on the clocking resources).

A typical shared-logic design would include one master and one or more slave instances within a quad. Unlike most other communication IP, Aurora doesn't limit itself to a single-quad sharing. The shared-logic definition for the Aurora core can be extended to any number of supported lanes.

Here are some examples that showcase applications based on Aurora's shared-logic feature.

MULTIPLE SINGLE-LANE DESIGNS

Multiple single-lane designs in a single FPGA differ from multilane designs in that they require channel bonding. Intuitively, it seems clear that resources needed for multiple single-lane designs linearly add up at the system level. Let us consider different scenarios and examine how the shared-logic feature helps in each case.

We will start with a design that has four single lanes. You could build this kind of design straightaway by instantiating four single-lane Aurora cores. If we actually ran through the implementation, each Aurora design would have one instance of GT common; therefore, the placement and resource utilization of this design would be spread across four GT quads. This might not always be a feasible solution because it is resource-intensive. For a better placement and optimized solution in terms of power and resources, the four GTs selected should be from the same GT quad.

Without the shared-logic feature, handcrafting the generated design to suit this requirement is a focused effort. To use the shared-logic feature effectively, you will need to generate one Aurora core in master mode and the other three Aurora cores in slave mode, as shown in Figure 2. Additional system-level considerations need to be taken into account, such as resetting the cores because the master core controls the clocking to the slave cores. This configuration and resource optimization are possible out of the box only if the Aurora cores are configured with the same line rate. Table 2 quantifies the benefits of using the shared-logic feature for four single-lane designs in a system.

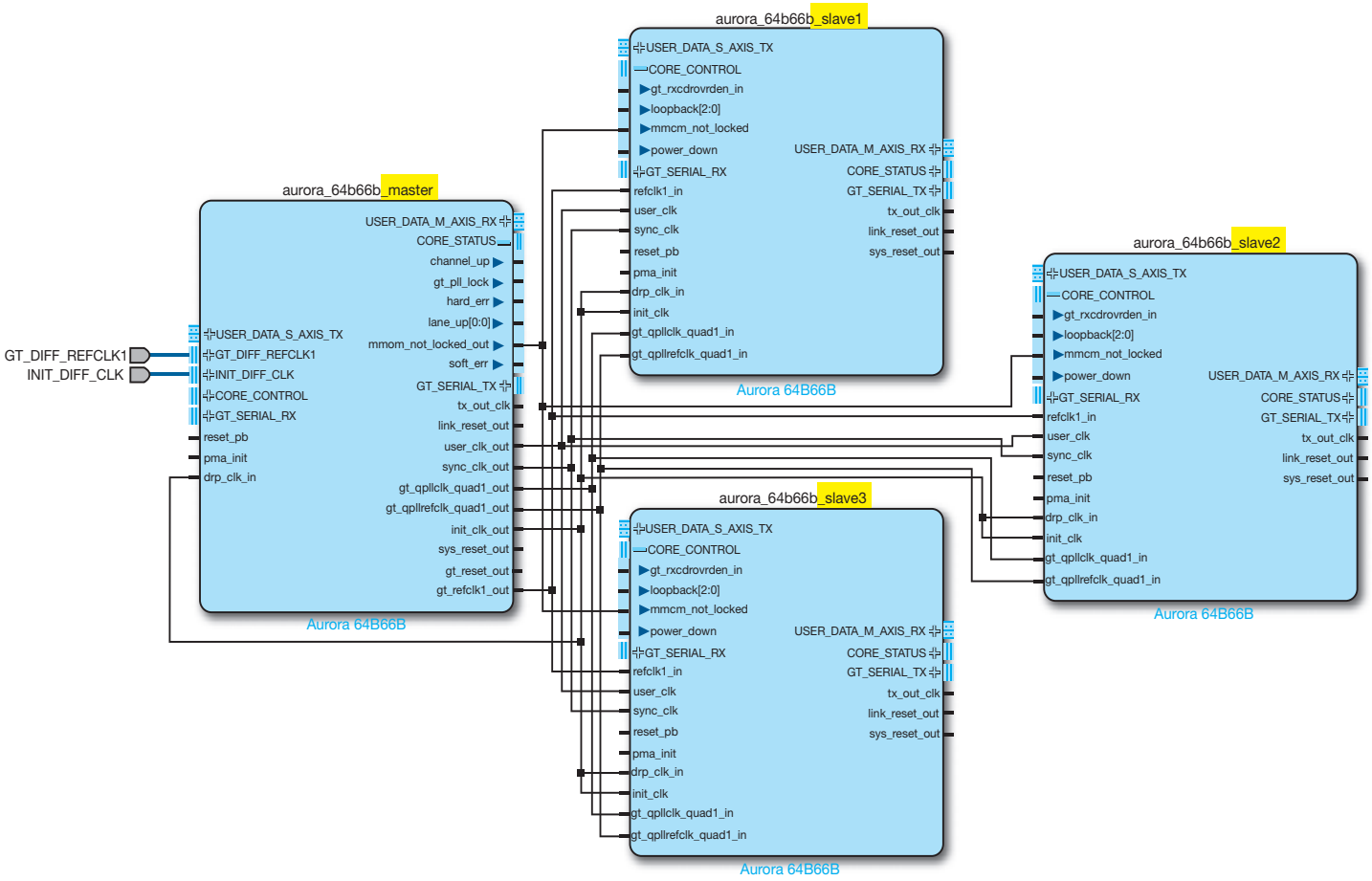


Figure 2 – Shared-logic design using one master Aurora core (left) and three slaves

DESIGNS OCCUPYING 12 GT CHANNELS

For a 7 series FPGA, the GT requirement based on north-south clocking is that a single reference clock source can serve up to maximum of 12 GT channels if it is chosen in a middle quad.

Let us consider the use case where the requirement is to have 12 single-lane designs utilizing as few clocking resources as possible. You can save on clocking resources if you try to extend the one-master-plus-three-slave configurations as shown in Figure 2. But if this 1+3 configuration is extended for the three quads, the design will need a total of six differential-clocking resources. However, more savings are possible if you select two of the master designs such that they accept a single-ended INIT_CLK and GT reference clock. This way, potentially we could reduce the differential-clock input requirement from six to two for this system, saving IBUFDS/IBUFDS_GTE2 resource requirements (see Table 3). IBUFDS_GTE2 resource reduction in the design actually would also mean a reduction in external

Design with Four Single Lanes		
Resource	**Without Shared Logic**	**With Shared Logic**
MMCME2_ADV	4	1
IBUFDS_GTE2	4	1
GTE2_COMMON	4	1
GTXE2_CHANNEL	4	4

Table 2 – Resource usage benefits with shared logic for designs with four single lanes

12-Single-Lane Designs			
Resource	**Without Shared Logic**	**With Shared Logic (default)**	**With Shared Logic (using single-ended master input clocks)**
MMCME2_ADV	12	3	3
IBUFDS_GTE2	12	3	1
GTE2_COMMON	12	3	3
GTXE2_CHANNEL	12	12	12

Table 3 – Resource benefit with shared-logic feature for designs with 12 single lanes

clocking resources as well as design pinouts. Similar optimization can be imagined for the MMCM as well.

THREE-BY-FOUR-LANE DESIGNS

If there is a requirement of having three four-lane designs, without the shared-logic feature you might end up creating three four-lane Aurora cores in master mode and then handcrafting the generated design for optimal utilization of clocking resources. What if you could achieve the same result out of the box? You can do just that by customizing one master core and two slave cores as shown in Figure 3.

Moving up in size to large (16 and over) single-lane Aurora designs, the need for shared logic becomes even more acute. Sometimes the requirement could be as large as having 48 single-lane independent duplex links. The number of allowable Aurora single-lane links is limited only by the number of GT resources available in a chosen device. In such use cases, it is difficult to realize this system design without making effective use of the shared-logic feature.

This design would spread over 12 quads; hence, there could be a requirement of 2*12 differential-clocking resources, which could be a daunting task from the point of view of board design. By using the techniques mentioned in the 12-single-lane design use case, you could reduce the differential clocks and MMCM requirements of the overall system (see Table 5).

ASYMMETRIC LANES AND OTHER CUSTOM OPTIMIZATIONS

In applications like video projectors, mainstream data will flow in one direction with high throughput while a back channel with lower throughput is used to transmit auxiliary or control information. In such applications,

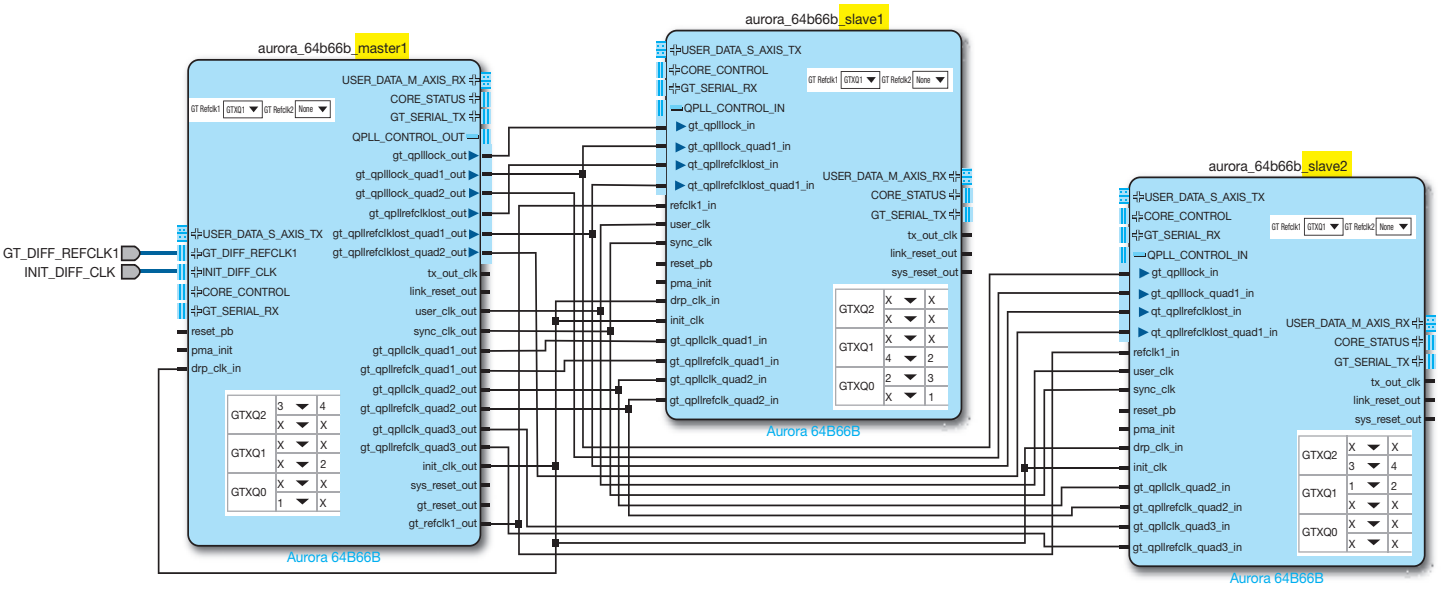


Figure 3 – Single-master, two-slave configuration for a four-lane Aurora design over three consecutive quads

System designers implementing multiple Aurora cores need to be aware of the device-specific clocking requirements and limitations. Conserving clocking resources will make the system more cost-effective.

having a full-blown duplex link would mean lesser usage of bandwidth and essentially result in lower ROI of the system design. An ideal solution for this kind of problem could be, as shown in Figure 4, to have an asymmetric link width with optimized GT resource utilization, where the number of lanes in one direction of data flow (higher throughput) could be higher than that of the other direction (lower throughput).

With the current available data flow modes (simplex/duplex) in the Aurora cores, it is possible only to configure the cores with an equal number of TX and RX lanes. To have a different number of lanes in the two directions, you need to generate two Aurora simplex cores for each direction. One way of building these kinds of asymmetric-lane designs on 7 series FPGAs is discussed in Xilinx application note 1227, "Asymmetric Lane Design with Aurora 64B/66B IP Core" (XAPP1227).

Another useful design strategy is BUFG resource optimization. Often, system designers implementing multiple Aurora cores that operate at the same or different line rates need to be aware of the device-specific clocking requirements and limitations. Implementing numerous Aurora links demands generation of clocks for the respective links. Conserving clocking resources will make the system more cost-effective. If the system design has multiple blocks and if there is a clocking resource (BUFG) crunch, you could consider replacing the BUFG with BUFR/BUFH. It is recommended, though, that you drive both TX path user clocks of the GT cores with the same buffer type.

The 7 series Aurora core requires an additional dynamic reconfiguration port (DRP) clock input that otherwise would need to use one BUFG. If Aurora's free-running clock frequency is chosen in the allowable range of the DRP clock, then the available output free-running clock from Aurora could be reused and connected back to the DRP clock. As a result, you could save on the number of BUFGs in the generated designs.

When selecting the line rates across multiple Aurora designs, keep in mind

Optimized Lane Selection for 3 x 4-Single-Lane Designs		
GTQ2	Master1_3	Master1_4
	Slave2_3	Slave2_4
GTQ1	Slave2_1	Slave2_2
	Slave1_4	Master1_2
GTQ0	Slave1_3	Slave1_2
	Master1_1	Slave1_1

Table 4 – Optimized lane selection for three four-lane designs

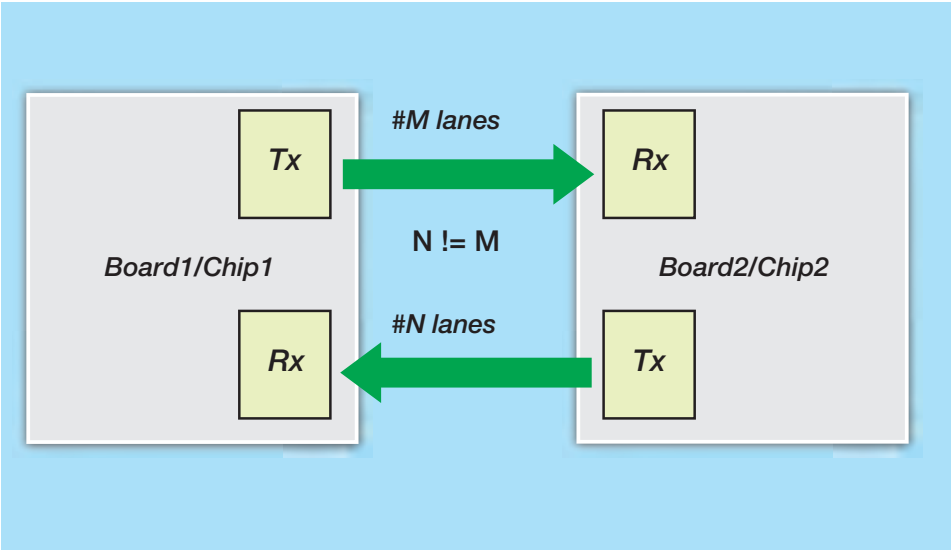


Figure 4 – Asymmetric data transfer across links made possible with Aurora

that you can share clocking resources if line rates are integral multiples for easier clock derivation and sharing across the links. If the shared-logic feature is to be extended to harmonic line rates, then by designing in few extra clock dividers, you could generate the required input frequencies for the slave Aurora cores.

FUTURE POSSIBILITIES

Aurora's flexibility opens up possibilities of creating a variety of system configurations and applications. Aided by a powerful tool like Xilinx's Vivado IP Integrator, design-entry productivity and system-level resource sharing are speeding up innovation in the All Programmable application space. With the Xilinx UltraScale™ architecture, there are devices with many more GT channels with enhanced GT line rate support and hence the possibilities and effective resource utilization are even greater.

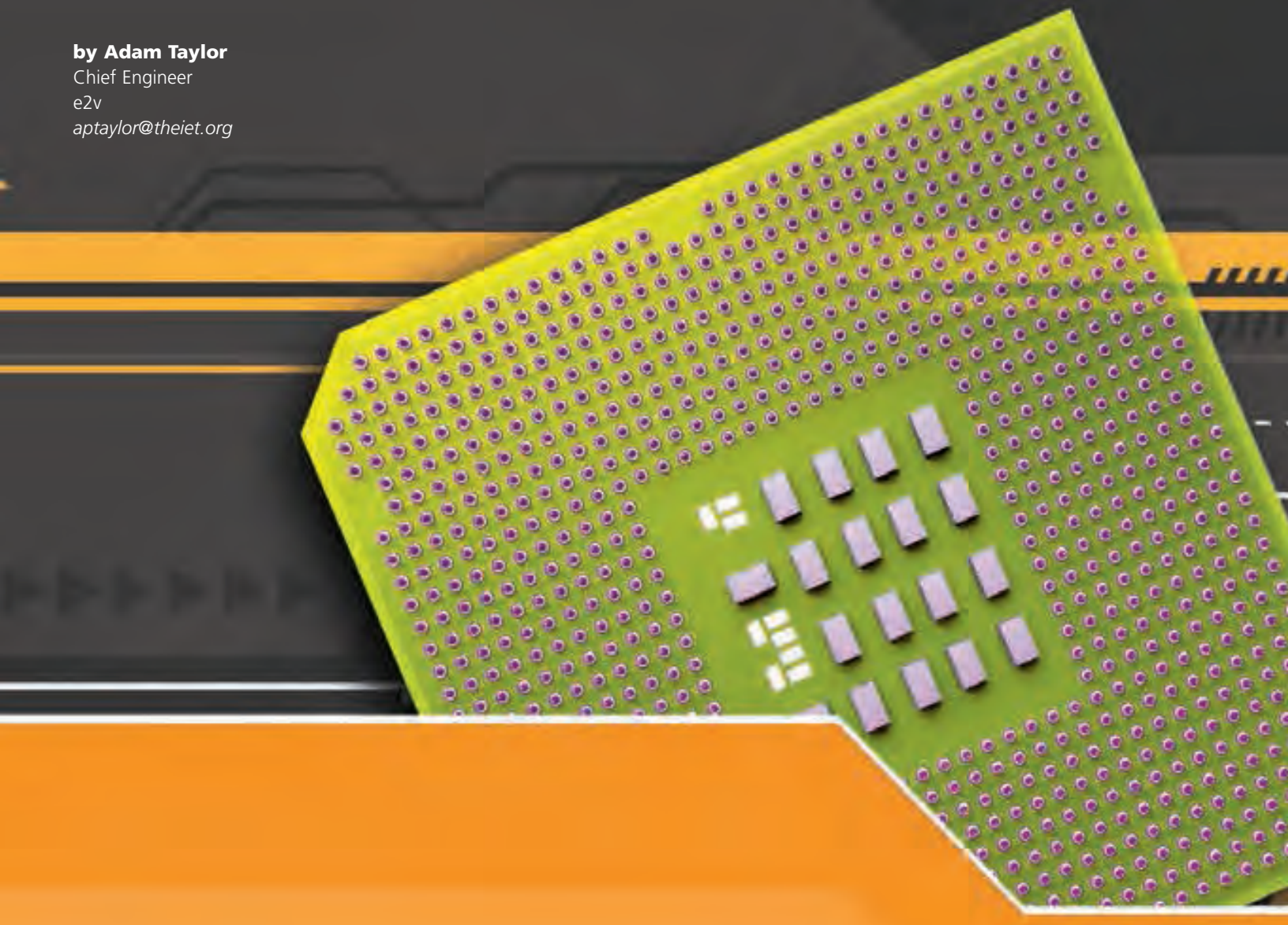
48-Single-Lane Designs		
Resource	Without Shared Logic	With Shared Logic
MMCME2_ADV	48	4
IBUFDS_GTE2	48	4
GTE2_COMMON	48	12
GTXE2_CHANNEL	48	48

Table 5 – Resource benefit with shared-logic feature for 48-single-lane designs

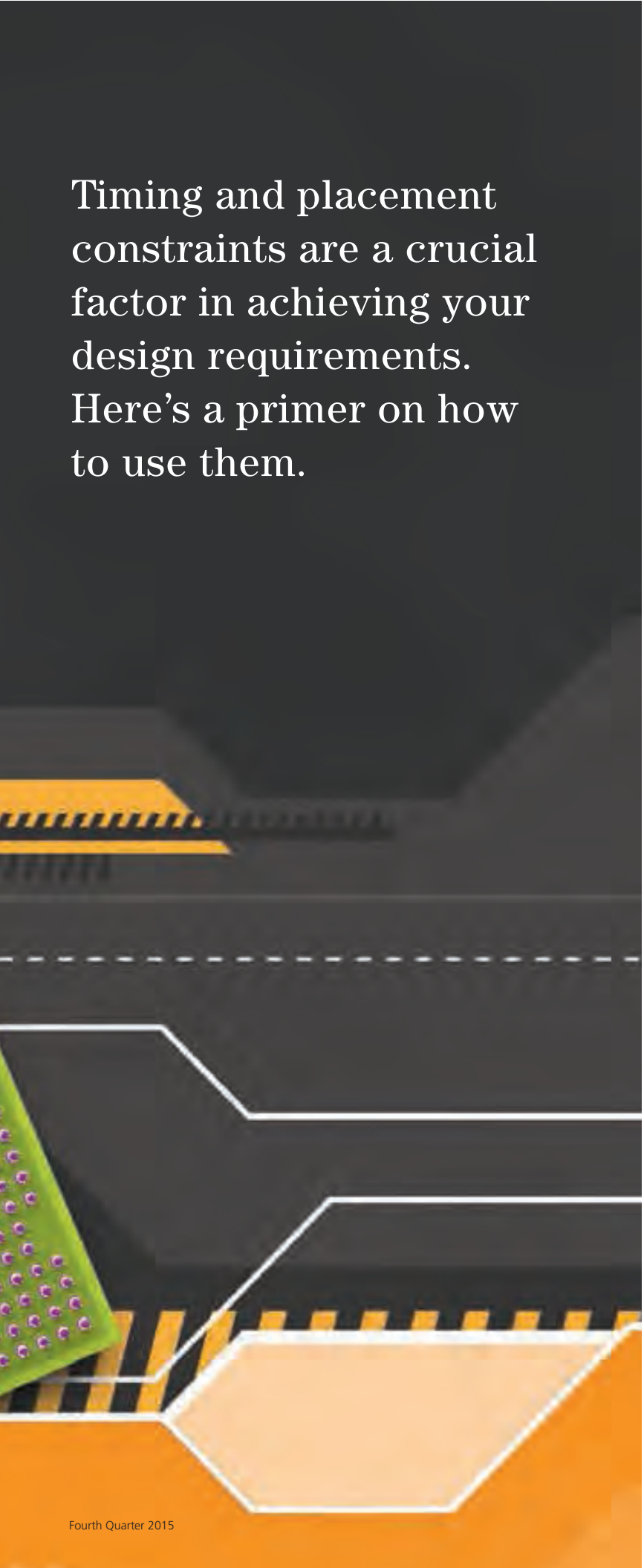
To evaluate the Aurora cores, check out the IP Catalog, IPI and Aurora product Web page at *http://www.xilinx.com/products/design_resources/conn_central/grouping/aurora.htm*.

Making XDC Timing Constraints Work for You

by Adam Taylor
Chief Engineer
e2v
aptaylor@theiet.org



Timing and placement constraints are a crucial factor in achieving your design requirements. Here's a primer on how to use them.



Completing the RTL design is one part of getting your FPGA design production-ready. The next challenge is to ensure the design meets its timing and performance requirements in the silicon. To do this, you will often need to define both timing and placement constraints.

Let's take a look at how to create and use both of these types of constraints when designing systems around Xilinx® FPGAs and SoCs.

TIMING CONSTRAINTS

At their most basic level, timing constraints define the operating frequency of your system's clock or clocks. However, more advanced constraints establish the relationships between clock paths. Engineers include these types of constraints to determine if it will be necessary to analyze the path or—if there is no valid timing relationship between those clock paths—discount it.

By default, Xilinx's Vivado® Design Suite will analyze all relationships. However, not all clocks within a design will have a timing relationship that can be accurately analyzed. One example is clocks that are asynchronous, since it's not possible to accurately determine their phase, as shown in Figure 1.

You can manage the relationships between clock paths using a constraints file and declaring clock groups. When a clock group is declared, the Vivado tools perform no timing analysis in either direction between the clocks defined within it.

To aid in the generation of timing constraints, the Vivado tools define clocks as being within one of three categories: synchronous, asynchronous or unexpandable.

- Synchronous clocks have a predictable timing/phase relationship. This is normally the case for a primary clock and its generated clocks, as they share a common root and will have a common period.
- Asynchronous clocks have no predictable timing/phase relationship between them. This is normally the case for different primary (and their generated) clocks. Asynchronous clocks will have different roots.
- Two clocks are unexpandable if, over 1,000 cycles, a common period cannot be determined. If this is the case, then the worst-case setup relationship over these 1,000 cycles will be used. However, there is no guarantee it is the worst case in reality.

To determine which type of clock you are dealing with, use the clock report that Vivado produces. This report will aid you in identifying asynchronous and unexpandable clocks.

Declaring a multicycle path results in a more appropriate and less restrictive timing analysis, allowing the timing engine to focus upon other, more critical paths.

With these clocks identified, you can now use the "set clock group" constraint to disable timing analysis between them. The Vivado suite uses Xilinx Design Constraints (XDC), which are constraints based upon Synopsys Design Constraints (SDC), a widely used Tcl-based constraints format. With the XDC constraints, you can use the command below to define a clock group:

```
set_clock_groups –name –logically_exclusive –physically_exclusive –asynchronous –group
```

The –name is the name given to the group. The –group option is the place where you can define the members of the group—that is, the clocks that have no timing relationship. The logically and physically exclusive options are used when you have multiple clock sources from which to select in order to drive a clock tree, including BUFGMUX and BUFGCTL. Therefore, the clocks cannot be present upon the clock tree at the same time. As such, we do not want Vivado to analyze the relationship between these clocks as they are mutually exclusive. Finally, the –asynchronous constraint is used to define asynchronous clock paths.

The final aspect of establishing the timing relationship is to take into account the nonideal relationship of the clocks, in particular jitter. You need to consider jitter in two forms: input and system jitter. Input jitter is present upon the primary clock inputs and is the difference between when the transition occurs against when it should have occurred under ideal conditions. System jitter results from noise existing within the design.

You can use the set_input_jitter constraint to define the jitter for each primary input clock. Meanwhile, the system jitter is set for the whole design (that's all the clocks) using the set_system_jitter constraint.

TIMING EXCEPTIONS

You must also focus upon what happens within a defined clock group when you have an exception. But what is an exception?

One common example of a timing exception would be a result being captured only every other clock cycle. Another would be transferring data from a slow to a faster clock (or vice versa) where both clocks are synchronous. In fact, both of these situations are examples of a timing exception commonly referred to as a multicycle path, as shown in Figure 2.

Declaring a multicycle path for these paths results in a more appropriate and less restrictive timing analysis, allowing the timing engine to focus upon other, more critical paths. The upshot is to increase the quality of results.

Within your XDC file, you can declare a multicycle path using the following XDC command:

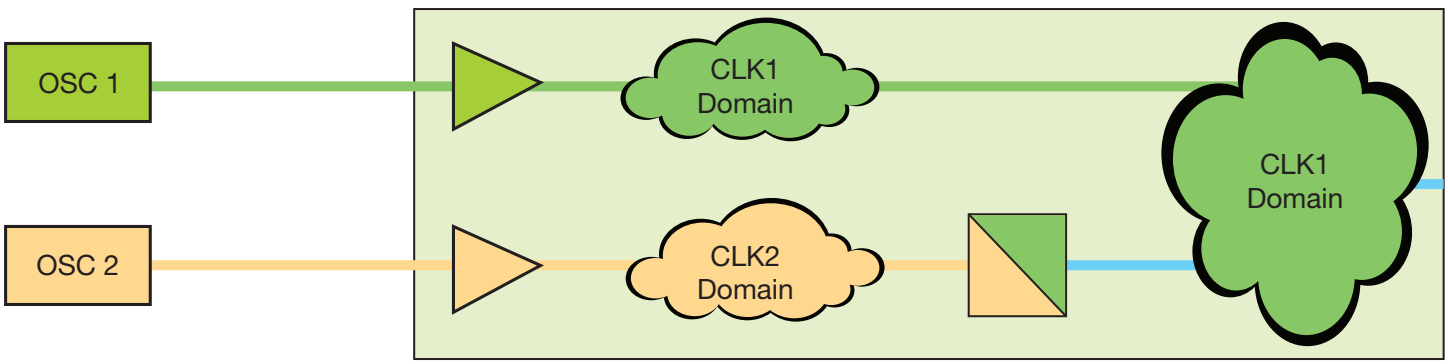


Figure 1 – Domains CLK1 and CLK2 are asynchronous to each other.

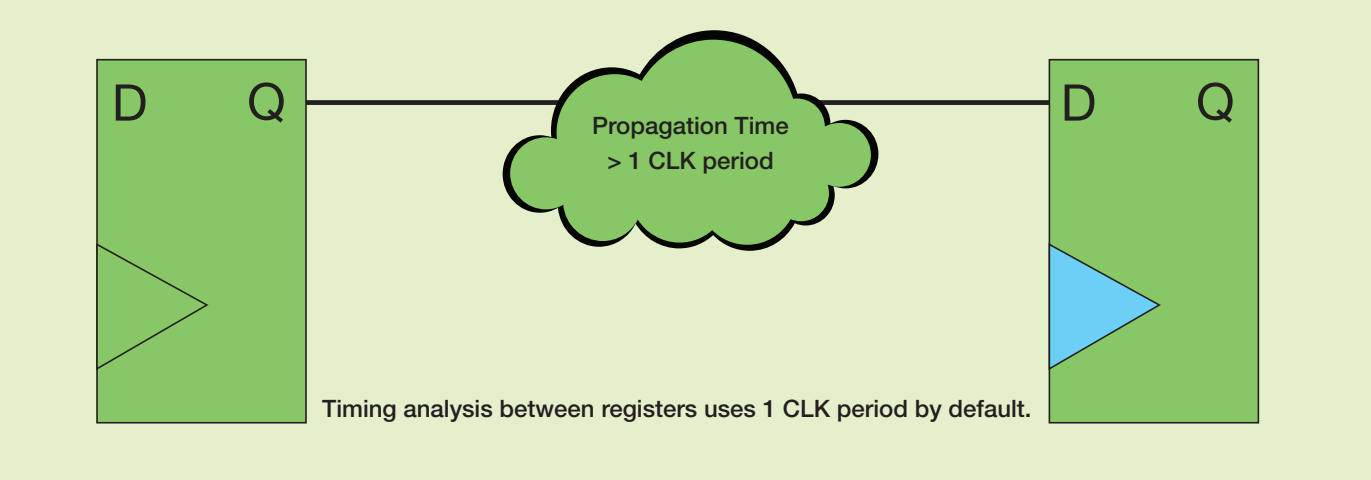


Figure 2 – The multicycle path is one example of a timing exception.

```
set_multicycle_path path_
multiplier [-setup|-hold]
[-start|-end][-from <start-
points>] [-to <endpoints>]
[-through <pins|cells|nets>]
```

When you declare a multicycle path, you are in effect multiplying the requirements for the setup and hold (or both) analysis by the path_mutiplier. For instance, in the first example above, where the output occurs every two clock cycles, the path_multiplier would be two in the case of the setup timing.

Since the multicycle path can be applied to either setup or hold, you then have the choice of where to apply it. When you declare a setup multiplier, it is often best practice to also declare a hold time multiplier using the equation below.

```
hold cycles = setup multi-
plier – 1 – hold multiplier
```

What this means for the simple example we have been following is that the hold multiplier is defined by this equation:

```
hold multiplier = setup
multiplier – 1 When using a
common clock.
```

To demonstrate the importance of multicycle paths, I have created a simple example that you can downloaded here. Within the XDC file there is one example which contains two multicycle paths declared, both setup and hold.

PHYSICAL CONSTRAINTS

The most commonly used physical constraints are the placement of I/O pins and the definition of parameters associated with the I/O pins, for instance standard drive strength. However, there are other types of physical constraints, including placement, routing, I/O and configuration constraints. Placement constraints make it possible to define the locations of cells, while routing constraints allow you to define the routing of signals. I/O constraints let you define the location of I/Os and their parameters. Finally, configuration constraints offer a way to define your configuration methods.

As always, there are a few constraints that sit outside of these groups. The Vivado Design Suite has three such constraints, and they are predominantly used on the netlist.

- DONT_TOUCH – This constraint is used to prevent optimization, and as such it can be of great use when implementing a safety-critical or high-reliability system.
- MARK_DEBUG – This constraint is used to preserve an RTL net such that it can be used for debugging later.
- CLOCK_DEDICATED_ROUTE – This constraint identifies a route for the clock routing.

The most commonly used constraints relate to I/O placement and configuration of the I/O. Placing an I/O on an FPGA involves the use of both placement constraints to locate the physical pin and I/O constraints to configure the I/O properties such as the I/O standard, slew rate, etc.

Modern FPGAs support a number of single and differential I/O standards. These are defined via the I/O constraints. However, you must take care to ensure you are following I/O banking rules, which depend upon the final pin placement.

But what are I/O banking rules? The user I/Os within an FPGA are grouped together into a number of banks consisting of a number of I/Os. These banks have independent voltage supplies, enabling the support of the wide range of I/O standards. On the Zynq®-7000 All Programmable SoC (and other 7 series devices), I/O banks are further classified as belonging to one of two overall groups—high performance and high range. These categories further constrain their performance and require that the engineer use the correct class for the correct interface.

The high-performance (HP) class is optimized for higher data rates. As such, it uses lower operating voltages and does not support LVCMOS 3v3 and 2v5. The other class, high range (HR), is optimized to handle wider I/O standards not supported by HP. HR therefore supports

traditional 3v3 and 2v5 interfacing. Figure 3 demonstrates these banks.

Once you have determined which banks to use for which signal, you still have the ability to change the signal drive strength and slew rate. These metrics will be of great interest to your hardware design team as they strive to ensure that the signal integrity upon the board is optimal. These selections will also affect the timing of the board design. As such, you may opt to use a signal integrity tool.

SI tools require an IBIS model. You can extract an IBIS model of your design from the Vivado tools when you have the implemented design open using the File->Export->Export IBIS model option. You can then use this file to close the system-level SI issues and timing analysis of the final PCB layout.

Once the design team is happy with the SI performance and timing of the system as a whole, you will end up with a number of constraints like the ones below for the I/Os in the design.

```
set_property PACKAGE_PIN
G17 [get_ports {dout}]
set_property IOSTAN-
DARD LVCMOS33 [get_ports
{dout}]
set_property SLEW SLOW
[get_ports {dout}]

set_property DRIVE 4 [get_
ports {dout}]
```

With the HP I/O banks, you can also use the digitally controlled impedance to terminate the I/O correctly and increase the SI of the system without the need for external termination schemes. You must also consider the effects of the I/O if there is no signal driving it, for instance if it is attached to an external connector. In that case, you can use the I/O constraints to implement a pull-up or pull-down resistor to prevent the FPGA input signal from floating, which can cause system issues.

Of course, you can also use physical constraints to improve the timing of your design by implementing the final output flip-flop within the I/O block itself. Doing so reduces the clock-to-output timing. You can do the same thing on input signals as well, which will allow the design to meet the pin-to-pin setup-and-hold timing requirements.

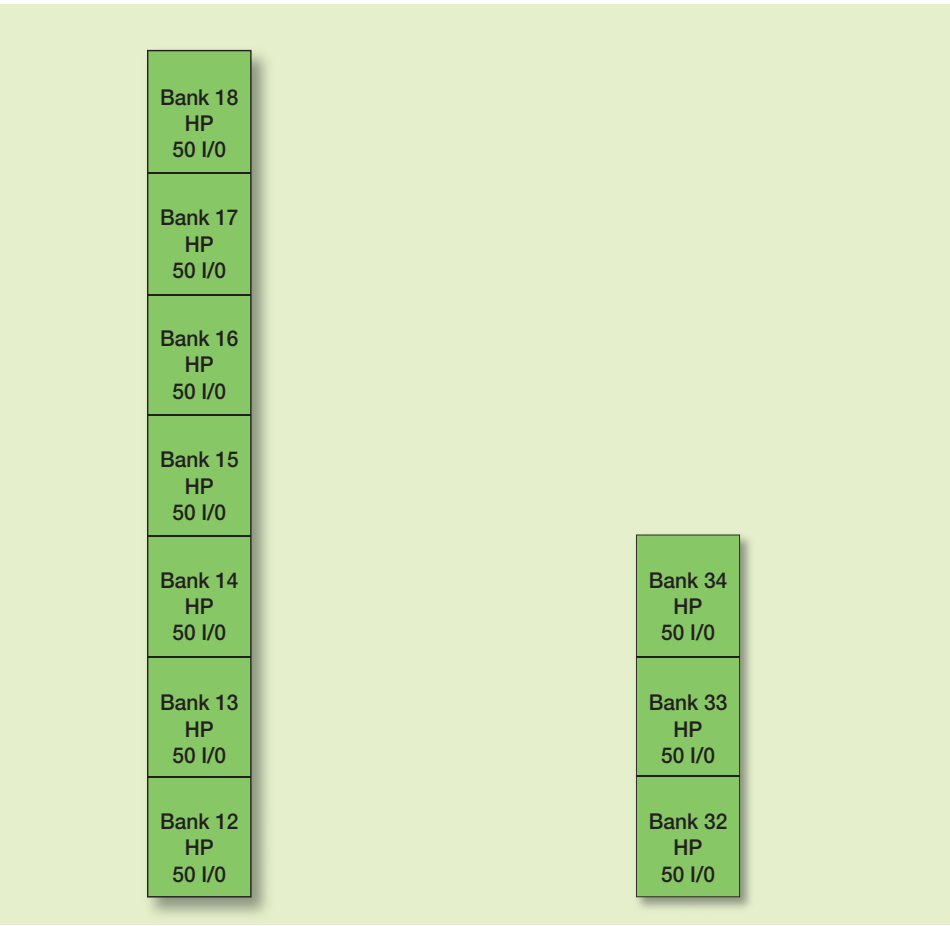


Figure 3 – High-performance (left) and high-range I/O banks on a Xilinx 7 series device.

PHYSICAL CONSTRAINTS START WITH PLACEMENT

You may wish to constrain the placement for a number of reasons—perhaps to help achieve timing or maybe to provide isolation between sections of the design. In this regard, three types of constraints will be important:

- BEL – The basic element of logic allows a netlist element to be placed within a slice.
- LOC – Location places an element from the netlist to a location within a device.
- PBlock – You can use the physical (or "P") block to constrain logic blocks to a region of the FPGA.

Thus, while a LOC allows you to define a slice or other location within the device, a BEL constraint lets you target at a finer granularity the flip-flop to use within the slice. PBlocks can be used to group logic together when segmenting large areas of the design. Another use for PBlocks is to define logical regions when you wish to perform partial reconfiguration.

In some instances, you will wish to group together smaller logic functions to ensure the timing is optimal. While it's possible to do so using PBlocks, it is more common in this scenario to use relatively placed macros.

Relatively placed macros (RPMs) allow design elements such as DSPs, flip-flops, LUTs and RAMs to be grouped to-

```
SIGNAL ip_sync : STD_LOGIC_VECTOR(1 DOWNTO 0) :=(OTHERS =>'0');
SIGNEL shr_reg : STD_LOGIC_VECTOR(31 DOWNTO 0) :=(OTHERS =>'0');

ATTRIBUTE RLOC : STRING;
ATTRIBUTE HU_SET : STRING;
ATTRIBUTE HU_SET OF ip_sync : SIGNAL IS "ip_sync";
ATTRIBUTE HU_SET of shr_reg : SIGNAL IS "shr_reg";
```

Figure 4 – Constraints within the source code

gether in the placement. Unlike PBlocks, RPMs do not constrain the location of these elements to a specific area of the device (unless that's what you want). Instead, RPMs group these elements together when they are placed.

Placing design elements close together makes it possible to achieve two goals. It improves resource efficiency and it allows you to fine-tune interconnection lengths to enable better timing performance.

To co-locate design elements, you can use three types of constraints, which can be defined with the HDL source files.

- U_SET makes it possible to define an RPM set of cells regardless of hierarchy.
- HU_SET allows definition of an RPM set of cells with hierarchy.
- RLOC allows the assignment of relative locations to the SET.

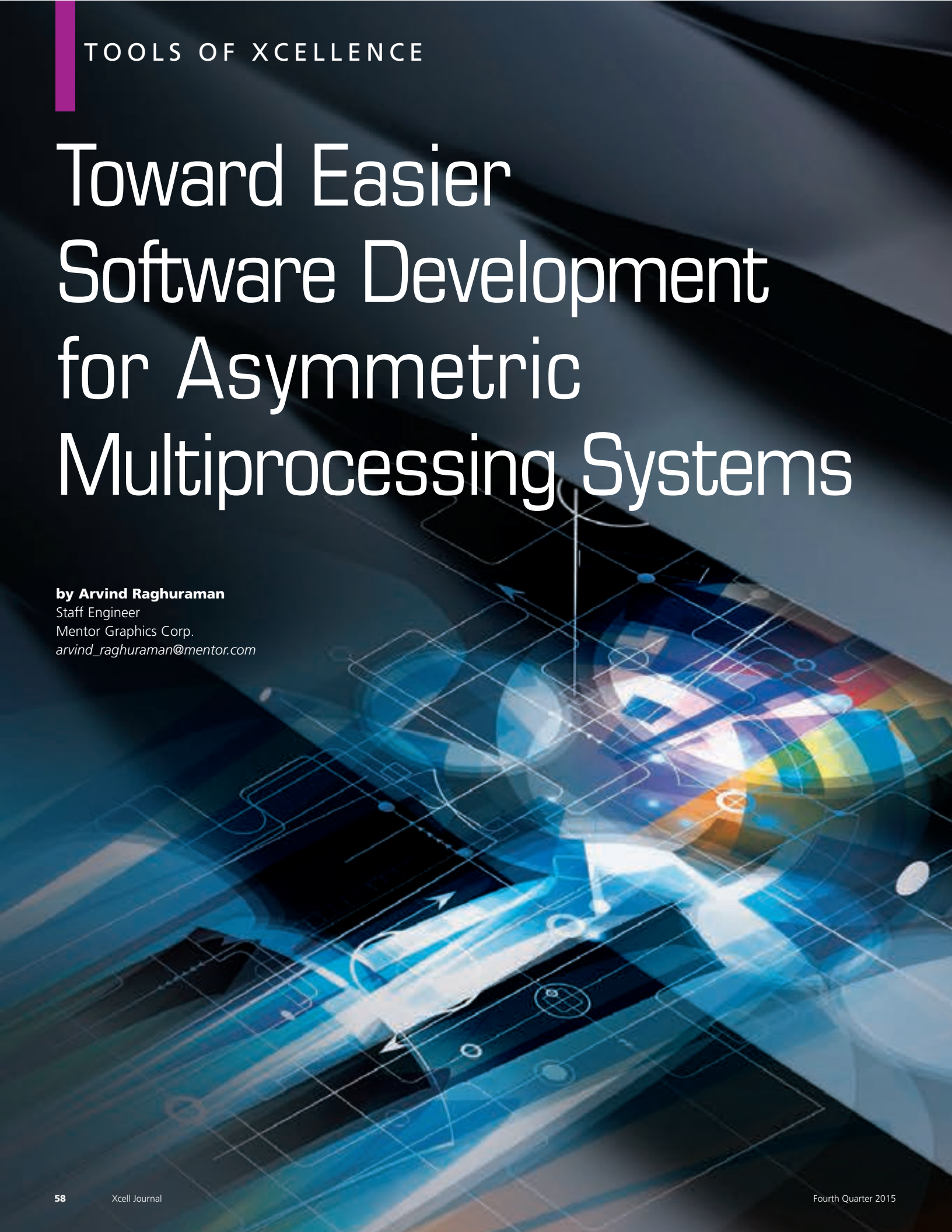
The RLOC constraints use the definition RLOC = XmYm, where the X and Y relate to the coordinates of the FPGA array. When you define an RLOC, you can make this either a relative or an absolute coordinate, depending upon whether you add in the RPM_GRID attribute. Including this attribute makes the definition absolute and not relative.

As these constraints are defined within the HDL as in Figure 4, it is often necessary to run a place-and-route iteration initially, before adding the constraints to the HDL file, so as to correctly define the placement.

In short, understanding timing and placement constraints and learning how to correctly use them are the keys to obtaining the best quality of results in your Xilinx-based programmable logic design.

Toward Easier Software Development for Asymmetric Multiprocessing Systems

by Arvind Raghuraman
Staff Engineer
Mentor Graphics Corp.
arvind_raghuraman@mentor.com



The Mentor Embedded Multicore Framework eases SoC system design by hiding the complexities of managing heterogeneous hardware and software environments.

Heterogeneous multiprocessing is becoming increasingly important to embedded applications today. System-on-chip (SoC) architectures such as Xilinx®'s Zynq® UltraScale+™ MPSoC provide a powerful heterogeneous multiprocessing infrastructure consisting of quad ARM® Cortex®-A53 cores and dual ARM Cortex-R5 cores. In addition to the core compute infrastructure, the SoC contains a rich collection of hardened peripheral IP and FPGA fabric, enabling flexible design paradigms for system developers to create high-performance multiprocessing systems.

Various software development paradigms are available that enable developers to leverage the multiprocessing capabilities offered by SoCs such as the Zynq MPSoC. Symmetric multiprocessing (SMP) operating systems provide the infrastructure required to balance application workloads symmetrically or asymmetrically across multiple homogeneous cores present in a multiprocessing system. However, to leverage the compute bandwidth provided by the heterogeneous processors present in the system, asymmetric multiprocessing (AMP) software architectures are needed.

AMP architectures typically entail a combination of dissimilar software environments such as Linux, a real-time operating system (RTOS) or bare-metal software running on dissimilar processing cores present in the SoC—all working in concert to achieve the design goals of the end application. Typical designs involve a software context on a master core bringing up a remote software context on a remote core in a demand-driven manner to offload computation. The master, remote processors and their associated software contexts (that is, their OS environments) could

Mentor chose the remoteproc and rpmsg API present in the Linux 3.4.x kernel and newer.

be homogeneous or heterogeneous in nature. Effectively dealing with the complexities of managing the life cycles of several operating systems on possibly dissimilar processors, while also providing an enabling interprocessor communication (IPC) infrastructure for offloading compute workload, demands new and improved software capabilities and methods.

The Mentor Embedded Multicore Framework from Mentor Graphics is a software framework that provides two key capabilities for AMP system developers: the remoteproc component and API for life cycle management of remote processors and their associated software contexts; and the rpmsg component and API for IPC between OS contexts in the AMP environment. The framework hides the complexities of managing heterogeneous hardware and software environments, providing the user with a simplified application-level interface.

Let's take a closer look at how you can use this new development framework to manage heterogeneous computation in AMP systems.

COMPATIBILITY AND ORIGINS

Compliance to open standards and adoption by the Linux community were important criteria when selecting an appropriate API for the Mentor Embedded Multicore Framework. Mentor chose the remoteproc and rpmsg API present in the Linux 3.4.x kernel and newer. The Linux remoteproc and rpmsg infrastructure was originally conceived and committed to the Linux kernel by Texas Instruments. The infrastructure allowed Linux OS on a master processor to manage the life cycle and communications with a remote software context on a remote processor.

However, the Linux-provided infrastructure had some limitations. For starters, Linux rpmsg implicitly assumed that Linux would always be the master operating system, and it did not support Linux as a remote OS in an AMP configuration. Furthermore, the remoteproc and rpmsg APIs were available from the Linux kernel space only—there were no equivalent APIs or libraries usable with other OSes and runtimes.

The Mentor Embedded Multicore Framework is a standalone library written in the C language. It provides a clean-room implementation of the remoteproc and rpmsg functionality usable with RTOS or bare-metal software environments, with API-level compatibility and functional symmetry to its Linux counterpart. Figure 1a shows a software stack diagram of the Mentor Embedded Multicore Framework and its usage in RTOS or bare-metal environments. As

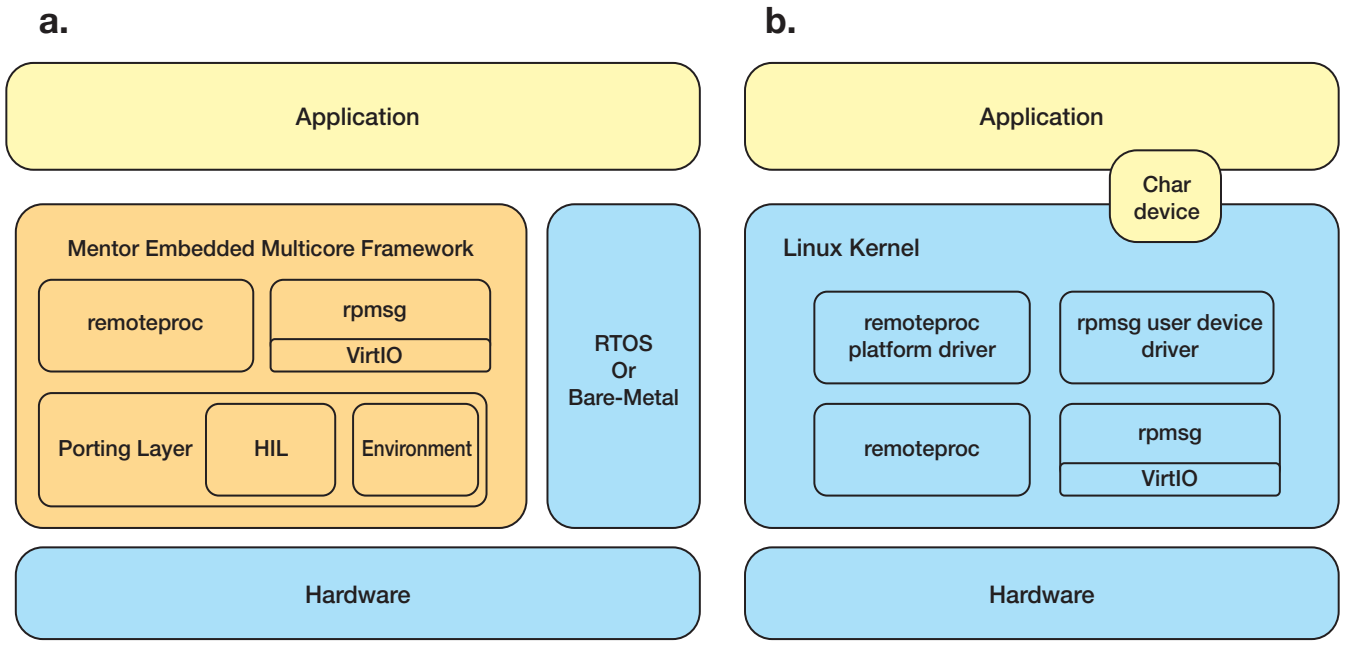


Figure 1 – Mentor Embedded Multicore Framework in RTOS and bare-metal environments (a), and remoteproc and rpmsg in the Linux kernel (b)

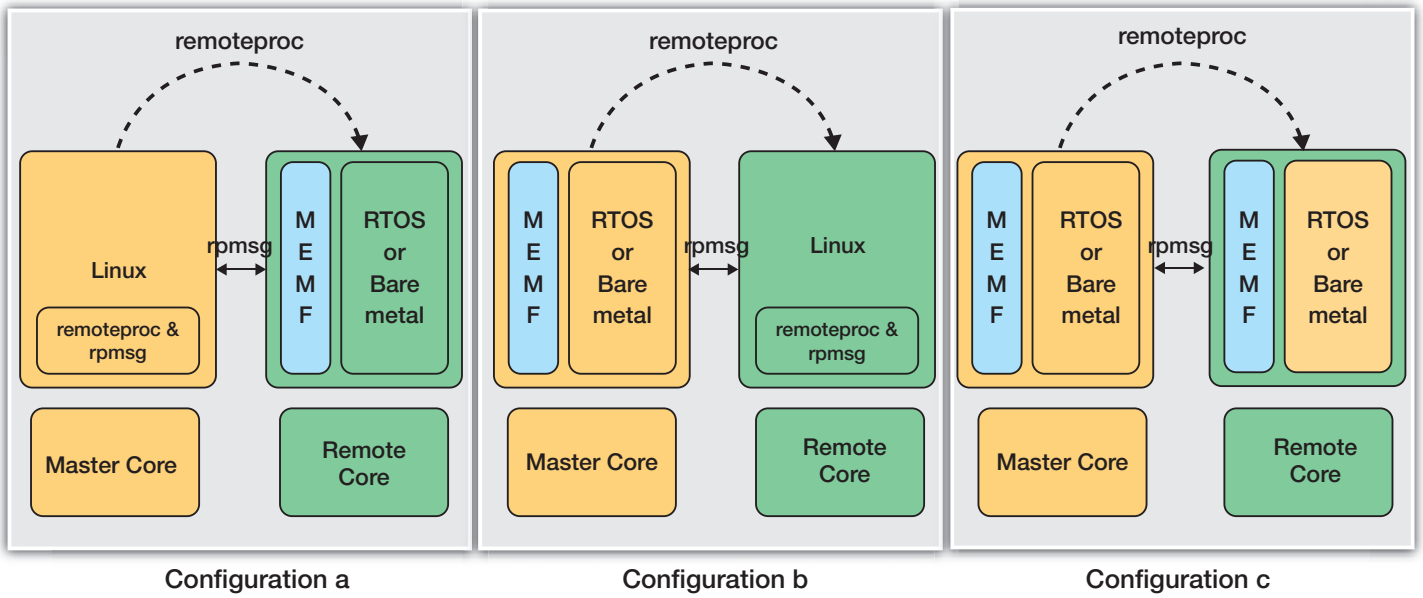


Figure 2 – AMP configurations that the Mentor Multicore Framework supports

shown, the framework's well-abstracted porting layer consists of a hardware interface layer and an OS abstraction (environment) layer, allowing users to easily port the framework to other processors and operating systems.

Figure 1b shows the remoteproc and rpmsg infrastructure present in the Linux kernel. The remoteproc and rpmsg kernel-space drivers provide services to the remoteproc platform driver and rpmsg user device driver. The remoteproc platform driver allows for remote life cycle management, and the rpmsg user device driver exposes IPC services to user-space applications.

In addition to enabling RTOS and bare-metal environments to interoperate with Linux remoteproc/rpmsg infrastructure in AMP architectures, the Mentor Embedded Multicore Framework provides work flows and runtime infrastructure to package and boot Linux as a remote OS in AMP configurations. Figure 2 shows the various AMP configurations the framework supports.

USE CASES AND APPLICATIONS

The Mentor Embedded Multicore Framework is well suited for both unsupervised and supervised AMP architectures.

The unsupervised AMP (uAMP) architecture is useful in applications that do not require a strong separation between the participating OS contexts. In this architecture, the participating operating systems run natively on the processors present in the system. As shown in Figure 3a, the Mentor Embedded Multicore Framework provides a simple and effective infrastructure in which a master software context on a master (boot) processor can manage the life cycle and offload computation to other compute resources present in the SoC.

A supervised asymmetric multiprocessing (sAMP) architecture is best for applications that require isolation of software contexts and virtualization of system resources. In sAMP, the participating guest operating systems run in guest virtual machines that are managed and scheduled by a hypervisor (aka virtual machine monitor). The hypervisor provides isolation and virtualization services for the virtual machines. The Mentor Embedded Multicore Framework enables sAMP architectures to manage computation on heterogeneous compute resources present in the SoC.

As illustrated in Figure 3b, the framework can be used in two ways: from the guest OS context for unsupervised management of heterogeneous compute resources; and from within the hypervisor for supervised management of heterogeneous compute resources, allowing the hypervisor to supervise interactions between the guest operating systems and remote contexts involved.

In general, the Mentor Embedded Multicore Framework is well-suited for applications requiring demand-driven offload of compute functions to specialized cores present on a multiprocessing chip. In the case of power-constrained devices, the framework enables on-demand bring-up and shutdown of compute resources, allowing for optimal power usage.

The framework also provides an easy path for consolidation of legacy single-core embedded systems onto powerful and more capable multiprocessing SoCs. With very little effort, the framework allows for migration of legacy software originally developed for unicore silicon to easily interoperate with enhanced system functionality developed on newer and more powerful multiprocessing chips.

Lastly, the framework facilitates implementation of fault-tolerant system architectures. For example, the framework can enable an RTOS context (master)

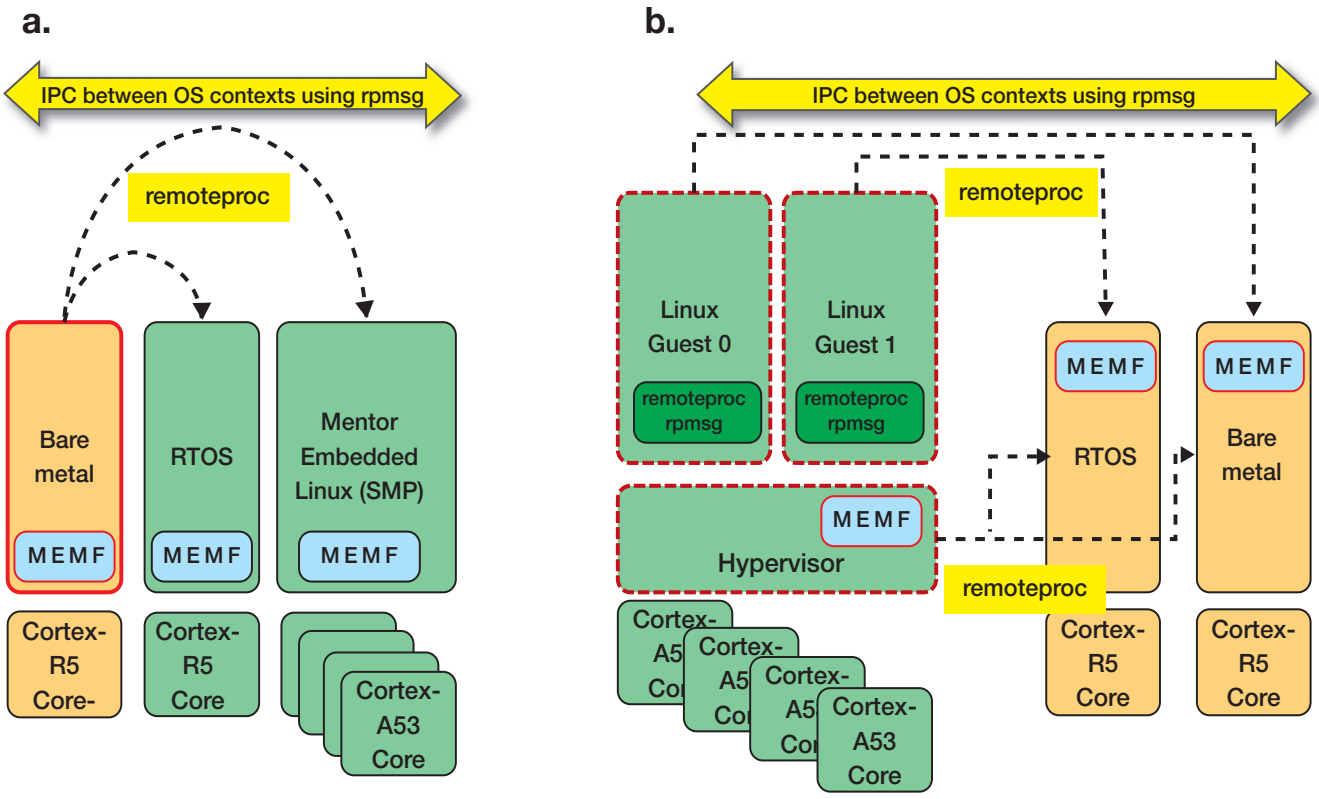


Figure 3 – Mentor Embedded Multicore Framework use cases, including the uAMP (a) and sAMP (b) architectures

that's handling critical system functionality to manage a Linux context handling noncritical system functions. Upon failure of the Linux-based subsystem, the RTOS can simply reboot the failed subsystem without causing any adverse effects to critical system functions.

SYSTEM-LEVEL CONSIDERATIONS

Mentor Embedded Multicore Framework APIs provide the required software infrastructure to manage computation in AMP systems. However, those designing AMP systems must take certain system-level considerations into account before developing application software using these APIs.

During the initial design phase, you will be determining the AMP topology. The framework can be used in a star topology—a single master managing multiple remotes—or in a chain topology, with chained master and remote nodes.

Once you have chosen a suitable topology, the next step is to determine the memory layout. You should assign memory regions for each participating OS runtime, and assign shared-memory regions for IPC between the OS instances. Once the memory layout is finalized, you will need to update the platform-specific configuration data provided to the framework to reflect the chosen memory architecture.

Off-the-shelf operating systems generally assume they own the entire SoC, and are not readily suited for operation in unsupervised AMP environments, where cooperative usage of shared resources and mutually exclusive usage of nonshared resources are key requirements. Each participating OS in an AMP system needs to be modified so that shared resources are used in a cooperative fashion. For example, the remote OS should not reset and reinitialize a shared global interrupt controller that might already be in use within the master context; nor can shared clock trees or peripherals be modified to cause conflicts. These changes will typically require modifications to the participating OS kernel, BSP sources or both.

The next step is to perform system partitioning. You must partition system resources such as memory and nonshared I/O devices between the participating operating systems so that each OS has visibility and can access only the resources that are assigned to it. You can accomplish this task by modifying the platform-specific data (device and memory definitions) provided to the participating OSes. For example, modify memory and device definitions in a Linux device tree source (DTS) file for the Linux OS; in a platform definition file for the Nucleus RTOS; and perhaps in the platform-specific headers in bare-metal environments.

LIFE CYCLE MANAGEMENT USING REMOTEPROC

Once you have made the system-level design decisions and modifications to

The framework provides work flows that make it possible to package Linux, RTOS or bare-metal software images along with required bootstrapping firmware to produce remote firmware images in ELF format.

the participating operating systems, you are now ready to start using the Mentor Embedded Multicore Framework from application software. The framework provides work flows that make it possible to package Linux, RTOS or bare-metal-based software images along with required bootstrapping firmware to produce remote firmware images in ELF format.

A remote firmware ELF image contains a special section called the resource table. The resource table is a static data structure with predefined bindings where users can specify resources required by the remote firmware. Some key definitions supplied in resource tables include memory required by the remote firmware and IPC capabilities supported by remote firmware. The remoteproc component in the master software context will use the resource table definitions to allocate resources and establish communications with the remote context.

The framework master initializes the remote processor context using the remoteproc_init API. On invocation, the remoteproc master fetches the remote firmware image, decodes it, obtains the resource table and parses it to discover the remote firmware's resource requirements. Based on resource table definitions, remoteproc carves out the physical memory required for the remote firmware and performs certain initialization functions specific to rpmsg/VirtIO-based IPC.

Once remoteproc is initialized, you can use the remoteproc_boot API to boot the remote processor with the associated software context. On invocation, the firmware image is located to execute in place in memory, and the remote processor is released from reset to execute the image. The remoteproc_shutdown and remoteproc_deinit APIs allow applications to shut down the remote processor and de-initialize resources, respectively. (The pseudo-code block in Figure 5 shows an example of remoteproc API usage from the master context.)

In the remote context, the boot and shutdown APIs are irrelevant. To initialize and de-initialize the remoteproc component, you must use remoteproc_resource_init, remoteproc_resource_deinit APIs. For information on how remoteproc is used from the Linux context, please refer to Linux kernel documentation.

RPMSG AND INTERPROCESSOR COMMUNICATION

Once the remote firmware is up and running on the remote processor, you can use the rpmsg APIs for interprocessor communication between the master and remote software contexts. The key abstractions and concepts to be understood when using rpmsg are as follows:

- From the perspective of the master, an rpmsg device represents a remote processor.
- An rpmsg channel is a bidirectional communication channel between the master and the remote processor (aka rpmsg device).
- An rpmsg endpoint is a logical abstraction that can be present on either side of an rpmsg channel.
- The endpoints provide the infrastructure for sending targeted messages between master and remote contexts.
- When an endpoint is created, the user provides a unique endpoint index or allows the rpmsg component to assign an index for the endpoint. In addition, the user provides an application-defined callback to be associated with the endpoint being created.
- When a message is received targeted to a given endpoint index, rpmsg invokes the associated receive callback with a reference to the data payload received.
- Users can create any number of endpoints on either side of an rpmsg channel.
- Messages that are not explicitly targeted to a destination endpoint index reach a default endpoint associated with the rpmsg channel.
- The rpmsg component notifies user applications of events such as channel creation and deletion using user provided callbacks registered during initialization.

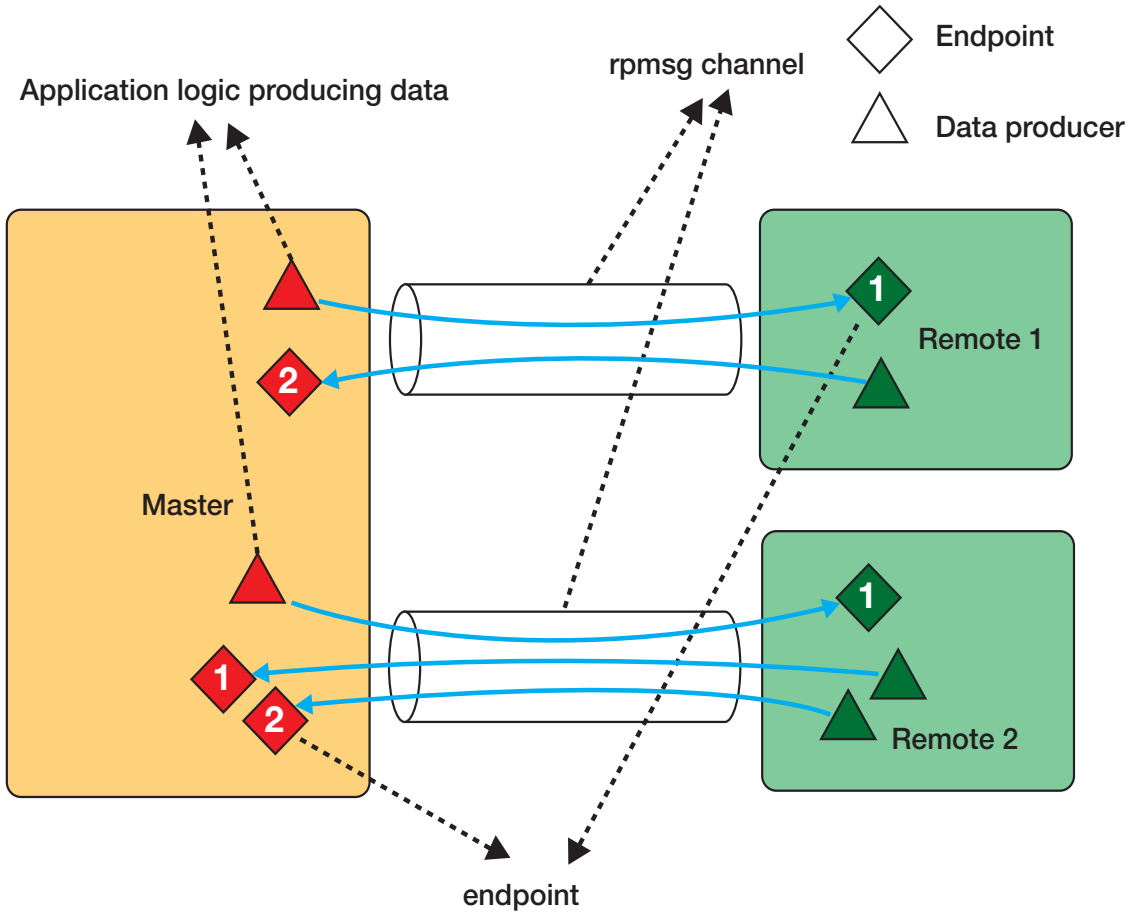


Figure 4 – rpmsg channel and endpoint abstractions

Figure 4 illustrates rpmsg channel and endpoint abstractions and their usage. The rpmsg component works in concert with remoteproc to establish and manage the rpmsg communication channel between master and remote contexts. Once remoteproc on master brings up the remote context, rpmsg on the remote context sends a name service announcement. Upon receiving the name service announcement, the master registers the announced rpmsg device and establishes an rpmsg channel. Once the channel is established, the rpmsg channel-created callback is invoked on both sides, notifying the master and remote applications of channel creation.

At this point, the master and remote context can transmit data to each other using the rpmsg_sendxx and rpmsg_trysendxx APIs for blocking and non-blocking transmit requests respectively. When the remote context invokes remoteproc_resource_deinit, the master application is notified of the event by the rpmsg channel deleted callback, allowing for graceful termination of the rpmsg-based communication link. The master can choose to asynchronously shut down the remote processor using the remoteproc_shutdown API in situations where the remote context becomes unresponsive. The pseudo-code segment in Figure 5 shows usage of rpmsg APIs in concert with remoteproc APIs in a master context.

The rpmsg component uses VirtIO as a shared-memory-based transport abstraction. VirtIO has its roots as an I/O virtualization standard used for guest-to-host communications in lguest, KVM and the Mentor Embedded Hypervisors. The rpmsg driver uses services provided by the VirtIO layer for shared-memory-based communications with its counterpart. The rpmsg driver instantiates an rpmsg VirtIO device and uses the VirtQueue interface to push and consume data w.r.t. its communicating counterpart.

TOOLS FOR DEVELOPMENT OF AMP SYSTEMS

Development of AMP application software presents a unique set of challenges. System developers typically find themselves having to simultaneously debug different OS environments deployed on dissimilar processors on heterogeneous SoCs. Having a unified debugging environment with awareness of the operating systems involved will not only enhance the debug experience, but improve productivity. Mentor Embedded Sourcery CodeBench tools provide a unified IDE with OS awareness for all supported OS environments (including

Mentor Embedded Linux and Nucleus RTOS). Sourcery CodeBench also supports a multitude of debug options, which include JTAG-based debug for debugging Linux kernel space, Nucleus RTOS and bare-metal contexts; and GDB-based debug for Linux user space and Nucleus RTOS-based applications.

```
/* rpmsg channel created callback - invoked on channel creation */
void rpmsg_channel_created(struct rpmsg_channel *rp_chnl) {
    ..
      /* Use RTOS provided primitives (ex., semaphores) to
    release the application context blocked on rpmsg
    channel creation */
}

/* rpmsg channel deletion callback - invoked on channel deletion */
void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl) {
    ..
      /* Use RTOS provided primitives (ex., semaphores) to
    release the application context blocked on rpmsg
    channel deletion */
}

/* rpmsg receive callback - invoked when data received on
default endpoint */
void rpmsg_rx_cb(struct rpmsg_channel *rp_chnl, void *data,
     int len, void * priv, unsigned long src) {
    ..
      /* Copy received data to application buffer and use
      RTOS provided primitives (ex., semaphores) to release
      the application IPC data processing context */
}

/* Initialize remote context */
int Initialize_Remote_Context(..) {
    ...
    /* Initialize remote context */
    remoteproc_init(remote_fw_info,
        rpmsg_channel_created, rpmsg_channel_deleted,
        rpmsg_rx_cb, &proc);

      /* Boot remote context */
    remoteproc_boot(proc);
      ...
}

/* Send data to remote context after rpmsg channel creation */
int Send_Data_To_Remote(..) {
    ..
      rpmsg_send(app_rp_chnl, user_buff, sizeof(user_buff));
      ..
}

/* Finalize remote context after rpmsg channel deletion */
int Finalize_Remote_Context(..) {
    ..
      /* Shut down and finalize the remote processor */
      remoteproc_shutdown(proc);
      remoteproc_deinit(proc);
      ..
}
```

Figure 5 – Pseudo code that illustrates the usage of key remoteproc and rpmsg APIs from the master context

While developing AMP systems, software profiling is a valuable tool to gain insight into how various applications deployed on heterogeneous OSes interact with each other during runtime. Each OS instance typically uses an independent clock reference, and any profiling data collected within a given OS context will be based on a time base that is local to the OS. Mentor Embedded Sourcery Analyzer host-based tools and Mentor's operating systems contain built-in algorithms that enable users to graphically visualize and analyze trace data collected from disparate OS sources in a unified time reference. This capability allows users to gain interesting insights into complex interactions and hard-to-find timing issues typically encountered in developing AMP software.

AN OPEN-SOURCE RUNTIME COMPONENT

The Mentor Embedded Multicore Framework is tightly integrated with Mentor's development tools and operating systems. It supports a diverse set of ARM-based SoCs and platforms. Using the framework with Mentor tools and operating systems frees users from having to design their AMP system from scratch—that is, perform the tasks discussed under the system-level considerations section above. Users can get started with AMP application development with one of the reference configurations and later customize the system configuration to fit their needs.

For AMP system design, there is a clear need for a standards-based software framework that enables development of RTOS or bare-metal software that can interoperate with interfaces adopted by the open-source Linux community. To address this need and to promote industry adoption, Mentor Graphics and Xilinx have jointly open-sourced the Mentor Embedded Multicore Framework's runtime component under the OpenAMP open-source project, with platform support for the the Zynq-7000 All Programmable SoC. This project is currently jointly maintained by Mentor Graphics and Xilinx.

What's New in the Vivado 2015.3 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 2015.3 is available from the Xilinx Download Center at www.xilinx.com/download.

VIVADO DESIGN SUITE 2015.3 RELEASE HIGHLIGHTS

The Vivado Design Suite 2015.3 release introduces new market-tailored plug-and-play IP subsystems. These new subsystems, in combination with enhancements to Vivado IP Integrator (IPI) and high-level synthesis (HLS) for C/C++ and SystemC-based design, significantly decrease design creation and integration efforts by abstracting time-consuming RTL development.

Also, the Vivado 2015.3 implementation has new features including pipeline reporting, automated pipeline insertion and retiming, and enhanced cross-probing, as well as core engine enhancements.

Device Support

New Devices

The following UltraScale™ devices are introduced in this release:

- Kintex® UltraScale devices: XCKU095, XCKU025, XCKU085

New General Access Supported Devices

The following devices are production ready (in -1 and -2 speed grades)

- Kintex UltraScale devices: XCKU095, XCKU025, XCKU085
- Virtex® UltraScale devices: XCVU095, XCVU080

VIVADO DESIGN SUITE: DESIGN EDITION UPDATES

Partial Reconfiguration and Tandem Configuration

The new release of the Vivado Design Suite features expanded support for UltraScale devices, including place-and-route support for the KU085, KU095, VU065 and VU080, bringing the total number of UltraScale devices supported to 12.

Partial-bit-file generation is now enabled for the KU060, KU095, KU115 and VU095 production silicon, bringing the total number of devices enabled for bitstreams up to six.

Also new is this release is the Partial Reconfiguration Decoupler IP. This new core makes it easy for designers to isolate reconfigurable partitions from the static design during reconfiguration.

See the Xilinx IP page for more details: *http://www.xilinx.com/products/intellectual-property/pr-decoupler.html*.

Vivado IP Integrator

To speed design integration, the new release offers easy access to IP example designs from IP Integrator (IPI), along with enhanced configurable example designs. Among these example designs is an option to configure the MicroBlaze™ processor.

Vivado Simulator

The latest simulator version offers up to a 3x improvement in elaboration runtime performance.

Vivado Debug and Reporting

Xilinx has added a host of new reports to speed timing closure and debug. The report_design_analysis command reports a list of paths that are critical at both the current design stage and the prior stage. This provides a way to check on which critical paths the tools will focus on at each stage. Another new command, report_pipeline_analysis, evaluates potential design performance improvements by hypothetically adding latency (pipeline stages) to the design and reporting the new, resulting Fmax.

VIVADO DESIGN SUITE: SYSTEM EDITION UPDATES

Vivado High-Level Synthesis

To accelerate development of C-based IP, Vivado HLS can now launch the Vivado waveform viewer after running a C/RTL co-simulation to visualize the simulation waveforms. Just click on the Open Wave Viewer toolbar button. Also new is support for half-precision floating-point through the hls_half.h header file. The half-precision feature allows for smaller and faster designs while in many cases retaining sufficient numerical precision.

See the Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.

System Generator for DSP

New for System Generator is support for MATLAB® 2015B, including tighter integration that allows the HDL Coder to automate the generation of a combined model containing high-level RTL and target-optimized IP.

For better usability, Xilinx has enhanced the System Generator's blockset for digital upconverters and digital downconverters (DUC/DDC), greatly simplifying this blockset for wireless algorithm development. Enhancements to improve verification and compile runtime have been added to the new blocks, all of which are configured with seven or fewer parameters. The digital FIR filter block tightly integrates with the Filter Design and Analysis Tool from MathWorks to build area-efficient filters, including fixed-fractional interpolation or decimation types. The sine-wave and complex-product blocks greatly simplify modulator design for frequency conversion at high sample rates. The requantize block enables quick manipulation of data types to maximize dynamic range at any point in the data path.

System Generator for DSP also features new interactive cross-probing that accelerates design exploration and provides iterative design closure. With the cross-probing of timing analysis, algorithm developers can quickly identify their critical paths and single out bottlenecks that may affect throughput and latency of their algorithms to make swift adjustments. Also new to this release are improvements in System Generator's hardware-based co-simulation, improving verification runtime by 45x.

XILINX INTELLECTUAL PROPERTY (IP) UPDATES

Xilinx's new LogiCORE™ IP subsystems are highly configurable, market-tailored building blocks that integrate up to 80 individual IP cores, software drivers, design examples and testbenches. Available with the Vivado Design Suite 2015.3 release are new IP subsystems for Ethernet, PCIe®, video processing, image sensor processing and OTN development. These IP subsystems are based on industry standards such as the AMBA® AXI-4 interconnect protocol, IEEE P1735 encryption and IP-XACT to enable interoperability with Xilinx and Alliance member IP and to accelerate integration.

The highly configurable Video Processing Subsystem supports a 4K2K video pipe. Comprehensive video support includes VDMA, deinterlacer, chroma resampler and scaler. The subsystem can also easily source and sync DisplayPort, HDMI and MIPI interfaces by leveraging the automatically generated AXI interfaces and Vivado IPI. The new video IP subsystem replaces Xillinx's VIPP cores.

LEARN MORE

See the Vivado Design Suite 2015.3 Release Notes for more information.

QuickTake Video Tutorials

Vivado Design Suite QuickTake tutorials are how-to videos that take a look inside the features of the Vivado Design Suite and UltraFast™ Design Methodology.

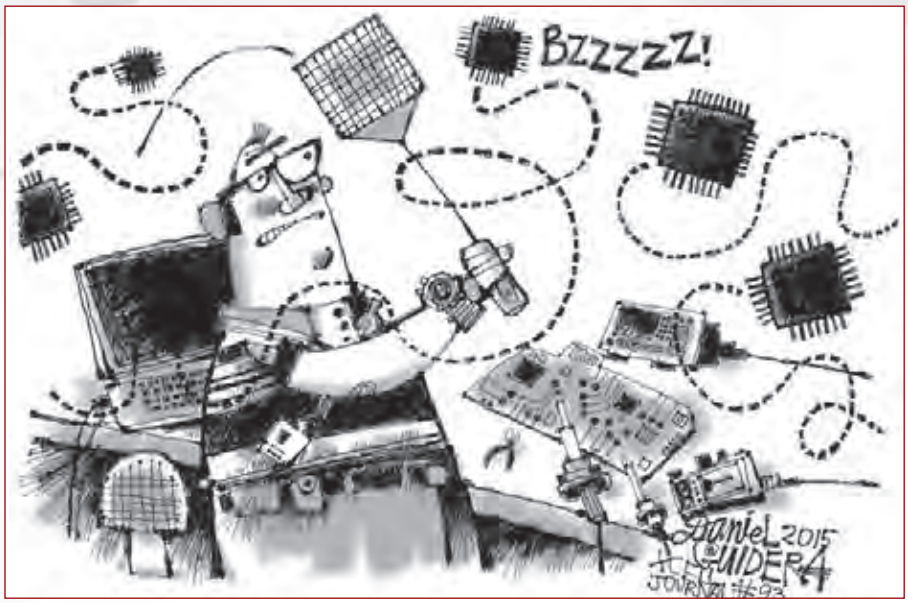
See all QuickTake Videos here: *www.xilinx.com/training/vivado*.

Training

For instructor-led training on the Vivado Design Suite, UltraFast Design Methodology and more, visit *www.xilinx.com/training*.

Download Vivado Design Suite 2015.3 today at *http://www.xilinx.com/download*.

Xpress Yourself in Our Caption Contest



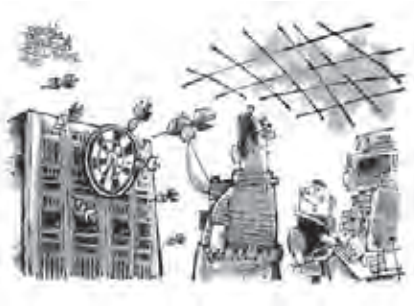
DANIEL GUIDERA

Every designer knows that debugging is one of the trickiest steps in the flow, but at this lab, things have gotten out of hand. Xercise your funny bone as you help our beleaguered cartoon engineer swat some of the most recalcitrant bugs we've ever seen. We invite readers to submit an engineering- or technology-related caption for this cartoon showing a worst-case debugging situation. The image might inspire a caption like "Henry longed for an automated way to debug his latest design, because doing it by hand just wasn't working."

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 have their captions, names and affiliations featured in the next issue.

The contest begins at 12:01 a.m. Pacific Time on Oct. 15, 2015. All entries must be received by the sponsor by 5 p.m. PT on Jan. 6, 2016.

TRAVIS ROTHLISBERGER, director of device development at Cerevast Medical (Redmond, Wash.), won a shiny new Digilent Zynq Zybo board with this caption for the dartboard cartoon in Issue 92 of *Xcell Journal*:



"To Ed's delight, nobody questioned him when he said that his Monte Carlo analysis would take several months to complete."

Congratulations as well to our two runners-up:

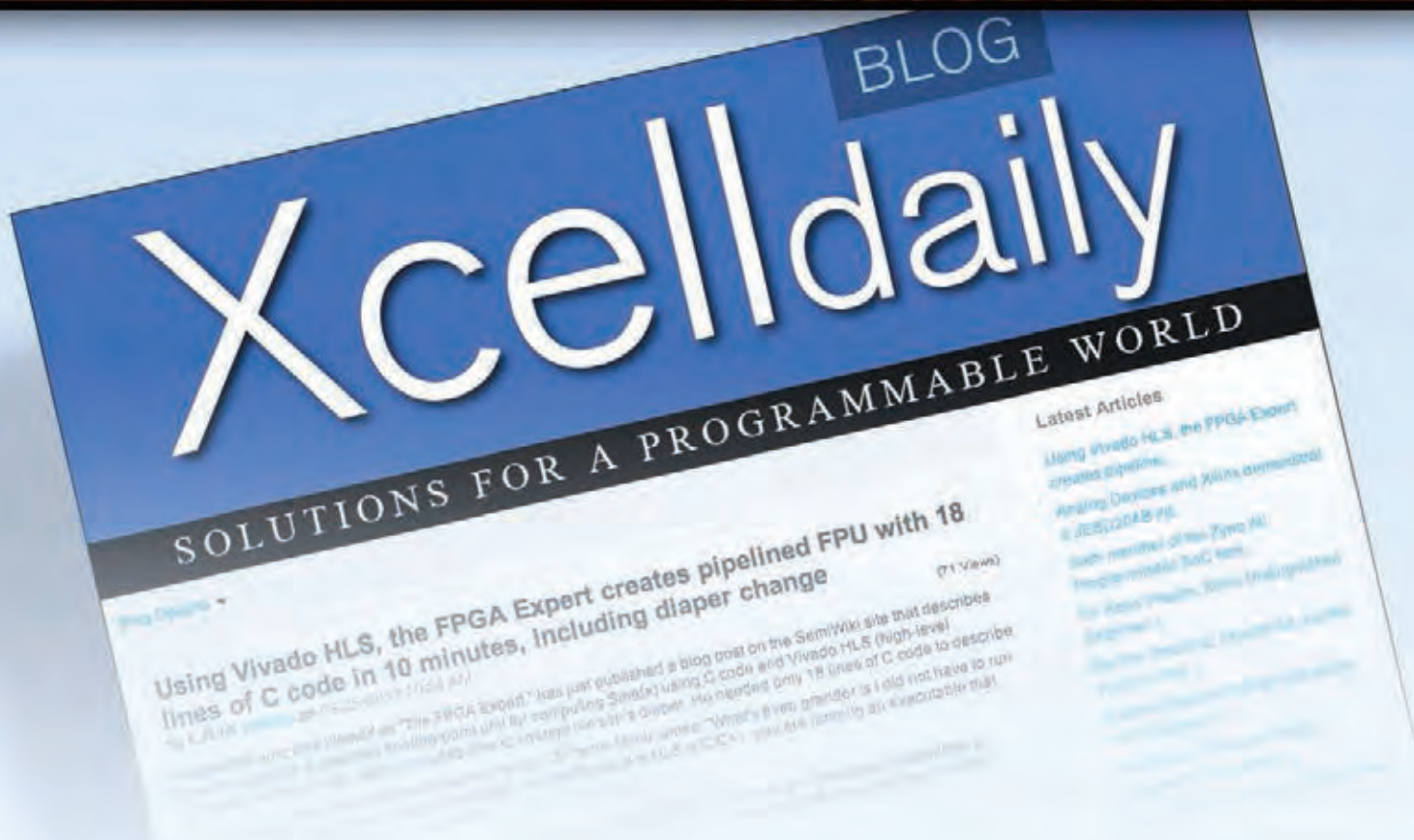
"Keep debugging, I'm finalizing the feature set and pricing right now!"

— Lee Courtney, CTO/VP-Product, Qurasense (Menlo Park, Calif.)

"I figure this is a better way to meet targets than what goes on upstairs!"

– Larry Standage, principal applications engineer, Microchip Technology Inc. (Chandler, Ariz.)

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

- *$55 Zynq-based Wireless Snickerdoodle single-board computer with WiFi, Bluetooth launched today on CrowdSupply*
- *Tiny 100x62mm, Zynq-based Avnet PicoZed SDR implements 2x2 MIMO, 70MHz to 6GHz radio using ADI RF Agile Transceiver*
- *ARTY—the $99 Artix-7 FPGA Dev Board/Eval Kit with Arduino I/O and $3K worth of Vivado software. Wait, What????*
- *Lift-off! 16nm Zynq UltraScale+ MPSoC ships to customers. From tapeout to "Hello World" in 2.5 months.*
- *Zynq-based, $179 Skreens Nexus on Kickstarter allows you to safely cross the (HDMI video) streams*

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

XILINX
ALL PROGRAMMABLE