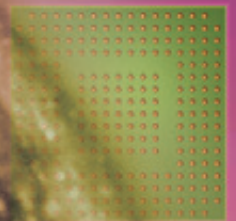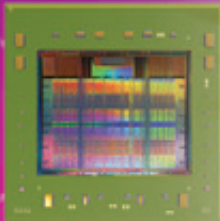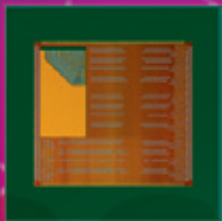# Xcell journal

SOLUTIONS FOR A PROGRAMMABLE WORLD

Xilinx Moves a Generation Ahead with All Programmable Devices

Vivado HLS Eases Design of Floating-Point PID Controller

Image Sensor Color Calibration Using the Zynq-7000 SoC

Floating-Point Design with Xilinx's Vivado HLS

Tips on Implementing State Machines in Your FPGA

How to Test and Debug Zynq SoC Designs with BFMs

page **22**

XILINX®

So Much More Than Gate Arrays

A couple of weeks after I joined Xilinx back in 2008, one of my new colleagues gave me an original copy of the press release announcing Xilinx's first product, the XC2064, which would eventually come to be known as the world's first FPGA. The public release date was Nov. 1, 1985 and the announcement came from legendary Silicon Valley public relations firm Regis McKenna. Every once in a while, I'll pick up the press release and give it another read, because it reminds me how far the FPGA industry has come. The contrast between what FPGAs were like in 1985 and what they can do today is remarkable—so much so that I think many of them have outgrown the moniker "field-programmable gate array." Well, at least the "gate array" part.

It's no easy task naming innovative technologies. Back when Xilinx and Regis McKenna were launching the XC2064, it appears that they hadn't quite figured out what to call the device. In fact, the headline for the press release reads: "Xilinx Develops New Class of ASIC." Meanwhile, the first paragraph describes the device as follows: "The new device, called a logic cell array, offers a high level of integration together with the versatility of a gate-array-like architecture." The release goes on to describe many truly revolutionary features, including "unlimited reprogramming without removal from the system," "64 configurable logic blocks and 58 I/O blocks…[and] 1,000 to 1,500 gate equivalents." These were all quite impressive for the time. Clearly, Xilinx had set its sights on displacing PALs and "bipolar and CMOS programmable products, LS TTL [low-power Schottky transistor-transistor logic] components and gate arrays." Who knew at the time that this new class of device would one day compete with ASICs and consolidate the functions of many chips into one?

I started covering the EE design space in 1995, and so I missed the point in time when these "logic cell array ASICs" became commonly known as field-programmable gate arrays. What's a bit puzzling to me is how "gate array" got into the name? In 1985, the public relations team likely called the devices "a new class of ASIC" because even back then, ASICs were starting to rapidly displace gate arrays. By the time I began writing about electronic design, gate arrays were gone, daddy, gone.

I get the field-programmable part. That's an engineering-esque way of saying that the device was reprogrammable. But "gate arrays"? Really?

Certainly, the devices Xilinx offers today are so much more than gate arrays that are reprogrammable. As you'll read in this issue's cover story, back in 2008 Xilinx began executing on a plan that redefines the possibilities of programmability. At the 28-nanometer mode, Xilinx delivered three lines of All Programmable FPGAs (the Virtex®-7, Kintex™-7 and Artix-7™ devices), but didn't stop there. In its 28-nm generation, Xilinx also delivered the first homogeneous and heterogeneous All Programmable 3D ICs, which shatter capacity and bandwidth records, respectively. What's more, Xilinx is delivering to customers today the Zynq™-7000 All Programmable SoC, which marries an ARM® dual-core Cortex™-A9 MPCore, programmable logic and key peripherals all on a single device. On top of these silicon innovations, Xilinx launched a fresh, state-of-the-art design suite called Vivado™ to help customers leverage the new device families to achieve new levels of system integration, reduce BOM costs, lower power consumption and speed system performance.

Not only do the latest Xilinx offerings have innovative programmable logic, they also boast programmable I/O, DSP slices and embedded processors, making them also software programmable—they are truly All Programmable, enabling truly reprogrammable systems, not just reprogrammable logic. They certainly are not gate arrays.

Mike Santarini
Publisher

HES-7™
ASIC Prototyping

Scalable, Flexible Solution

HES-7™ provides SoC/ASIC hardware verification and software validation teams with a scalable and high quality FPGA-based ASIC prototyping solution backed with an industry leading 1-year limited warranty. Each HES-7 board with dual Xilinx® Virtex®-7 2000T has 4 million FPGA logic cells or up to 24 million ASIC gates of capacity, not including the DSP and memory resources.

The HES-7 prototyping solution was architected to allow for easy implementation and expansion. Using only one or two large FPGAs, rather than multiple low density FPGAs, HES-7 does not require as much labor-intensive partitioning or tool expense.

Using a non-proprietary HES-7 backplane connector, HES-7 can easily expand prototype capacity up to 96 million ASIC gates and can include the expansion of daughterboards.

5 Global CLKS (Diff Pairs)
10x Single Region Clocks
10x Multi Region Clocks
20x Single Region Clocks
20x Multi Region Clocks

PLLs

16x GTX
240/120 (SE/Diff)
28x GTX
480/240 (SE/Diff)

SPARTAN® Prog. FPGA
KINTEX®-7 FPGA0
VIRTEX®-7 FPGA1
VIRTEX®-7 FPGA2

96/48 (SE/Diff)
226/113 (SE/Diff)

HES-7 Prototyping Board
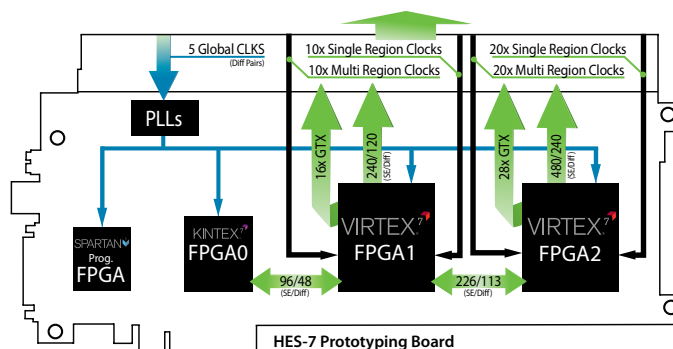
HES-7 is available in four different, single or dual, FPGA configurations, includes popular interface protocols, abundant I/O and GTX resources that make this prototyping board the perfect fit for any design verification and validation project.

XILINX
| ALLIANCE PROGRAM

ALDEC
THE DESIGN VERIFICATION COMPANY®

CONTENTS

THE XILINX XPERIENCE FEATURES

22

58

XTRA READING

46

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

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

Moving a Generation Ahead with All Programmable Devices

At 28 nm, Xilinx pioneered All Programmable SoCs and 3D ICs, introduced innovative low-power FPGAs and built a new tool suite to move a generation ahead of the competition.

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

At the 28-nanometer node, Xilinx® introduced several new technologies that have created an extra generation of value for customers and moved Xilinx a generation ahead of its competition. Rather than simply migrate the existing FPGA architecture to the next node, Xilinx delivered numerous FPGA innovations and pioneered the first commercial All Programmable 3D ICs and SoCs.

These All Programmable devices, all shipping today, employ "All" forms of programmable technologies—going well beyond programmable hardware to software, beyond digital to analog and mixed signal (AMS), and beyond single-die to multidice 3D IC implementations (Figure 1). With these new All Programmable devices, design teams can achieve greater degrees of programmable systems integration, increase overall system performance, reduce BOM costs and get ever smarter, more innovative products to market more quickly.

The transformation of Xilinx's product portfolio can be traced back to 2008. Under the leadership of new CEO Moshe Gavrielov, Xilinx set in motion a comprehensive strategy to broaden its technology portfolio, expand market reach and move a generation ahead starting at the 28-nm node. This mandate included the go-ahead to commercially produce two entirely new classes of devices that Xilinx Labs and product-engineering teams had been prototyping and evaluating for years. The company also engaged with foundry TSMC to create a new silicon process at 28 nm called HPL (High Performance, Low Power), tailored for the sweet spot of FPGAs with the optimal mix of performance and low power. Recognizing power as a top concern for customers, Xilinx implemented the entire All Programmable product line on this process (see cover story, *Xcell Journal*

issue 76). Xilinx also assembled a first-class EDA team to develop an entirely new, modern design suite. The goal was not only to boost customer productivity for the five different 28-nm device families, but also to provide the scalability required for the next decade of All Programmable devices.

FIRST UP: ZYNQ-7000 ALL PROGRAMMABLE SOC

The first of these new classes of devices, the Zynq™-7000 All Programmable SoC (system-on-chip), is an *EE Times* Innovation of the Year award winner for 2011, in addition to being an industry game-changer. An All Programmable SoC combines three forms of programmability for mass customization: hardware, software and I/O programmability. The idea for the device was born from customer feedback as well as lessons Xilinx learned through the many years of delivering FPGAs that had soft- and hard-core processors on-chip.

Starting in the late 1990s, Xilinx and competitors began offering soft processor cores that customers could synthesize into the logic structures of FPGAs. In this way, design teams were able to closely link processing with logic functionality in the same fabric and achieve greater degrees of BOM cost reduction. In practice, designers used many of these soft processors for embedded state machines, rather than running the OSes and software stacks typically associated with more-complex systems. In the mid-2000s, as new semiconductor processes allowed FPGA vendors to offer higher-capacity devices, vendors began to improve the processing performance of these FPGAs by embedding hard processor cores alongside FPGA logic. Xilinx, for example, introduced the Virtex®-4 FX and later Virtex-5 FX families, each of which included a PowerPC® CPU core embedded alongside the FPGA logic.

Portfolio: All Programmable
FPGAs, SoCs, and 3D ICs
Available Today

All Programmable
SoC

■ First shipped: Q4, 2011

■ Integrating FPGA, CPU,
DSP, AMS

■ Competitor silicon
not available

All Programmable
FPGA Family

■ First shipped: Q1, 2011

■ Delivering one extra node
of power and performance

All Programmable
3D ICs

■ First shipped: Q3, 2011

■ Integrating 2x logic and
serdes bandwidth

■ Competitor working only
on test chips

Figure 1 – At 28 nm, Xilinx has expanded "programmability" beyond logic to create a line of ALL Programmable devices.

While the Virtex FX families great-ly improved processor performance over soft implementations, they required design teams to first pro-gram the FPGA logic before they could program the processor. Once the FPGA logic was programmed, the designers then needed to create their own peripherals, memory subsys-tems and, ultimately, "embedded sys-tems," along with the associated plumbing to and from the logic. While expert design teams well-versed in FPGA design welcomed the resulting boost in processor performance, the architecture was counterintuitive to more popular, traditional embedded-system design methodologies. Learning from this experience, in 2008 Xilinx began architecting the Zynq-7000 All Programmable SoC and, equally important, the associat-ed ecosystem—the firmware and software development tools and infrastructure—to facilitate the pro-gramming of the device.

For the Zynq-7000 All Prog-rammable SoC, Xilinx chose the highly popular and well-supported 1-GHz ARM® A9 dual-core processor system, and worked with ARM to create the AXI4 interface standard to facilitate the plug and play of third-party, Xilinx and customer-developed cores in the logic portion of the architecture. The Zynq family boots directly from the processor. This allows system designers to work in a familiar fashion and helps design teams get an early jump on software development—speeding time-to-mar-ket. Because the processor boots first, even software designers not familiar with FPGA logic or hardware design can begin to use the device and perhaps expand their programming repertoire. Xilinx also gave the Zynq-7000 a rich set of peripheral IP and programmable, high-speed I/O—deliv-ering to customers not just an FPGA or an FPGA with a processor, but all told, a truly All Programmable SoC.

Xilinx announced the architecture in 2010 to give customers and ecosys-tem partners a jump on product devel-opment. The company delivered the first All Programmable SoC to cus-tomers in the winter of 2011. One of the first customers to receive this device, a company that had been defin-ing and developing its design using a Zynq emulation platform for more than a year, was able to get the design oper-ating successfully within a few hours of receiving the silicon. Today, Zynq is in as much demand as the other Xilinx FPGA families, with most applications integrating systems functions that used to be done in separate CPU, DSP, FPGA and AMS components. For details on the Zynq-7000 All Programmable SoC, see the cover story in *Xcell Journal* issue 75. As of September 2012, one company had dis-closed plans to release a competitive device, but has yet to announce the delivery of silicon or a significant ecosystem to adequately support it.

ALL PROGRAMMABLE 3D ICS

The second of the radical new device classes that Xilinx pioneered at 28 nm is the category of "All Programmable 3D ICs." Back in 2004, Xilinx Labs began to explore and eventually prototype the stacking of multiple silicon dice in a single-IC configuration as a way to go beyond the scaling limits of Moore's Law to create new levels of programmable systems integration. Xilinx's scientists created test chips of various 3D IC architectures, exploring alternative ways of stacking silicon dice and using through-silicon vias (TSVs) to both power the dice and support die-to-die communications. Through extensive prototyping and a view toward reliable manufacturing, the company concluded that the most practical near-term commercially viable architecture would be one that Xilinx called "stacked silicon interconnect" (SSI). In this architecture, multiple dice are placed side-by-side on top of a passive silicon interposer, which facilitates the interconnect/communication among the many dice. With more than 10,000 interconnects that can be programmed among the dice, along with programmability of each die as well as the I/O, Xilinx has created not only the first commercial 3D IC, but the first All Programmable 3D IC. To learn more about the SSI technology, read the cover story in *Xcell Journal* issue 77.

In early 2012, Xilinx delivered the very first 3D ICs to customers. The Virtex-7 2000T device stacks four FPGA logic slices side-by-side. The device established a new record for IC transistor counts (more than 6.8 billion transistors) at the 28-nm node and smashed the record for FPGA logic capacity, offering 2 million logic cells (the equivalent of 20 million ASIC gates). The device is double the size of the competition's largest FPGA, essentially offering today a device with a logic capacity that one would not have expected until the 20-nm process node. What's more, this SSI technology

> **Xilinx Labs began to explore and eventually prototype the stacking of multiple silicon dice in a single-IC configuration as a way to go beyond the scaling limits of Moore's Law to create new levels of programmable systems integration.**

architecture will allow Xilinx to offer capacity that exceeds Moore's Law for future-generation products as well.

The Virtex-7 2000T device has been eagerly received by customers who have designed it into applications that include ASIC prototyping, storage and high-performance computing systems. These applications all require the highest capacity of programmable logic the industry can offer. However, Xilinx has also extended its 3D IC technology to enable another innovation a generation ahead of the competition, targeted for the highest-performance applications in the communications market.

In the summer of 2012, Xilinx announced the Virtex-7 H580T, the first of three heterogeneous All Programmable 3D ICs tailored for the communications market (see cover story, *Xcell Journal* issue 80). Where the Virtex-7 2000T is a homogeneous 3D IC in that all four of its dice/slices are primarily composed of FPGA logic, the Virtex-7 H580T is the first heterogeneous 3D IC.

To make the Virtex-7 H580T, Xilinx placed a dedicated 28G transceiver die alongside two FPGA dice on the passive silicon interposer. The result is a device that packs eight 28-Gbps transceivers, forty-eight 13.1-Gbps transceivers and 580k logic cells. For applications such as a 2x100G optical-transport line card based on a CFP2 optical module, the Virtex-7 H580T provides an astounding five-chip-to-one reduction in BOM and associated board area over previous implementations.

The Virtex-7 H580T is just the first heterogeneous 3D device Xilinx is delivering in its 28-nm family. The Virtex-7 H870T device comprises two eight-channel transceiver dice alongside three FPGA logic dice on a single chip, yielding a total of sixteen 28-Gbps transceivers, seventy-two 13.1-Gbps transceivers and 876,160 logic cells on one chip. The Virtex-7 H870T device is targeted at the next generation in wired communications—the 400G market. Xilinx's 3D IC technology will allow customers to begin development of 400G applications as the market begins to take shape and gain a significant market advantage—perhaps moving a generation ahead of their competitors.

The All Programmable 3D IC is yet another class of All Programmable device that has yet to see competition. Although one company recently announced it has produced a test chip with foundry TSMC, the company has yet to publicly announce the delivery of samples or production devices to customers.

BEYOND GLUE LOGIC

FPGAs have come a long way since Xilinx introduced the industry's first FPGA, the 1,000 ASIC-gate-equivalent XC2064, back in November of 1985. The earliest FPGAs, positioned as an alternative to gate arrays and ASICs, were primarily used as "glue logic" to facilitate communications between two devices not originally meant to talk to each other, or to add last-minute functionality errantly left off the larger ASIC.
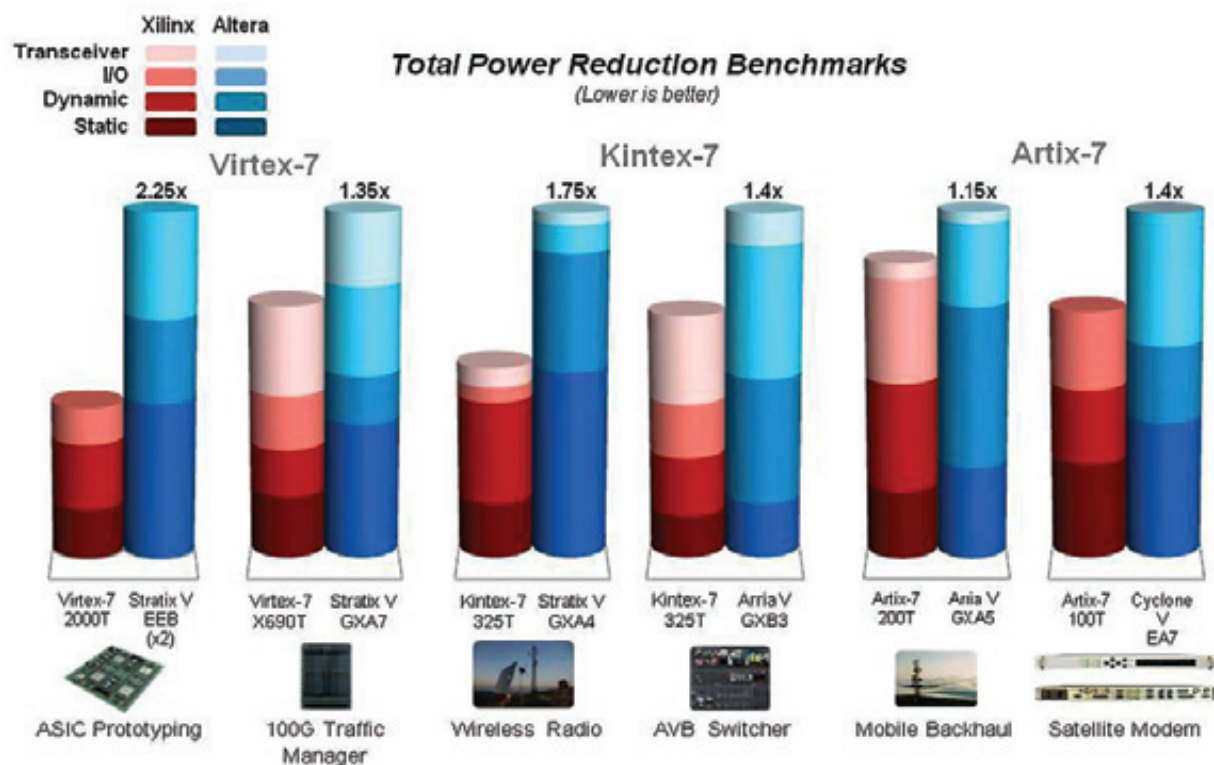
Figure 2 – Customer designs show 35 percent lower power on average vs. the competition at the same performance level.

Fast-forward to today and it is evident that modern devices have far outgrown the comparison to gate arrays. Today's All Programmable FPGAs include not only millions of gates of programmable logic, but also embedded memory controllers, high-speed I/O and, increasingly, analog/mixed-signal circuitry. With yesterday's FPGAs, design teams could fix a problem in their system or "glue" elements together. With today's All Programmable FPGAs, customers can create high-performance packet processing, waveform processing, image/video processing or high-performance computing functions that can be reprogrammed dynamically in the system or upgraded in the field.

Where Xilinx has yet to see competition in the All Programmable SoC and 3D IC device markets, to move a generation ahead in the traditional FPGA market could be considered an even more impressive achievement. To accomplish this goal, Xilinx set in motion a clear strategy to be first to 28 nm with FPGA silicon, with an expanded portfolio to cover low-end, midrange and high-end requirements. Xilinx also set goals to differentiate all 28-nm silicon with (1) a generation worth of advantages vs. the competition in system performance and integration, (2) an additional generation of power reduction, (3) a leapfrog in serdes with the lowest jitter and unmatched channel equalization and (4) a next-generation tool suite offering a step function in productivity and quality-of-result (QoR) advantages, and scalability for the future.

Xilinx in fact has already accomplished all of these tasks, delivering the first 28-nm-family device (which happens to be the first of the midrange devices), the Kintex™-7 K325T, in March of 2011. Xilinx was actually the first company in the entire semiconductor industry to tape out 28-nm silicon. Xilinx's decision to implement its entire 28-nm line of All Programmable devices on TSMC's HPL process, along with key architectural innovations targeting further power reduction, have enabled Xilinx to ship All Programmable FPGAs to customers today that deliver power savings of 35 to 50 percent over competing devices running at the same performance levels—that's a generation ahead in terms of power efficiency (Figure 2). Xilinx's 28-nm FPGAs also deliver unmatched performance and integration (Figure 3). The key contributors to system performance and levels of integration—including Block RAM, DSP, memory interfaces, transceivers and logic elements—all outperform the competition by 1.2x to 2x, with an average of 1.5x. Again, that's approximately a generation ahead.

What's more, Xilinx's All Programmable FPGAs contain features that competing FPGAs simply don't have. For example, all of Xilinx's 28-nm FPGAs include programmable

Figure 3 – Xilinx's 28-nm FPGAs have a generation-ahead performance and integration advantage over the competition. The company has delivered its All Programmable devices to specifications, with no errata to FPGA production devices.

Figure 4 – Xilinx's Vivado gives designers a state-of-the-art tool suite to vastly speed productivity and time-to-market.

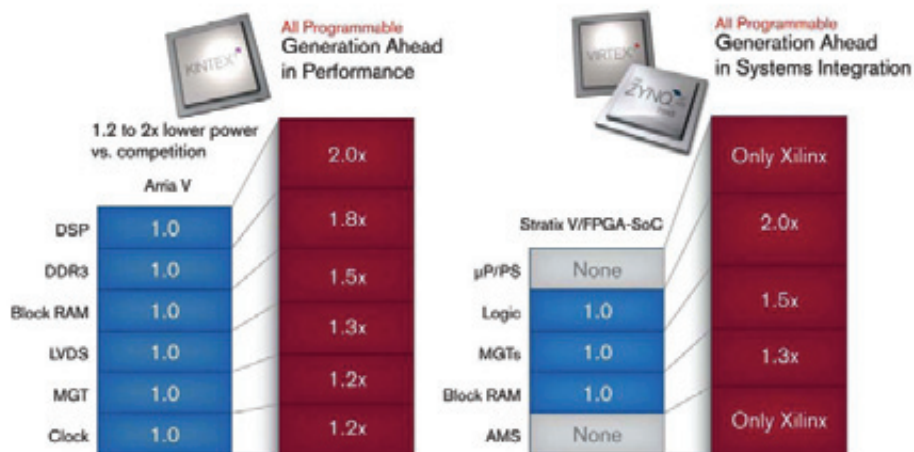QoR, leveraging state-of-the-art analytical engines and closure automation. To enable significant levels of integration, Vivado includes support for hierarchy, IP packaging and reuse, automated IP stitching and high-speed verification. To further speed time-to-market and raise the level of design abstraction, Vivado supports flows that start with C-based design and verification. It also leverages high-level synthesis and automated AXI interface generation to speed the time from C to RTL IP creation and integration. In this way, Vivado not only accelerates time to implementation, but time to C and RTL integration at the front end of a design.

Xilinx now develops "All" forms of programmable technologies—going well beyond programmable hardware to software, beyond digital to AMS and beyond single-die to multidice 3D IC implementations. Xilinx infuses these technologies into All Programmable FPGAs, SoCs and 3D ICs, enabling design teams to achieve greater degrees of programmable systems integration, increase overall system performance, reduce BOM costs and get more innovative products to market more quickly. The transformation of Xilinx's product portfolio can be traced back to 2008, with some innovations starting as far back as 2006. The result is a portfolio that today is delivering an extra generation of value for customers and moving Xilinx a generation ahead of its competition.

At the cusp of 20 nm, Xilinx is expanding on its leadership with even more advanced FPGAs, second-generation SoCs and 3D ICs, and the Vivado design system, enabling Xilinx to stay a generation ahead. Xilinx is benefiting from a multiyear head start in fine-tuning SoC and 3D IC technology with customers, redefining how to develop and deliver critical core technology such as high-speed serial transceivers, improving design methodologies and tools, expanding system-level ecosystems and supply chains, and assuring both quality and reliability.

analog/mixed-signal blocks that further reduce BOM costs by supporting the implementation of analog system functions within the FPGA rather than by means of external discrete analog devices.

PRODUCTIVITY WITH VIVADO

To improve designer productivity with its All Programmable devices at 28 nm and beyond, Xilinx also developed from the ground up a next-generation design environment and tool suite, Vivado™ (Figure 4). This development took more than four years by the calendar, and 500 man years of effort. Without this design suite, design teams could not effectively leverage Xilinx's 3D ICs. For FPGAs and SoCs, the Vivado Design Suite further improves the QoR of designs by up to three speed grades, cuts dynamic power by up to 50 percent, improves routability and resource utilization by over 20 percent, and speeds time to integration and implementation as much as fourfold. To learn more about Vivado, read the cover story in *Xcell Journal* issue 79.

Vivado is critical to enabling the new All Programmable portfolio of devices and levels of "programmable systems integration" they make possible. As a result, there is a focus that goes beyond accelerating implementation flows and

Image Sensor Color Calibration Using the Zynq-7000 All Programmable SoC

by Gabor Szedo
Staff Video Design Engineer
Xilinx Inc.
gabor.szedo@xilinx.com

Steve Elzinga
Video IP Design Engineer
Xilinx Inc.
steve.elzinga@xilinx.com

Greg Jewett
Video Marketing Manager
Xilinx Inc.
greg.jewett@xilinx.com

Xilinx image- and video-processing cores and kits provide the perfect prototyping platform for camera developers.

I mage sensors are used in a wide range of applications, from cell phones and video surveillance products to automobiles and missile systems. Almost all of these applications require white-balance correction (also referred to as color correction) in order to produce images with colors that appear correct to the human eye regardless of the type of illumination—daylight, incandescent, fluorescent and so on.

Implementing automatic white-balance correction in a programmable logic device such as a Xilinx® FPGA or Zynq™-7000 All Programmable SoC is likely to be a new challenge for many developers who have used ASIC or ASSP devices previously. Let's look at how software running on an embedded processor, such as an ARM9 processing system on the Zynq-7000 All Programmable SoC, can control custom image- and video-processing logic to perform real-time pixel-level color/white-balance correction.

To set the stage for how this is done, it's helpful to first examine some basic concepts of color perception and camera calibration.

CAMERA CALIBRATION

The measured color and intensity of reflections from a small, uniform surface element with no inherent light emission or opacity depend on three functions: the spectral power distribution of the illuminant, $I(\lambda)$; the spectral reflective properties of the surface material, $R(\lambda)$; and the spectral sensitivities of the imager, $S(\lambda)$.

The signal power measured by a detector can be expressed as:

$$P=\int_0^\infty I(\lambda)R(\lambda)S(\lambda)d\lambda$$

In order to get a color image, the human eye, as well as photographic and video equipment, uses multiple adjacent sensors with different spectral responses. Human vision relies on three types of light-sensitive cone cells to formulate color perception. In developing a color model based on human perception, the International Commission on Illumination (CIE) has defined a set of three color-matching functions, $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$. These can be thought of as the spectral sensitivity curves of three linear light detectors that yield the CIE XYZ tristimulus values P_x, P_y, and P_z, known collectively as the "CIE standard observer."

Digital image sensors predominantly use two methods to measure tristimulus values: a color filter array overlay above inherently monochromatic photodiodes; and stacked photodiodes that measure the absorption depth of photons, which is proportional to wavelength λ.

However, neither of these methods creates spectral responses similar to those of the human eye. As a result, color measurements between different photo detection and reproduction equipment will differ, as will measurements between image sensors and human observers when photographing the same scene—the same $(I\lambda)$ and $(R\lambda)$.

Thus, the purpose of camera calibration is to transform and correct the tristimulus values that a camera or image sensor measures, such that the spectral responses match those of the CIE standard observer.

WHITE BALANCE

You may view any object under various lighting conditions—for example, illuminated by natural sunlight, the light of a fire, fluorescent or incandescent bulbs. In all of these situations, human vision perceives the object as having the same color, a phenomenon called "chromatic adaptation" or "color constancy." However, a camera with no adjustment or automatic compensation for illuminants may register the color as varying. When a camera corrects for this situation, it is referred to as white-balance correction.

According to the top equation at the right of Figure 1, describing spectra of the illuminants, the reflective properties of objects in a scene and the spectral sensitivity of the detector all contribute to the resulting color measurement. Therefore, even with the same detectors, measurement results will mix information from innate object colors and the spectrum of the illuminant. White balancing, or the separation of innate reflective properties $R(\lambda)$ from the spectrum of the illuminant $I(\lambda)$, is possible only if:

• Some heuristics, e.g. the spatial frequency limits on the illuminant, or object colors are known a priori. For example, when photographing a scene with natural sunlight, it is expected that the spec-
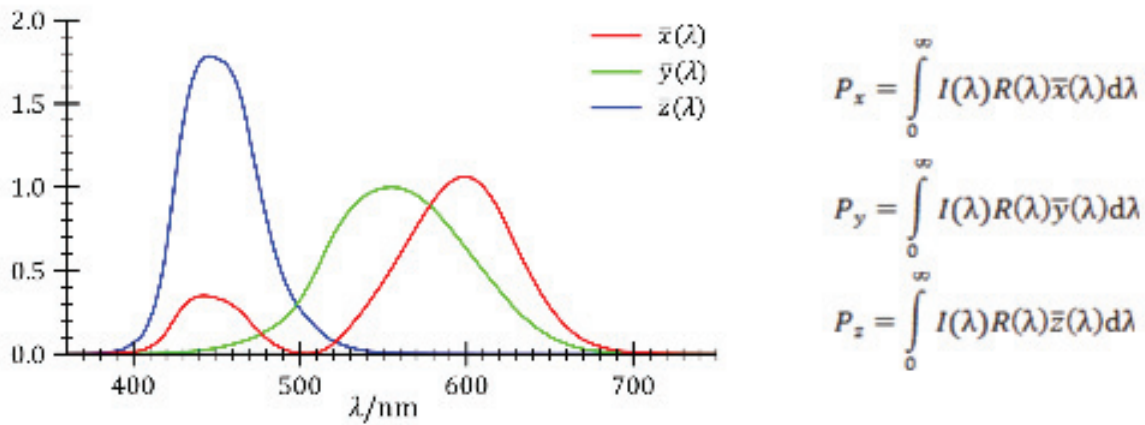
Figure 1 – Spectral responses of the "standard observer"

tral properties of the illuminant will remain constant over the entire image. Conversely, when an image is projected onto a white screen, spectral properties of the illuminant change dramatically from pixel to pixel, while the reflective properties of the scene (the canvas) remain constant. When both illuminant and reflective properties change abruptly, it is very difficult to isolate the scene's objects and illuminants.

- Detector sensitivity $S(\lambda)$ and the illuminant spectrum $I(\lambda)$ do not have zeros in the range of spectrum observed. You cannot gain any information about the reflective properties of objects outside the illuminant spectrum. For example, when a scene is illuminated by a monochromatic red source, a blue object will look just as black as a green one.

PRIOR METHODS

In digital imaging systems, the problem of camera calibration for a known illuminant can be represented as a discrete, three-dimensional vector function:

$$\underline{x}'=F(x)$$

where $F(x)$ is the mapping vector function and $\underline{x}$ is the discrete (typical-ly 8-, 10- or 12-bit) vector of R,G,B principal color components. Based on whether you are going to perform mapping linearly and whether color components are corrected independently, the mapping function can be categorized as shown in Table 1.

THE VON KRIES HYPOTHESIS

The simplest, and most widely used method for camera calibration is based on the von Kries Hypothesis [1], which aims to transform colors to the LMS color space, then performs correction using only three multipliers on a per-channel basis. The hypothesis rests on the assumption that color constancy in the human visual system can be achieved by individually adapting the gains of the three cone responses; the gains will depend on the sensory context, that is, the color history and surround. Cone responses from two radiant spectra, f_1 and f_2, can be matched by an appropriate choice of diagonal adaptation matrices D_1 and D_2 such

that $D_1\, S\, f_1 = D_2\, S\, f_2$, where S is the cone sensitivity matrix. In the LMS (long-, medium-, short-wave sensitive cone-response space),

$$D = D_1^{-1}D_2 = \begin{bmatrix} L_2/L_1 & 0 & 0 \\ 0 & M_2/M_1 & 0 \\ 0 & 0 & S_2/S_1 \end{bmatrix}$$

The advantage of this method is its relative simplicity and easy implementation with three parallel multipliers as part of either a digital image sensor or the image sensor pipeline (ISP):

$$\begin{bmatrix} L' \\ M' \\ S' \end{bmatrix} \quad \begin{bmatrix} k_L & 0 & 0 \\ 0 & k_M & 0 \\ 0 & 0 & k_S \end{bmatrix}\begin{bmatrix} L \\ M \\ S \end{bmatrix}$$

In a practical implementation, instead of using the LMS space, the RGB color space is used to adjust channel gains such that one color, typically white, is represented by equal R,G,B values. However, adjusting the perceived cone responses or R,G,B values for one color does not guarantee that other colors are represented faithfully.

	Linear	Nonlinear
Independent	von Kries	Component correction
Dependent	Color-correction matrix	Full lookup table

Table 1 – Camera calibration methods

COMPONENT CORRECTION

For any particular color component, the von Kries Hypothesis can only represent linear relationships between input and output. Assuming similar data representation (e.g. 8, 10 or 12 bits per component), unless k is 1.0, some of the output dynamic range is unused or some of the input values correspond to values that need to be clipped/clamped. Instead of multipliers, you can represent any function defining input/output mapping using small, component-based lookup tables. This way you can address sensor/display nonlinearity and gamma correction in one block. In an FPGA image-processing pipeline implementation, you can use the Xilinx Gamma Correction IP block to perform this operation.

FULL LOOKUP TABLE

Camera calibration assigns an expected value to all possible camera input tristimulus values. A brute-force approach to the problem is to use a large lookup table containing expected values for all possible input RGB values. This solution has two drawbacks. The first is memory size. For 10-bit components, the table is 2^{30} word (4 Gbytes) deep and 30 bits wide. The second problem is initialization values. Typically only a few dozen to a few hundred camera input/expected-value pairs are established via calibration measurements. The rest of the sparse lookup-table values have to be interpolated. This interpolation task is not trivial, as the heterogeneous component input-to-output functions are neither mono-tone nor smooth. Figure 2a presents the measured vs. expected-value pairs for R,G,B input (rows) and output (columns) values.

A visual evaluation of empirical results interpolated (Figure 2b) did not show significant quality improvement over a gamma-corrected, color-correction matrix-based solution. Most image- or video-processing systems are constrained on accessible bandwidth to external memory. The large size of the lookup table, which mandates external memory use; the significant bandwidth demand the per-pixel accesses pose; and the static nature of lookup-table contents (difficult to reprogram on a frame-by-frame basis) limit practical use of a full LUT-based solution in embedded video- and image-processing applications.

COLOR-CORRECTION MATRIX

The calibration method we describe in this article demonstrates how you can use a 3x3-matrix multiplier to perform a coordinate transformation aiming to orthogonalize measured red, green and blue components. The advantage of this method over the von Kries approach is that all three color channels are involved in the calibration process. For example, you can incorporate information from the red and blue channels when adjusting green-channel gains. Also, this solution lends itself well for camera calibration and white-balance correction to be performed simultaneously using the same module, updating matrix coefficients to match changing illuminants smoothly on a frame-by-frame basis.

The two simplest algorithms for white-balance correction—the Gray World and the White Point algorithms—use the RGB color space.

The Gray World algorithm [2] is based on the heuristics that although different objects in a scene have different, distinct colors, the average of scene colors (average of red, green and blue values) should result in a neutral, gray color. Consequently, the differences in R,G,B color values
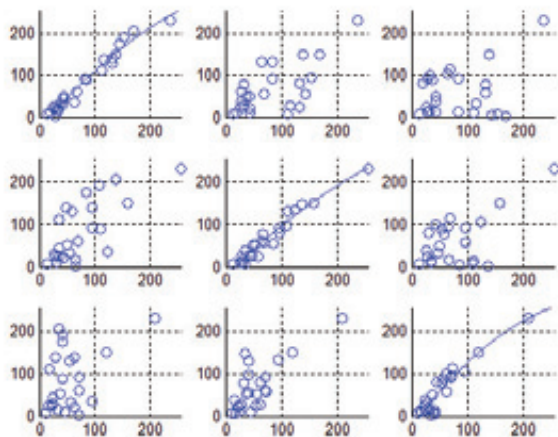

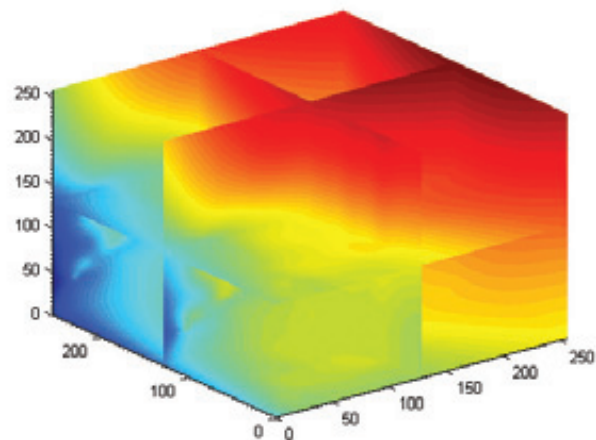Figure 2a – R,G,B measured vs. expected mapping values

Figure 2b – R component output as a function of R,G,B inputs
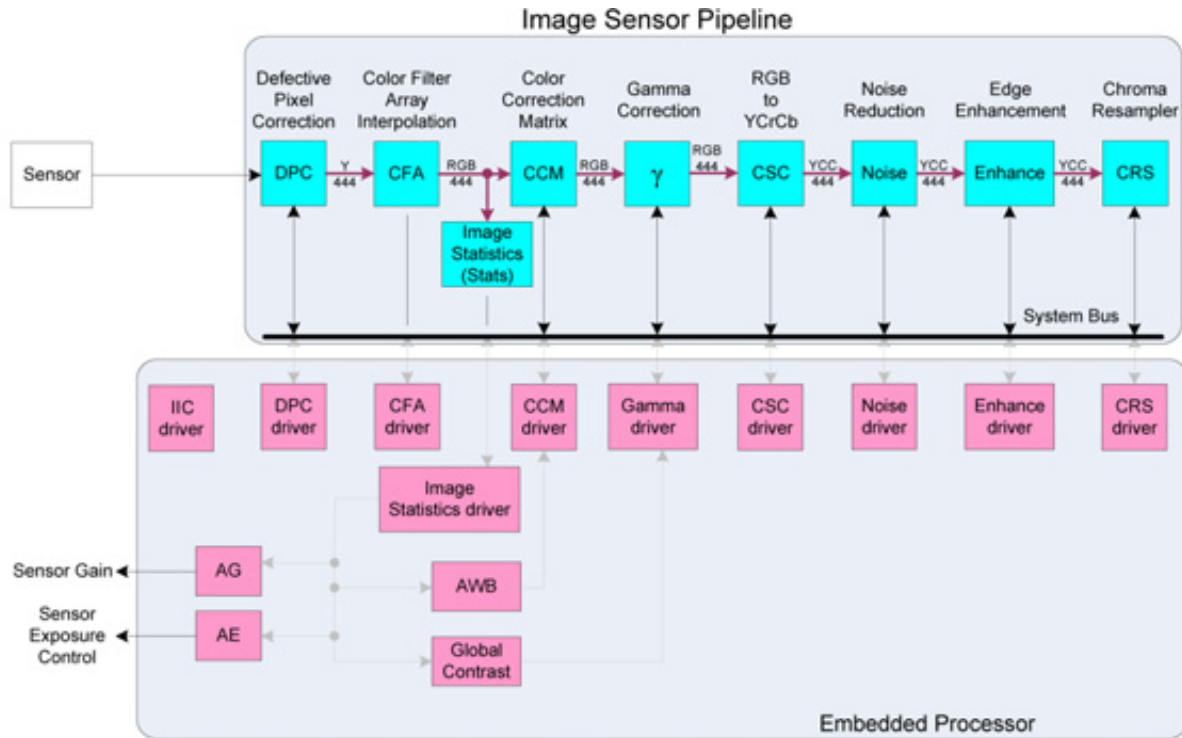
Image Sensor Pipeline

Figure 3 – Typical image sensor pipeline

averaged over a frame provide information about the illuminant color, and correction should transform colors such that the resulting color averages are identical. The Gray World algorithm is relatively easy to implement. However, it introduces large errors in which inherent scene colors may be removed or altered in the presence of large, vivid objects.

The White Point [2] algorithm is based on the assumption that the lightest pixels in an image must be white or light gray. The difference in red, green and blue channel maxima provides information about the illuminant color, and correction should transform colors such that the resulting color maxima are identical. However, to find the white point, it's necessary to rank pixels by luminance values. In addition, you may also have to perform spatiotemporal filtering of the ordered list to suppress noise artifacts and aggregate ranked results into a single, white color triplet. The advantage of using the White Point algorithm is easy implementation. The downside is that it too can introduce large errors and may

remove inherent scene colors. Also, the method is easily compromised by saturated pixels.

More-refined methods take advantage of color-space conversions, where hue can be easily isolated from color saturation and luminance, reducing three-dimensional color correction to a one-dimensional problem.

For example, color gamut mapping builds a two-dimensional histogram in the YCC, YUV, L*a*b* or Luv color spaces, and fits a convex hull around the base of the histogram. The UV or (Cr, Cb) averages are calculated and used to correct colors, such that the resulting color UV, or CbCr histograms are centered on the neutral, or gray point in the YUV, YCC, Luv or Lab space. The advantage of these methods is better color performance. The disadvantage is that implementation may require floating-point arithmetic.

All of the methods described above may suffer from artifacts due to incorrect exposure settings or extreme dynamic ranges in scene illumination. For example, saturated pixels in an image that's illuminated by a bright

light source with inherent hue, such as a candlelit picture with the flame in focus, may lead to fully saturated, white pixels present on the image.

OTHER WAYS TO IMPROVE WHITE-BALANCE RESULTS

Separating foreground and background is another approach to color correction. The autofocus logic, coupled to multizone metering, in digital cameras allows spatial distinction of pixels in focus around the center and the background around the edges. The assumption is that the objects photographed, with only a few dominant colors, are in focus at the center of the image. Objects in the distance are closer to the edge, where the Gray World hypothesis prevails.

Another technique centers on shape detection. Face or skin-color detection helps cameras identify image content with expected hues. In this case, white-balance correction can be limited to pixels with known, expected hues. Color correction will take place to move the colors of these pixels closer to the expected colors. The disad-

vantage of this method is the costly segmentation and recognition logic.

Most commercial applications combine multiple methods, using a strategy of adapting to image contents and photographic environment. [2]

ISPs FOR CAMERA CALIBRATION AND COLOR CORRECTION

Our implementation uses a typical image sensor pipeline, illustrated in Figure 3. We built the hardware components of the ISP (the blue blocks) with Xilinx image-processing cores using configurable logic. Meanwhile, we designed the camera calibration and white-balancing algorithms as C code (pink blocks) running on one of the embedded ARM processors. This same ARM processor runs embedded Linux to provide a user interface to a host PC.

The portion of the ISP relevant to white balancing and camera calibration is the feedback loop, including:

• The image statistics module, which gathers zone-based statistical data on a frame-by-frame basis;

• The embedded drivers and the application software, which analyzes the statistical information and programs the color-correction module on a frame-by-frame basis;

• The color-correction module, which performs color transformations on a pixel-by-pixel basis.

We implemented the ISP as part of the Zynq Video and Imaging Kit (ZVIK) 1080P60 Camera Image Processing Reference Design.

DETAILED ALGORITHM DESCRIPTION

In order to calibrate the colors of our sensor, we used an off-the-shelf color-viewing booth (X-Rite Macbeth Judge II), or light box, which has four standard illuminants with known spectra: simulated day-

light, cool-white fluorescent, warm fluorescent and incandescent. We also used an off-the-shelf color target (an X-Rite ColorChecker 24 Patch Classic) with color patches of known reflective properties and expected RGB and sRGB values.

To begin the process of implementing the camera-calibration algorithm, we first placed the color target in the light booth, flat against the gray background of the light box. We made sure to position the color target such that illumination from all light sources was as even as possible.

Next we captured the images taken by the sensor to be calibrated, with the all illuminants, with no color correction (using "bypass" color-correction settings: identity matrix loaded to the color-correction matrix).

We then used MATLAB® scripts available from Xilinx to assist with compensating for barrel (geometric) lens distortion and lens shading (light intensity dropping off toward the corners). The MATLAB script allows us to identify control points on the recorded images, then warps the image to compensate for barrel distortion. The rest of the script estimates horizontal and vertical light

drop-off using the background around the registered ColorChecker target.

In order to attenuate measurement noise, we identified rectangular zones within the color patches. Within these zones, we averaged (R,G,B) pixel data to represent each color patch with an RGB triplet. A MATLAB script with a GUI helps identify the patch centers and calculates averaged RGB triplets corresponding to the expected RGB values of each color patch (R_e, G_e, B_e).

We implemented the simulated annealing optimization method to identify color-correction coefficients and offsets. The measured uncalibrated (R,G,B) color triplets are transformed to corrected (R',G',B') triplets using the Color Correction module of Figure 3.

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} R_{offs} \\ G_{offs} \\ B_{offs} \end{bmatrix}$$

The simulated annealing algorithm minimizes an error function returning a scalar. In the following discussion (R_k, G_k, B_k) reference a subset or superset of measured color-patch pixel values. The user is free to limit the number of patches included in the optimization (subset), or include a particular patch multiple
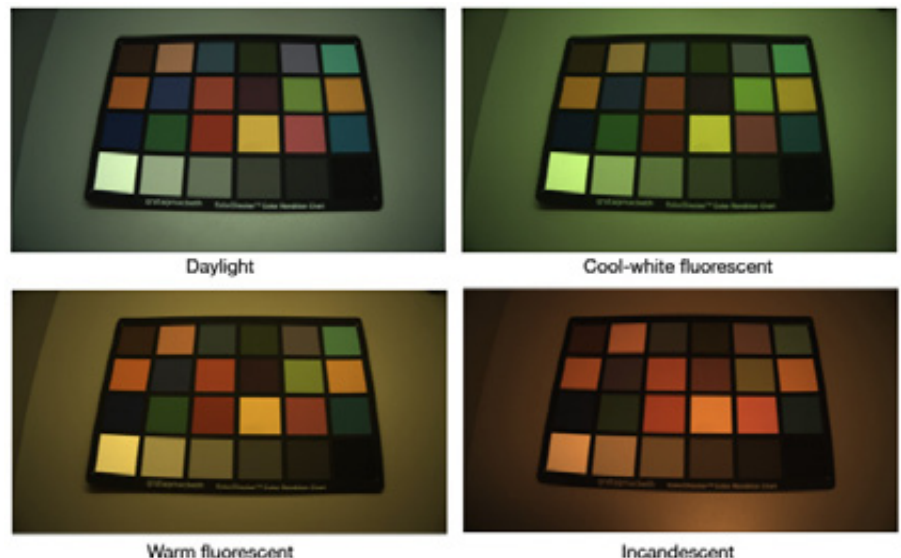

Figure 4 – Sensor images with different illuminants before lens correction

times, thereby increasing its relative weight during the optimization process. The integer n represents the number of color patches selected for inclusion in the optimization. If all patches are included exactly once in the optimization process, for the X-Rite Color-Checker 24 Patch Classic, $n=24$.

As the optimization algorithm has freedom to set 12 variables (CCM coefficients and offsets) only, typically no exact solution exists that maps all measured values to precisely the expected color patch values. However, the algorithm seeks to minimize an error function to provide optimal error distribution over the range of patches used.

We set up error functions that calculate one of the following:

- The sum of squared differences between expected and transformed triplets in the RGB color space:

$$E=\sum_{k=0}^{n} (R_k'-Re_k)^2 + (G_k'-Ge_k)^2 + B_k'-Be_k)^2$$

- The sum of absolute differences between expected and transformed triplets in the RGB color space:

$$E=\sum_{k=0}^{n} |R_k'-Re_k| + |G_k'-Ge_k| + |B_k'-Be_k|$$

- The sum of squared differences between expected and transformed triplets in the YUV color space:

$$E=\sum_{k=0}^{n} (U_k'-Ue_k)^2 + (V_k'-Ve_k)^2$$

- Or absolute differences between expected and transformed triplets in the YUV color space:

$$E=\sum_{k=0}^{n} |U_k'-Ue_k| + |V_k'-Ve_k|$$

where U' and V' correspond to R'G'B' values transformed to the YUV color space. Similarly, error functions can be set up to the L*u*v* or L*a*b* color spaces. You can use any of the above error functions in the simulated annealing minimization.

Figure 5 – Color-calibrated, lens-corrected images with different illuminants

WHITE BALANCING

Using the camera-calibration method above, we established four sets of color-correction coefficients and offsets, CCMk, k={1,2,3,4}, that result in optimal color representation assuming that the illuminant is correctly identified. The white-balancing algorithm, implemented in software running on the embedded processor, has to perform the following operations on a frame-by-frame basis. Using statistical information, it estimates the illuminant weights (W_k). Weights are low-pass filtered to compensate for sudden scene changes, resulting in illuminant probabilities (p_k). The color-correction matrix module is programmed with the combined CCM_k values according to weights p_k.

The advantage of this method is that a linear combination of calibration CCM_k values will limit color artifacts in case scene colors and illuminant colors are not properly separated. In the case of underwater photography, for example, where a strong blue tinge is present, a simple white-balancing algorithm such as Gray World would compensate to remove all blue, severely distorting the innate colors of the scene.

For all illuminants $k=\{1,2,3,4\}$ with different scene setups in the light booth, we also recorded the two-dimensional YUV histograms of the scenes by binning pixel values by chrominance, and weighing each pixel by its luminance value (luminance-weighed chrominance histogram). This method de-prioritizes dark pixels, or those in which a small difference in R,G,B values results in large noise in the chrominance domain.

Using a mask, we eliminated histogram bins which pertain to vivid colors that cannot possibly originate from a neutral (gray or white) object illuminated by a typical illuminant (Figure 6). A typical mask contains nonzero values only around the neutral (white) point, where most illuminants are located. We hard-coded the masked two-dimensional histogram values H_k (x,y), as well as the CCM_k values, into the white-
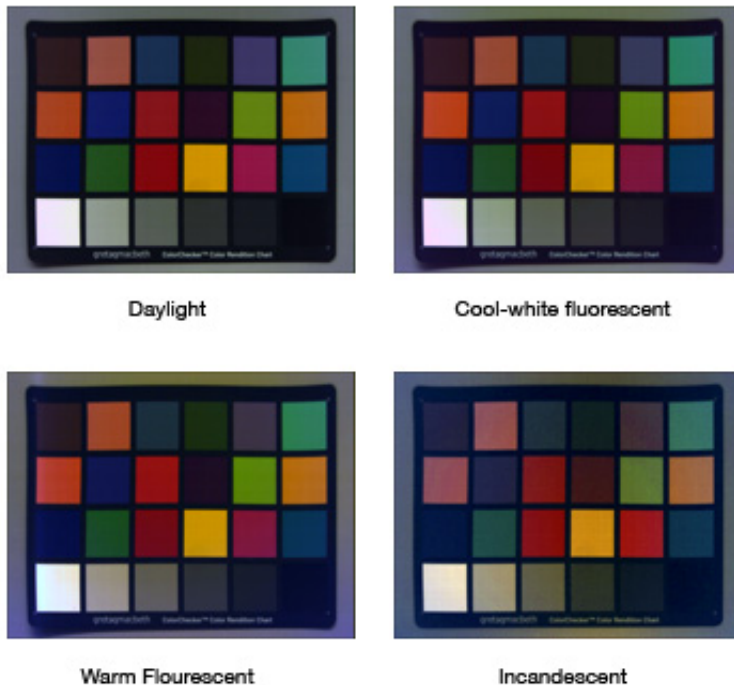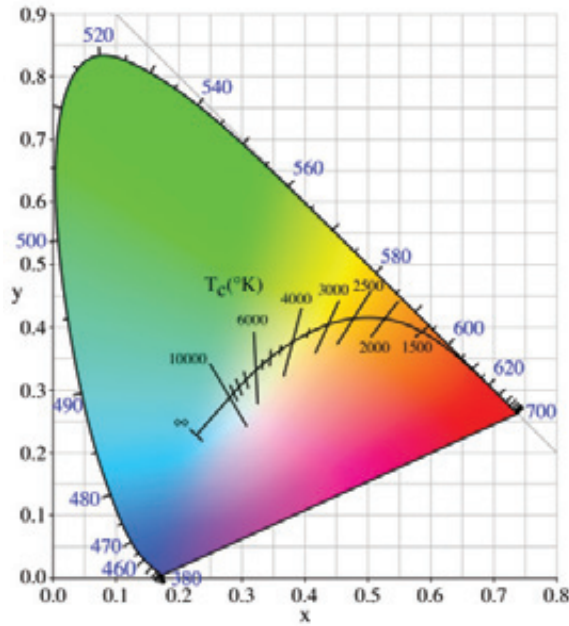
Figure 6 – Illuminants with different temperatures in CIE color space

balancing application running on the embedded processor.

During real-time operation, the white-balancing application collects similar two-dimensional, luminance-weighed chrominance histograms. The measured two-dimensional histograms are also masked, and the sum of absolute differences or sum of squared differences is calculated among the four stored histograms and the measured one:

$$D_k = \sum_{k=0}^{15} \sum_{k=0}^{15} \left(H_k(x,y) - H(x,y) \right)^2$$

where $H_k(x,y)$ are the precalculated reference two-dimensional histograms pertaining to known illuminants $\{k=1,2,3,4\}$, and $H(x,y)$ is the real-time histogram measurement.

Based on the measured histogram differences D_k, normalized similarity values are calculated using:

$$w_i = \frac{1/D_i}{\sum_{k=1}^{4} 1/D_k}$$

To avoid abrupt frame-by-frame tone changes, we smoothed normalized similarity values over time. We used a simple low-pass IIR filter, implementing

$$p_i = cw_i + (1-c)p_{i-1}$$

where $0 < c < 1$ controls the impulse response of the IIR filter. The smaller the values of c, the smoother the transitions. The larger the value, the quicker the filter responds to changes in lighting conditions.

Finally, we programmed the color-correction module of the ISP (Figure 3) with a linear combination of the precalculated color-correction coefficients and offsets (CCM_k):

$$CCM = \sum_{k=1}^{4} p_k CCM_k$$

Real-time white-balance implementation results (Figure 7) from a scene illuminated by both natural daylight and fluorescent light show significant improvement in perceived image quality and color representation.

The Zynq Video and Imaging Kit, along with MATLAB scripts available from Xilinx, complement and provide an implementation example for the algorithms we have presented.

Real-time color balancing is becoming increasingly challenging as resolutions and frame rates of industrial, consumer and automotive video applications improve. The algorithm we have described illustrates how software running on an embedded processor, such as the ARM9 cores of the Zynq processing platform, can control custom image- and video-processing logic performing pixel-level color correction. ⁂

References

1. H.Y. Chong, S.J. Gortler and T. Zickler, "The von Kries Hypothesis and Basis for Color Constancy," Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2007.

2. S. Bianco, F. Gasparini and R. Schettini, "Combining Strategies for White Balance," Proceedings of SPIE (2007), Volume 39, pages 65020D-65020D-9
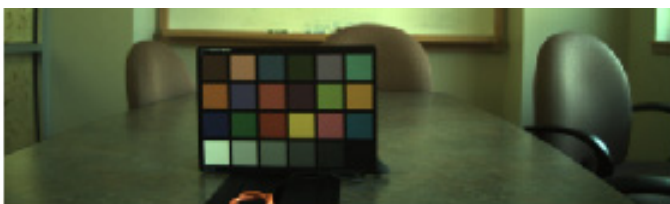
Figure 7 – Scene captured with no correction (left) and with white-balance correction

Testing and Debugging Zynq SoC Designs with BFMs

by **Martin Hofherr**
Design Engineer
Missing Link Electronics
martin.hofherr@missinglinkelectronics.com

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

AXI bus functional models make it easy to verify Zynq-7000 All Programmable SoC components and subsystems. Here's how to do it, with a step-by-step example based on the Xilinx tool chain.

The powerful dual-core ARM® Cortex™ A9 processors that Xilinx packs inside the Zynq™-7000 All Programmable SoC make it possible to build your own performance system-on-chip with this one piece of silicon. The software engineers among us will enjoy the rich software ecosystem of ARM, including GNU/Linux, while the hardware designers can add coprocessors and digital signal processing inside the programmable logic fabric. The challenge now lies in the verification and debugging of such a system-on-chip, especially at the interfaces between the hardware and the software—territory occupied by the device drivers and interrupt service routines, for example.

With the advent of ARM CPUs, the Advanced Microcontroller Bus Architecture (AMBA®) and, more importantly, the Advanced eXtensible Interface (AXI) have become the de facto standard for connecting components inside FPGAs. AXI is not a bus in the classical sense, but a transaction-based interconnect among multiple masters and multiple slaves using crossbars and arbitration. AXI comes in three flavors—AXI4 (regular), AXI4-Lite and AXI4-Stream. Xilinx® ships its ISE® 14.2 and Vivado™ 2012.2 design tool suites with v1.06.a AXI interconnect, compatible with AXI3 and AXI4.

Bus functional modeling (BFM) is an electronic-system-level verification methodology very suitable for system-on-chip designs. The value of BFM lies in abstracting the bus interconnect and providing a high-level API to implement testbenches that stimulate your RTL block, saving you precious time. BFM enables designers to verify blocks of RTL that interface with the Zynq-7000 device's processing system. Running inside an RTL simulator, such as Xilinx's ISim, the BFM enables you to verify one block at a time or many blocks together, thereby following a bottom-up design

flow. Xilinx and Cadence Design Systems have collaborated to provide a verification environment that builds on top of proven industry standards. The AXI BFM has been available for almost two years and was recently upgraded to version 2.1. [1]

Let's take a deeper look at this powerful means of system-level verification and the steps involved in adopting this methodology. First, we will list the tools and components you will need for a working BFM verification environment. This will include a list of documentation that helped us when we started using BFM. We will then describe a verification flow that uses the AXI BFM to verify your RTL blocks.

As we engineers learn best by example, finally we will go step-by-step into the details using the Xilinx ISE design environment. The example we picked—a simple design consisting of an AXI4 master and a memory controller with attached BRAM—can serve as a starting point for your next BFM verification project. We have made this example available to you for download from Missing Link Electronics' Developer Zone, at *http://www.missinglinkelectronics.com /devzone/axi-bfm*.

STEP SAVER

A BFM can significantly reduce the work required during the verification phase of your SoC design. The methodology basically enables you to directly connect your RTL block, as a so-called device under test (DUT), to the BFM and to stimulate and check back responses from your DUT at a high level of abstraction, without the need to delve into the details of the AXI interconnect. One key advantage of the AXI BFM from Xilinx and Cadence is that it relieves you from developing code that matches the AXI4-Lite IP Interface (IPIF). Neither will you need handwritten testbenches for your RTL block. The AXI BFM is nicely integrat-
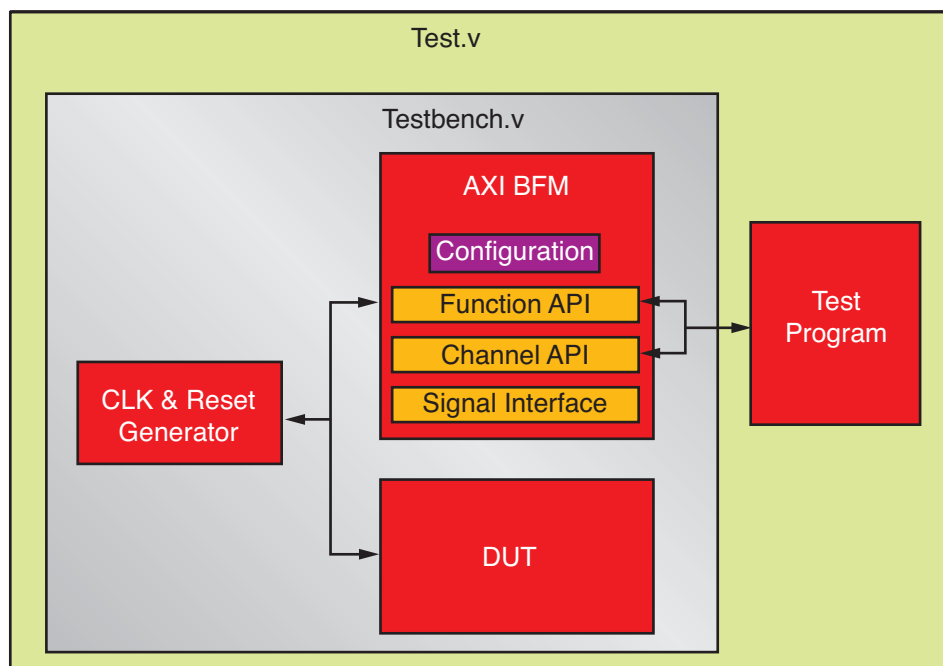
Figure 1 – The structure of the test system

ed into the FPGA design environment. You generate the top-level HDL for your embedded system and most of the necessary files for a BFM simulation project using Xilinx Platform Studio; XPS also relieves you of the task of wiring up your DUT properly. You can integrate your BFM project with Xilinx ISim using the "fuse.sh" script, which will compile the final test program.

The DUT consists of one or more VHDL or Verilog files that make up your RTL block. The AXI4 interface of your RTL block (which can be AXI4, AXI4-Stream or AXI4-Lite) is connected via the AXI BFM inside the Verilog "Testbench.v," which again is combined with your test program inside the top-level design file "Test.v." This test program is driving, and checking, your DUT. The good news: All the details of AXI4 are abstracted away. You can write your test program based on a very convenient and rich API that is divided into two levels of abstraction: the Function API and the Channel API.

While the language behind this API is Verilog HDL, you will quickly find out that this API is not only apt for HDL designers but is also very suitable for software engineers who are chartered with the development of firmware or drivers. And, even better: Because the AXI BFM fully understands the AXI4, AXI4-Lite and AXI4-Stream protocols, it performs additional checks and during simulation you may see warnings when your RTL block has AXI4 "misunderstandings." This process greatly helps you to become an AXI4 expert, too.

The AXI BFM environment is not limited to verifying one single RTL block but can handle many blocks quite efficiently. Therefore, we suggest that you use the AXI BFM to verify components, subsystems or even your entire system-on-chip and make this part of your regression testing as well. But be forewarned—as in all complex verification projects, it is easy to miss the forest for the trees. Check back with your colleagues (or yourself) to make sure that you are testing the same feature during simulation that you use in hardware.

AXI4 BFM AND XPS

While you can use the AXI BFM to verify the bus interface of single RTL block cores, it is as well suited to simulate the bus transactions for entire embedded designs. The AXI BFM provides models for AXI3, AXI4, AXI3-Lite, AXI4-Lite and AXI4-Streaming masters and slaves.

The different AXI interfaces are tailored to fit the needs of different types of RTL blocks. While the regular AXI3 and AXI4 interfaces are burst based and allow different data widths and out-of-order transactions, simpler RTL blocks that only need a register-style access from software can use the Lite versions of the protocol. Hardware that is used to process stream-oriented data, such as video data from a camera interface, often implements the AXI-Stream protocol, which is more suited to manage the peculiarities of streaming data than the regular AXI interfaces. In this example, however, we will focus on the regular AXI4 interface and the use of the bus functional model for this version of the bus. The workflow to test

RTL blocks with the other types of AXI interfaces from within Xilinx Platform Studio is exactly the same, and you can use this text as a guideline for these interfaces as well.

We will demonstrate how to set up the BFM from within XPS, so that you can effectively generate the HDL code that instantiates and interconnects the devices under test and the bus functional models. This approach minimizes the time-consuming act of writing interconnection HDL code by hand. This generated test system will consist of the peripherals to be tested and the BFMs that provide the bus stimulus. Thus, the master BFMs allow you to simulate the bus transfers that a bus master in the system would usually initiate, and to check the connected slave RTL block for correct behavior. On the other hand, slave BFMs are used to verify the correct behavior of RTL blocks with AXI master interfaces.

In the rest of this article, we will create a simple system containing one slave and an AXI master BFM that will provide the stimulus for that slave.

INSIDE THE AXI4 ARCHITECTURE
Before starting with the practical example, though, let's take a brief tour of the AXI4 bus architecture. For more-detailed information, see the AXI bus specification. [2] The AXI4 bus system is divided into five independent transaction channels: the write-address channel, write-data channel, write-response channel, read-address channel and read-data channel. Besides the actual source or destination address, the address channels convey information that describes the type of burst transfer that will take place on the associated data channel. This information comprises the number of transfers in the burst, the size of the data and the ID of the burst. For each burst, the slave sends a response to the master with the ID that was transferred through the address channel, and informs the master whether the transaction was successful or not.

For each of the five channels, a pair of handshake signals, READY and VALID, controls the actual data transfer. As the names of the two signals imply, the transmitting side asserts VALID when the data on the bus is valid and the receiving side asserts READY when it is prepared to receive data. The actual transaction takes place on the rising bus clock edge at which both READY and VALID are high. When you implement the access to the AXI bus by yourself, you must take into account that the assertion of the VALID signal must not depend on the READY signal, or a deadlock will occur.

CREATING A SIMPLE TEST SYSTEM IN XPS
Now that we have a basic understanding of the AXI4 bus system, we can begin to create our simple test system in XPS and see the BFMs and the AXI bus in action. The system will consist of one AXI4 Master BFM, a Block RAM controller, a Block RAM and an AXI interconnect that will link all the components. Here is a list of ingredients you will need for doing the AXI BFM example:

• Xilinx ISE 14.2, or newer, with XPS version 14.2

• Xilinx ISim (version 14.2)

• License key for the AXI BFM (part number DO-AXI-BFM)

• Xilinx DS824, "AXI Bus Functional Models v2.1" (replaces Xilinx UG783) [1]

• Xilinx DS768, "LogiCORE™ IP AXI Interconnect (v1.06.a)" [3]

• AXI BFM example project from *http://www.missinglinkelectronics.com/devzone/axi-bfm*

Just as if we were creating any new embedded design with XPS, we start a new blank project and name it "bfm_system." But instead of a processor like the MicroBlaze® or the Zynq-7000 All Programmable SoC, we will

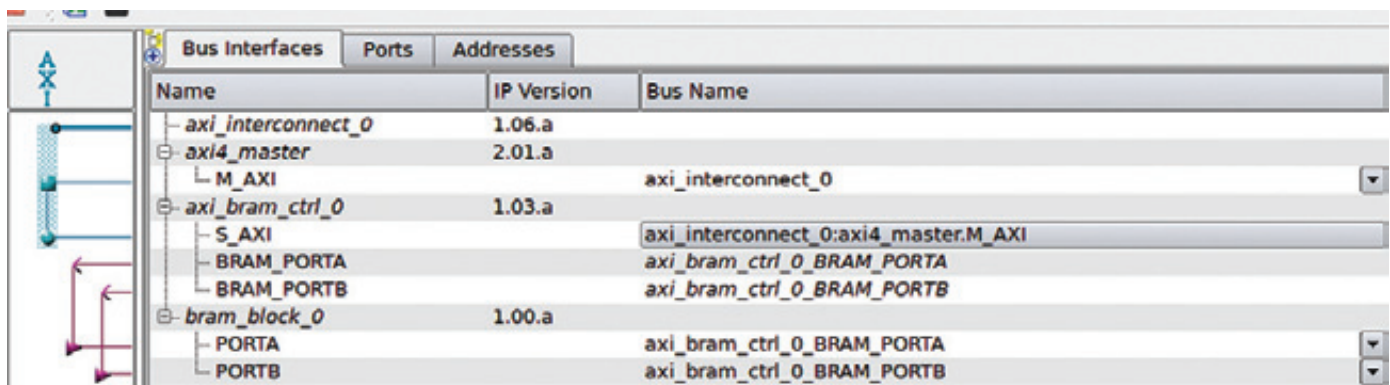| Name	IP Version	Bus Name
axi_interconnect_0	1.06.a	
axi4_master	2.01.a	
L M_AXI		axi_interconnect_0
axi_bram_ctrl_0	1.03.a	
S_AXI		axi_interconnect_0:axi4_master.M_AXI
BRAM_PORTA		axi_bram_ctrl_0_BRAM_PORTA
BRAM_PORTB		axi_bram_ctrl_0_BRAM_PORTB
bram_block_0	1.00.a	
PORTA		axi_bram_ctrl_0_BRAM_PORTA
PORTB		axi_bram_ctrl_0_BRAM_PORTB

Figure 2 – The Bus Interfaces tab of XPS showing the bus layout of the test system
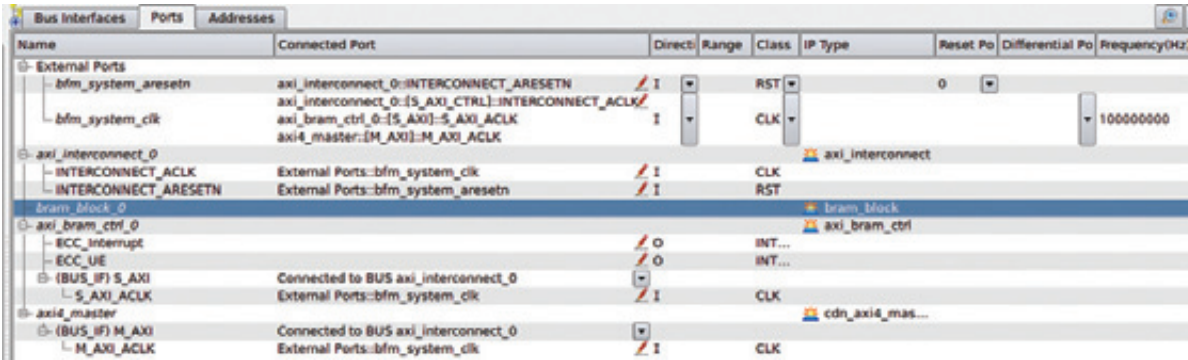
Figure 3 – The Ports tab of XPS showing the external ports for clock and reset

instantiate an AXI4 Master BFM that will initiate the transactions on the AXI4 bus. You can find the AXI4 Master BFM in the Verification node of the IP Core tree in XPS.

Next we will add an AXI4 interconnect from the "Bus and Bridge" category and an AXI BRAM controller and an associated Block RAM. You will find these two components in the "Memory and Memory Controller" category of the XPS IP tab. Interconnect the components in the Bus view tab of XPS such that the system looks like the one in Figure 2.

In the Ports tab of XPS, make the clock and reset port of the AXI interconnect external and connect the clock port of the BRAM controller and the Master BFM to the external clock ports as well. You can leave the ports "ECC_Interrupt" and "ECC_UE" of the AXI BRAM controller open, as we will not use the ECC feature. Set the frequency of the clock port to 100 MHz. The final system should look like the one in Figure 3.

As a next step, set the address range of the BRAM controller in the Addresses tab of XPS. This will determine the size of the BRAM block as well. Here we will create a 32k Block RAM starting at address 0x00000000.

Now that we have set up our basic system, we can let XPS generate the HDL code for it by clicking on the "Generate HDL Files" button in the tool bar on the left. This will create a directory called Simulation in the folder of your XPS project. If you want to make

changes on the system in XPS and regenerate the HDL, make sure to back up any files that you have added manually to this folder, as XPS will overwrite it completely when generating the simulation files. In the folder titled "behavioral" inside the simulation folder, we will find a Verilog or VHDL (depending on the preferred language in your project's settings) file named identically to our XPS design, "bfm_system.vhd." This is the top level of our system that contains all the instantiated components and the connections just made in XPS.

RUNNING THE SIMULATION

The data sheet of the AXI BFM suggests instantiating this system in a testbench that provides clock and reset signals, and starting the transactions on the AXI bus from a separate test module. (The structure of this approach is displayed in Figure 1.) You start the transactions on the bus by calling the respective Verilog tasks from the AXI BFM API. The API is layered into a channel level and a function level. The channel-level API allows you to initiate transfers on the different channels, like the read-

Figure 4 – Excerpt of bfm_system_test.v. You can call the functions and tasks of the BFM API by referring to the instance of the AXI BFM with help of the Verilog '.' operator.

Figure 5 – The master labels the last word in the burst through the assertion of **s_axi_wlast** (1, top circle in screen shot). The slave informs the master whether the burst was successful through the write-response channel (2 , bottom circle).

address and write-address channels, and thus gives you control over each phase of a write or read burst separately. For its part, the function-level API provides convenience functions to start entire data transfers, like a read or write burst. Furthermore, this API provides Verilog functions to change settings in the BFM, such as the verbosity of output on the ISim console. DS824 provides detailed information about the API along with a programmer's reference. [1]

You call the functions and tasks of this API from within the test program by referring to the BFM instance with the help of the Verilog "." operator. Figure 4 provides an example of such a function call. The full code is included in the project bundle accompanying this article, along with a shell script that takes care of compiling the example and running ISim to view the waveforms. Please consult the README.txt file in the project bundle for instructions on how to use this script. Once you execute the script, the ISim main window will show up. You can inspect the signals on the different channels of the AXI bus by loading the "axi_waveforms.wcfg" file in ISim and simulating the system for 15 microseconds.

Our test program will first initiate a write burst of two hundred fifty-six 32-bit words to the Block RAM and then read back the data. By searching the rising clock edge where, at the write-address channel, both the **s_axi_awready** and **s_axi_awvalid** signals are high, we can see where the information describing the burst that will follow is transmitted to the Block RAM controller. The master labels the last word in the burst by asserting the **s_axi_wlast** signal. Following the write burst, the slave—here, our Block RAM controller—signals that the transaction was successful through the write-response channel (Figure 5).

You can examine the same transaction structure for the read burst. The information describing the burst is transmitted over the read-address channel and the actual data over the read-data channel. In contrast to the write burst, there is no separate read-response channel. But the slave asserts for every word in the read transfer the read-response signal **s_axi_rresp**, indicating whether the current read was successful or not.

Of course, the bus transactions for our small example system with off-the-shelf components are all successful, but when developing RTL blocks, it's crucial to determine whether the RTL block adheres to the AXI standard. The BFMs provide protocol checking to make sure that the connected RTL block behaves correctly. So for example, the master BFM checks whether the connected RTL block applies the correct reset values to its output ports after reset. You can examine this action in the console tab of ISim. The protocol-checking API of the BFMs provides some more functionality. Again, the AXI Bus Functional Models product specification [1] provides an in-depth explanation.

POWERFUL TOOL

Xilinx and Cadence have provided Zynq-7000 All Programmable SoC designers with a very powerful tool for system-on-chip verification: bus functional models for AXI4. Early on during the design and ongoing during regression testing, we can verify our RTL blocks by writing test programs on top of a convenient Function API.

Missing Link Electronics develops solutions for embedded systems realization via prevalidated IP and expert application support, combining off-the-shelf FPGA devices with open-source software. MLE is a certified member of the Xilinx Alliance Program and an Early Access Partner of the Zynq-7000 series All Programmable SoC.

References

1. *Xilinx, DS824 AXI Bus Functional Models v2.1, April 24, 2012*

2. *ARM, IHI0022D AMBA AXI and ACE Protocol Specification, 2011*

3. *Xilinx, LogiCORE AXI Interconnect IP (v1.01.a), Dec. 14, 2010*

Floating-Point Design with Xilinx's Vivado HLS

by James Hrica
Senior Staff Software Applications Engineer
Xilinx, Inc.
jhrica@xilinx.com

The ability to easily implement floating-point arithmetic hardware from C/C++ source code on Xilinx FPGAs is a powerful feature of the Vivado HLS tool. However, floating-point math is not as straightforward as it might seem.

Most designers use fixed-point arithmetic logic to perform mathematical functions in their designs because the method is fast and very area-efficient. However, there are many cases where implementing mathematical calculations using a floating-point numerical format is the better choice. While fixed-point formats can achieve precise results, a given format has a very limited dynamic range, and so designers must perform a deep analysis in order to determine the bit-growth patterns throughout a complex design. And in implementing fixed-point formats, designers must also introduce many intermediate data types (of varying fixed-point formats) to achieve an optimal quality of results (QoR). Alternatively, floating-point formats represent real numbers in a much wider dynamic range, allowing designers to use a single data type through the long sequences of calculations that many algorithms require.

From a hardware design perspective, implementing a floating-point format manually comes at a cost. This format requires greater area to implement in a design and it increases latency, as the logic needed to implement a given arithmetic operation is considerably more complex than the logic required for implementing integer (fixed-point) arithmetic.

Luckily, a new tool from Xilinx called Vivado™ High-Level Synthesis (HLS) helps designers transform C/C++ design specifications into a register-transfer-level (RTL) implementation for designs that require floating-point calculations. Vivado HLS drastically reduces the design effort it takes to get floating-point algorithms implemented in hardware. While the basics of performing HLS on floating-point designs are reasonably straightforward, certain more-subtle aspects merit detailed explanation. Let's look at some of the topics, both basic and advanced, relating to design performance, area and verification of implementing floating-point logic in Xilinx® FPGAs using the Vivado HLS tool.

FLOAT AND DOUBLE

The Vivado HLS tool supports the C/C++ float and double data types, which are based on the single- and double-precision binary floating-point formats as defined by the IEEE-754 standard. [1] PG060, the Xilinx LogiCORE™ IP Floating-Point Operator v6.1 Product Guide [2], likewise offers a good summary of floating-point formats and arithmetic implementation. A very important consideration when designing with floating-point operations is that these numerical formats cannot represent every real number and therefore have limited precision.

This point is more subtle and complicated than it might first seem, and much has been written on this topic—designers are encouraged to peruse the references [3, 4, 5 and 6]. Generally speaking, you should not expect an exact match (at the binary representation level) for results of the same calculation accomplished

by different algorithms or even differing implementations (microarchitectures) of the same algorithm, even in a pure-software context. There are several sources for such mismatches, including the accumulation of a rounding error, which can be sensitive to the order in which operations are evaluated. Also, FPU support of extended precision can affect rounding of results. With the x87 80-bit format, for example, SIMD (SSE, etc.) instructions behave differently to x87. Moreover, many floating-point literal values can only be approximately represented, even for rational numbers. Other factors that can lead to mismatches include library function approximations, for example trigonometric functions, and constant propagation or folding effects.

Additionally, some floating-point arithmetic implementations support "subnormals" to represent numbers smaller than those the normal floating-point format can represent. For example, in the single-precision format, the smallest normal floating-point value is 2^{-126}. However, when subnormals are supported, the mantissa bits are used to represent a fixed-point number with a fixed exponent value of 2^{-127}.

Let's look at some simple software examples that validate the results of floating-point calculations.

Example 1 below demonstrates that different methods (and even what appears to be the same method) of doing the same calculation can lead to slightly different answers. Meanwhile, Example 2 helps illustrate that not all numbers, even whole (integer) values, have exact representations in binary floating-point formats.

EXAMPLE 1: DIFFERENT RESULTS FOR THE SAME CALCULATION

```
// Simple demo of floating point predictability problem
int main(void) {
  float fdelta = 0.1f; // Cannot be represented exactly
  float fsum = 0.0f;
  while (fsum < 1.0f)
    fsum += fdelta; float fprod = 10.0f * fdelta;
  double dprod = float(10.0f * fdelta);
  cout.precision(20);
  cout << "fsum: " << fsum << endl;
  cout << "fprod: " << fprod << endl;
  cout << "dprod: " << dprod << endl;
  return 0;

}
```
Program output:
fsum: 1.0000001192092895508
fprod: 1
dprod: 1.0000000149011611938

The first output result in Example 1 is the result of summing the approximation of 0.1 ten times, leading to the accumulation of rounding errors. On each iteration, we add the inexact single-precision value 0.1 to the running sum, which is then stored in a single-precision (32-bit) register. As the (base-2) exponent of the sum grows (from –4 to 0), rounding the intermediate sum occurs four times, regardless of the internal precision of the floating-point unit (FPU).

For the second value, the computation is carried out using the x87 extended precision and the result is rounded before being stored in single-precision format. For the third value, the multiplication is also done at extended precision, but it is rounded and stored in the double-precision format, leading to a different inaccuracy in the result.

Note: This code might produce different results from those shown when compiled for different machine architectures or with different compilers.

EXAMPLE 2: EVEN WHOLE NUMBERS CAN LOSE PRECISION

```
// Another demo of floating-point predictability problem
int main(void)
{
  int i;
  float delta = 1.0f;
  for (int i = 0; i < 100000000; i++) {
    float x = (float)i + delta;
    if (x / (float)i <= 1.0f) {
      // (x / i) should always be > 1.0
      printf("!!! ERROR on iteration #%d !!!\n", i + 1);
      return -1;
    }
  }
  return 0;
}
```
Program output:
!!! ERROR on iteration #16777217 !!!

This example shows that integer values above 16,777,216 (2^{24}) lose precision when stored in single-precision floating-point format. That's because the significand has 24 bits and beyond that point, least-significant bits of i must be dropped when cast to float format.

THE BASICS OF FLOATING-POINT DESIGN USING THE VIVADO HLS TOOL

Native support for HLS of the basic arithmetic operators (+, -, *, /), relational operators (==, !=, <, <=, >, >=) and format conversions (e.g., integer/fixed to float and float to double) is accomplished by mapping these operations onto Xilinx LogiCORE IP Floating-Point Operator cores instantiated in the resultant RTL. Additionally, calls to the sqrt()

Designers can use all floating-point operators and functions that HLS suppports in a fully pipelined context and produce one result per clock cycle, provided there is no feedback path through the operation.

family of functions (from the C/C++ standard math library), code of the form 1.0/x and 1.0/sqrt(x), are mapped onto appropriate Floating-Point Operator cores. Although the CORE Generator™ can build these cores for custom, precision floating-point types, the Vivado HLS tool generates only the single-and double-precision versions of the cores described by the IEEE-754 standard. A potential source of (small) differences in results generated by software vs. hardware based on the Floating-Point Operator cores is that these cores handle subnormal inputs by "flushing to zero"; that is, they are replaced by 0.0 when encountered. For details regarding the behavior of these cores, see PG060, LogiCORE IP Floating-Point Operator v6.1 Product Guide. [2]

The Vivado HLS tool also supports many of the other functions from the C(99)/C++ standard math library, for which there are no Floating-Point Operator cores available. For the complete lists of functions, see UG902, Vivado Design Suite User Guide: High-Level Synthesis v2012.2. [7] The approximation algorithms underlying many of these functions have been selected and optimized for implementation on Xilinx FPGAs and their accuracy characteristics might differ somewhat from those specified in software standards.

Vivado HLS only implements the standard math library functions as single- and double-precision floating-point operations in hardware. If the user calls any of these functions with integer or fixed-point arguments or return variables, the tool will implement format conversions (to or from floating point) where necessary and carry out the calculations in floating point.

Designers can use all floating-point operators and functions that HLS supports in a fully pipelined context and produce one result per clock cycle, provided there is no feedback path through the operation. Although the Vivado HLS tool can schedule these operations for infrequent, sequential execution, the implementation might not be the most area-efficient possible, especially with respect to flip-flop utilization. This applies even for "/" and sqrt(), for which low-throughput cores are available.

USING <MATH.H> IN ANSI/ISO-C BASED PROJECTS

To use the supported standard math library functions in ANSI/ISO-C based projects, you will need to include the math.h header file in all source files making calls to them. The base functions are intended to operate on (and return) dou-

ble-precision values—for example, double sqrt(double). Single-precision versions of most functions have an "f" appended to the function name, for example, float sqrtf(float), float sinf(float) and float ceilf(float). It is important to bear this in mind, because otherwise Vivado HLS will implement a much larger (in FPGA resources) double-precision version even though the argument and return variables are single-precision. What's more, there are format conversions that use additional resources and add latency to the calculation.

Another consideration when working in ANSI/ISO-C is that when compiling and running the code as software (including the C testbench side during RTL co-simulation), different algorithms are used than are implemented in the HLS-produced RTL. In the software, the GCC libc functions are called; on the hardware side, the Vivado HLS tool math library code is used. This can lead to bit-level mismatches between the two, when both results might be quite close to the real (in the analytical sense) answer.

EXAMPLE 3: UNINTENDED USE OF DOUBLE-PRECISION MATH FUNCTION

```
// Unintended consequences?
#include <math.h>
float top_level(float inval)
{
  // double-precision natural logarithm
  return log(inval);
}
```

This example leads to an RTL implementation that converts the input into double-precision format, calculates the natural logarithm at double-precision, then converts the result to single-precision for output.

EXAMPLE 4: EXPLICIT USE OF SINGLE-PRECISION MATH FUNCTION

```
// Be sure to use the right version of math functions...
#include <math.h>
float top_level(float inval)
{
  // single-precision natural logarithm
  return logf(inval);
}
```

Because the single-precision version of the logarithm function is called, that version is implemented in the RTL with no need for input/output format conversions.

USING <CMATH> IN C++ PROJECTS

When designing with C++, the most straightforward way to get support for the standard math library is to include the <cmath> system header file in all source files that call its functions. This header file provides versions of the base (double-precision) functions overloaded to take as arguments and returns single-precision (float) values in the std namespace. To use the single-precision version, the std namespace must be in scope, either by using the scope resolution operator (::) or by importing the entire namespace with the using directive.

EXAMPLE 5: EXPLICIT SCOPE RESOLUTION

```
#include <cmath>
float top_level(float inval)
{
   // single-precision natural logarithm
   return std::log(inval);
}
```

EXAMPLE 6: EXPOSING CONTENTS OF A NAMESPACE TO FILE SCOPE

```
#include <cmath>
using namespace std;
float top_level(float inval)
{
   // single-precision natural logarithm
   return log(inval);
}
```

As with the usage of <math.h> in ANSI/ISO-C projects, when using functions from <cmath> in code that the Vivado HLS tool is to synthesize, results might differ between that code run as software vs. the RTL implementation, because different approximation algorithms are used. For this reason, the Vivado HLS programming environment provides access to the algorithms used to synthesize the RTL for use in C++ modeling.

When validating changes to C++ code destined for HLS and later co-simulating the resultant RTL with a C++ based testbench, Xilinx recommends that the HLS sources use the same math library calls and that the testbench code uses the C++ standard library to generate reference values. This provides an added level of verification of the HLS models and math libraries during development.

To follow this methodology, you should include the <hls_math.h> Vivado HLS tool header only in any source files to be synthesized to RTL. For source files involved only in validating the HLS design, such as the test program and supporting code, include the <cmath> system header file. The HLS version of the functions in the <hls_math.h> header file is part of the hls:: namespace. For the HLS versions to be compiled for software modeling and validation, use hls:: scope resolution for each function call.

Note: It is not a good idea to import the hls:: namespace (via "using namespace hls") when using C++ standard math library calls, because this can lead to compilation errors during HLS. Example 7a illustrates this usage.

EXAMPLE 7A: TEST PROGRAM USES STANDARD C++ MATH LIBRARY

```
// Contents of main.cpp - The C++ test bench
#include <iostream>
#include <cmath>
using namespace std;
extern float hw_cos_top(float);
int main(void)
{
   int mismatches = 0;
   for (int i = 0; i < 64; i++) {
      float test_val = float(i) * M_PI / 64.0f;
      float sw_result = cos(test_val); //float
      std::cos(float)
         float hw_result = hw_cos_top(test_val);
      if (sw_result != hw_result) {
         mismatches++;
         cout << "!!! Mismatch on iteration #" << i;
         cout << " -- Expected: " << sw_result;
         cout << "\t Got: " << hw_result;
         cout << "\t Delta: " << hw_result - sw_result;
         cout << endl;
      }
   }
   return mismatches;
}
```

EXAMPLE 7B: THE HLS DESIGN CODE USES THE hls_math LIBRARY

```
// Contents of hw_cos.cpp
#include <hls_math.h>
float hw_cos_top(float x)
{
   // hls::cos for both C++ model and RTL co-sim
   return hls::cos(x);
}
```

When this code is compiled and run as software (e.g., "Run C/C++ Project" in the Vivado HLS GUI), the results returned by hw_cos_top() are the same values as those the HLS-gener-

Because floating-point operations use considerable resources relative to integer or fixed-point operations, the Vivado HLS tool utilizes those resources as efficiently as possible.

ated RTL produces, and the program tests for mismatches against the software reference model—that is, std::cos(). If you had included <cmath> in hw_cos.cpp instead, there would be no mismatches when the C/C++ project was compiled and run as software, but there would be during RTL co-simulation.

It is important not to assume that the Vivado HLS tool makes optimizations that seem obvious and trivial to human eyes. As is the case with most C/C++ software compilers, expressions involving floating-point literals (numeric constants) might not be optimized during HLS. Consider the following example code.

EXAMPLE 8: ALGEBRAICALLY IDENTICAL; VERY DIFFERENT HLS IMPLEMENTATIONS

```
// 3 different results
void top(float *r0, float *r1, float *r2, float inval)
{
  // double-precision multiplier & conversions
  *r0 = 0.1 * inval;
  // single-precision multiplier
  *r1 = 0.1f * inval;
  // single-precision divider
  *r2 = inval / 10.0f;
}
```

If this function is synthesized to RTL, three very different circuits result for computing each of r0, r1 and r2. By the rules of C/C++, the literal value 0.1 signifies a double-precision number that cannot be represented exactly. Therefore, the tool instantiates a double-precision (double) multiplier core, along with cores to convert inval to double and the product back to float (*r0's type). When you want a single-precision (float) constant, append an f to the literal value, for example, 0.1f. Therefore, the value of r1 above is the result of a single-precision multiplication between the (inexact) float representation of 0.100 and inval. Finally, r2 is produced by a single-precision division core, with inval as the numerator and 10.0f as the denominator. The real value 10 is represented exactly in binary floating-point formats. Therefore, depending on the value of inval, the calculation r2 might be exact, whereas neither r0 nor r1 is likely to be exact.

Because the order in which floating-point operations occur potentially impacts the result (for example, due to rounding at different times), multiple floating-point literals involved in an expression might not be folded together.

EXAMPLE 9: ORDER OF OPERATIONS CAN IMPACT CONSTANT FOLDING

```
// very different implementations
void top(float *r0, float *r1, float inval)
{
  // *r0 = inval; constants eliminated
  *r0 = 0.1f * 10.0f * inval;
  // two double-precision multiplies
  *r1 = 0.1f * inval * 10.0f;
}
```

In this example, because of the order of evaluation of the expression assigned to r0, the compiler recognizes the entire expression to be identity and does not generate any hardware. However, the same does not apply to r1; two multiplications are done.

EXAMPLE 10: AVOID FLOATING-POINT LITERALS IN INTEGER EXPRESSIONS

```
void top(int *r0, int *r1, int inval)
{
  *r0 = 0.5 * inval;
  *r1 = inval / 2;
}
```

For this example, HLS implements the logic to assign r0 by converting inval to double-precision format in order to multiply it by 0.5 (a double-precision literal value) and then convert it back to an integer. On the other hand, HLS optimizes multiplication and division by integer powers of two into left- and right-shift operations respectively, and implements them in hardware as simple wire selections (with zero-padding or sign-extension as appropriate for the direction and type of the operand). Therefore, the logic created to assign r1 is much more efficient while achieving the same arithmetic result.

PARALLELISM, CONCURRENCY AND RESOURCE SHARING

Because floating-point operations use considerable resources relative to integer or fixed-point operations, the Vivado HLS tool utilizes those resources as efficiently as possible. The tool will often share Floating-Point Operator cores among multiple calls to the source operation when data dependencies and constraints allow. To illustrate this concept, we sum four float values in the following example.

EXAMPLE 11: MULTIPLE OPERATIONS USE SINGLE CORE

```
// How many adder cores?
void top (float *r, float a, float b, float c, float d)
{
    *r = a + b + c + d;
}
```


Figure 1 – This Vivado HLS report shows a single adder core instantiated.
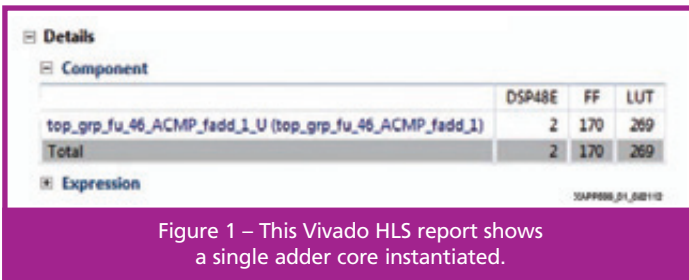
Sometimes, when data bandwidth allows, it might be desirable to perform more work in a given time by doing many operations concurrently that would otherwise be scheduled sequentially. In the following example, the RTL that the tool has created generates values in the result array by summing the elements of two source arrays in a pipelined loop. Vivado HLS maps top-level array arguments to memory interfaces, thus limiting the number of accesses per cycle (for example, two per cycle for dual-port RAM, one per cycle for FIFO, etc.).

EXAMPLE 12: INDEPENDENT SUMS

```
// Independent sums, but I/O only allows
// throughput of one result per cycle
void top (float r0[32], float a[32], float b[32])
{
#pragma HLS interface ap_fifo port=a,b,r0
  for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
    r0[i] = a[i] + b[i];
  }
}
```

By default, the Vivado HLS tool schedules this loop to iterate 32 times and to implement a single adder core, provided input data is available continuously and the output FIFO never gets full. The resulting RTL block requires 32 cycles, plus a few to flush the adder's pipeline. This is essentially as fast as possible, given the I/O data rate. If on the other hand the data rates are increased, then designers can use HLS techniques to increase the processing rate as well. Extending the previous example, you can increase the I/O bandwidth by doubling the width of the interfaces, using the Vivado HLS tool's array reshape directive. To increase the processing rate, partially unroll the loop by a factor of two to match the increase in bandwidth.

EXAMPLE 13: INDEPENDENT SUMS

```
// Independent sums, with increased I/O
// bandwidth -> high throughput and area
void top (float r0[32], float a[32], float b[32])
{
#pragma HLS interface ap_fifo port=a,b,r0
#pragma HLS array_reshape cyclic factor=2 \
  variable=a,b,r0
  for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
#pragma HLS unroll factor=2
    r0[i] = a[i] + b[i];
  }
}
```

With these added directives, the Vivado HLS tool synthesizes RTL that has two adder pipelines working concurrently over half as many iterations to produce two output samples per iteration. This is possible because each calculation is completely independent and the order in which the addition operations occur cannot affect the accuracy of the results. However, when more-complex calculations occur, perhaps as a result of a chain of dependent floating-point operations, the Vivado HLS tool cannot rearrange the order of those operations. The result can be less concurrency or sharing than expected. Furthermore, when there is feedback or recurrence in a pipelined datapath, increasing design throughput via concurrency might require some manual restructuring of the source code.

Example 14 presents a detailed instance of how the Vivado HLS tool deals with feedback and recurrence through floating-point operations. The following code involves a floating-point accumulation in a pipelined region.

EXAMPLE 14: DEPENDENCY THROUGH AN OPERATION

```
// Floating-point accumulator
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
  float acc = 0;
  for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
    acc += x[i];
  }
  return acc;
}
```

Because this form of accumulation results in a recurrence and the latency for floating-point addition is generally greater than one cycle, this pipeline cannot achieve a throughput of one accumulation per cycle.

For example, if the floating-point adder has a latency of four cycles, the pipeline initiation interval is also four cycles (as seen in the Vivado HLS tool's synthesis report shown in Figure 2), due to the dependency requiring that each accumulation complete before another can start. Therefore, the best throughput achievable in this example is one accumulation every four cycles. The accumulation loop iterates 32 times, taking four cycles per trip, leading to a total of 128 cycles plus a few to flush the pipeline.

A higher-performance alternative might be to interleave four partial accumulations onto the same adder core, each completing every four cycles, thereby reducing the time it takes to accomplish the 32 addition operations. However,

Figure 2 – In this Vivado HLS report, the notation "Pipeline II = 4" indicates a bottleneck.

the Vivado HLS tool cannot infer such an optimization from the code provided in Example 14, because it requires a change to the order of operations for the accumulation. If each partial accumulation takes every fourth element of x[] as input, the order of the individual sums changes, which could lead to a different result.

You can work around this limitation by making minor modifications to the source code to make your intent more explicit. The following example code introduces an array, acc_part[4], to store the partial sums; these are subsequently summed, and the main accumulation loop is partially unrolled.

EXAMPLE 15: EXPLICIT REORDERING OF OPERATIONS FOR BETTER PERFORMANCE

```
// Floating-point accumulator
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
  float acc_part[4] = {0.0f, 0.0f, 0.0f, 0.0f};
  // Manually unroll by 4
  for (int i = 0; i < 32; i += 4) {
```

```
    // Partial accumulations
    for (int j = 0; j < 4; j++) {
#pragma HLS pipeline
      acc_part[j] += x[i + j];
    }
    // Final accumulation
    for (int i = 1; i < 4; i++) {
#pragma HLS unroll
      acc_part[0] += acc_part[i];
    }
  }
  return acc_part[0];
}
```

With this code structure, the Vivado HLS tool recognizes that it can schedule the four partial accumulations onto a single adder core on alternating cycles, which is a more efficient use of resources (see Figure 3). The subsequent final accumulation might also use the same adder core, depending on other factors. Now the main accumulation loop takes eight iterations (32/4), each taking four cycles to produce the four partial accumulations. The same amount of work takes place in much less time with a small increase in FPGA resources. The final accumulation loop, while using the same adder core, adds extra cycles, but the number is fixed and small relative to the savings from unrolling the main accumulation loop, especially when the data set is large. We could further optimize this final accumulation step but with diminishing returns relative to performance vs. area.

Figure 3 – In this report, Pipeline II is still 4, but the trip count drops from 32 to 8.

When greater I/O bandwidth is available, we can specify larger unrolling factors in order to bring more arithmetic cores to bear. If in the previous example, two x[] elements were available per clock cycle, we could increase the unrolling factor to 8, in which case two adder cores would be implemented and eight partial accumulations done per cycle. The target device selection and user timing con-

straints can have an impact on the exact operator latency. Generally, it is necessary to run HLS and do some performance analysis of a simpler base case (e.g., Example 14) to determine the optimal amount of unrolling.

CONTROLLING IMPLEMENTATION RESOURCES

The Xilinx LogiCORE IP Floating-Point Operator cores allow control over the DSP48 utilization of some of the supported operations. For example, the multiplier core has four variants that trade logic (LUT) resources for DSP48 usage. Normally, the Vivado HLS tool automatically determines which type of core to use based on performance constraints. Designers can use the Vivado HLS tool's RESOURCE directive to override automatic selection and specify which type of Floating-Point Operator cores to use for a given instance of an operation. For example, for the code presented in Example 14, the adder would generally be implemented employing the "full-usage" core, using two DSP48E1 resources on a Kintex™-7 FPGA, as shown in the "Component" section of the synthesis report (see Figure 4).
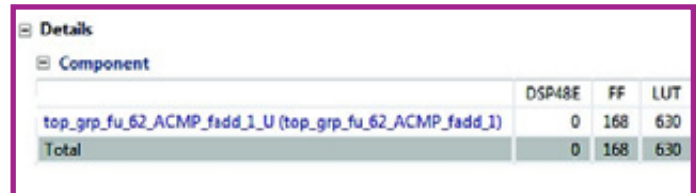

Figure 4 – In this Vivado HLS report, the default adder core uses two DSP48E resources.

The following example code forces the addition operation to be mapped onto an FAddSub_nodsp core, as shown in Figure 5.

EXAMPLE 16: USING THE 'RESOURCE' DIRECTIVE TO SPECIFY FLOATING-POINT OPERATOR CORES VARIANT

```
// Floating-point accumulator
float top(float x[32])
{
#pragma HLS interface ap_fifo port=x
  float acc = 0;
  for (int i = 0; i < 32; i++) {
#pragma HLS pipeline
#pragma HLS resource variable=acc \
    core=FAddSub_nodsp
    acc += x[i];
  }
  return acc;
}
```


Figure 5 – Now the adder uses no DSP48E resources, as the report shows.

See UG902, the Vivado HLS user guide [7], for full details on the usage of the RESOURCE directive and also for a list of available cores.

VALIDATING THE RESULTS OF FLOATING-POINT CALCULATIONS

There are many reasons to expect bit-level (or greater) mismatches between floating-point results of the same calculation done by different means. Mismatches can occur in a number of places, including different approximation algorithms, reordering of operations leading to rounding differences and the handling of subnormal (which the Floating-Point Operator cores flush to zero).

In general, the result of comparing two floating-point values, especially for equality, can be misleading. The two values being compared might differ by just one "unit in the last place" (ULP; the least-significant bit in their binary format), which can represent an extremely small relative error, yet the "==" operator returns false. For example, using the single-precision float format, if both operands are non-zero (and non-subnormal) values, a difference of 1 ULP represents a relative error on the order of 0.00001 percent. For this reason, it is good practice to avoid using the "==" and "!=" operators to compare floating-point numbers. Instead, check that the values are "close enough," perhaps by introducing an acceptable-error threshold.

For most cases, setting an acceptable ULP or relative-error level works well and is preferable to absolute-error (or "epsilon") thresholds. However, when one of the values being compared is exactly zero (0.0), this method breaks down. If one of the values you are comparing has or can take on a constant-zero value, then you should use an absolute-error threshold instead. The following example code presents a method you can use to compare two floating-point numbers for approximate equality, allowing you to set both ULP and absolute-error limits. This function is intended for use in the C/C++ "testbench" code for validating modifications to HLS source code and verifying during the Vivado HLS tool's RTL co-simulation. You can also use similar techniques in code destined for HLS implementation as well.

EXAMPLE 17: C CODE TO TEST FLOATING-POINT VALUES FOR APPROXIMATE EQUIVALENCE

```
// Create a union based type for easy access to
// binary representation
typedef union {
  float fval;
  unsigned int rawbits;
} float_union_t;
bool approx_eqf(
    float x, float y,
    int ulp_err_lim, float abs_err_lim
    )
{
  float_union_t lx, ly;
  lx.fval = x;
  ly.fval = y;
  // ULP based comparison is likely to be meaningless
  // when x or y is exactly zero or their signs
  // differ, so test against an absolute error
  // threshold this test also handles (-0.0 == +0.0),
  // which should return true. N.B. that the
  // abs_err_lim must be chosen wisely, based on
  // knowledge of calculations/algorithms that lead
  // up to the comparison. There is no substitute for
  // proper error analysis when accuracy of results
  // matter.
  if (((x == 0.0f) ^ (y == 0.0f)) ||
      (__signbit(x) != __signbit(y))) {
    return fabs(x - y) <= fabs(abs_err_lim);
  }
  // Do ULP base comparison for all other cases
  return abs((int)lx.rawbits -
      (int)ly.rawbits) <= ulp_err_lim;
}
```

There is no single answer as to at what level to set the ULP and absolute-error thresholds, since the settings will vary depending upon the design. A complex algorithm can have the potential to accumulate many ULPs of inaccuracy in its output, relative to a reference result. Other relational operators can also be prone to misleading results. For example, when testing whether one value is less (or greater) than another and the difference is only a few ULPs, is it reasonable to resolve the comparison in this case? We could use the function presented above in conjunction with a less-than/greater-than comparison to flag results that might be ambiguous.

POWERFUL FEATURE

The ability to easily implement floating-point arithmetic hardware (RTL code) from C/C++ source code on Xilinx FPGAs is a powerful feature of the Vivado HLS tool. However, use of floating-point math is not as straightforward as it might seem, whether from a software, hardware or mixed perspective. The imprecise nature of the IEEE-754 standard binary floating-point formats can make it difficult to validate computed results. Additionally, designers must take extra care, both at the C/C++ source code level and when applying HLS optimization directives, to get the intended QoR, whether with respect to FPGA resource utilization or design performance.

For more hands-on advice, see the Vivado High-Level Synthesis product page at *http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/hls/*; the Vivado Video Tutorials at *http://www.xilinx.com/training/vivado/*; and the Floating-Point Operator product page at *http://www.xilinx.com/products/intellectual-property/FLOATING_PT.htm.*

References

1. *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008; IEEE-754*

2. *PG060, LogiCORE IP Floating-Point Operator v6.1 Product Guide*

3. http://randomascii.wordpress.com/category/floating-point/

4. http://docs.oracle.com/cd/E19957-01/806 3568/ncg_goldberg.html

5. http://www.lahey.com/float.htm

6. *Adam Taylor, "The Basics of FPGA Mathematics," Xcell Journal Issue 80; http://issuu.com/xcelljournal/docs/xcell80*

7. *UG902, Vivado Design Suite User Guide: High-Level Synthesis v2012.2, and UG871, Vivado Design Suite Tutorial: High-Level Synthesis v2012.2*

Vivado HLS Eases Design of Floating-Point PID Controller

by **Daniele Bagni**
DSP Specialist
Xilinx, Inc.
daniele.bagni@xilinx.com

Giulio Corradi
ISM Senior System Architect
Xilinx, Inc.
giulio.corradi@xilinx.com

The new Xilinx synthesis tool automates an otherwise manual process, eliminating the source of many design errors and accelerating a long and iterative part of the development cycle.

F PGA technology is a powerful and flexible way of implementing a PID controller. With its massive, parallel resources, an FPGA device can allow multiple PID (proportional integral derivative) instances for concurrent operations. An FPGA also offers flexibility in adding extra PID loops if the application needs them, without affecting the performance of other PIDs previously designed. When you implement a PID in the programmable logic (or fabric) of the new Xilinx® Zynq™-7000 All Programmable SoC, you will reap additional advantages, because the power of the FPGA can be directly exploited by the powerful ARM® Cortex™ dual A9 core processing system onboard.

However, FPGA devices require a register transfer-level (RTL) design language, typically VHDL or Verilog, and this might represent a gap in the control engineer's background, discouraging the usage of FPGA technology. To eliminate this gap, Xilinx's new Vivado™ High-Level Synthesis (HLS) design tool makes it possible to transform a C, C++ or SystemC design specification into an RTL implementation that's ready for synthesis into a Xilinx FPGA. The transformation requires only a minor adaptation of regular C or C++ code, and therefore it does not present any significant knowledge gap.

A UBIQUITOUS DEVICE

Virtually all natural and man-made control systems employ PID feedback or one of its variations: PI (proportional integral) and PD (proportional derivative). Large factories use thousands of PID controllers to oversee their chemical or physical processes. In cars and transport systems, PIDs control and maintain engine speed, ensure smooth braking action and control many steering functions. In electrical motors, PID controls the motor's current and torque, while in robots PIDs are used to drive and stabilize the robot's arm or leg trajectory. The PID is so ubiquitous that it is even applied in medical systems, for instance to control an artificial pancreas for a patient with Type 1 diabetes, emulating the natural insulin secretion profile. Indeed, biological systems themselves use feedback to control the response to stimulation, for example the retinal system, which adapts to the light.

A typical feedback control system consists of a plant—which is the mechanical or electrical system to be controlled—and the PID controller. D/A converters transform the control output into the proper plant input, while A/D converters transform the plant output into the feedback signal. Figure 1 sums up how PID works. In a nutshell, the PID controller processes the information difference e(n)—
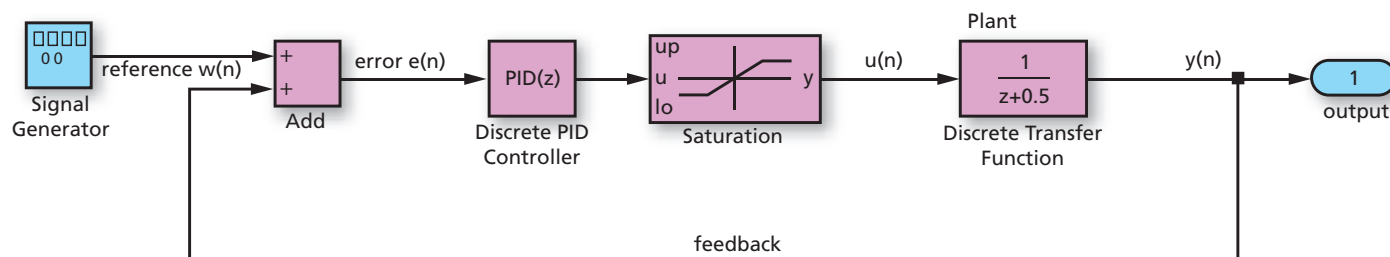

Figure 1 – General discrete control system in a closed loop

Additionally, in recent years electrical drives and robots are finding their way into networked control systems using PID in the loop with a communication channel. In this application, the determinism and speed of FPGA implementation is a great advantage. But again, due to interaction with the software communication protocol stack, the system architects and control engineers often sacrifice performance to get an all-software implementation. Vivado HLS can nullify this trade-off by offering a simpler, more versatile method of implementation. It will be possible to dramatically improve the performance of the developed networked control system just by remapping the C or C++ code into the FPGA fabric of the Zynq-7000 All Programmable SoC device.

called "error"—between a reference input value w(n) and the plant output value y(n), measured by a sensor, applying corrective action to the system's actuator for reaching the desired commanded output value. Every part of the PID contributes to a specific action, or "mode." The P action drives the controller output u(n) according to the size of the error. The I action eliminates the steady-state offset but may worsen the transient response. The D action evaluates the trend to anticipate the output correction, thus increasing the stability of the system, reducing the overshoot and improving the transient response.

A system without feedback control is defined as open-loop, and its transfer function (the way the system's input
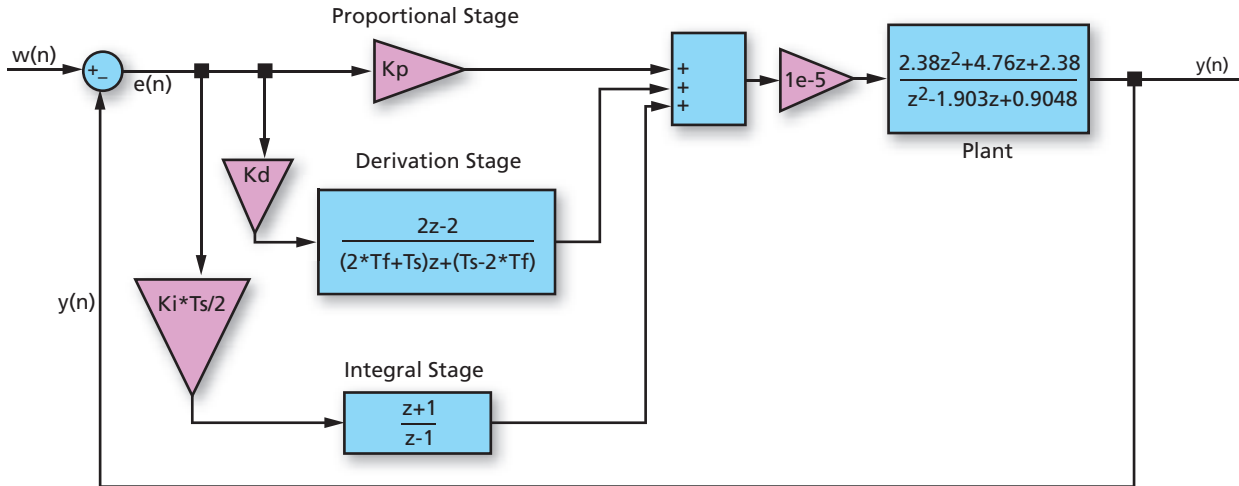
Figure 2 – An example of a digital PID controller and plant in a closed loop

maps into its output) must exhibit a phase shift of less than 180° at unity gain. The difference between the phase lag of the open loop and 180° at unity gain is called the "phase margin." System gain and phase shift are commonly expressed in the analog domain s of the Laplace transform or in the discrete domain z of the Z transform. Assuming P(z) and H(z) to be, respectively, the discrete transfer functions of the plant and of the PID controller, the transfer function T(z) of the whole closed-loop system can be expressed as:

$$T(z) = \frac{P(z) \cdot H(z)}{1 + P(z) \cdot H(z)}$$

The values in z of the numerator and denominator of T(z) are respectively called zeros and poles. The phase lag of the PID enters into the loop and adds up to the total phase lag; thus, a fast PID is desirable to keep the lag at a minimum. Ideally, the PID's response time should be immediate, as with an analog controller. Therefore, the PID computational speed is paramount. In a closed-loop system, stability is mandatory, especially for highly sophisticated applications like robotic systems or electrical motor drives. Without stability the control loop may respond with undesired oscillations, or by generating a sluggish response. Stability can be achieved via compensation of the poles and zeros of the PID controller in order to get the best achievable performance from the closed-loop system (gain and phase characteristics).

In robots and positioning systems, the level of complexity extends not only to a single PID loop but also to cascaded loops. For instance, the torque is managed by the current loop PID, the motor speed by the velocity PID cascaded with the current PID and the position by the space PID cascaded

with the velocity PID. In these cases, sequential software execution of every PID loop becomes a less and less effective way to minimize the overall computational delay.

Many PID designs for electric drives and robots rely on a floating-point C, C++ implementation, and this is often the most familiar representation for the control engineer. It's easy to modify the software using a fast microprocessor, microcontroller or DSP processor, and you can implement many difficult control structures in software without spending too much time with the design of additional hardware.

REFERENCE MODEL OF THE PID CONTROLLER
A practical example can show the advantages of using Vivado HLS to ease the implementation of a digital PID controller. For this example design, we consider only one loop and an electrical DC motor to be the plant. Therefore, the rotational speed is the output and the voltage is the input.

Without entering into the details of the mathematical derivation of the DC motor and rotor transfer functions, it is possible to express the plant as in Equation 1, showing the analog open-loop transfer function in the Laplace domain:

$$P(s) = \frac{Y(s)}{U(s)} = \frac{a}{s^2 b + c + d}$$

Equation 1

where a, b, c, d are the plant numerical parameters. Equation 2 illustrates the PID controller transfer function:

$$H(s) = \frac{U(s)}{E(s)} = K_P + \frac{1}{s} \cdot K_I + s \cdot K_D$$

Equation 2
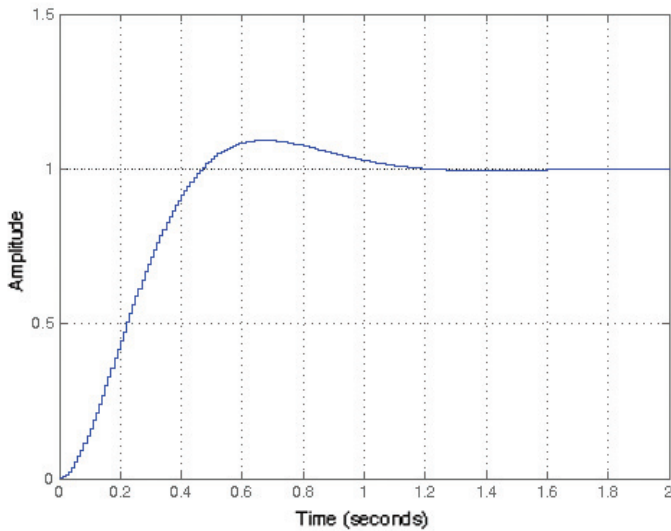
where U(s) and E(s) are the Laplace transforms of the PID

Figure 3 – Closed-loop system response to a step input signal

output and input signals, respectively u(n) and e(n). The terms K_P, K_I and K_D are the gains of the proportional, integral and derivative stages.

One of the methods for transforming transfer functions from the Laplace domain to the Z domain is the Tustin approximation with trapezoidal integration. The plant (Equation 1) and PID (Equation 2) transfer functions are shown respectively in Equations 3 and 4 in their digital form:

$$P(z) = 10^{-5} \cdot \frac{2.38z^2 + 4.76z + 2.38}{z^2 - 1.903z + 0.9048}$$

Equation 3

$$H(z) = K_P + K_I \frac{T_S}{2} \frac{z+1}{z-1} + \frac{K_D}{T_F + \frac{T_S}{2} \frac{z+1}{z-1}}$$

Equation 4

where T_F and T_S are, respectively, the derivative filter time and the sampling time. Figure 2 shows the discrete system made up of the PID controller block and the plant.

The MathWorks' Control System toolbox of MATLAB® and Simulink® is a powerful tool to design and simulate analog and digital PID control systems. The following MATLAB code delivers the PID controller parameters. Figure 3 illustrates the response of the closed-loop system to a step input signal and the PID parameters set to K_P=35.3675, K_I=102.2398, K_D=0.29161.

```
Ts = 1/100; t = 0 : Ts : 2.56-Ts;

% (Laplace transform) transfer function of
```

```
% the continuous system to be controlled
a=1; b=1; c=10; d=20; num=a; den=[b c d];
plant = tf(num,den);

% (Z transform) transfer function of
% the discrete system
plant_d = c2d(plant, Ts, 'tustin')

% dummy parameters to generate a PID
Kp=1; Ki=1; Kd=1; Tf=20;
C_be = pid(Kp, Ki, Kd, Tf, Ts, ...
    'IFormula','Trapezoidal', ...
    'DFormula','Trapezoidal');

% tuning the PID with more
% suitable parameters
contr_d = pidtune(plant_d, C_be)
Kp = contr_d.Kp;
Ki = contr_d.Ki;
Kd = contr_d.Kd;
Tf = contr_d.Tf;

sys_d = feedback(contr_d*plant_d,1);
% closed loop system
figure; step(sys_d);
title(['Closed-loop output to step ' ...
    'signal: Kp=',num2str(contr_d.Kp), ...
    ' Ki=', num2str(contr_d.Ki), ...
    ' Kd=', num2str(contr_d.Kd)]);
axis([0 2.0 0 1.5]); grid;
```

Equations 3 and 4 can be formally rewritten as shown, respectively, in Equations 5 and 6:

$$P(z) = 10^{-5} \cdot \frac{2.38 + 4.76z^{-1} + 2.38z^{-2}}{1 - 1.903z^{-1} + 0.9048z^{-2}}$$

Equation 5

$$H(z) = \frac{U(z)}{E(z)} = G_P + G_I \frac{1+z^{-1}}{1-z^{-1}} + G_D \frac{1-z^{-1}}{1+Cz^{-1}}$$

Equation 6

with

$$G_P = K_P$$
$$G_I = K_I \frac{T_S}{2}$$
$$G_D = \frac{2K_D}{T_S + 2T_F}$$
$$C = \frac{T_S - 2T_F}{T_S + 2T_F}$$

By anti-transforming Equations 5 and 6 we obtain the formulas of Equation 7, which we will use to model the PID controller and the plant blocks in the discrete time domain, as shown by the fragment of MATLAB code that follows:

$$
\begin{aligned}
e(n) &= w(n) & - y(n) \\
y_D(n) &= e(n) & - e(n-1) & - C \cdot y_D(n-1) \\
y_I(n) &= e(n) & + e(n-1) & + y_I(n-1) \\
y(n) &= 1.903 \cdot y(n-1) & - 0.9048 \cdot y(n-2) & + \\
& 10^{-5} \cdot (2.38 \cdot u(n) & + 4.76 \cdot u(n-1) & + 2.38 \cdot u(n-2))
\end{aligned}
$$

Equation 7

```
w = ones(1, numel(t)); w(1:4) = 0;
C = (contr_d.Ts - 2*contr_d.Tf) / ...
    (contr_d.Ts + 2*contr_d.Tf);
Gd = 2*contr_d.Kd / (contr_d.Ts + ...
    2*contr_d.Tf);
Gi = contr_d.Ki * contr_d.Ts/2;
Gp = contr_d.Kp;

% closed loop
e_prev  = 0; % e(n-1)
yi_prev = 0; % yi(n-1)
yd_prev = 0; % yd(n-1)
y_z1    = 0; % y(n-1)
y_z2    = 0; % y(n-2)
u_z1    = 0; % u(n-1)
u_z2    = 0; % u(n-2)

for i = 1 : numel(w)

% error
e(i) = w(i) - y_z1; % CLOSED LOOP

% derivation
yd(i) = -C*yd_prev + e(i) - e_prev;
yd_prev = yd(i);

% integration
yi(i) = yi_prev + e(i) + e_prev;
yi_prev = yi(i); e_prev = e(i);

% PID
u(i) = e(i) * Gp + Gd*yd(i) + Gi*yi(i);

% plant
y(i) = 1.903*y_z1 -0.9048*y_z2 + ...
    1e-5*(2.38*u(i) + 4.76*u_z1 + ...
    2.38*u_z2);
y_z2 = y_z1; y_z1 = y(i);
u_z2 = u_z1; u_z1 = u(i);
```

```
end

figure; plot(t, y, 'g'); grid;
title 'Closed Loop Step: plant+contr';
```

PERFORMANCE OF THE PID DESIGN WITH VIVADO HLS

Vivado HLS is the latest Xilinx design tool. It generates production-quality RTL implementations automatically from high-level specifications written in C, C++ and SystemC. In other words, Vivado HLS automates an otherwise manual process, eliminating the source of many design errors and accelerating a very long and iterative part of the development cycle.

Vivado HLS performs two distinct types of synthesis upon the design. Algorithm synthesis takes the content of the functions and synthesizes the functional statements into RTL statements over a number of clock cycles. Interface synthesis transforms the function arguments (or parameters) into RTL ports with specific timing protocols, allowing the design to communicate with other designs in the system. You can perform interface synthesis on global variables, top-level function arguments and the return value of the top-level function.

The synthesis process executes in multiple steps. The first step is to extract the control and data path inferred by the C code. The interface synthesis affects what is achievable in algorithm synthesis and vice versa. Like the numerous decisions made during any manual RTL design, the result will be a large number of available implementations and optimizations, and an even larger number of variations depending on how they impact each other in combination. Vivado HLS abstracts the user away from these details and allows you to productively get to the best design in the shortest time. Vivado HLS quickly creates the most optimal implementation based on its own default behavior and the constraints and directives you have specified.

Scheduling and binding are the processes at the heart of Vivado HLS. The scheduling process assigns every operation to a specific clock cycle. The decisions made during scheduling take into account, among other things, the clock frequency and clock uncertainty, timing information from the device technology library, as well as area, latency and throughput directives. Binding is the process that determines which hardware resource, or core, is used for each scheduled operation. For example, Vivado HLS will automatically determine whether both an adder and subtractor will be used or if a single adder-subtractor can handle both operations. Since the decisions in the binding process can influence the scheduling of operations—for example, using a pipelined multiplier instead of a standard combinational multiplier—binding decisions are considered during scheduling.

Vivado HLS provides productivity benefits in both verification and design optimization by:

- shortening the previous manual RTL creation process and avoiding translation errors by automating the creation of the RTL from the functional C specification;

- allowing you to evaluate multiple architectures quickly and easily before committing to an optimum solution.

The acceleration of simulation time by using the functional C specification instead of the RTL one enables the early detection of design errors.

The C code implementation, very similar to the MATLAB model, is shown in the following. We assume that in the real world, the PID input and output signals are saturated by an amount that the user can control. The PID coefficients (G_I, G_P, G_P, G_D and C of Equation 7) and the maximum and minimum value for e(n) and u(n) signals are assumed to be loaded on the PID core sequentially at any function call. The same holds true for the two input and output signals.

```
void PID_Controller(bool ResetN, float
coeff[8], float din[2], float dout[2])
{

// local variables for I/O signals
float Gi, Gd, C, Gp, Y, W, E, U;
// previous PID states:
// Y1(n-1), X1(n-1), INT(n-1)
static float prev_X1, prev_Y1;
static float prev_INT;

// current local states:
// X1(n), X2(n)
float X1, X2, Y1, Y2, INT;

// local variables
float max_limE, max_limU;
float min_limE, min_limU;
float tmp, pid_mult, pid_addsub;

// get PID input coefficients
Gi = coeff[0]; Gd = coeff[1];
C  = coeff[2]; Gp = coeff[3];
max_limE = coeff[4];
max_limU = coeff[5];
min_limE = coeff[6];
min_limU = coeff[7];

// get PID input signals
// effective input signal
W = din[0];
```

```
// closed loop signal
Y = din[1];

if (ResetN==0)
{
  // reset INTegrator stage
  prev_INT = 0;
  // reset Derivative stage
  prev_X1 = 0;
}
// compute error signal E = W - Y
pid_addsub = W - Y;
pid_addsub = (pid_addsub>max_limE) ?
  max_limE : pid_addsub;
E = (pid_addsub<min_limE) ?
  min_limE : pid_addsub;

// Derivation
// Y1(n) = -C * Y1(n-1) + X1(n) -
// X1(n-1) = X1 - (prev_X1+C*Y1)
X1 = Gd * E;
pid_mult = C * prev_Y1;
pid_addsub = pid_mult + prev_X1;
pid_addsub = X1 - pid_addsub;
// update Y1(n)
Y1 = pid_addsub;

// Integrator
// INT(n) = CLIP(X2(n) + INT(n-1))
// Y2(n) = INT(n-1) + INT(n)
X2 = Gi * E;
pid_addsub = prev_INT + X2;
pid_addsub=(pid_addsub>max_limE)?
  max_limE : pid_addsub;
INT = (pid_addsub<min_limE)?
  min_limE : pid_addsub;
Y2 = INT + prev_INT;

// output signal U(n)
pid_mult = Gp * E;
pid_addsub = Y1 + Y2;
tmp = pid_addsub + pid_mult;
tmp = (tmp > max_limU) ?
  max_limU : tmp;
U = (tmp < min_limU) ?
  min_limU : tmp;

// PID effective
// output signal
dout[0] = U;
// test the PID error
// signal as output
dout[1] = E;
```
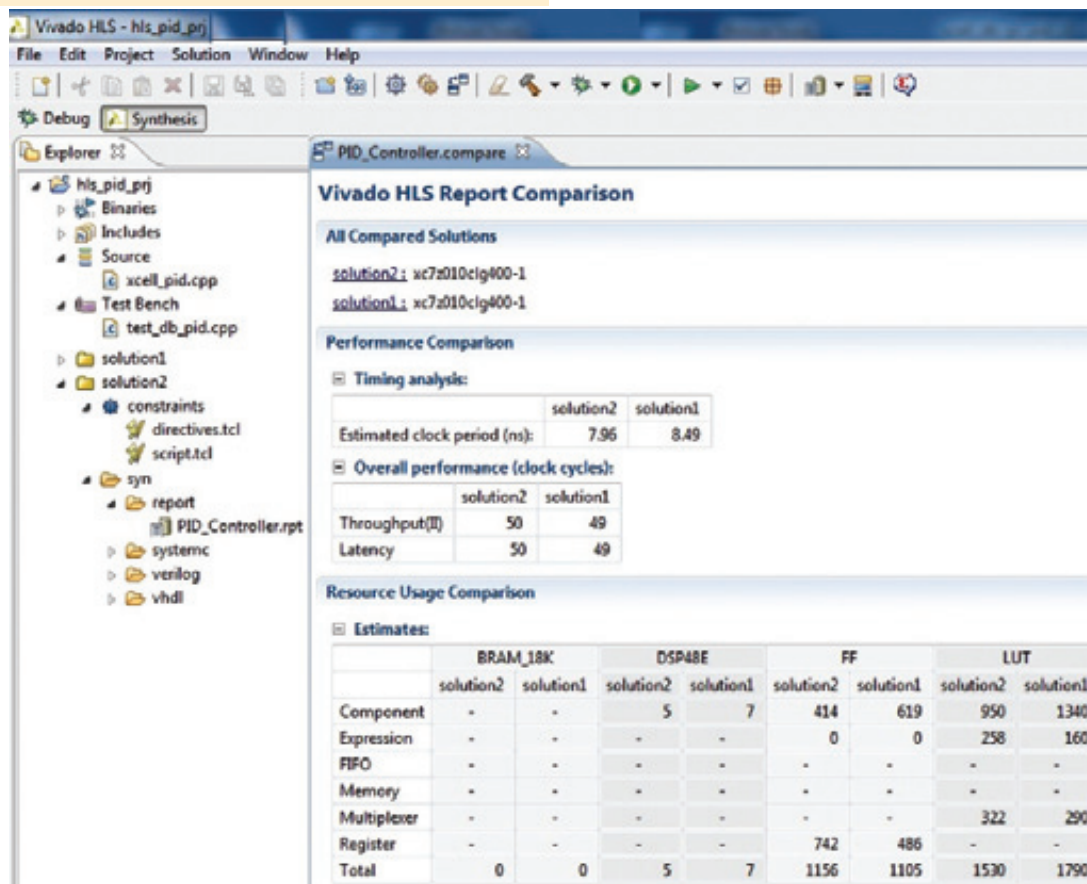
Figure 4 – The Vivado HLS screen shows a performance estimation of two alternative solutions for the same PID C++ design.

```
// update internal states
// for the next iteration
prev_X1 = X1;
prev_Y1 = Y1;
prev_INT= INT;

return;
}
```

We targeted the Zynq-7010 CLG400-1 device with a clock period of 10 nanoseconds; the PID is implemented in the FPGA in 32-bit floating-point arithmetic. The first run of Vivado HLS (named solution1 in Figure 4) exhibits an estimated clock period of 8.49 ns, which corresponds to a 118-MHz FPGA clock frequency. Since 49 clock cycles of latency are needed to produce an output, the effective data rate is 2.4 Msamples per second. The FPGA area occupation estimation (before place-and-route) is seven DSP48E slices, 1,105 flip-flops and 1,790 lookup tables.

By analyzing the report file produced by Vivado HLS, we realized that the tool has generated two floating-point adder-subtractor cores. Therefore, we applied the following directives:

```
set_directive_interface -mode ap_fifo
"PID_Controller" coeff
set_directive_interface -mode ap_fifo
"PID_Controller" din
set_directive_interface -mode ap_fifo
"PID_Controller" dout
set_directive_allocation -limit 1 -type
core "PID_Controller" fAddSub
set_directive_allocation -limit 1 -type
core "PID_Controller" fMul
```

The first three directives set the I/O function parameters to be mapped as FIFO ports in the automatically generated RTL design, while the last two directives limit the amount of floating-point multipliers and adder-subtract cores to a single instantiation of each one.

As a result, the second run of Vivado HLS (named solution2 in Figure 4) exhibits a performance of 7.96 ns of estimated clock period, thus corresponding to a 125-MHz FPGA clock frequency. The 50 clock cycles of latency for any output value make the effective data rate 2.5 MSPS. The FPGA area occupation estimation is five DSP48Es, 1,156 flip-flops

and 1,530 LUTs, which represents the best result. Figure 4 shows a screen shot of a Vivado HLS synthesis estimation report comparing the two solutions.

The following fragment of RTL code shows the VHDL that Vivado HLS has automatically generated for the top-level function. The tool generates the interfacing signals as *clock reset* and *start* as input ports, *done* and *idle* as output ports; the input arrays *din* and *coeff* are mapped as input FIFO ports, thus having *empty* and *read* signals, while the output array *dout* becomes an output FIFO with its *full* and *write* signals.

```
— RTL generated by Vivado(TM) HLS - High-
— Level Synthesis from C, C++ and SystemC
— Version: 2012.2
— Copyright (C) 2012 Xilinx Inc. All
— rights reserved.

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
library work;
use work.AESL_components.all;

entity PID_Controller is
port (
ap_clk          : IN  STD_LOGIC;
ap_rst          : IN  STD_LOGIC;
ap_start        : IN  STD_LOGIC;
ap_done         : OUT STD_LOGIC;
ap_idle         : OUT STD_LOGIC;
coeff_empty_n   : IN  STD_LOGIC;
coeff_read      : OUT STD_LOGIC;
dout_full_n     : IN  STD_LOGIC;
dout_write      : OUT STD_LOGIC;
din_empty_n     : IN  STD_LOGIC;
din_read        : OUT STD_LOGIC;

ResetN       : IN
  STD_LOGIC_VECTOR ( 0 downto 0);
coeff_dout : IN
  STD_LOGIC_VECTOR (31 downto 0);
din_dout   : IN
  STD_LOGIC_VECTOR (31 downto 0);
dout_din   : OUT
  STD_LOGIC_VECTOR (31 downto 0));

end;
```

THREE WORKING DAYS

You can efficiently implement digital PID controllers into Xilinx FPGA devices with a limited amount of resources starting from a C-model specification, even in 32-bit floating-point arithmetic. The RTL that Vivado HLS automatically generates consumes little area—just five DSP48E slices, 1,156 flip-flops and 1,530 LUTs on a Zynq-7000 device, with an FPGA clock frequency of 125 MHz and an effective data rate of 2.5 MSPS. The total amount of design time to produce these results was only three working days, most of this time devoted to building the MATLAB and C models rather than running the Vivado HLS tool itself. That took about half a day of work.

This methodology presents a significant advantage over other alternatives; in particular, Vivado HLS is taking care of directly mapping the floating-point PID into the fabric. This eliminates the intermediate step of realizing such mapping by hand, improving the project portability and consistency, and dramatically reducing the total development time that, in the case of handmade translation, surely would have taken more than three working days.

Dr. Daniele Bagni is a DSP Specialist FAE for Xilinx EMEA in Milan, Italy. After earning a degree in quantum electronics from the Politecnico of Milan, he worked for seven years at Philips Research labs in real-time digital video processing, mainly motion estimation for field-rate upconverters. For the next nine years he was project leader at STMicroelectronics' R&D labs, focusing on video coding algorithm development and optimization for VLIW architecture embedded DSP processors, while simultaneously teaching a course in multimedia information coding as an external professor at the State University of Milan. In 2006 he joined the Xilinx Milan sales office. What Daniele enjoys most about his job is providing customers with feasibility studies and facing a large variety of DSP applications and problems. In his spare time, he likes playing tennis and jogging.

Dr. Giulio Corradi is ISM (industrial, scientific and medical) senior system architect for Xilinx in Munich, Germany. He brings 25 years of experience in management, software engineering and development of ASICs and FPGAs in industrial, automation and medical systems, specifically in the field of control and communication for the major corporations. DSP algorithms, applied chromatography, motor control, real-time communication and functional safety have been his major focus. From 1997 to 2005, he managed several European-funded research projects for train communication networking and wireless remote diagnostic systems. Between 2000 and 2005 he headed the IEC61375 conformance test standard. In 2006, Giulio joined Xilinx's Munich office, contributing to the Industrial Networking and Motor Control Xilinx reference platforms, and providing customer guidance on functional safety applications. In his spare time, he enjoys swimming and playing the piano.

Understanding the Major Clock Resources in Xilinx FPGAs

by Sharad Sinha

PhD Candidate
Center for High Performance Embedded Systems
Nanyang Technological University, Singapore
sharad_sinha@pmail.ntu.edu.sg

Knowledge of DCMs, PLLs, PMCDs and MMCMs is the foundation for designing robust clocking strategies.

Xilinx offers abundant clocking resources in its FPGAs, and most designers have used one or all of them at one point or another in their FPGA designs. However, it can be perplexing for a new FPGA designer to figure out which of the four major types to use when. No existing Xilinx® FPGA contains all four of these resources (see Table 1).

Each of the four categories lends itself to particular applications. For example, the digital clock manager (DCM) is ideal for implementing a delay-locked loop, digital frequency synthesizer, digital phase shifter or digital spread spectrum. The DCM is also a good choice to mirror, forward or rebuffer a clock signal. Choose a different clocking resource—the phase-matched clock divider (PMCD)—meanwhile, to implement phase-matched divided clocks or phase-matched delay clocks.

The phase-locked loop (PLL) and mixed-mode clock manager (MMCM) handle many of the same jobs, such as frequency synthesis, jitter filtering for internal and external clocks, and deskewing clocks. You can also use these resources to mirror, forward or rebuffer a clock signal.

Keeping these general use cases in mind will help you sort through your clocking choices as you contemplate the implementation details of your design. It's also important to consider compatibility among the varying device families when designing a proper clocking strategy within a long-term product evolution plan. Let's take a look at each of these clocking resources in more depth.

Device Family	DCM	PLL	PMCD	MMCM
Virtex-4	Y	N	Y	N
Virtex-5	Y	Y[1]	N	N
Virtex-6	N	N	N	Y[2]
7 Series FPGAs	N	Y	N	Y[2]
Spartan-6	Y	Y	N	N

1. PLL in the Virtex-5 supports the Virtex-4 FPGA PMCD mode of operation
2. MMCM in the 7 series is backward compatible

Table I – Clock resources in some major Xilinx FPGA families

You can use a DCM to generate a clock of a higher frequency by multiplying the input clock from the clock source. Similarly, you can also generate a clock with a lower frequency by dividing the input clock from a higher-frequency clock source.

DIGITAL CLOCK MANAGER

As the name suggests, a DCM is a module that manages clocking architectures and is helpful in shaping and manipulating clock signals. The DCM contains a delay-locked loop (DLL), which eliminates clock distribution delays by deskewing the output clocks from the DCM with respect to the input clocks.

The DLL contains a chain of delay elements and control logic. Output tapped from a delay element is a delayed version of the incoming clock. The amount of delay depends on the delay element's position in the delay chain. The delay is expressed as a change in phase or phase shift with respect to the original clock. This is called the "digital phase shift." Figure 1 shows a typical DCM block in a Virtex®-4 device. There are three different kinds of DCM primitives in the Virtex-4, as detailed in the Virtex-4 FPGA User Guide (UG070, v2.6).

A DLL is like a PLL in a general sense. However, unlike a PLL, a DLL does not contain a voltage-controlled oscillator (VCO). A PLL would always store both phase and frequency information, while the DLL has only phase information. As a result, a DLL is a little bit more stable than a PLL. You can design either type using analog as well as digital techniques, or a mix of the two. However, DCMs in Xilinx devices are entirely digital in design.

Since a DCM can introduce a delay in the clock path, you can use one to precisely generate the timing of the row- and the column-access strobe signals for a DRAM, for example. Similarly, individual data bits on a data bus can arrive at different times. To correctly sample the data bits, the clock signal at the receiving end must be properly aligned with the arrival of all the data bits. This might require delaying the clock signal from the transmit end to the receive end if the receiver uses the transmitted clock.

Sometimes a design might require a higher clock frequency to run the logic on the FPGA. However, only a clock source with a lower-frequency output may be available. In this case, you can use a DCM to generate a clock of a higher frequency by multiplying the input clock from the clock source. Similarly, you can also generate a clock with a lower frequency by dividing the input clock from a higher-frequency clock source, a technique known as "digital frequency synthesis."
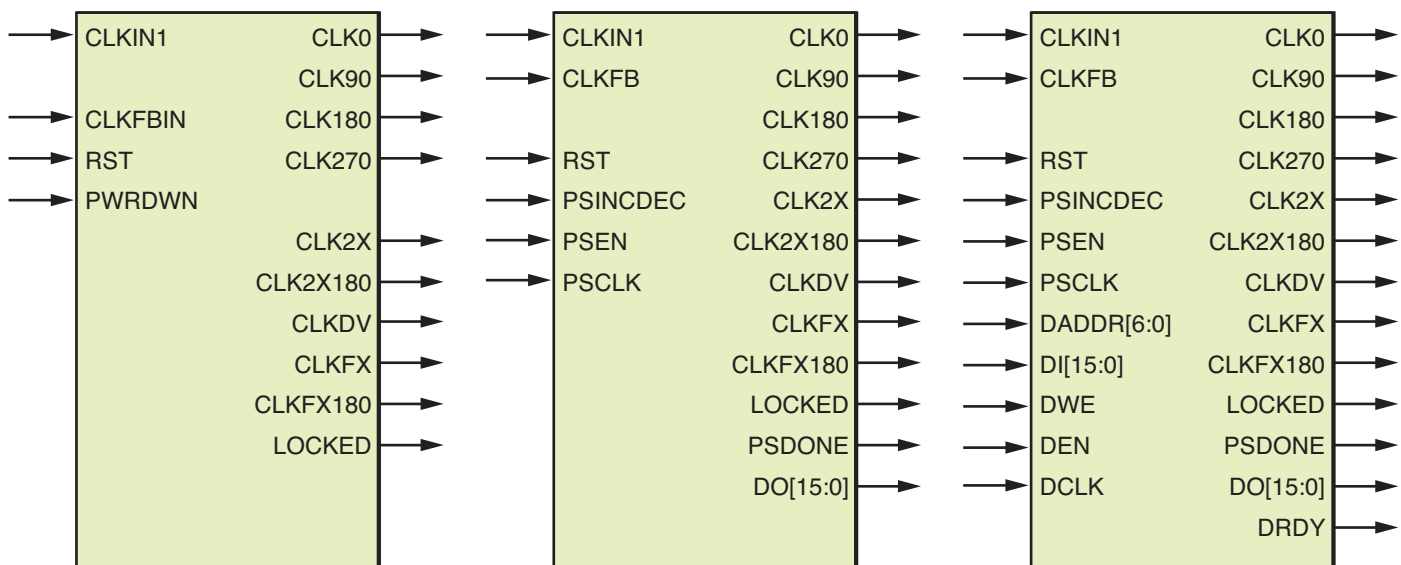

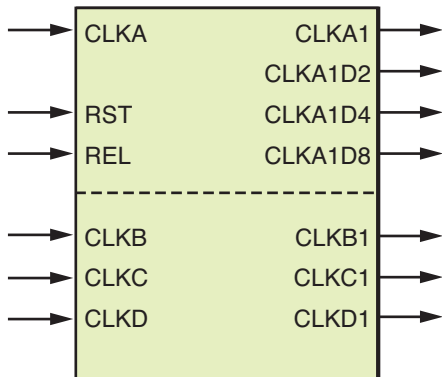Figure 1 – DCM primitives in the Virtex-4 FPGA

Figure 2 – PMCD primitive in the Virtex-4 FPGA

Designers use spread-spectrum clocking to reduce the peak electromagnetic radiation from a clock signal by modulating the clock signal. An unmodulated clock signal will have a peak with a high electromagnetic radiation. However, when the signal is modulated, the EM radiation is spread over a range of clock frequencies, thus reducing radiation at any one frequency. Generally, you will use spread-spectrum clocking when you have to meet certain maximum electromagnetic-radiation requirements and perform high-speed processing on the FPGA— for instance, in a deserializer in a receiver for a communications system. Hence, a DCM in your FPGA would multiply the input spread-spectrum clock and create high-frequency clocks internally. The DCM output must follow the spread-spectrum clock accurately to maintain phase and frequency alignment and update the deskewing and phase shifting. A deterioration in the DCM's phase and frequency alignments would reduce the receiver's skew margin.

Mirroring a clock involves sending the clock out of the FPGA device and then receiving it back again. You can use this technique to deskew a board-level clock among multiple devices. DCMs can forward a clock from the FPGA to another device. This is so because an input clock to the FPGA cannot be routed to an output pin straight away; there is no such routing path available. If clock forwarding is the only requirement, use a DCM to forward the clock to an output pin to preserve the signal fidelity. Optionally, you can also connect the DCM output to an ODDR flip-flop before the clock is sent out. Of course, you may choose not to use a DCM and forward the clock using just an ODDR. Quite often, a clock driver needs to drive a clock to many parts of a design. This increases the load on the clock driver, resulting in clock skew and other problems. In this case, you should employ clock buffering to balance the load.

A clock may be connected to a number of logic blocks on an FPGA. In order to ensure that the clock signal has proper rise and fall times (and consequently, that the latency between input and output is within bounds) at a register far away from the clock source, insert clock buffers between the clock driver and the load. DCMs can serve as clock buffers between the clock input pin and the logic blocks.

Finally, you can also use DCMs to convert an input clock signal to a different I/O standard. For example, a DCM can forward an incoming LVTTL clock as an LVDS clock.

PHASE-MATCHED CLOCK DIVIDER
Designers use PMCDs to generate frequency-divided, but phase-matched, versions of an input clock. This is similar to the DCM frequency synthesis for divided clocks. PMCDs can also generate phase-matched but delayed clocks in a design. In the latter case, they preserve edge alignments, phase
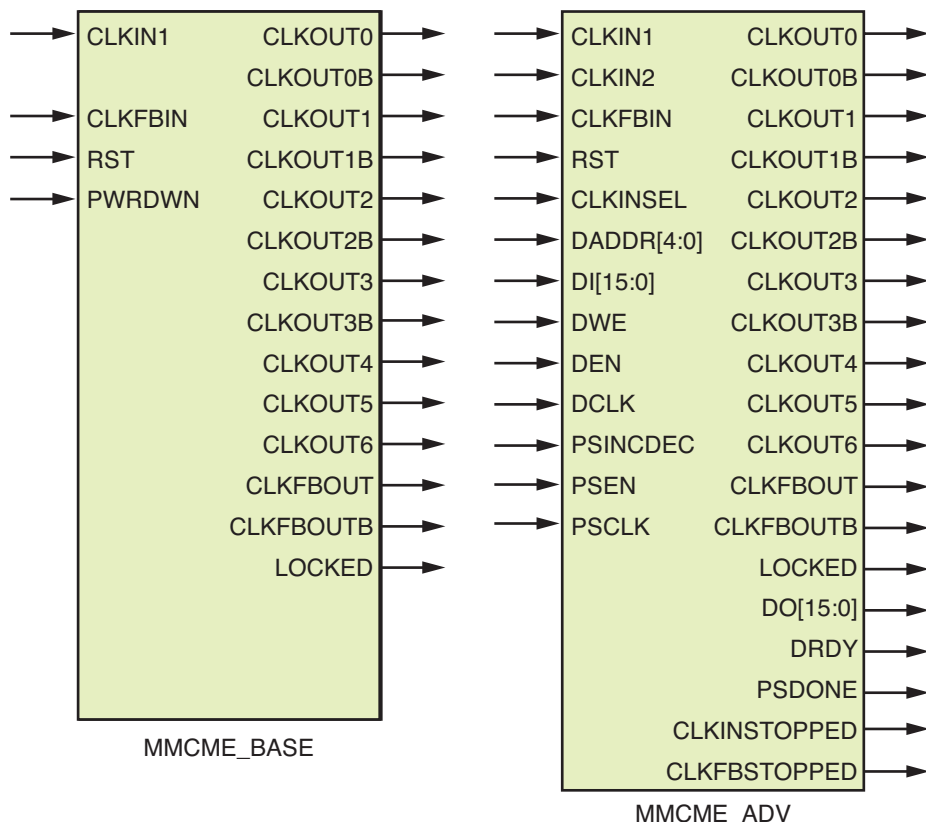

MMCME_BASE

MMCME_ADV

Figure 3 – MMCM primitives in the Virtex-6 architecture

relationships and skews between the input clock and other PMCD input clocks. Unlike a DCM, where the divider value is configurable, existing PMCDs in Xilinx devices generate clocks divided by only 2, 4 and 8. This means that the PMCD-generated clocks will have a frequency that is half, one-fourth or one-eighth of the input clock. In Xilinx devices like the Virtex-4 FPGA, PMCDs reside next to DCMs in a column; each of the columns has two PMCD-DCM pairs. Hence, a DCM output can drive a PMCD input.

Since DCMs also handle deskewing, designers can use the PMCD without a DCM whenever deskewed clocks are not required. Connections between the two PMCDs in a column are also allowed through a dedicated pin. Figure 2 shows the PMCD primitive in the Virtex-4 device. Details are available in the Virtex-4 FPGA User Guide (UG070, v2.6).

MIXED-MODE CLOCK MANAGER

Another variety of clocking resource, the MMCM, is used to generate different clocks with defined phase and frequency relationships to a given input clock. However, unlike a DCM, the MMCM uses a PLL to accomplish this task. The clock-management tile (CMT) in the Virtex-6 FPGA has two MMCMs, while that in the Virtex-7 has one MMCM and one PLL. The MMCMs in the Virtex-6 device do not have spread-spectrum capability; therefore, a spread spectrum on the input clock would not be filtered and will be passed on to MMCM output clocks. However, the Virtex-7 FPGA's MMCM does have the spread-spectrum capability.

The MMCMs in the Virtex-6 FPGA require that you insert a calibration circuit that is needed for proper operation of the MMCM after a user reset or a user power down. Version 11.5 and higher of Xilinx ISE® design tools automatically inserts the necessary calibration circuit during the MAP phase of implementation. For earlier versions of Xilinx ISE, you will need to insert the circuit manually, using design files supplied by Xilinx Technical Support. A final wrinkle is that when migrating such a design for implementation with ISE 11.5 and higher, you must manually remove the calibration circuit or disable its automatic insertion through the appropriate synthesis attribute on each MMCM. For details, refer to Xilinx Answer Record AR#33849.

There is no such concern for MMCMs in 7 series devices because these FPGAs are supported only by ISE versions 13.1 and higher, along with the new Vivado™ Design Suite. The presence of dedicated routing between MMCMs in the Virtex-6 family enables you to use global clocking resources for other parts of the design.

Figure 3 shows the MMCM primitives in the Virtex-6 FPGA. The details of various ports are spelled out in the Virtex-6 FPGA Clocking Resources User Guide (UG362, v2.1). Figure 4 shows the MMCM primitives in 7 series Xilinx FPGAs. Again, the 7 Series FPGA Clocking Resources User Guide (UG472, v1.5) offers all the details.

PHASE-LOCKED LOOPS

Designers use PLLs primarily for frequency synthesis. Hence, you can use one to generate multiple clocks from an input clock. A PLL can also be a way to filter jitter, together with a DCM. PLLs are available in Spartan®-6, Virtex-5 and 7 series FPGAs. There are dedicated DCM–to-PLL and PLL-to-DCM routing lines in both the Spartan-6 and Virtex-5. The PLL outputs in Spartan-6 and Virtex-5 are not spread spectrum. In both these devices, you may select a PLL over a DCM in a
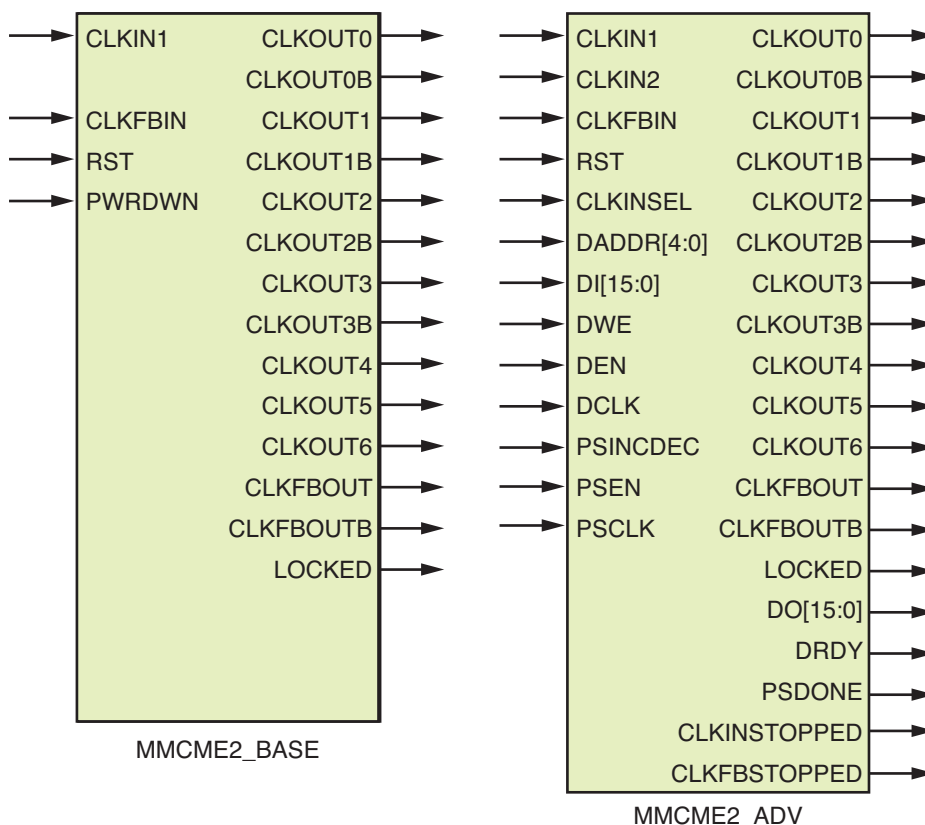
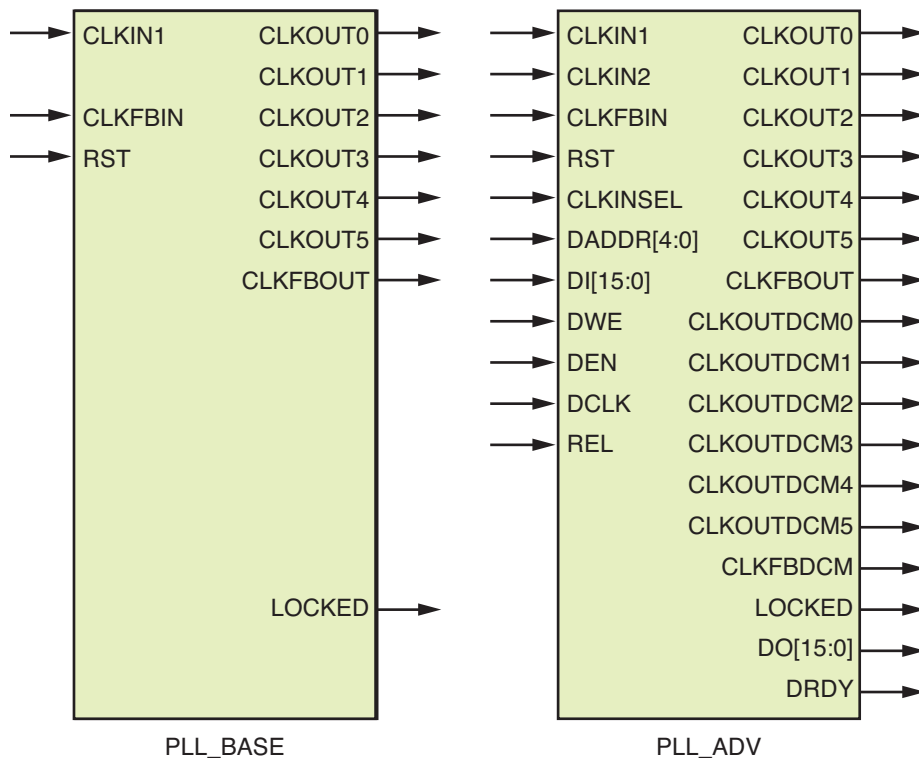Figure 4 – MMCM primitives in 7 series Xilinx FPGAs

Figure 5 – PLL primitives in the Virtex-5 FPGA

design that uses several different clocks. The PLL clock outputs are configurable over a wide range, whereas the DCM output clocks are predetermined and not configurable. Again, this choice depends on the requirements of the design. However, when a phase shift is necessary, a DCM is the clear choice.

The PLLs in 7 series devices, meanwhile, implement a smaller subset of functions than the MMCM implements. So while the MMCM is based on a PLL architecture, there are standalone PLLs as well in 7 series devices. Figure 5 shows the PLL primitive in the Virtex-5 FPGA. The details of the various ports are spelled out in the the Virtex-5 User Guide (UG190, v5.4).

DESIGN MIGRATION
It is important to understand the differences among the four major clock resources as well as their availability in different device families. At the same time, similar resources (like a

DCM) in different families may not be exactly the same in functionalities. For instance, the DCM in a Spartan-6 FPGA supports the generation of spread-spectrum clocks, but the DCM in Virtex-5 and Virtex-4 devices does not.

Apart from ensuring functionality, selecting the correct clock resource for a given design is also important when planning future design migration to a higher-end family. As can be seen from Table 1, the MMCM in Virtex-6 and 7 series devices provides backward compatibility with DCMs in previous families. However, you will need to determine the degree to which backward compatibility is supported, as all these clock resources are multifunctional in nature, providing different functions related to clocking. Understanding the fine points of compatibility is important in planning for long-term product evolution.

To read more by Sharad Sinha, follow his blog at http://sharadsinha.wordpress.com.

How to Implement State Machines in Your FPGA

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

State machines are often the backbone of FPGA development. Choosing the right architecture and implementation methods will ensure that you obtain an optimal solution.

F PGAs are often called upon to perform sequence- and control-based actions such as implementing a simple communication protocol. For a designer, the best way to address these actions and sequences is by using a state machine. State machines are logical constructs that transition among a finite number of states. A state machine will be in only one state at a particular point in time. It will, however, move between states depending upon a number of triggers.

Theoretically, state machines are divided into two basic classes—Moore and Mealy—that differ only in how they generate the state machine outputs. In a Moore type, the state machine outputs are a function of the present state only. A classic example is a counter. In a Mealy state machine, the outputs are a function of the present state and inputs. A classic example is the Richards controller (see *http://en.wikipedia.org/wiki/Richards_controller*).

DEFINING A STATE MACHINE
When faced with the task of defining a state machine, the first step is to develop a state diagram. A state diagram shows the states, the transitions between states and the outputs from the state machine. Figure 1 shows two state diagrams, one for a Moore state machine (left) and the other for a Mealy state machine.

If you had to implement these diagrams in physical components (as engineers did before FPGAs were available), you would start by generating the present-state and next-state tables, and producing the logic required to implement the state machine. However, as we will be using an FPGA for implementation, we can work directly from the state transition diagram.

ALGORITHMIC STATE DIAGRAMS
While many state machines are designed using the state diagram approach shown in Figure 1, another method of describing a state machine's behavior is the algorithmic state chart. This ASM chart (Figure 2) is much closer in appearance to a software-engineering flow chart. It consists of three basic constructs:

1. **State box,** which is associated with the state name and contains a list of the state outputs (Moore)

2. **Decision box,** which tests for a condition being true and allows the next state to be determined

3. **Conditional output box,** which allows the state machine to describe Mealy outputs dependent upon the current state and inputs

Some engineers feel that a state machine described in ASM format is easier to map to implementation in a hardware description language such as VHDL.
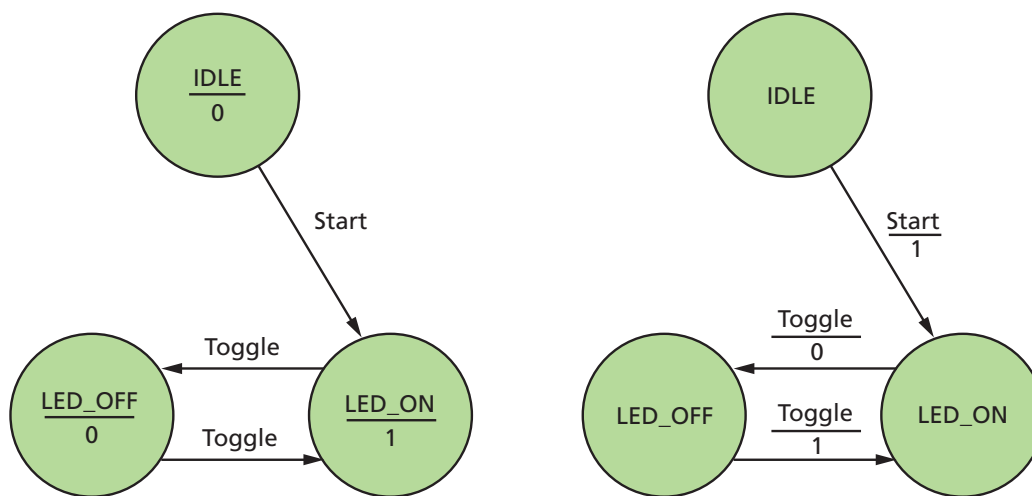
Figure 1 – State diagrams for a Moore (left) and a Mealy state machine designed to turn LEDs on and off

MOORE VS. MEALY: WHICH SHOULD I CHOOSE?

The decision to implement a Moore or a Mealy machine will depend upon the function the state machine is required to perform along with any specified reaction times. The major difference between the two is in how the state machines react to inputs. A Moore machine will always have a delay of one clock cycle between an input and the appropriate output being set. This means a Moore machine is incapable of reacting immediately to a change of input, as can be seen clearly in Figure 3. This ability of the Mealy machine to react immediately to the inputs often means that Mealy machines require fewer states to implement the same function as a Moore implementation would need. However, the drawback of a Mealy machine is that when communicating with another state machine, there is the danger of race conditions, which occur when the output is unexpectedly and critically dependent on the sequence or timing of other events.

Of course, a state machine does not have to be just Moore or Mealy. It is possible to create a hybrid state machine that uses both styles to deliver a more efficient implementation of the function required. For example, a state machine that is used to receive RS232 serial data could be a hybrid.

IMPLEMENTING THE STATE MACHINE

Using a high-level language like VHDL, it is easy to implement a state machine directly from the state diagram. VHDL supports enumerated types, allowing you to define the actual state names. Here's an example:

```
TYPE state IS (idle, led_on, led_off) ;
```
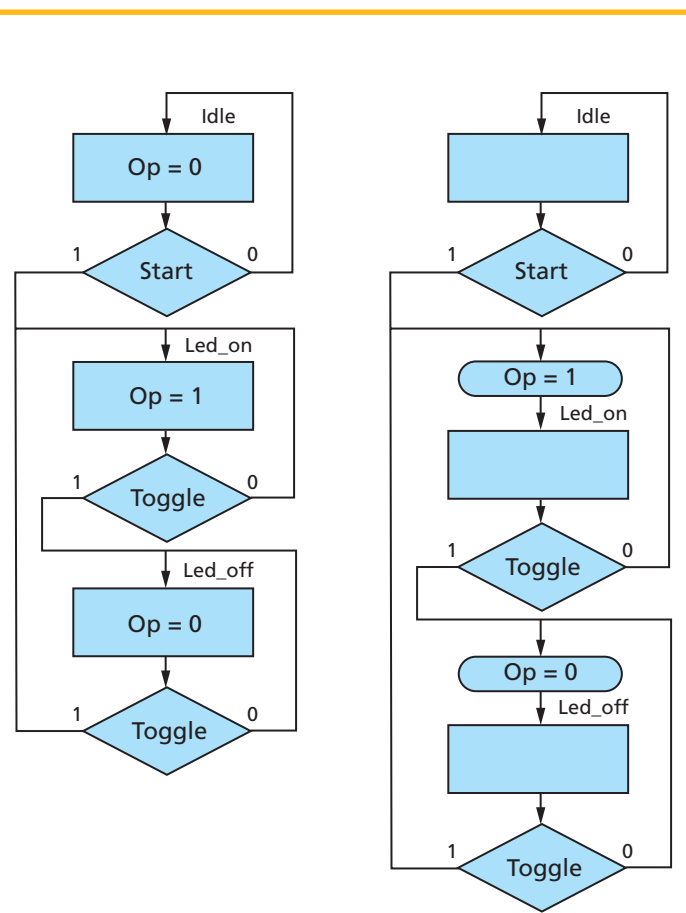

Figure 2 – Algorithmic state charts for the state machines shown in Figure 1: Moore (left) and Mealy

The above type declaration corresponds to the state diagrams shown in Figure 1, for a machine that toggles a light-emitting diode on and off when a button is pushed.

There are many methods of implementing a state machine. However, the choices basically fall into two categories. The first basic technique uses a one-shot approach that contains everything within a single process. The alternative is a two-process approach, which separates the combinatorial and sequential logic.

As a general rule, most engineers prefer to implement a single-process state machine. These have several benefits over the traditionally taught two-process approach:

- They remove the risk of incomplete coverage of signals in the combinatorial process, creating latches.

- State machine outputs are aligned to the clock.

- They are generally easier to debug than a two-process implementation.

Regardless of which approach you decide upon to implement your state machine, you will evaluate the next-state determination and any outputs using a CASE statement, as seen in Figure 4. The figure shows a side-by-side comparison between a Moore (left) and Mealy machine using a single-process approach.

STATE MACHINE ENCODING

The state variable is stored within flip-flops, which are updated with the next state on the next clock edge (even if there is no state change). How you use these flip-flops to represent the state value depends upon the number of states and on whether you choose to direct your synthesis tool in one particular method. The three most common types of state encoding are:

- **Sequential**—State encoding follows a traditional binary sequence for the states.

- **Gray**—Similar to the sequential encoding scheme except that the state encoding uses Gray code, where only one bit changes between state encodings.

- **One-Hot**—This technique allots one flip-flop for each state within the state machine. Only one flip-flop is currently set high and the rest are low; hence the name "one-hot."

Both sequential and Gray encoding schemes will require a number of flip-flops, which you can determine by the equation below.

$$FlipFlops = Ceil\left(\frac{LOG10(States)}{LOG10(2)}\right)$$

By contrast, one-hot encoding schemes require the same number of states as there are flip-flops.

The automatic assignment of state encoding depends upon the number of states the state machine contains. While this will vary depending upon the synthesis tool you have selected, you can use this general rule of thumb for encoding:

- Sequential, less than five states

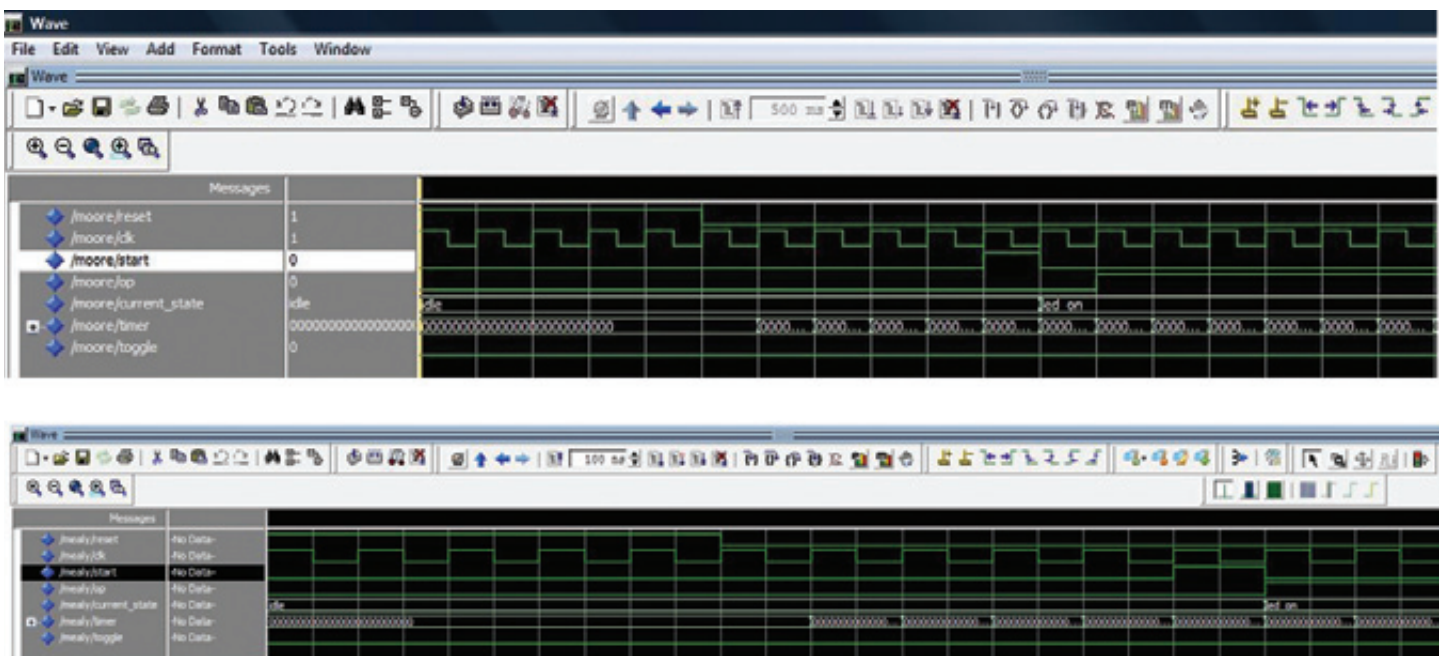- One-hot, five to 50 states

- Gray, greater than 50 states

Figure 3 – Screens show simulation results for Moore (top) and Mealy outputs.

Figure 4 – Moore (left) and Mealy state machines in VHDL

Often you will not necessarily think about what state encoding to use, instead allowing the synthesis engine to determine the correct implementation—getting involved only if the chosen style causes an issue. However, should you need to take things into your own hands and define the state encoding, there is no need to do so long-hand, defining constants for each state with the state encoding. Instead, you can use an attribute within the code to drive the synthesis tool to choose a particular encoding style, as demonstrated below.

```
TYPE state IS (idle, led_on, led_off) ;

SIGNAL current_state : state := idle;

ATTRIBUTE syn_encoding STRING;

ATTRIBUTE syn_encoding OF current_state :
SIGNAL IS "sequential";
```

where "sequential" can be also "gray" and "onehot." You can also combine these three choices—sequential, Gray and one-hot—with the "safe" attribute to ensure the state machine can recover to a valid state should it enter an illegal state.

In addition, you can also use the syn_encoding attribute to define the values of the state encoding directly. For example, suppose you desired to encode a three-state machine using the following state encoding: Idle = "11," led_on = "10," led_off = "01," as opposed to the more traditional sequence of "00," "01" and "10."

```
TYPE state IS (idle, led_on, led_off) ;

SIGNAL current_state : state := idle;

ATTRIBUTE syn_encoding STRING;

ATTRIBUTE syn_encoding OF current_state :
SIGNAL IS "sequential";
```

As the engineer, you are responsible for using the correct settings in the synthesis tool to ensure the tool does not ignore any attributes. For example, the Xilinx® XST tool requires that you set the FSM Option to USER while Synopsys' Synplify requires that the FSM Compiler be turned off.

The equation given earlier determines the number of flip-flops needed for a state machine implementation. Since not all state machines are to a power of two, this means that

some states will not be used within the design. As the engineer implementing the state machine, you must be responsible for ensuring these unused states are correctly handled within the design. There are several basic techniques to implement to accomplish this goal that will serve a broad range of designs, and other, more advanced techniques to employ if you are working in high-reliability, safety-critical fields. (See *Xcell Journal* issue 73 for an in-depth article titled "Using FPGA in Mission-Critical Systems," which looks at state machine protection.)

However, for most applications you will simply need to ensure your state machine correctly addresses unused states and recovers properly should it enter an illegal state. There are two main options for doing so. The first method is to implement a safe state machine using the synthesis tool. The tool will typically insert additional logic to detect an illegal state and return the state machine to a valid state. The second option, which gives you more control over the logic being implemented, is to declare all 2n states and use another attribute to ensure they are not optimized out despite having no entry condition. This means the state is never entered by any condition within the state machine except an error (single-event upset, etc.). The code below shows the use of attributes to prevent removal of these unused states.

```
TYPE state IS (idle, led_on, led_off) ;

SIGNAL current_state : state := idle;

ATTRIBUTE syn_keep BOOLEAN;

ATTRIBUTE syn_keep OF current_state :
SIGNAL IS TRUE";
```

In short, safe, efficient state machine design is a key skill for every engineer working with FPGAs. The choice among Moore, Mealy or even mixed machines depends upon the needs of your overall system. Whichever type of state machine you select, understanding the tools and techniques available for implementation will ensure you achieve the optimal solution.
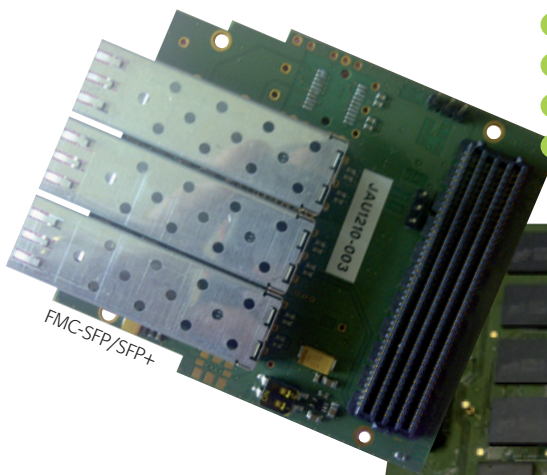
Bugs Be Gone!

Smarter debug and synthesis techniques will help you isolate errors and get your FPGA design to work on the board.

by Angela Sutton
Staff Product Marketing Manager
Synopsys
sutton@synopsys.com

When your FPGA design fails to synthesize or fails to operate as expected on the board, the cause may not be obvious and the source of the failure may be hard to pinpoint among potentially thousands of RTL and constraint source files, many of which may have been authored by another designer. Given how lengthy FPGA design iterations and runtimes have become, designers need to find and root out what may be a large group of errors early in the design process and seek methods to focus the validation of the design on the board.

It is essential to apply smarter techniques to isolate specific errors under specific conditions, relate parts of the circuit that are behaving badly back to the source and apply the incremental fixes. You can save time by performing rudimentary design setup checks on clocks, constraints and module-level interfaces with a view to making your design specification good, before you waste hours in synthesis and place-and-route (P&R).

Synopsys' Synplify Premier and Synplify Pro FPGA design tools and Identify RTL Debugger are among the products available to designers to handle these tasks. Such tools include features that enable designers to quickly isolate bugs and effectively reduce the length of runtimes and the number of iterations needed to bring up their boards.

PINPOINTING A PROBLEM ON THE BOARD

When functional errors become apparent on the board, it may be quite challenging to narrow down the cause of the problem. To debug the design, it is necessary to create additional circuitry and preserve certain nodes so that you can probe, tap and analyze data produced by the design under operation. Let's examine how to find errors using board-level debug software.

By following a four-step approach with an RTL debugger, you can pinpoint and then sample signals and conditions of interest. You can then relate your observations back to the original RTL to narrow down the cause of the problem, which may reside in the RTL specification or constraints setup.

Step 1: Specify probes. Specify within the RTL exactly what signals and conditions you want to monitor. Here you state watchpoints (signals or nodes to be observed) and breakpoints (RTL control flow statements such as IF, THEN and CASE statements) whose operation is of interest to you.

Step 2: Build the design with probes. Synthesize the FPGA design with additional monitoring circuitry: an intelligent in-circuit emulator (IICE), which captures and exports debug data according to the specification of what you wanted to monitor.

Step 3: Analyze and debug. After you have synthesized the design, run the design and observe the data using the RTL debugger. As you run your tests on the board, the watchpoints and breakpoints together will trigger data sampling, allowing you to observe and debug the circuit's behavior at very specific nodes under very specific conditions of interest. You can write the observed sampled data to a VCD file and relate it back to the RTL.

Step 4: Apply an incremental fix. Once you have found a bug, you can fix it incrementally in the RTL or constraints, using hierarchical and incremental flows.

VISUAL INSPECTION OF TIMING AND FUNCTIONAL ERRORS

FPGA design and debug tools have the added benefit of allowing the display of RTL and netlist-level schematics. For example, a schematic viewer tool with interactive debug capabilities displays RTL and netlist schematic representations of the design to allow you to visualize and relate reports of timing and VCD data (from your design operating on the board) back to the RTL source. The viewer includes an RTL View, available after the synthesis RTL compile phase, in which a design is schematically represented by technology-independent components such as adders, registers, large muxes and state machines. From this RTL schematic you cross-probe back to your original RTL, to adjust it if the design does not appear to be specified as intended, and to the constraints editor, where you can update and more easily specify constraints (Figure 1).
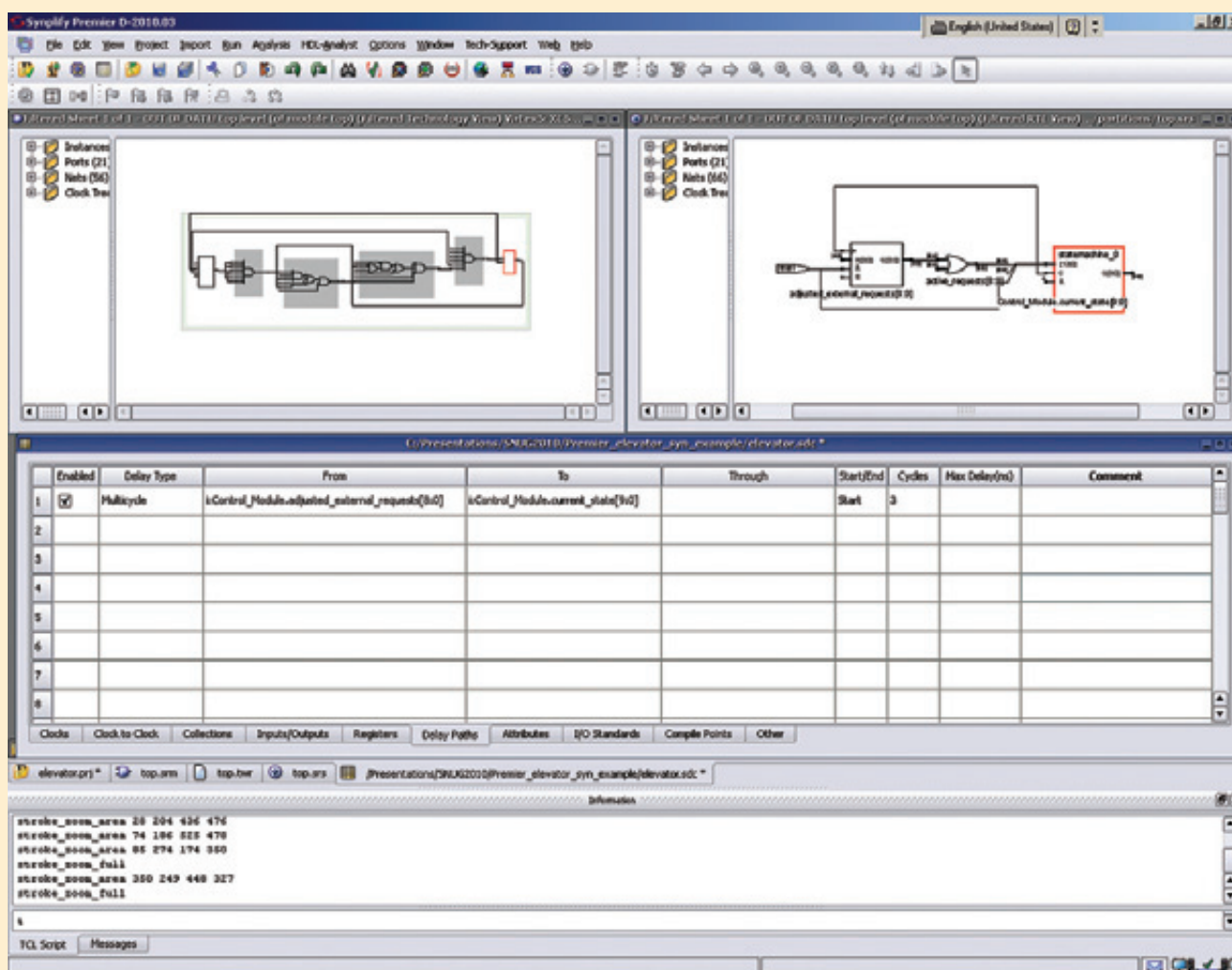

Figure 1 – You can debug RTL and constraints in the RTL View in a schematic viewing tool. Here you can relate constraints to the design spec and then drag and drop them directly into the constraints editor, where you can adjust or update them.

Gated clock structures in an ASIC that you may be prototyping in an FPGA are not necessary in the FPGA implementation, and cause inefficient use of FPGA resources. An efficient way to deal with this problem is to have the FPGA synthesis software convert the clocks.

To trace the source of any incorrect operation back to the RTL itself, you can use the RTL debugger to superimpose observed operation data live on top of the RTL schematic view.

Schematic viewers include a netlist-level Technology View showing the actual design implementation after synthesis. In the HDL Analyst schematic viewer, this view is based on Xilinx® device primitives such as lookup tables, registers and DSP slices. You can cross-probe paths in this schematic back to your original RTL, as well as your eventual timing reports post-synthesis and post-P&R to assess and improve overall performance.

DEBUGGING BIG DESIGNS

Probing all signals is out of the question in a big design, since the amount of data generated would be astronomical and the amount of additional debug logic required to probe the data too great. A common problem with on-chip debug methodologies is that it can be difficult to predict ahead of time which signals to probe and monitor.

Some debug software solves this problem to a degree by allowing you to take a divide-and-conquer approach. Multiplexed sample groups allow designers to selectively sample and switch between groups of signals through a multiplexed path and shared IICEs. This strategy increases the signals and conditions that you can observe without increasing data storage requirements. You can switch between signal groups of interest on the fly, without having to spend time re-instrumenting, and resynthesize a newly instrumented design.

Unfortunately, the very debug IICE logic that enables you to probe and sample data will consume chip resources, including memory Block RAMs. You can reduce on-chip BRAM usage by storing IICE sampled data off-chip in SRAM memory cards. This approach has the added benefit of increasing the depth of sample data that you can capture.

MY DESIGN WON'T SYNTHESIZE

Design errors may be occurring that prevent a clean synthesis and P&R run. Thousands of RTL and constraint source

files can turn that first successful synthesis and place-and-route run into a multiweek endeavor. When performing FPGA-based prototyping, it is necessary to render ASIC design source files "FPGA ready." One example is the need to achieve gated clock conversion.

Gated clock structures in an ASIC that you may be prototyping in an FPGA are not necessary in the FPGA implementation, and cause inefficient use of FPGA resources. An efficient way to deal with this problem is to have the FPGA synthesis software convert the clocks. For example, a gated/generated clock-conversion feature will move generated clocks and gated clock logic from the clock pin of a sequential element to the enable pin. This allows you to tie sequential elements directly to the source clock, removing skew issues and reducing the number of clock sources required in the design, saving resources.

To enable the Gated Clocks option in the Synplify Premier software:
 – Select **Project->Implementation Options**
 – On the **GCC & Prototyping Tools** tab, click on the **Clock Conversion checkbox**

Or in TCL, the command

```
set_option -fix_gated_and_generated_
clocks 1
```

performs gated and generated clock conversion in Synplify Pro/Premier, while

```
set_option -conv_mux_xor_gated_clocks 1
```

performs gated-clock conversion on muxes or OR gates in the clock tree in Synplify Premier for Synopsys HAPS-based designs.

A "complete" set of clock constraints includes having the clock defined in all the correct places and having defined relationships between the generated clocks. Clock conversion may fail upon the first attempt due to missing clock constraints, where a clock that feeds a sequential element is missing, or due to misplaced clock constraints, in which

```
#### START OF CLOCK OPTIMIZATION REPORT #####[

9 instances converted, 16 sequential instances remain driven by non-clean clocks

============================================ Gated/Generated Clock Combined Report ============================================
Clock Tree ID     Driving Element     Drive Element Type     Fanout     Sample Instance     Explanation
------------------------------------------------------------------------------------------------------------------------------
ClockId0001       comb1_u0.gclk       LUT4                    8          ff_reg1_u0.out1[2]  Multiple clocks on instance
ClockId0002       comb2_u2.gclk       LUT2                    8          ff_reg1_u3.out1[1]  No clock input found
==============================================================================================================================


##### END OF CLOCK OPTIMIZATION REPORT ######]
```

Figure 2 – Check the Gated Clock Report right after synthesis compilation to find out which clocks failed to convert successfully and why. Adding a more complete set of clock constraints that defines all clocks in the right places and specifies relationships between generated clocks will often fix the problem.

a clock has become disassociated from its true source for some reason, such as the creation of a black box between the clock source and its destination. In many cases, the failure to convert is due to incomplete constraints. For example, there might be a combinatorial loop in the gating logic that needs to be broken via an exception constraint before clock conversion can occur. A Gated Clock Report, available after the compile stage of synthesis, tells you which gated and generated clocks converted and their clock name, type, group and associated constraints. Another list of clocks that did not convert contains an error message explaining why. Figure 2 shows an example of the report.

If, for example, there are black boxes in your design, you can assist automated gated clock conversion further by specifying software-specific directives in your RTL. For example, by using the `syn_gatedclk_clock_en` directive you can specify the name of the enable pin within a black box, while the `syn_gatedclk_clock_en_polarity` directive indicates polarity of the clock enable port on a black box. Each converted instance and clock pin driving the instance is assigned a searchable property and can thus be identified in the design database and extracted into a custom TCL/Find script-generated report.

PORT MISMATCHES

A design can include files that originate from multiple sources both inside and outside of your company. When instantiating IP or preverified hierarchical blocks in a design, "port mismatch" specification mistakes are a common problem and can be difficult to fathom, especially when they occur in mixed-language designs. For example, if a top-level VHDL entity "Top" instantiates a Verilog module "sub," the top-level VHDL declares sub to have 4-bit ports whereas the actual Verilog module only has 3-bit ports. Synplify Premier software, for example, will flag a mismatch immediately and cite it in a separate log report as a hyperlinked error:

```
Interface Mismatch between View
work.sub.syn_black_box and View
work.sub.verilog

Details:

========

The following bit ports in the Source View
work.sub.syn_black_box do NOT exist in the
Target View work.sub.verilog

=========================================

        Bit Port in1[4]

        Bit Port in2[4]

        Bit Port dout[4]
```

With multiple levels of hierarchy, how do you trace the mismatch back to the offending module's RTL definition? It is necessary for the tool to tag all module instances in some way, for example with an orig_inst_of property. The property's value contains the original RTL name of the module for easy reference back to the RTL. Suppose, for example, an instance of sub called sub_3s incurs a port mismatch error. You can retrieve the RTL module's original name, "sub," using the following TCL commands:

```
get_prop -prop orig_inst_of {v:sub_3s}
which returns the value "sub"
```

CONSTRAINTS CLEANUP

Specifying adequate and correct constraints will affect both the quality of results and functionality. Constraints declarations should generally include three components: definition of primary clocks and clock groups; asynchronous clock declarations; and false and multicycle path declarations.
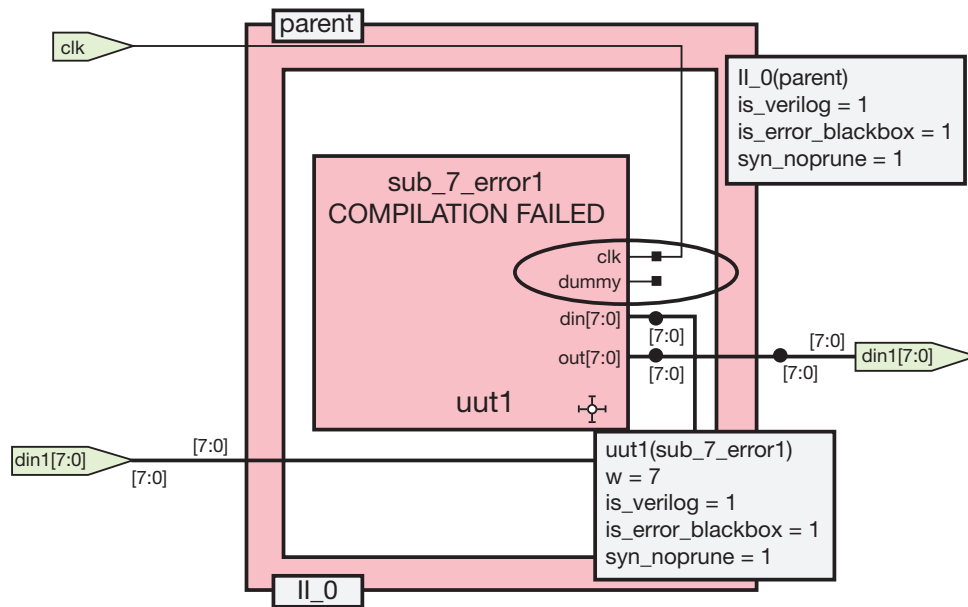
Figure 3 – The HDL Analyst RTL view highlights modules that contain errors,
as well as the parent module of an instance with an interface error, as shown.

Checking these constraints prior to synthesis is a good idea. A tool with a constraints checker that discovers syntax errors and looks at the applicability of timing constraints and instance names will alert you to problems. It will, for example, report on how constraints will be applied after wild-card expansions, as well as the clock relationships that resulted after you defined clock constraints. It will also flag timing constraints that aren't being applied due to nonexistent or invalid types of arguments and objects.

To generate a constraints checker report titled `projectName_cck.rpt` in the Synplify Pro/Premier software, prior to synthesis:

Synplify Pro/Premier GUI: **Run -> Constraint check**
Or using TCL: `project -run constraint_check`

Be sure to also avoid potential meta-instabilities by running an "asynchronous clock report" that will alert you to paths that start in one clock domain and end in another.

To generate the Clock Synchronization Report `projectName_async_clk.rpt.csv` in the Synplify Pro/Premier software:

Synplify Pro/Premier GUI: **Analysis->Timing Analyst** and select "**Generate Asynchronous Clock Report**" check box.

Using TCL: `set_option -reporting_async_clock`

Good practices include checking that you have adequately and completely constrained your design without having overconstrained it (which will result in longer runtimes and, potentially, the reporting of false critical paths). Be sure to check that you have completely specified multicycle and false paths, and that you have set constraints on derived clocks (**set_multicycle_path, set_false_path**).

SHORTENING THE DEBUG TIME

It can now take multiple hours to see the results of implementing a potential RTL or constraints fix. Let's examine how to reduce iterations using a combination of hierarchical "divide-and-conquer" design approaches and a continue-on-error (CoE) capability that allows you to uncover multiple errors in a single synthesis iteration.

Block-based flows have become necessary in order to save runtime. These flows also allow for design preservation, or the ability to lock down parts of the design that already work. A tool that supports block-based flows allows you to create RTL partitions, called compile points, prior to synthesis. Some software also enables designers to black-box a portion of the design with errors and completely export that portion as its own separate design subproject for rework. Once fixed, the subproject will be merged back either as a netlist using a bottom-up flow or as RTL using a top-down flow—or even using a hybrid of both top-down and bottom-up flows.

To integrate and troubleshoot large designs, it is important to find groups of specification errors as early as possible in the design process. For example, a CoE feature provides an aggregate report of errors with each synthesis

pass. CoE tolerates nonfatal, nonsyntax HDL compilation problems and certain mapping errors so that designers can analyze and complete as much of the design as possible with each synthesis iteration. To invoke the CoE feature in the Synplify Premier software with the Synplify Pro/Premier GUI, enable the Continue-on-Error checkbox on the left side of the Project View.

With TCL: `set_option –continue_on_error 1`

Modules containing errors and the parent module of instances that have interface errors are flagged with a property `is_error_blackbox=1` and highlighted graphically as shown in Figure 3.

To find all "Erroneous Instances" using TCL:
```
c_list [find –hier –inst * -filter
@is_error_blackbox==1]
```

To list all the "Erroneous Modules" with TCL:
```
get_prop -prop inst_of [find –hier –inst
* -filter @is_error_blackbox==1]
```

To view Erroneous Modules ready for black-boxing or export, look for red boxes in the HDL Analyst RTL View (Figure 3).

ISOLATING PROBLEMS BY EXPORTING A MODULE

You can export modules that have errors as a completely standalone synthesis project in order to perform focused debug of the module. The export process creates an isolated synthesis project containing all of the individual module's source files, language standard and compiled libraries, including directory paths and the path order for included files, required to synthesize and debug the module on its own. Modules that incur errors are automatically tagged with an error property in the design database, as indicated in the previous section, and are highlighted in the design schematic. This makes it easy to find the modules that you want to extract for rework.

To export a module and all associated source files for debug in isolation, in the Synplify Pro/Premier software GUI (Figure 4), first select a design block or instance in the Design Hierarchy view or RTL view. Then, right click and select "Generate Dependent File List" from the pop-up menu.

As each hierarchical module is fixed you can reintegrate it into the design, either as RTL to be resynthesized in the context of the entire design (top-down synthesis flow) or as a netlist (bottom-up flow). See Figure 5, opposite.

Meeting timing with the inevitable use of design hierarchies can be a challenge. Hierarchical boundaries may limit performance unless timing budgets are created for each hierarchical partition of the design. Some tools automatical-

ly set timing budgets when you use RTL partitions called manual locked compile points. Synplify Pro/Premier software also has automatic compile points that create automatic partitions under the hood, for example for runtime speedup using multiprocessing. The budgeting feature produces interface logic models (ILMs) for each RTL partition, with the result that the software can figure out how to meet timing targets for each partition. You can override manual locked compile point automatic timing budgets by specifying a constraints file for each compile point.

In a recent Global User Survey conducted by Synopsys, 59 percent of designers identified "getting the design specification right" as one of their most significant design challenges. This challenge can translate to design delays and, in worst-case scenarios, design failure. Design tools must adapt to catch errors early and provide greater visibility into the design's operation so that the specification can be validated and fixed. These tools must also provide a means of feedback on proposed design fixes.
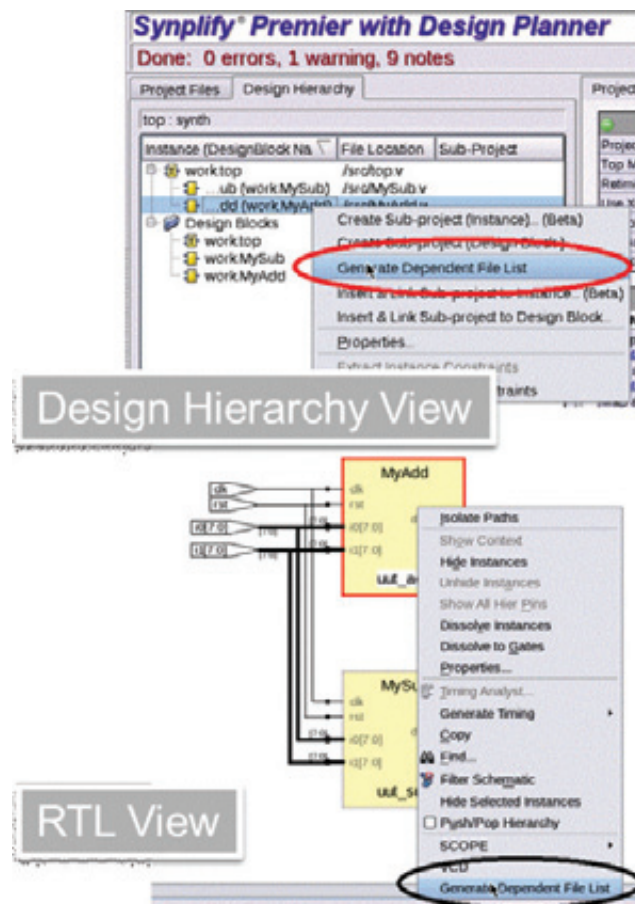

Figure 4 – Problematic modules in the design can be exported as their own separate synthesis project for rework, allowing you to isolate the problem.
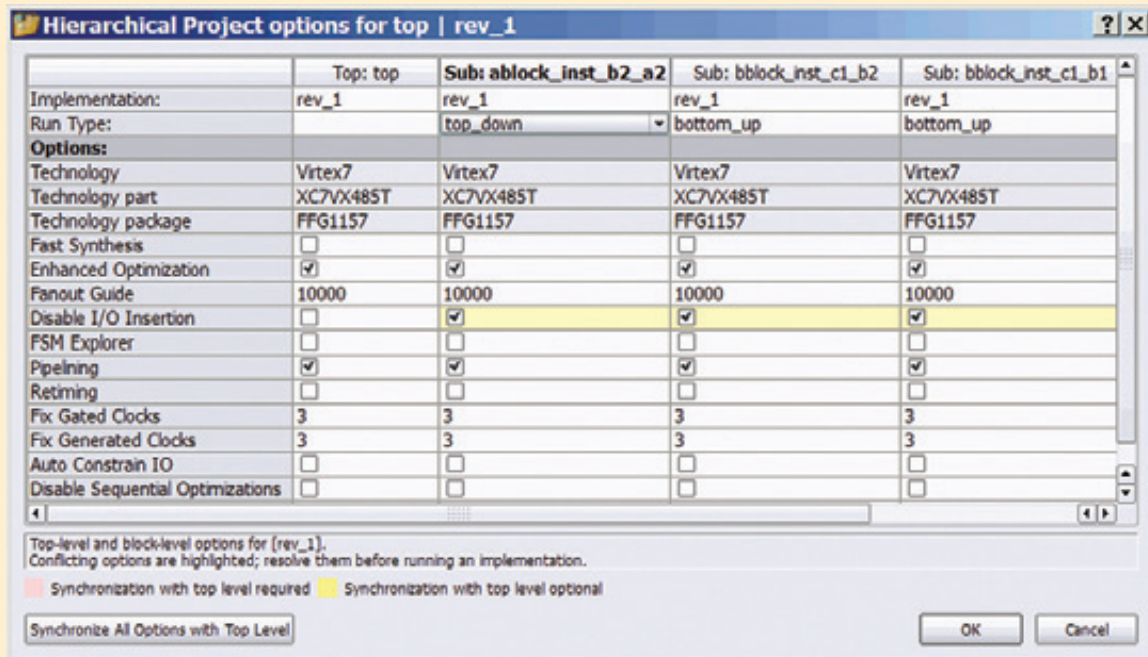
Figure 5 – As working modules become available, you can reintegrate them into the design, using either a top-down (RTL) or bottom-up (netlist-level) integration.

Application Notes

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

XAPP741: DESIGNING HIGH-PERFORMANCE VIDEO SYSTEMS IN 7 SERIES FPGAS WITH THE AXI INTERCONNECT

http://www.xilinx.com/support/documentation/application_notes/xapp741-high-performance-video-AXI-interconnect.pdf

This application note describes the design considerations of a video system using the performance features of the LogiCORE™ IP Advanced eXtensible Interface (AXI) interconnect core. The design focuses on high system throughput using approximately 80 percent of DDR memory bandwidth through the AXI interconnect core, with Fmax and area optimizations in certain portions of the design.

Eight AXI video direct-memory access (VDMA) engines simultaneously move 16 streams (eight transmit video streams and eight receive video streams), each in 1,920 x 1,080-pixel format, at 60- or 75-Hz refresh rates and up to 32 data bits per pixel. Each VDMA is driven from a video test pattern generator (TPG), with a video timing controller (VTC) block to set up the necessary video timing signals. Data read by each AXI VDMA is sent to a common on-screen display (OSD) core capable of multiplexing or overlaying multiple video streams to a single output video stream. The output of the OSD core drives the onboard HDMI video display interface through the color-space converter.

Authors Sateesh Reddy Jonnalagada and Vamsi Krishna add a performance monitor block to their design to capture DDR performance metrics. DDR traffic passes through the AXI interconnect to move 16 video streams over eight VDMA pipelines. All 16 video streams are buffered through a shared DDR3 SDRAM and are controlled by a MicroBlaze™ processor. The reference system is targeted for the Kintex™-7 FPGA XC7K325TFFG900-1 on the Xilinx® KC705 evaluation board (Revision C or D).

XAPP742: AXI VDMA REFERENCE DESIGN

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

The same authors here describe how to create video systems by using Xilinx native video IP cores to process configurable frame rates and resolutions in Kintex-7 FPGAs. Among the cores that are useful in such applications are the AXI video direct-memory access (VDMA), the video timing controller (VTC), the test pattern generator (TPG) and the DDR3 memory controller.

The reference design focuses on runtime configuration of an onboard clock generator for a video pixel clock and on video IP blocks for running selected combinations of video resolution and frame rate, common metrics that system designers use. The design displays system-level bandwidth utilization and video latency for each combination of frame rate and resolution. Authors Sateesh Reddy Jonnalagada and Vamsi Krishna discuss the configuration of each video IP block in detail, helping designers to make effective use of such IP for processing various video features. The reference design is targeted for the Kintex-7 FPGA XC7K325TFFG900-1 on the Xilinx KC705 evaluation board (Revision C).

XAPP552: PARAMETERIZABLE CORDIC-BASED FLOATING-POINT LIBRARY OPERATIONS

http://www.xilinx.com/support/documentation/application_notes/xapp552-cordic-floating-point-operations.pdf

This application note presents a way to use the Xilinx System Generator for DSP tool to create a parameterizable floating-point library computation method for trigonometric, power and logarithmic operations based on the coordinate rotational digital computer (CORDIC)

algorithm. The design methodology leverages the fixed-point CORDIC LogiCORE IP v5.0 block, along with floating-point building blocks such as adders, multipliers, comparators, ROM and FIFOs to create a set of floating-point CORDIC functions that can serve as building blocks in applications. These functions are essential in a wide range of engineering applications such as image processing, manipulator kinematics, radar signal processing, robotics and optimization processes, in which a large number of trigonometric or power operations must be computed efficiently.

Authors Nikhil Dhume and Ramakrishnan Srinivasakannan designed a floating-point library for computation of trigonometric, power and logarithmic operations using System Generator for DSP, version 13.4, by applying range-extension algorithms to underlying fixed-point blocks. The library supports single- and double-precision input as defined by the IEEE-754 floating-point standard. The authors demonstrate performance similar to that of the underlying fixed-point block.

XAPP518: IN-SYSTEM PROGRAMMING OF BPI PROM FOR VIRTEX-6 FPGAS USING PCI EXPRESS TECHNOLOGY

http://www.xilinx.com/support/documentation/application_notes/xapp518-isp-bpi-prom-virtex-6-pcie.pdf

Many systems use Byte-wide Peripheral Interface (BPI) flash memory for FPGA configuration and system data storage. Often it is not desirable or even possible to update the flash PROM directly after the system is deployed. One way to address this issue is to use the FPGA to program the PROM to which it is connected, a methodology called in-system programming (ISP). An example of ISP is the indirect programming capability supported by iMPACT, a tool featuring batch and GUI operations. In this case, iMPACT uses the JTAG interface port as the communication channel between a host and the FPGA. The iMPACT tool sends the BIT file to the FPGA, which in turn programs the PROM attached to it.

However, many embedded systems do not have JTAG interface connections. The FPGA is often an endpoint on the PCI Express® bus. Because no JTAG interface channel is available through the standard PCIe® peripheral, the only way to program a PROM on the endpoint is to program across the PCIe system. This application note by Simon Tan provides an ISP reference design to demonstrate the methodology and considerations of programming in-system BPI PROM in a Virtex®-6 PCIe system.

XAPP583: USING A MICROPROCESSOR TO CONFIGURE 7 SERIES FPGAS VIA SLAVE SERIAL OR SLAVE SELECTMAP MODE

http://www.xilinx.com/support/documentation/application_notes/xapp583-fpga-configuration.pdf

Many designers of embedded systems are looking to reduce component count and increase flexibility. To accomplish both of these goals, they can use a microprocessor already available in the system to configure an FPGA. This application note provides a thorough discussion of how to do so, covering the Xilinx 7 series FPGAs. Author Matt Nielson includes C code to illustrate an example application using either Slave Serial or Slave SelectMAP mode. The example configures a Kintex-7 XC7K325T device from a MicroBlaze processor.

This application note provides background on configuration as well as two complete sets of reference designs. Although the microprocessor C code targets a Xilinx MicroBlaze processor, it was written with portability in mind. Porting the code to another processor requires some effort, but all the design files are documented extensively.

XAPP587: BPI FAST CONFIGURATION AND IMPACT FLASH PROGRAMMING WITH 7 SERIES FPGAS

http://www.xilinx.com/support/documentation/application_notes/xapp587-bpi-fast-configuration.pdf

The 7 series FPGA Byte-wide Peripheral Interface (BPI) configuration mode with synchronous read and the External Master Configuration Clock (EMCCLK) options together provide the fastest configuration time from a direct FPGA interface. Author Stephanie Tapp refers to the combination as BPI Fast Configuration in this application note, which describes hardware setup, file generation and tool flow to indirectly program a parallel NOR flash through a 7 series FPGA. BPI Fast Configuration enables high-capacity nonvolatile storage and decreases the configuration time to less than 8 percent of the legacy BPI configuration with the asynchronous read. The application note uses the Virtex-7 FPGA and 28F00AG18F parallel NOR flash on the VC707 evaluation board to demonstrate the flow with the ISE® Design Suite 14.1.

What's New in the Vivado 2012.3 Release?

The Vivado™ Design Suite 2012.3 is now available, at no additional cost, to all Xilinx® ISE® Design Suite customers that are currently in warranty. The Vivado Design Suite provides a highly integrated design environment with a completely new generation of system-to-IC-level features, including high-level synthesis, analytical place-and-route and an advanced timing engine. These tools enable developers to increase design integration and implementation productivity.

THE VIVADO DESIGN SUITE 2012.3

■ New multithreaded place-and-route technology accelerates design productivity even further on multicore workstations, with 1.3x faster runtimes on a dual-core processor and 1.6x faster runtimes on a quad-core.

■ Vivado extends the portfolio of very popular Targeted Reference Designs (TRD) for the Kintex™-7 and Virtex®-7 families to further accelerate designer productivity.

  • The Kintex-7 FPGA Base TRD showcases the capabilities of Kintex-7 FPGAs through a fully integrated PCIe® design that delivers 10-Gbit/second (Gbps) performance end-to-end using a performance-optimized DMA engine and DDR3 memory controller.

  • The Kintex-7 FPGA Connectivity TRD delivers up to 20 Gbps of performance per direction featuring a dual network interface card (NIC) with a Gen2 x8 PCIe endpoint, a multichannel packet DMA, a DDR3 memory for buffering, a 10G Ethernet MAC and a 10GBASE-R standard compatible physical-layer interface.

  • The Kintex-7 FPGA Embedded TRD offers a comprehensive processor subsystem complete with Gigabit Ethernet, DDR3 memory controller, display controller and other standard processor peripherals.

  • The Kintex-7 FPGA DSP TRD includes a high-speed analog interface with digital up- and downconversion overclocked to run at 491.52 MHz.

  • You can find a complete list of Vivado Design Suite-supported TRDs at *http://www.xilinx.com/ise/design_tools/support.htm*.

NEW DEVICE SUPPORT

The following devices will be in production fully supporting design with both the Vivado Design Suite 2012.3 and ISE Design Suite 14.3:

  • Kintex-7 70T, 410T, 480T, 420T, 355T, 325T (Low Voltage), 160T (Low Voltage)

  • Virtex-7 X485T (Low Voltage)

The following devices will be in general engineering sampling (ES) supporting both Vivado Design Suite 2012.3 and ISE Design Suite 14.3:

  • Virtex-7 X690T

The following devices will be in general ES in Vivado Design Suite 2012.3:

  • Virtex-7 X1140T, 2000T

  • Artix™-7 100T, 200T

The following devices will be in initial ES in Vivado Design Suite 2012.3:

  • Virtex-7 H580T

For further information regarding the release, please visit *http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug910-vivado-getting-started.pdf*.

WHAT IS THE VIVADO DESIGN SUITE?

It's all about improving designer productivity. This entirely new tool suite was architected to increase your overall productivity in designing, integrating and implementing with the 28-nanometer family of Xilinx All

Programmable devices. With 28-nm manufacturing, Xilinx devices are now much larger and come with a variety of new technologies including stacked-silicon interconnect, high-speed I/O interfaces operating at up to 28 Gbps, hardened microprocessors and peripherals, and analog/mixed-signal. These larger and more-complex devices present developers with multidimensional design challenges that can prevent them from hitting market windows and increasing productivity.

The Vivado Design Suite is a complete replacement for the existing Xilinx ISE Design Suite of tools. It replaces all of the ISE Design Suite point tools. All of those capabilities are now built directly into the Vivado integrated development environment (IDE), leveraging a shared scalable data model. With the Vivado Design Suite, developers are able to accelerate design creation with high-level synthesis and implementation by using place-and-route to analytically optimize for multiple and concurrent design metrics, such as timing, congestion, total wire length, utilization and power. Built on Vivado's shared scalable data model, the entire design process can be executed in memory without the need to write or translate any intermediate file formats, accelerating runtimes, debug and implementation while reducing memory requirements.

Vivado provides users with upfront metrics that allow for design and tool-setting modifications earlier in the design process, when they have less overall impact on the schedule. This capability reduces design iterations and accelerates productivity. Users can manage the entire design process in a pushbutton manner by using the Flow Navigator feature in the Vivado IDE, or control it manually by using Tcl scripting.

SHOULD I CONTINUE TO USE THE ISE DESIGN SUITE OR MOVE TO VIVADO?

ISE Design Suite is an industry-proven solution for all generations of Xilinx's All Programmable devices. The Xilinx ISE Design Suite continues to bring innovations to a broad base of developers, and extends the familiar design flow for 7 series and Xilinx Zynq™-

> The Vivado Design Suite is a complete replacement for the existing ISE Design Suite. All of the capabilities of the ISE point tools are built directly into the Vivado IDE.

7000 All Programmable SoC projects. ISE 14.3, which brings new innovations and contains updated device support, is available for immediate download.

The Vivado Design Suite 2012.3, Xilinx's next-generation design environment, supports 7 series devices including Virtex-7, Kintex-7 and Artix-7 FPGAs. It offers enhanced tool performance, especially on large or congested designs.

IS VIVADO DESIGN SUITE TRAINING AVAILABLE?

Vivado is new and takes full advantage of industry standards such as powerful interactive Tcl scripting, Synopsys Design Constraints, SystemVerilog and more. To reduce your learning curve, Xilinx has rolled out 10 new instructor-led classes to show you how to use the Vivado tools. We also encourage you to view the Vivado Quick Take videos found at *www.xilinx.com/training/vivado*.

ARE THERE DIFFERENT EDITIONS OF THE VIVADO DESIGN SUITE?

The Vivado Design Suite comes in two editions: Design and System. In-warranty ISE Design Suite Logic and Embedded Edition customers will receive the new Vivado Design Edition. ISE Design Suite DSP and System Edition customers will receive the new Vivado System Edition. Vivado is not yet available for WebPACK™ users. Vivado WebPACK is currently planned for later this year. For more information about Xilinx design tools for the next generation of All Programmable devices, please visit *www.xilinx.com/design-tools*.

Xilinx recommends that customers starting a "new" design on a Kintex K410 or larger device contact their local FAE to determine if Vivado is right for the design. Xilinx does not recommend transitioning during the middle of a current ISE Design Suite project, as design constraints and scripts are not compatible between the environments.

For more information, please read the ISE 14.3 and Vivado 2012.3 release notes.

WHAT ARE THE LICENSING TERMS FOR VIVADO?

There is no additional cost for the Vivado Design Suite during the remainder of 2012. A single download at the Xilinx download center contains both ISE Design Suite 14.3 and Vivado 2012.3. All current, in-warranty seats of ISE Design Suite are entitled to a copy of the Vivado Design Suite.

For customers who generated an ISE Design Suite license for versions 13 or 14 after Feb. 2, 2012, your current license will also work for Vivado. Customers who are still in warranty but who generated licenses prior to February 2 will need to regenerate their licenses in order to use Vivado.

For license generation, please visit *www.xilinx.com/getlicense*.

Xpress Yourself
in Our Caption Contest

DANIEL GUIDERA

I f you're feeling frisky and looking to Xercise your funny bone, here's your opportunity. We invite readers to claw their way to our verbal challenge and submit an engineering- or technology-related caption for this cartoon showing what can happen when you turn your laboratory over to a clowder of cats. Anyone with cat hair gumming up their keyboards can relate. The image might inspire a caption like "It's amazing what they can do, with or without a mouse."

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 an Avnet Zedboard, featuring the Zynq™-7000 All Programmable SoC (*http://www.zedboard.org/*). Two runners-up will gain notoriety, fame and a cool, Xilinx-branded gift from our swag closet.

The contest begins at 12:01 a.m. Pacific Time on Oct. 23, 2012. All entries must be received by sponsor by 5 p.m. PT on Jan. 7, 2013.

So, get out your catnip and pounce!

WILLIAM BISHOP, continuing lecturer in the Department of Electrical and Computer Engineering at the University of Waterloo, in Ontario, won a shiny new Avnet Spartan®-6 LX9 MicroBoard with this caption for the cartoon of the mime from Issue 80 of *Xcell Journal*:

"I requisitioned an Artix, not an artist!"

Congratulations as well to our two runners-up:

"Misreading the FPGA design review checklist, Harold presents his static miming analysis."

– Greg Harrison, senior hardware engineer, Sealevel Systems. Liberty, S.C.

Age-old sales dilemma finally solved: How to bring an engineer along to a customer meeting without letting him say too much!

– Vinay Agarwala, PMTS, design engineering, SeaMicro (now AMD)