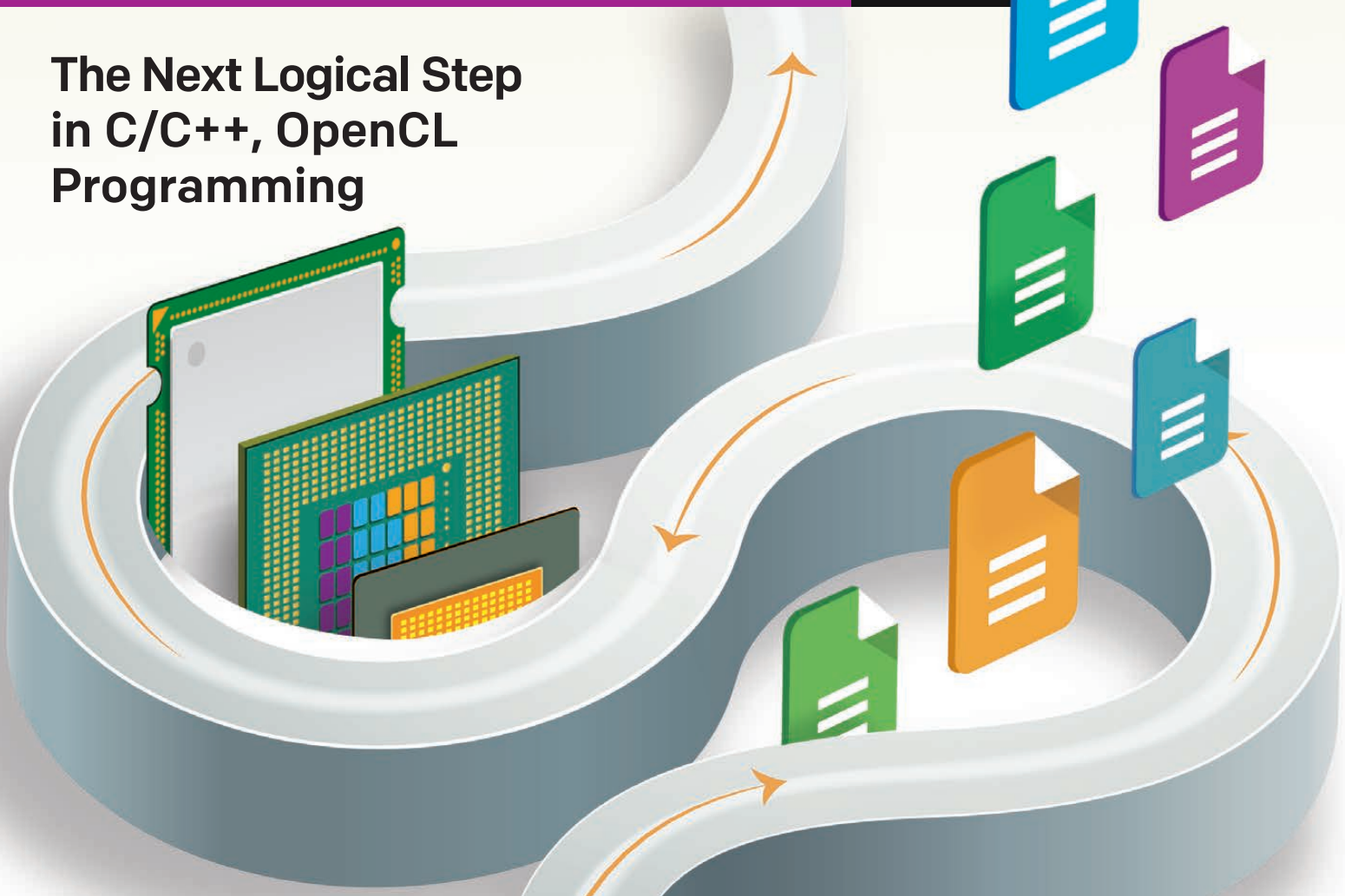
Xcell SOFTWARE journal

SOFTWARE SOLUTIONS FOR
A PROGRAMMABLE WORLD

ISSUE 1
THIRD QUARTER 2015

The Next Logical Step in C/C++, OpenCL Programming



Exploring the SDSoC Environment: Build a Sample Design

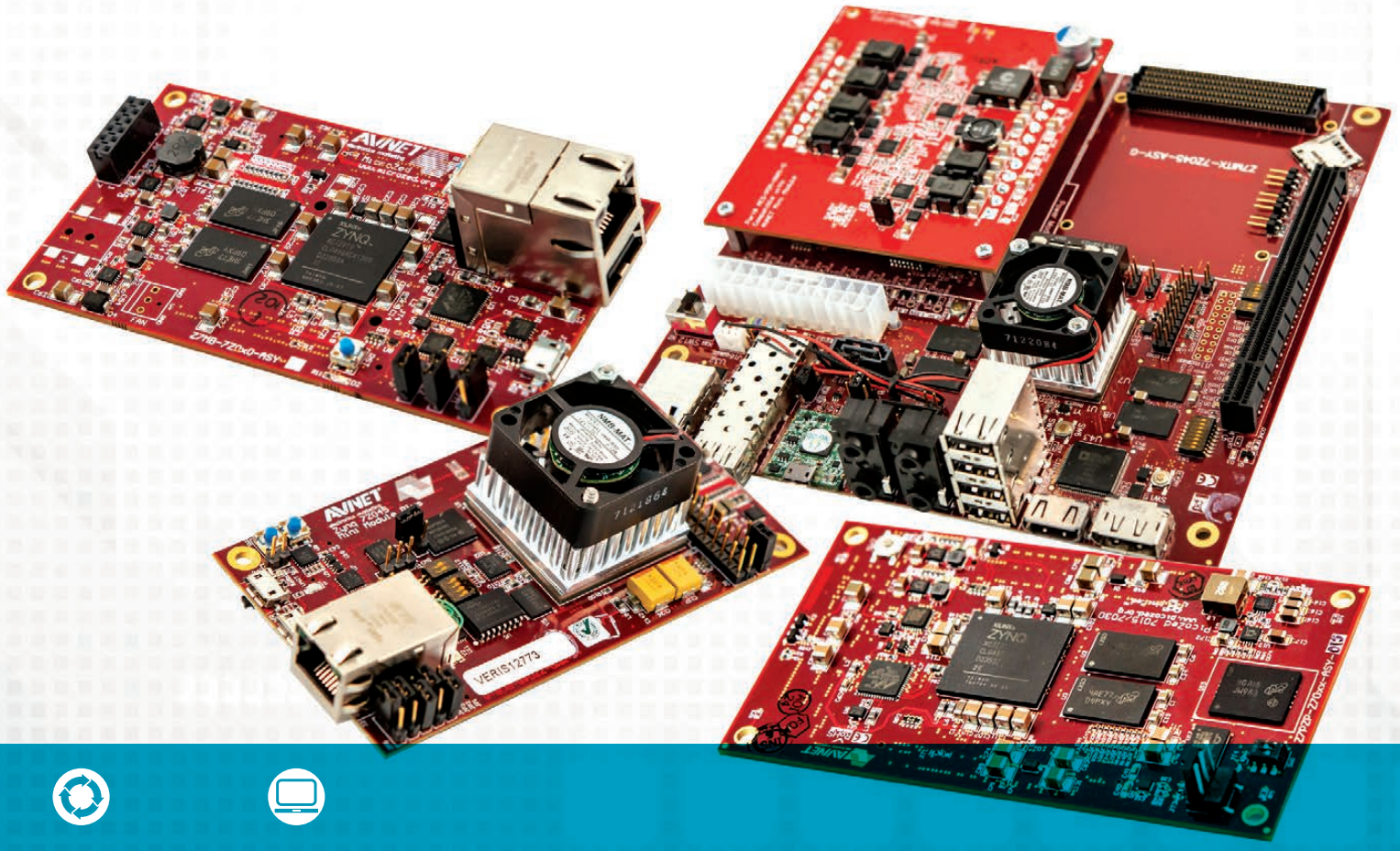
System-level HW-SW Optimization on the Zynq SoC

SDAccel Software Application Design Flows for FPGAs

MathWorks: Make Design Trade-offs at the Desktop, Not the Lab

XILINX
ALL PROGRAMMABLE™

www.xilinx.com/xcell



Lifecycle

Technology

Design it or Buy it?

Shorten your development cycle with Avnet's SoC Modules

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

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

DESIGNED BY AVNET

MMP

picoZed™

microZed™ FLEXIBILITY REDEFINED MICROZED.ORG

facebook.com/avnet twitter.com/avnet youtube.com/avnet

Accelerating Your Success™

Xcell SOFTWARE Journal

PUBLISHER Mike Santarini
mike.santarini@xilinx.com
1-408-626-5981

EDITOR Diana Scheben

ART DIRECTOR Scott Blair

DESIGN/PRODUCTION Teie, Gelwicks & Assoc.
1-408-842-2627

ADVERTISING SALES Judy Gelwicks
1-408-842-2627
xcelladsales@aol.com

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

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

Tomoko Suto,
Japan
tomoko@xilinx.com

REPRINT ORDERS 1-408-842-2627

EDITORIAL ADVISERS Tomas Evensen
Lawrence Getman
Mark Jensen

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

Letter from the Publisher

Welcome to *Xcell Software Journal*

Earlier this year, Xilinx® released its SDx™ line of development environments, which enable non-FPGA experts to program Xilinx device logic using C/C++ and OpenCL™. The goal of the SDx environments is to let software developers and embedded system architects program our devices as easily as they program GPUs or CPUs. Xilinx's FPGA hardware technology has long been able to accelerate algorithms, but it was only recently that the underlying software technology and hardware platforms reached a level where it was feasible to create these development environments for the broader software- and system-architect communities of C/C++ and OpenCL users.

The hardware platforms have evolved rapidly during this millennium. In the early 2000s, the semiconductor industry changed the game on software developers. To avoid a future in which chips reached the energy density of the sun, MPU vendors switched from monolithic MPUs to homogeneous multicore, distributed processing architectures. This switch enabled the semiconductor industry to continue to introduce successive generations of devices in cadence with Moore's Law and even to innovate heterogeneous multicore processing systems, which we know today as systems-on-chip (SoCs). But the move to multicore has placed a heavy burden on software developers to design software that runs efficiently on these new distributed processing architectures. Xilinx has stepped in to help software developers by introducing its SDx line of development environments. The environments let developers dramatically speed their C/C++ and OpenCL code running on systems powered by next-generation processing architectures, which today are increasingly accelerated by FPGAs.

Indeed, FPGA-accelerated processing architectures, pairing MPUs with FPGAs, are fast replacing power-hungry CPU/GPU architectures in data center and other compute-intensive markets. Likewise, in the embedded systems space, new heterogeneous multicore processors such as Xilinx's Zynq®-7000 All Programmable SoC and upcoming Xilinx UltraScale+™ MPSoC integrate multiple processors with FPGA logic on the same chip, enabling companies to create next-generation systems with unmatched performance and differentiation. FPGAs have traditionally been squarely in the domain of hardware engineers, but no longer.

Now that Xilinx has released its SDx line of development environments to use on its hardware platforms, the software world has the ability to unlock the acceleration power of the FPGA using C/C++ or OpenCL within environments that should be familiar to embedded-software and -system developers. This convergence of strong underlying compilation technology for our high-level synthesis (HLS) tool flow with programming languages and tools designed for heterogeneous architectures brings the final pieces together for software and system designers to create custom hardware accelerators in their own heterogeneous SoCs.

Xcell Software Journal is dedicated to helping you leverage the SDx environments and those from Xilinx Alliance members such as National Instruments and MathWorks®. The quarterly journal will focus on software trends, case studies, how-to tutorials, updates and outlooks for this rapidly growing user base. I'm confident that as you read the articles you will be inspired to explore Xilinx's resources further, testing out the SDx development environments accessible through the Xilinx Software Developer Zone. I encourage you to read the Xcell Daily Blog, especially Adam Taylor's chronicles of using the SDSoC development environment. And I invite you to contribute articles to the new journal to share your experiences with your colleagues in the vanguard of programming FPGA-accelerated systems.

— Mike Santarini
Publisher

There is a new bass player for the blues jam in the sky . . .
This issue is dedicated to analyst and ESL visionary Gary Smith, 1941 – 2015.

CONTENTS

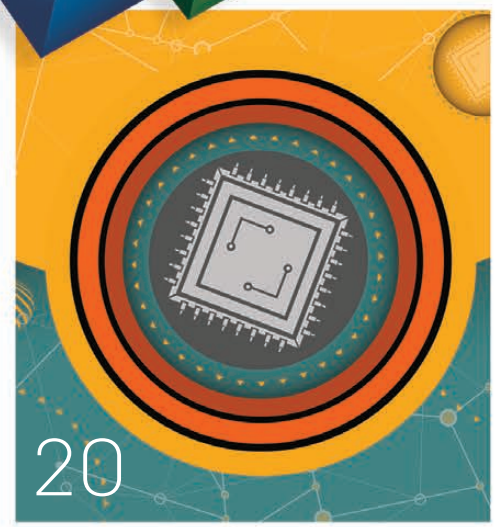
THIRD QUARTER
2015
ISSUE 1

VIEWPOINT

COVER STORY

XCELLENCE WITH SDSOC FOR EMBEDDED DEVELOPMENT

XCELLENCE WITH SDACCEL FOR APPLICATION ACCELERATION



XCELLENT ALLIANCE FEATURES

XTRA READING

The Next Logical Step in C/C++, OpenCL Programming

New environments allow you to maximize code performance.

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

Lawrence Getman
Vice President, Corporate Strategy and Marketing
Xilinx, Inc.
larryg@xilinx.com

Ever since Xilinx® invented and brought to market the world's first FPGAs in the early 1980s, these extraordinarily versatile programmable logic devices have been the MacGyver multipurpose tool of hardware engineers. With Xilinx's recent releases of the SDx™ line of development environments—SDAccel™, SDSoC™ and SDNet™—Xilinx is empowering a greater number of creative minds to bring remarkable innovations to the world by enabling software developers and systems engineers (non-FPGA designers) to create their own custom software-defined hardware easily with Xilinx devices.

Before we take a look at these new environments and other software development resources from Xilinx and its Alliance members, let's consider the evolution of processing architectures and their impact on software development.

IT'S A SOFTWARE PROBLEM ...

Prior to 2000, the typical microprocessor largely comprised one giant monolithic processor core with onboard memory and

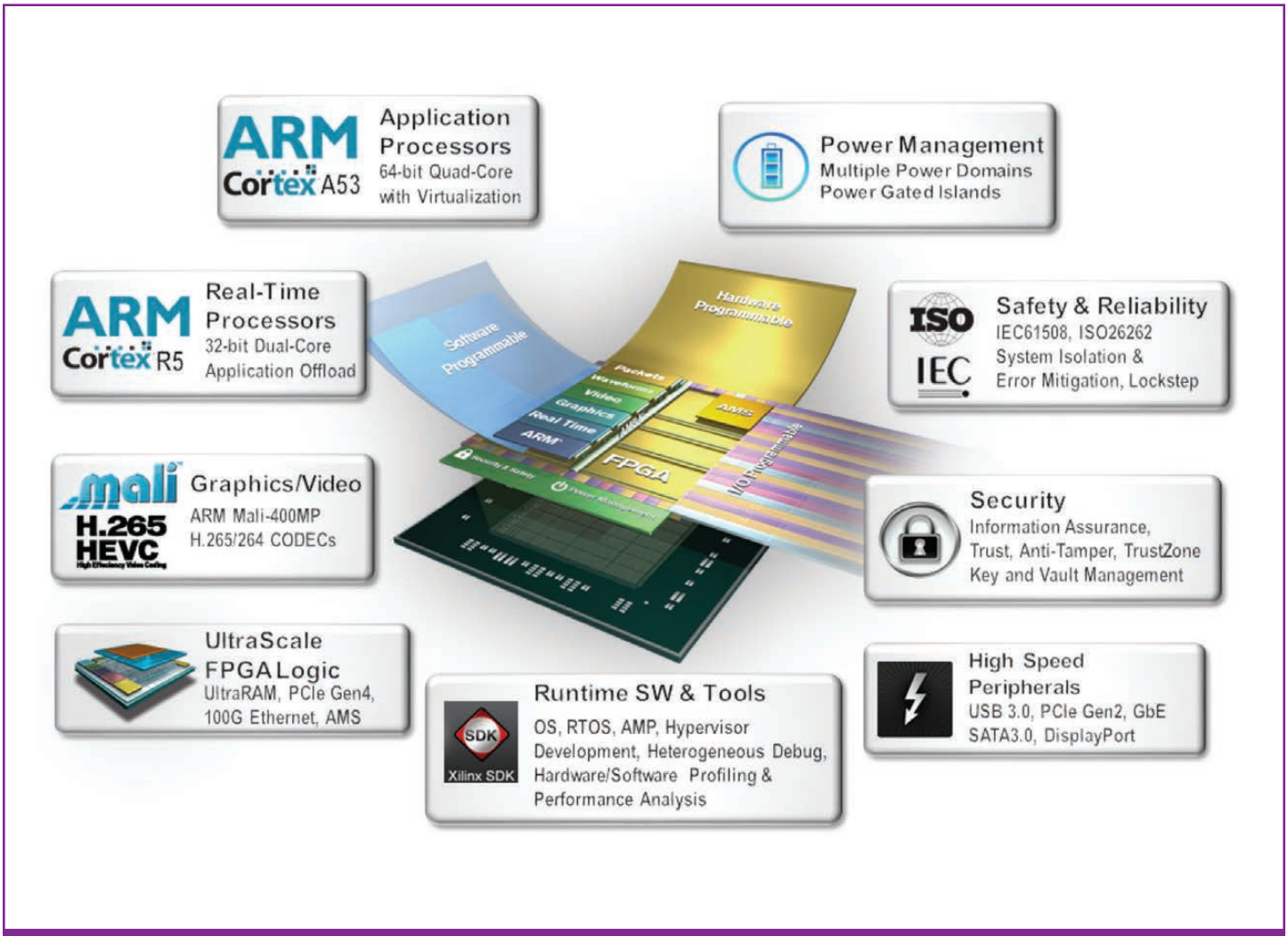


Figure 1 — The Zynq UltraScale+ MPSoC

a few other odds and ends, making MPUs relatively straightforward platforms on which to develop next-generation apps. For three decades leading up to that point, every 22 months—in step with Moore's Law—microprocessor vendors would introduce devices with greater capacity and higher performance. To increase the performance, they would simply crank up the clock rate. The fastest monolithic MPU of the time, Intel's Pentium 4 Pro, topped out at just over 4 GHz. For developers, this evolution was great; with every generation, their programs could become more intricate and perform more elaborate functions, and their programs would run faster.

But in the early 2000s, the semiconductor industry changed the game, forcing developers to adjust to a new set of rules. The shift started with the realization that if the MPU industry continued to crank up the clock on each new monolithic MPU architecture, given the silicon process technology road map and worsening transistor leakage, MPUs would soon have the same power density as the sun.

It was for this reason that the MPU industry quickly transitioned to a homogeneous multiprocessing architecture, in which computing was distributed to multiple smaller cores running at lower clock rates. The new processing model let MPU and semiconductor vendors continue to produce new generations of higher-capacity devices and reap more performance mainly from integrating more functions together in a single piece of silicon. Existing programs could not take advantage of the new distributed architectures, however, leaving software developers to figure out ways to develop programs that would run efficiently across multiple processor cores.

The SDAccel environment includes a fast, architecturally optimizing compiler that makes efficient use of on-chip FPGA resources.

Meanwhile, as these subsequent generations of silicon process technologies continued to double transistor counts, they enabled semiconductor companies to take another innovative step and integrate different types of cores on the same piece of silicon to create SoCs. These heterogeneous multiprocessor architectures posed additional challenges for embedded software developers, who now had to develop custom software stacks to get applications to run optimally on their targeted systems.

Today, the semiconductor industry is changing the game yet again—but this time software developers are welcoming the transition. Faced with another power dilemma, semiconductor and systems companies are turning to FPGA-accelerated heterogeneous processing architectures, which closely pair MPUs with FPGAs to increase system performance at a minimal power cost. This emerging architecture has been most notably leveraged in new data center processing architectures. In a now-famous paper, Microsoft researchers showed that the architectural pairing of an MPU and FPGA produced a 90 percent performance improvement with only a 10 percent power increase, producing far superior performance per watt than architectures that paired MPUs with power-hungry GPUs.

The advantages of FPGA-accelerated heterogeneous multiprocessing extend beyond data center applications. Numerous embedded systems using Xilinx's Zynq®-7000 All Programmable SoC have greatly benefited from the devices' on-chip marriage of ARM processors and programmable logic. Systems created with the upcoming Zynq UltraScale+™ MPSoC are bound to be even more impressive. Zynq UltraScale+ MPSoC integrates into one device multiple ARM® cores (quad Cortex™-A53 applications processors, dual Cortex-R5 real-time processors and a Mali™-400MP GPU), programmable logic, and multiple levels of security, increased safety and advanced power management (Figure 1).

But to make these FPGA-accelerated heterogeneous architectures practical for mass deployment and accessible to software developers, FPGA vendors have had to develop novel environments. In Xilinx's case, the company offers three development platforms: SDAccel for data center developers, SDSoC for embedded systems developers, and SDNet for network line card architects and developers. The new Xilinx environments give developers the tools to accelerate their programs by easily programming slow portions of their code onto programmable logic to create optimized systems.

SDACCEL FOR OPENCL, C/C++ PROGRAMMING OF FPGA-ACCELERATED PROCESSING

The new Xilinx SDAccel development environment gives data center application developers a complete FPGA-based hardware and software solution (Figure 2). The SDAccel environment includes a fast, architecturally optimizing compiler that makes efficient use of on-chip FPGA resources. The environment provides developers with a familiar CPU/GPU-like work environment and software-development flow, featuring an Eclipse-based integrated design environment (IDE) for code development, profiling and debugging. With the environment, developers can create dynamically reconfigurable accelerators optimized for different data center applications that can be swapped in and out on the fly. Developers can use the environment to create applications that swap many kernels in and out of the FPGA during run time without disrupting the interface between the server CPU and the FPGA, for nonstop application acceleration. The SDAccel environment targets host systems based on x86 server processors and provides commercial off-the-shelf (COTS), plug-in PCIe cards that add FPGA functionality.

With the SDAccel environment, developers with no prior FPGA experience can leverage SDAccel's familiar workflow to optimize their applications and take advantage of FPGA platforms. The IDE provides

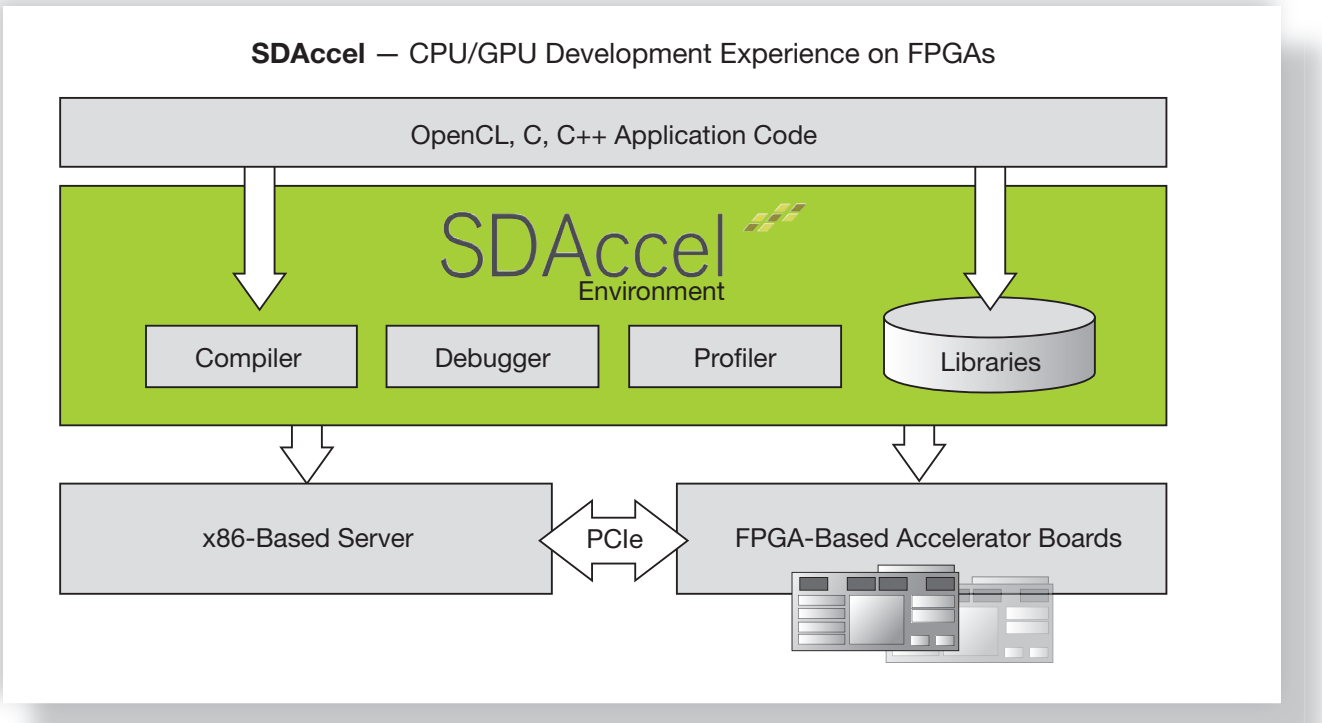


Figure 2 — The SDAccel development environment for OpenCL, C and C++ enables up to 25x better performance/watt for data-center-application acceleration leveraging FPGAs.

coding templates and software libraries, and it enables compiling, debugging and profiling against the full range of development targets, including emulation on the x86, performance validation using fast simulation, and native execution on FPGA processors. The environment executes the application on data-center-ready FPGA platforms complete with automatic instrumentation insertion for all supported development targets. Xilinx designed the SDAccel environment to enable CPU and GPU developers to migrate their applications to FPGAs easily while maintaining and reusing their OpenCL™, C and C++ code in a familiar workflow.

SDAccel libraries contribute substantially to the SDAccel environment's CPU/GPU-like development experience. They include low-level math libraries and higher-productivity ones such as BLAS, OpenCV and DSP libraries. The libraries are written in C++ (as opposed to RTL) so developers can use them exactly as written during all development and debugging phases. Early in a project, all development will be done on the CPU host. Because the SDAccel libraries are written in C++, they can simply be compiled along with the application code for a CPU target—creating a virtual prototype—which permits all testing, debugging and initial profiling to occur initially on the host. During this phase, no FPGA is needed.

SDSOC FOR EMBEDDED DEVELOPMENT OF ZYNQ SOC- AND MPSOC-BASED SYSTEMS

Xilinx designed the SDSoC development environment for embedded systems developers programming the Xilinx Zynq SoCs and soon-to-arrive Zynq UltraScale+ MPSoCs. The SDSoC environment provides a greatly simplified embedded C/C++ application

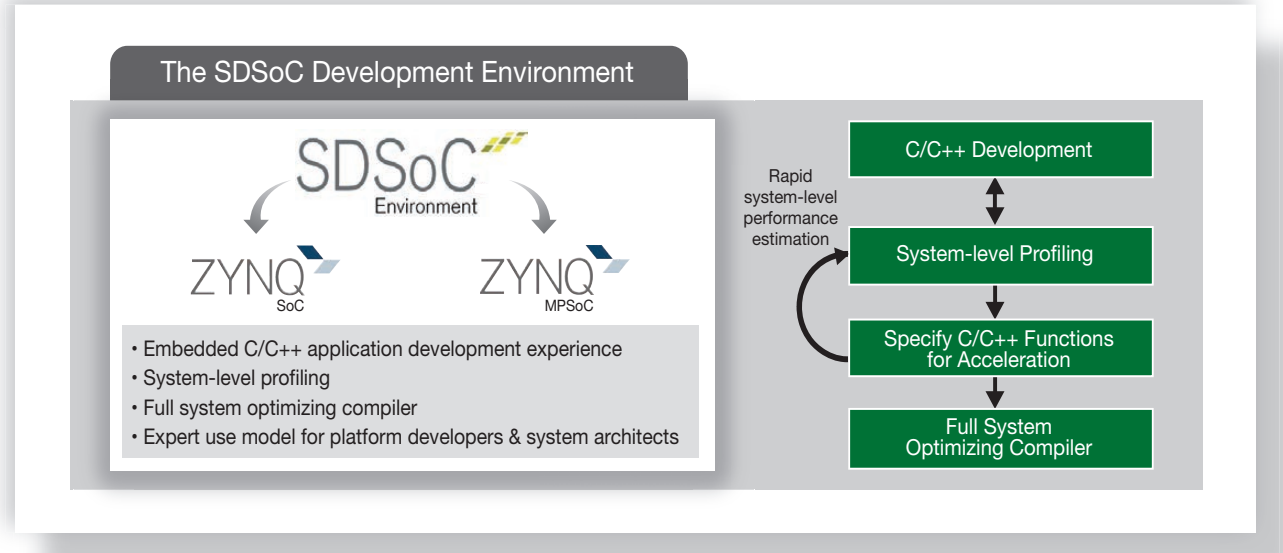


Figure 3— The SDSoC development environment provides a familiar embedded C/C++ application development experience, including an easy-to-use Eclipse IDE and a comprehensive design environment for heterogeneous Zynq All Programmable SoC and MPSoC deployment.

programming experience, including an easy-to-use Eclipse IDE running on bare metal or operating systems such as Linux and FreeRTOS as its input. It is a comprehensive development platform for heterogeneous Zynq SoC and Zynq MPSoC platform deployment (Figure 3). Complete with the industry's first C/C++ full-system optimizing compiler, the SDSoC environment delivers system-level profiling, automated software acceleration in programmable logic, automated system connectivity generation and libraries to speed programming. It also provides a flow for customer and third-party platform developers to enable platforms to be used in the SDSoC development environment.

SDSoC provides board support packages (BSPs) for Zynq All Programmable SoC-based development boards including the ZC702 and ZC706, as well as third-party and market-specific platforms including the ZedBoard, MicroZed, ZYBO, and video and imaging development kits. The BSPs include metadata abstracting the platform from software developers and system architects to ease the creation, integration and verification of smarter heterogeneous systems.

SDNET FOR DESIGN AND PROGRAMMING OF FPGA-ACCELERATED LINE CARDS

SDNet is a software-defined specification environment using an intuitive, C-like high-level language to design the requirements and create a specification for a network line card (Figure 4). The environment enables network architects and developers to create "Softly" Defined Networks, expanding programmability and intelligence from the control to the data plane.

In contrast to traditional software-defined network architectures, which employ fixed data plane hardware with a narrow southbound API connection to the control plane, Softly Defined Networks are based on a programmable data plane with content intelligence and a rich southbound API control plane connection. This enables multiple disruptive capabilities, including support of wire-speed services that are independent of protocol complexity, provisioning

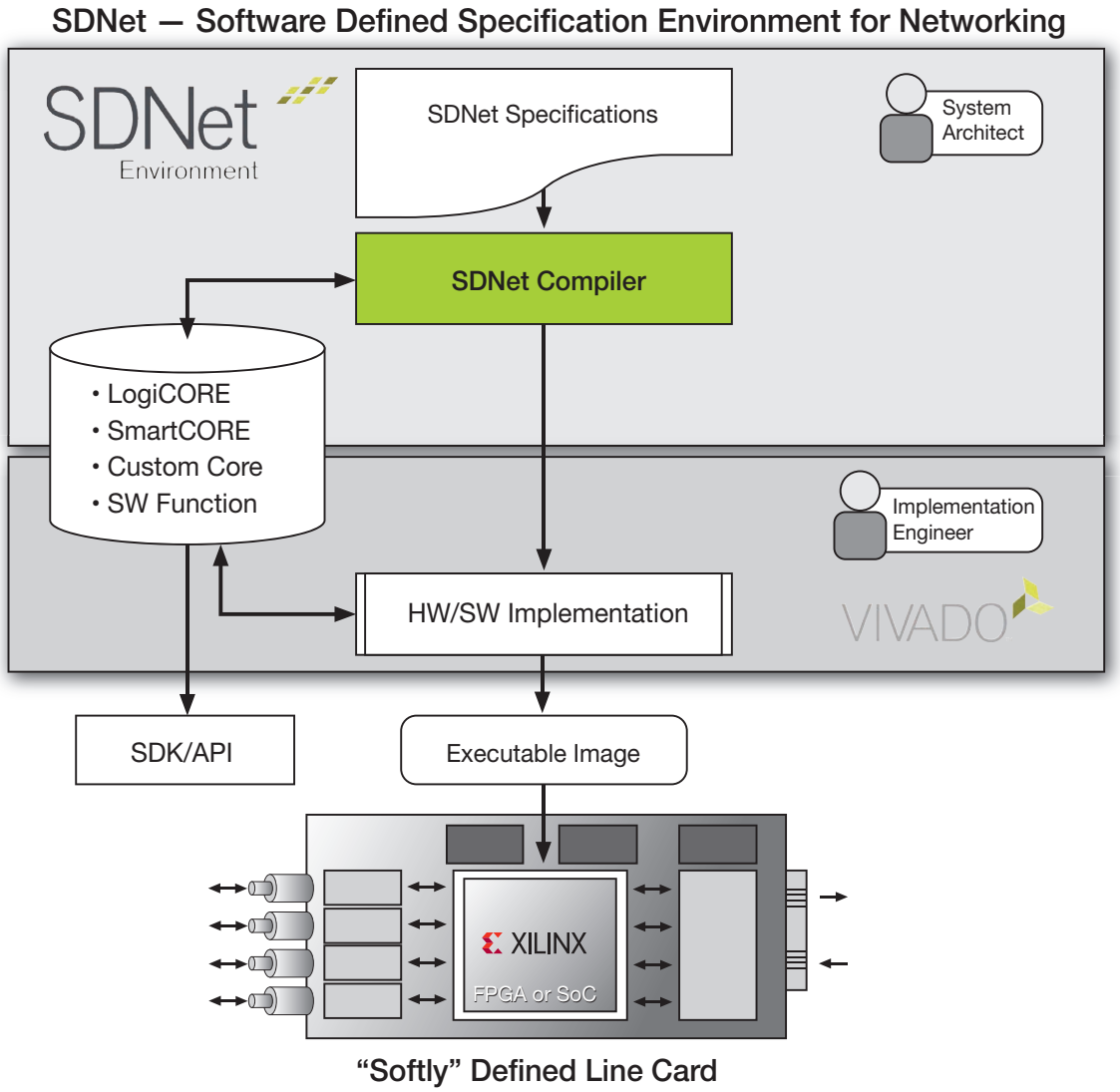


Figure 4 — The SDNet environment enables network architects to create a specification in a C-like language. After a hardware team completes the design, developers can use SDNet to update or add protocols to the card in the field.

of per-flow and flexible services, and support for revolutionary in-service "hitless" upgrades while operating at 100 percent line rates.

These unique capabilities enable carriers and multiservice system operators (MSOs) to provision differentiated services dynamically without any interruption to the existing service or the need for hardware requalification or truck rolls. The environment's dynamic service provisioning enables service providers to increase revenue and speed time to market while lowering capex and opex. Network equipment providers realize similar benefits from the Softly Defined Network platform, which allows for extensive differentiation through the deployment of content-aware data plane hardware that is programmed with the SDNet environment.

This combination of MathWorks and Xilinx technologies has helped customer companies produce thousands of innovative products.

EMBEDDED DEVELOPMENT ENVIRONMENTS

To further help embedded software engineers with programming, Xilinx offers a comprehensive set of embedded tools and run-time environments designed to enable embedded software developers to move efficiently from concept to production. Xilinx offers developers an Eclipse-based IDE called the Xilinx Software Development Kit (SDK), which includes editors, compilers, debuggers, drivers and libraries targeting Zynq SoCs or FPGAs with Xilinx's 32-bit MicroBlaze™ soft core embedded in them. The environment provides out-of-the-box support for advanced features such as security and virtualization software drivers built on Xilinx's unique Zynq SoCs and MPSoCs. This allows developers to innovate truly differentiated connected systems that are both smarter and secure.

Xilinx offers a comprehensive suite of open-source resources to develop, boot, run, debug and maintain Linux-based applications running on a Xilinx SoC or emulation platform. Xilinx provides example applications, kernel construction, Yocto recipes, multiprocessing and real-time solutions, drivers and forums, as well as many community links. Linux open-source developers will find a very comfortable environment in which to learn, develop and interact with others of like interests and needs.

A POWERFUL AND GROWING ALLIANCE OF PROGRAMMING ENVIRONMENTS

In addition to offering developers the new SDx development environments and SDK, Xilinx has built strong alliances over the past decade with companies that already have well-established development environments serving developers in specific market segments.

National Instruments (Austin, Texas) offers hardware development platforms fanatically embraced by control and test system innovators. Xilinx's FPGAs and Zynq SoCs power the NI RIO platforms. National Instruments' LabVIEW development environment is a user-friendly graphics-based program that runs Xilinx's Vivado® Design Suite under the hood so that National Instruments' customers need not know any of the details of FPGA design; indeed, some perhaps don't even know a Xilinx device is at the heart of the RIO platforms. They can simply program their systems in the LabVIEW environment and let NI's hardware speed the performance of designs they are developing.

MathWorks® (Natick, Mass.), for its part, added FPGA support more than a decade ago to its MATLAB®, Simulink®, HDL Coder™ and Embedded Coder® with Xilinx's ISE® and Vivado tools running under the hood and completely automated. As a result, the users—who are mainly mathematician algorithm developers—could develop algorithms and speed algorithm performance exponentially by running the algorithms succinctly on an FPGA fabric.

Xilinx added an FPGA-architecture-level tool called System Generator to its ISE development environment more than a decade ago and, more recently, added the tool to the Vivado Design Suite to enable teams with FPGA knowledge to tweak designs for further algorithm performance gains. This combination of MathWorks and Xilinx technologies has helped customer companies produce thousands of innovative products.

A number of members in Xilinx's Alliance ecosystem offer development tools in support of the SDx and Alliance environments; they include ARM, Lauterbach, Yokogawa Digital Computer Corp. and Kyoto Microcomputer Corp. As for OS and middleware support, Xilinx and its ecosystem of Alliance members provide customers with multiple software options, including Linux, RTOS, bare-metal, and even hypervisor and TrustZone-enabled solutions for safety and security.

For more information on the SDx environments and Xilinx's extensive and growing developer solutions, visit Xilinx's new Software Developer Zone. ■

SDSoC, Step by Step: Build a Sample Design

by Adam Taylor
Chief Engineer
e2v
aptaylor@theiet.org

A ZedBoard example proves quick to build and optimize using the seamless environment.

Until the release of the Xilinx® SDSoC™ development environment, the standard SoC design methodology involved a mix of disparate engineering skills. Typically, once the system architect had generated a system architecture and subsystem segmentation from the requirement, the solution would be split between functions implemented in hardware (the logic side) and functions implemented in software (the processor side). FPGA and software engineers would separately develop their respective functions and then combine and test them in accordance with the integration test plan. The approach worked for years, but the advent of more-capable SoCs, such as the Xilinx Zynq®-7000 All Programmable SoC and the upcoming Xilinx Zynq UltraScale+™ MPSoC, mandated a new design methodology.

The SDSoC methodology enables a wider user base of engineers to develop extremely high-performing systems. Engineers new to developing in the SDSoC development environment will discover that it's easy to get a system up and running quickly and just as easy to optimize it.

A simple, representative example will illustrate how to accomplish those tasks and reap the resultant benefits. We will target a ZedBoard running Linux and using one of the built-in examples: the Matrix Multiplier and Addition Template.

A BRIEF HISTORY OF DESIGN METHODOLOGIES

The programmable logic device segment has been fast-moving since the devices' introduction in the 1980s. At first engineers programmed the devices via schematic entry (although the earlier PLDs, such as the 22v10, were programmed via logic equations). This required that electronics engineers perform most PLD development, as logic design and optimization are typically the EE degree's domain. As device size and capability increased, however, schematic entry naturally began to hit boundaries, as both design time and verification time rose in tandem with design complexity. Engineers needed the capability to work at a higher level of abstraction.

Enter VHDL and Verilog. Both started as languages to describe and simulate logic designs, particularly ASICs. VHDL even had its own military standard. It is a logical step that if we are describing logic behavior within a hardware description language (HDL), it would be great to synthesize the logic circuits required. The development of synthesis tools let engineers describe logic behavior typically at a register transfer level. HDLs also provided a significant boost in verification approach, allowing the development of behavioral test benches that enabled structured verification. For the first time, HDLs also enabled modularity and vendor independence.

Again, the inherent concurrency of HDLs, the register transfer level design approach and the implementation flow, which required knowledge of optimization and timing closures, ensured that the PLD development task would largely fall to EEs.

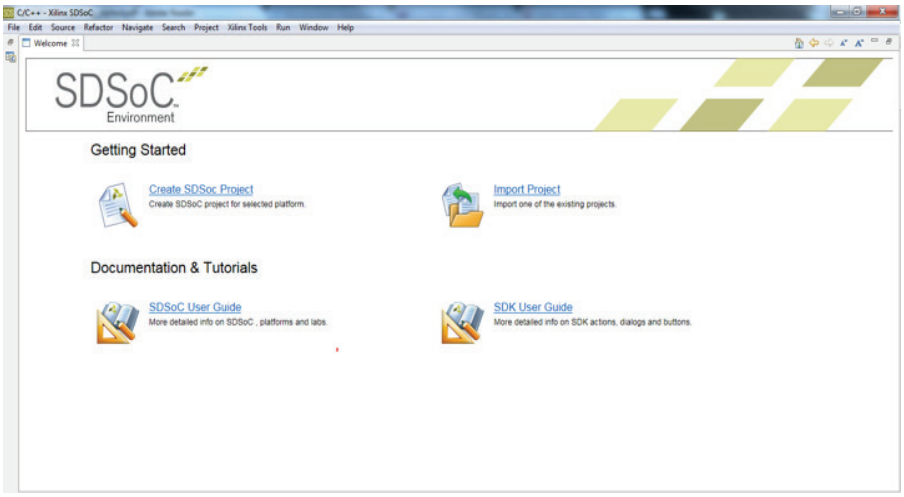


Figure 1 — SDSoC Welcome page

HDLs have long been the de facto standard for PLD development but have evolved over the years to take industry needs into account. VHDL alone underwent revisions in 1987 (the first year of IEEE adoption), 1993, 2000, 2002, 2007 and 2008. As happened with schematic design entry, however, HDLs are hitting up against the buffers of increases in development time, verification time and device capability.

As the PLD's role has expanded from glue logic to acceleration peripheral and ultimately to the heart of the system, the industry has needed a new design methodology to capitalize on that evolution. In recent years, high-level synthesis (HLS) has become increasingly popular; here, the design is entered in C/C++ (using Xilinx's Vivado® HLS) or tools such as MathWorks®' MATLAB® or National Instruments' LabVIEW. Such approaches begin to move the design and implementation out from the EE domain into the software realm, markedly widening the user base of potential PLD designers and cementing the PLD's place at the heart of the system as new design methodologies unlock the devices' capabilities.

It is therefore only natural that SoC-based designs would use HLS to generate tightly integrated development environments in which engineers could seamlessly accelerate functions in the logic side of the design. Enter the SDSoC environment.

FAMILIAR ENVIRONMENT

The SDSoC development environment is based on Eclipse, which should be familiar to most software developers (Figure 1). The environment seamlessly enables acceleration of functions within the PL side of the device. It achieves this by using the new SDSoC compiler, which can handle C or C++ programs.

The development cycle at the highest abstraction level used in the SDSoC environment is as follows:

1. We develop our application in C or C++.
2. We profile the application to determine the performance bottlenecks.
3. Using the profiling information, we identify functions to accelerate within the PL side of the device.
4. We can then build the system and generate the SD card image.
5. Once the hardware is on the board, we can analyze the performance further and optimize the acceleration functions as required.

We can develop applications in the SDSoC environment that function variously on bare metal, FreeRTOS or Linux operating systems. The environment comes with built-in support for most of the Zynq SoC development boards, including the ZedBoard, the MicroZed and the Digilent ZYBO Zynq SoC development board. Not only can we develop our applications faster as a result, but we can use this capability to define our own underlying hardware platform for use when our custom hardware platform is ready for integration.

When we compile a program within the SDSoC environment, the output of the build process provides the suite of files required to configure the Zynq SoC from an SD card. This suite includes first- and second-stage boot loaders, along with the application and images as required for the operating system.

SDSOC EXAMPLE

Let's look at how the SDSoC environment works and see how quickly we can get an example up and running. We will target a ZedBoard running Linux and using the built-in Matrix Multiplier and Addition Template.

The first task, as always, is to create a project. We can do so either from the Welcome screen (Figure 1) or by selecting File -> New -> SDSoC project from the menu. Selecting either option will open a dialog box that will let us name the project and select the board and the operating system (Figure 2).

This will create a project under the Project Explorer on the left-hand side of the SDSoC GUI. Under this project, we will see the following folders, each with its own, graphically unique symbol:

- SDSoC Hardware Functions: Here we will see the functions we have moved into the hardware. Initially, as we have yet to move functions, this folder will be empty.
- Includes: Expanding this folder will show all of the C/C++ header files used in the build.
- src: This will contain the source code for the demonstration.

To ensure that we have everything correctly configured not only with our SDSoC installation and environment, but also with our development board, we will build the demo so that it will run on only the on-chip processing system (PS) side of the device.

Of course, the next step is to build the project. With the project selected on the menu, we choose Project->Build Project. It should not take too long to build, and when we are done we will see folders as shown in Figure 3 appear under our project within the Project Explorer. In addition to the folders described above, we will have:

- Binaries: Here we will find the Executable and Linkable Format (ELF) files created from the software compilation process.
- Archives: The object files that are linked to create the binaries reside here.
- SDRelease: This contains our boot files and reports.

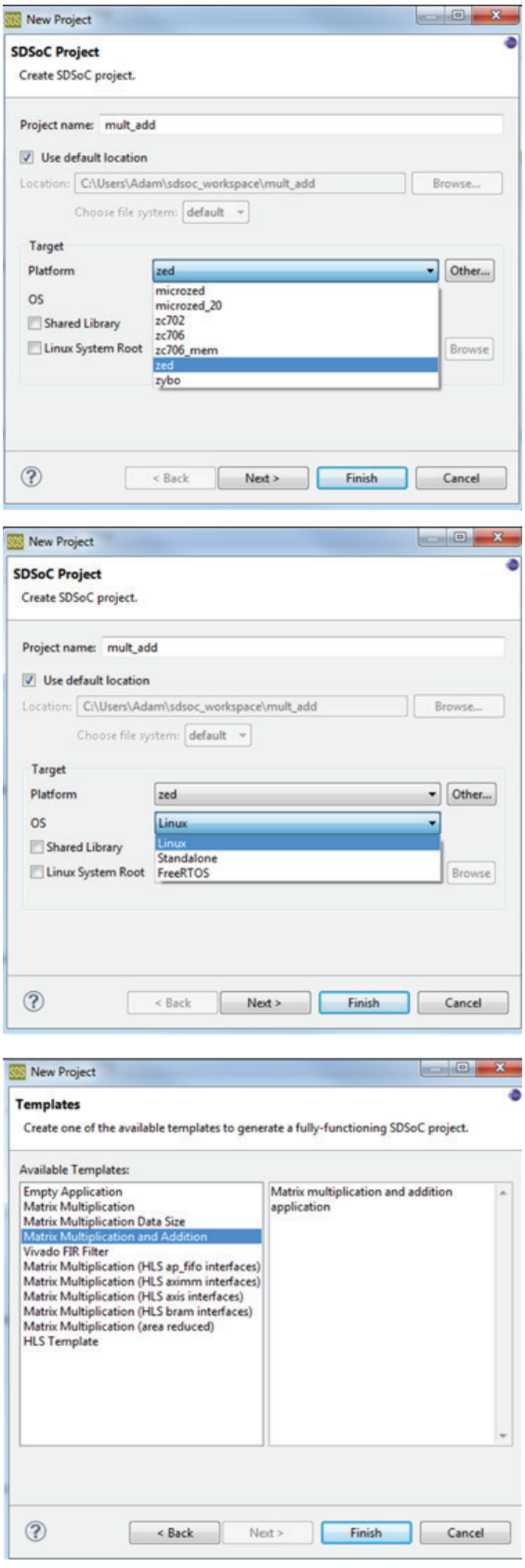


Figure 2 — Creating the project

With the first demo built such that it will run only on the Zynq SoC's PS, let's explore how we know it is working as desired. Recall that SDSoC acceleration works by profiling the application; the engineer then uses the profiled information to determine which functions to move.

We achieve profiling at the basic level by using a provided library called sds_lib.h. This provides a basic timestamp API, based on the 64-bit global counter, that lets us measure how long each function takes. With the API, we simply record the function start and stop times, and the difference constitutes the process execution time.

The source code contains two versions of the algorithm for matrix multiply and add. The so-called golden version is not intended for offloading to the on-chip programmable logic (PL); the other version is. By building and running these just within the PS, we can ensure that we are comparing eggs with eggs and that both processes take roughly the same time to execute.

With the build complete, we can copy all of the files in the SDRelease -> sd_card folder under the Project Explorer onto our SD card and insert the card into the ZedBoard (with the mode pins correctly set for SD card configuration). With a terminal program connected, once the boot sequence has been completed we need to run the program. We type /mnt/mult_add.elf (where mult_add is the name of the project we have created). When I ran this on my ZedBoard, I got the result shown in Figure 4, which demonstrates that the two functions take roughly the same time to execute.

Having confirmed the similar execution times, we will move the multiply function into the PL side of the SoC. This is simple to achieve.

Looking at the file structure within the src directory of the example, we will see:

- main.cpp, which contains the main function, golden calculation, timestamping, and calls to the mult and add functions used in the hardware side of the device;
- mmult.cpp, which contains the multiplication function to be offloaded into the hardware; and
- madd.cpp, which contains the addition function to be offloaded into the hardware.

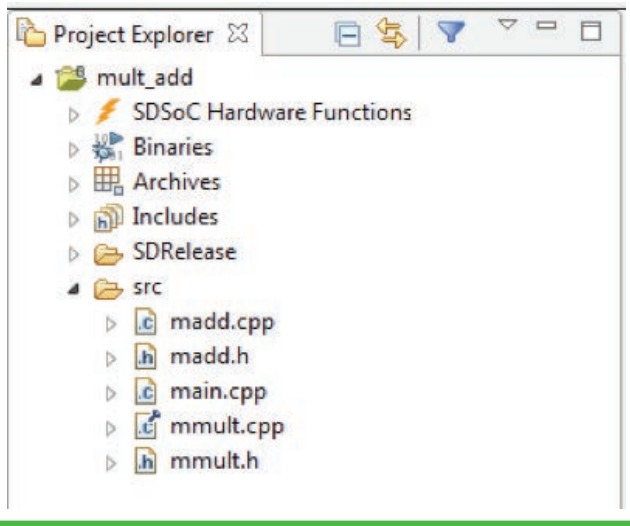


Figure 3 — Project Explorer view when built

```
sh-4.3# /mnt/mult_add.elf
Testing with A_NROWS = A_NCOLS = B_NCOLS = B_NROWS = 32
Testing mmult ...
Average number of processor cycles for golden version: 181607
Average number of processor cycles for hardware version: 183289
TEST PASSED
sh-4.3#
```

Figure 4 — Execution time of both functions in the PS

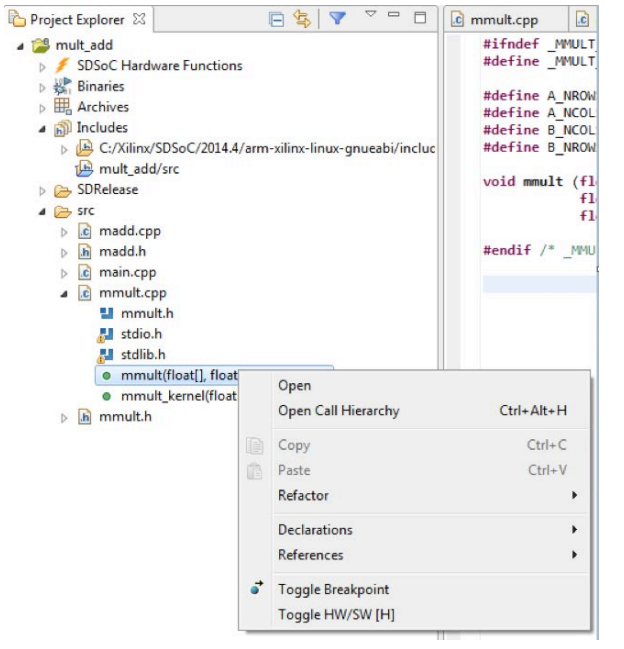


Figure 5 — Moving the multiplier kernel to the PL side using the Project Explorer

Once we have taken the steps described here, the next time we build the project the SDSoC linker will automatically call Xilinx Vivado HLS and Vivado to implement the functions within the PL side of the SoC.

The next step is to offload just one of these functions to the PL side of the SoC. We can achieve this by one of two methods:

1. Within the Project Explorer, we can expand the file such that we can see the functions within that file, select the function of interest, right click and select Toggle HW/SW [H] (Figure 5).
2. We can open the file and perform the same option under the outline tab on the right, which shows the functions as well (Figure 6).

Toggling the mmult() function to be accelerated within the hardware will result in an [H] being added to the back of the function (Figure 7).

We will also see the function we have selected under SDSoC Hardware Functions (beneath our project within the Project Explorer tab; Figure 8). This provides an easy way to see all of the functions that we have accelerated within our design.

Once we have taken the steps described here, the next time we build the project the SDSoC linker will automatically call Vivado HLS and the rest of the Vivado Design Suite to implement the functions within the PL side of the SoC. As it does so, it will create the relevant software drivers to support function acceleration. From our perspective, offloading the function to the PL side of the device becomes seamless, except for the increase in performance.

I moved the mmult() function into the hardware after compilation and SD card image generation, running it on my ZedBoard. As Figure 9 shows, the execution time (in processor cycles) was only 52,444 / 183,289 = 0.28, or 28 percent of the previous execution time of 183,289 processor cycles when executed within the PS side of the device (Figure 4). When we consider the performance of the same function when executed within the PS side of the device, we see that we achieve this considerable increase in execution time by a simple click of the mouse.

The straightforward example presented here demonstrates the power and seamlessness of the SDSoC environment and the tightly integrated HLS functions. ■

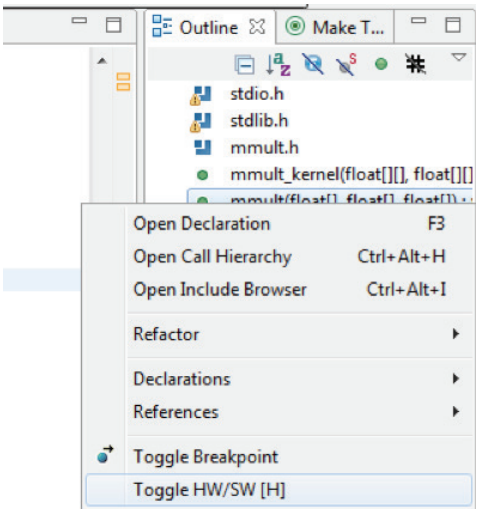


Figure 6 — Moving the multiplier kernel to the PL side using the outline window

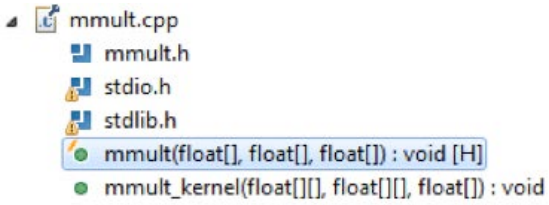


Figure 7 — The mmult() function in hardware

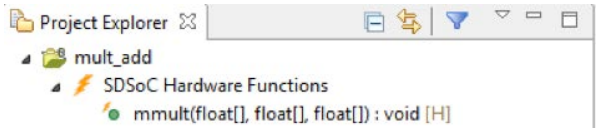


Figure 8 — Identifying our accelerated functions

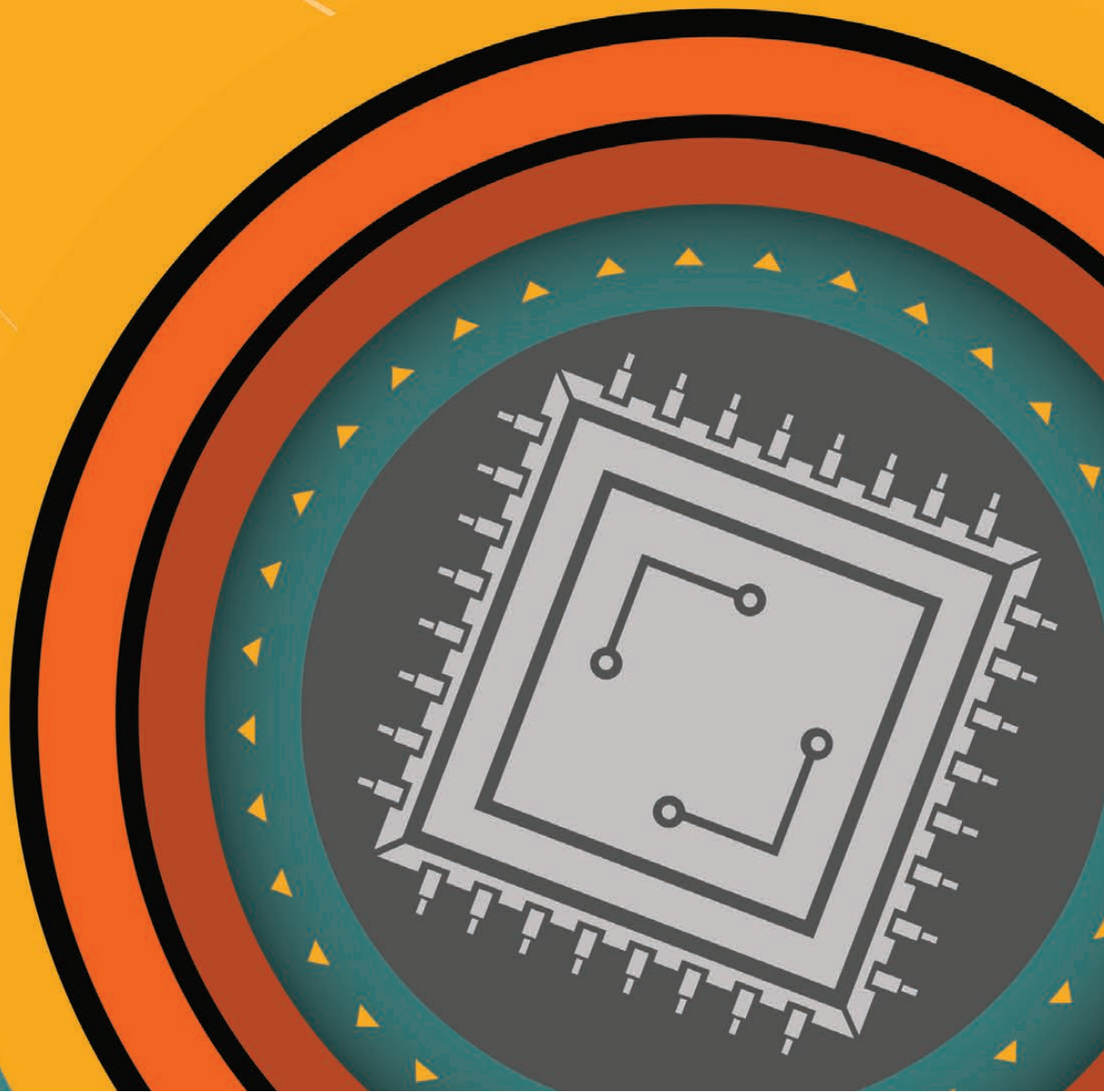
```
sh-4.3# /mnt/mult_add.elf
Testing with A_NROWS = A_NCOLS = B_NCOLS = B_NROWS = 32
Testing mmult ...
Average number of processor cycles for golden version: 181699
Average number of processor cycles for hardware version: 52444
TEST PASSED
```

Figure 9 — The accelerated results

Using the SDSoC IDE for System-level HW-SW Optimization on the Zynq SoC

by Daniele Bagni
DSP Specialist FAE
Xilinx, Inc.
danieleb@xilinx.com

Nick Ni
Product Manager, SDSoC
Development Environment
Xilinx, Inc.
nickn@xilinx.com



A Choleksy matrix decomposition example yields an acceleration estimate in minutes.

The Xilinx® Zynq®-7000 All Programmable SoC family represents a new dimension in embedded design, delivering unprecedented performance and flexibility to the embedded systems engineering community. These products integrate a feature-rich, dual-core ARM® Cortex™-A9 MPCore™-based processing system and Xilinx programmable logic in a single device. More than 3,000 interconnects link the on-chip processing system (PS) to on-chip programmable logic (PL), enabling performance between the two on-chip systems that simply can't be matched with any two-chip processor-FPGA combination. When Xilinx released the device in 2011, the Zynq SoC gained an instant following among a subset of embedded systems engineers and architects well-versed in hardware design languages and methodologies as well as in embedded software development. The first-of-its-kind Zynq SoC today is deployed in embedded applications ranging from wireless infrastructure to smart factories and smart video/vision, and it is quickly becoming the de facto standard platform for advanced driver assistance systems.

To make this remarkable device available to embedded engineers who have a strong software background but no HDL experience, Xilinx earlier this year introduced the Eclipse-based SDSoC™ integrated development environment, which enables software engineers to program the programmable logic as well as the ARM processing system of the Zynq SoC.

Let's take a closer look at the features of the Zynq SoC [1] and at how software engineers can leverage the SDSoC environment to create system designs not possible with any other processor-plus-FPGA system. For our investigation, we will use the Xilinx ZC702 evaluation board [2], containing a Zynq Z-7020-1 device, as the hardware platform.

As shown in Figure 1, the Zynq SoC comprises two major functional blocks: the PS (composed of the application processor unit, memory interfaces, peripherals and interconnect) and the PL (the traditional FPGA fabric).

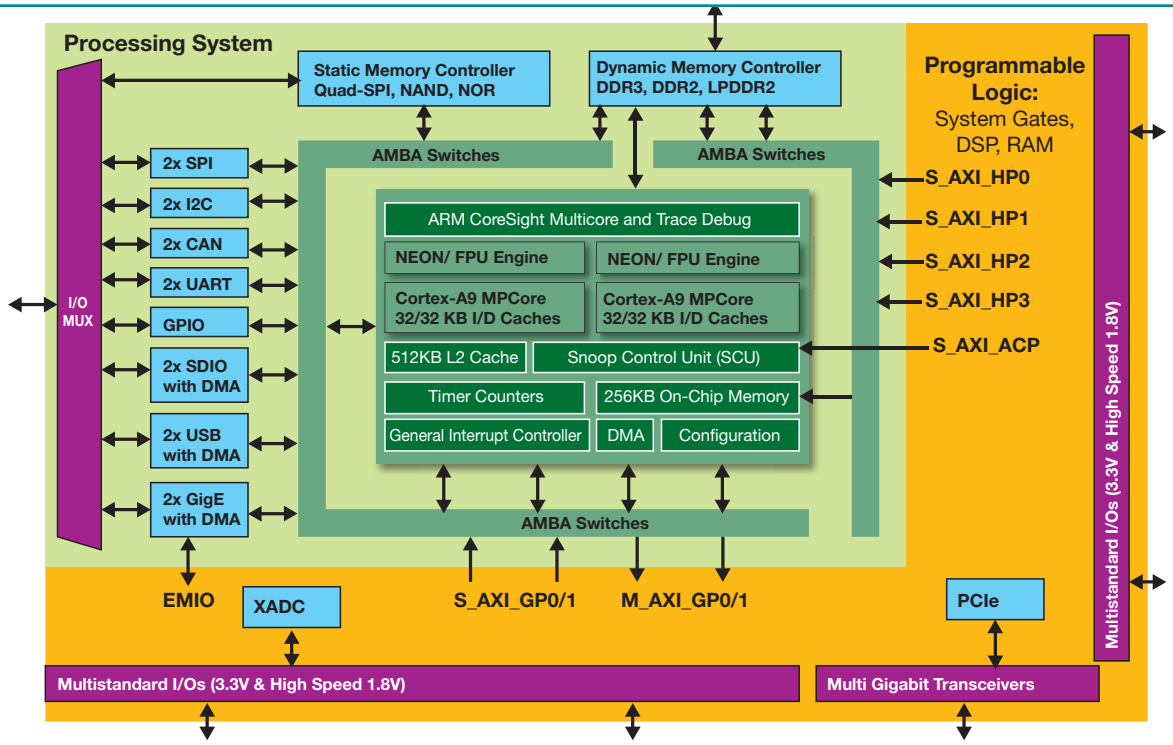


Figure 1 — Zynq high-level architecture overview

The PS and PL are tightly coupled via interconnects compliant with the ARM® AMBA® AXI4 interface. Four high-performance (HP) AXI4 interface ports connect the PL to asynchronous FIFO interface (AFI) blocks in the PS, thereby providing a high-throughput data path between the PL and the PS memory system (DDR and on-chip memory). The AXI4 Accelerator Coherency Port (ACP) allows low-latency cache-coherent access to L1 and L2 cache directly from the PL masters. The General Purpose (GP) port comprises low-performance, general-purpose ports accessible from both the PS and PL.

In the traditional, hardware-design-centric flow, using Xilinx's Vivado® Design Suite, designing an embedded system on the Zynq SoC requires roughly four steps:

1. A system architect decides a hardware-software partitioning scheme. Computationally intensive algorithms are the ideal candidates for hardware. Profiling results are used as the basis for identifying performance bottlenecks and running trade-off studies between data movement costs and acceleration benefits.

2. Hardware engineers take functions partitioned to hardware and convert/design them into intellectual-property (IP) cores—for example, in VHDL or Verilog using Vivado, in C/C++ using Vivado High Level Synthesis (HLS) [3] or in model-based design using Vivado System Generator for DSP [4].

3. Engineers then use Vivado IP Integrator [5] to create a block-based design of the whole embedded system. The full system needs to be developed with different data movers (AXI-DMA, AXI Memory Master, AXI-FIFO, etc.) and AXI interfaces (GP, HP and ACP) connecting the PL IP with the PS. Once all design rules checks are passed within IP Integrator, the project can be exported to the Xilinx Software Development Kit (SDK) [6].

4. Software engineers develop drivers and applications targeting the ARM processors in the PS using the Xilinx SDK.

In recent years, Xilinx made substantial ease-of-use improvements to the Vivado Design Suite that enabled engineers to shorten the duration of the IP development and IP block connection steps (step 2 and part of step 3 above). For IP development, the adoption of such new design technologies as C/C++ high-level synthesis in the Vivado HLS tool and model-based design with Vivado System Generator for DSP cut development

The SDSoC environment automatically orchestrates all necessary Xilinx tools to generate a full hardware-software system targeting the Zynq SoC—and does so with minimum user intervention.

and verification time dramatically while letting design teams use high-level abstractions to explore a greater range of architectures. Designs that took weeks to accomplish with VHDL or Verilog could be completed in days using the new tools.

Xilinx enhanced the flow further with Vivado IP Integrator. This feature of the Vivado Design Suite enables the design of a complicated hardware system, embedded or not, simply by connecting IP blocks in a graphical user interface (GUI), thereby allowing rapid hardware system integration.

The new Vivado Design Suite features made life a bit easier for design and development teams working with the Zynq SoC. But with a hardware-centric optimization workflow, not too much could be done to shorten the development time required to explore different data movers and PS-PL interfaces (part of step 3 above) and to write and debug drivers and applications (step 4). If the whole system did not meet the design requirement in terms of throughput, latency or area, the team would have to revisit the hardware architecture by modifying the system connectivity during step 3; those modifications inevitably would lead to changes in the software application in step 4. In some cases, a lack of acceleration or a hardware utilization overflow would force the team to revisit the original hardware-software partitioning. Multiple hardware and software teams would have to create another iteration of the system to explore other architectures that might meet the end requirement.

These examples show the time-to-market impact of system optimization done manually. System optimization nonetheless is critical for a tightly integrated system such as the Zynq SoC because bottlenecks often occur in the system connectivity between the PS and the PL.

The SDSoC environment greatly simplifies the Zynq SoC development process, slashing total development time by largely automating steps 2, 3 and 4. The development environment generates necessary hardware and software components to synchronize hardware and software and to preserve original program semantics, while enabling task-level parallelism and pipelined communication and computation to achieve high performance. The SDSoC environment automatically orchestrates all necessary Xilinx tools (Vivado, IP Integrator, HLS and SDK) to generate a full hardware-software system targeting the Zynq SoC—and does so with minimum user intervention.

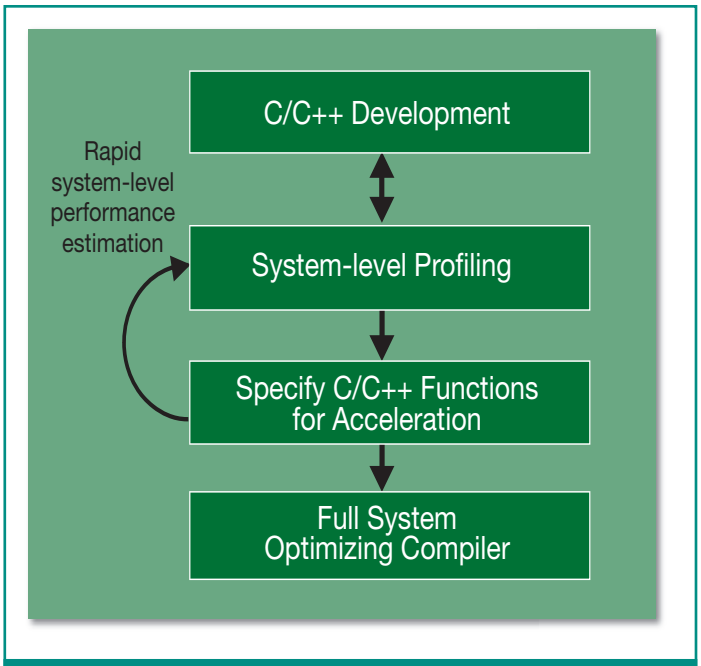


Figure 2 — The main steps in the SDSoC design flow

Assuming we have an application completely described in C/C++ targeting the PS and we have already decided which functions to partition to the PL for acceleration, the SDSoC development flow roughly proceeds as follows (Figure 2):

1. The SDSoC environment builds the application project using a rapid estimation flow (by calling Vivado HLS under the hood). This will provide the ballpark performance and resource estimation in minutes.

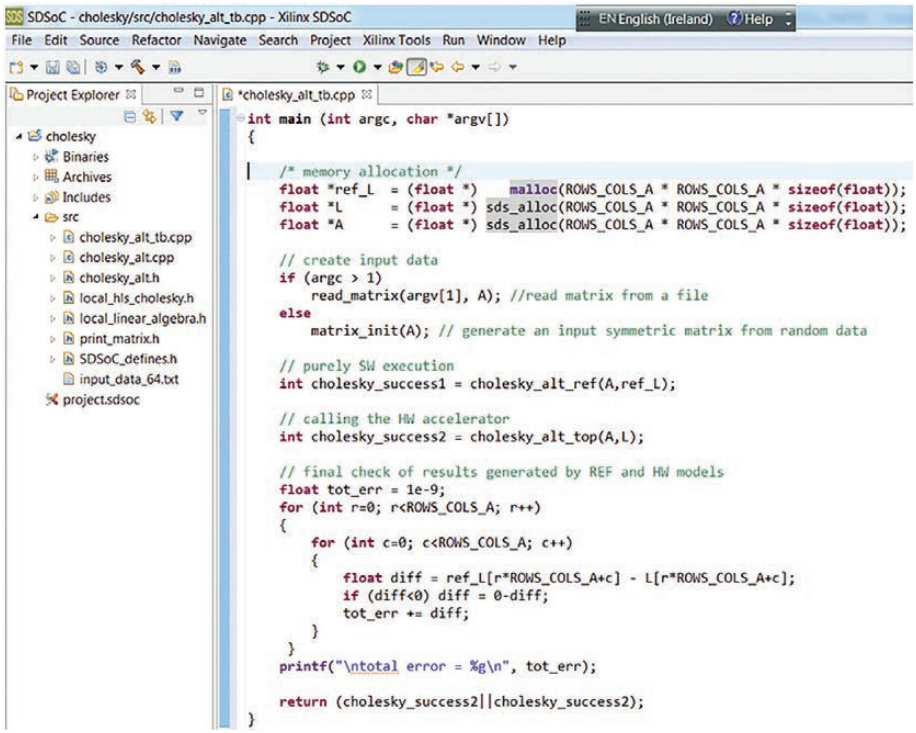


Figure 3 — Structure of the C/C++ test bench for the SDSoC environment

2. If we deem it necessary, we optimize the C/C++ application and the hardware functions with proper directives, and rerun the estimation until the desired performance and area are achieved.
3. The SDSoC environment then builds the full system. This process will generate the full Vivado Design Suite project and the bitstream, along with a bootable run-time software image targeting Linux, FreeRTOS or bare metal.

PERFORMANCE ESTIMATION OF HARDWARE VS. SOFTWARE WITH THE SDSOC ENVIRONMENT

Linear algebra is a fundamental and powerful tool in almost every discipline of engineering, allowing whole systems of equations with multidimensional variables to be solved computationally. For example, engineers can describe linear control theory systems as matrices of "states" and state changes. Digital signal processing of images is another classic example of linear algebra's application. In particular, matrix inversion through the Cholesky decomposition is considered one of the most efficient methods for solving a system of equations or inverting a matrix. Let's look closely at a Cholesky matrix decomposition of 64 x 64 real data in a 32-bit floating-point representation as an application example for hardware-software partitioning on the Zynq SoC.

The Cholesky decomposition transforms a positive definite matrix into the product of a lower and upper triangular matrix with a strictly positive diagonal. The matrix B is decomposed in the triangular matrix L, so that B = L' * L, with L' the transposed version of L, as illustrated in the following MATLAB® code for the case of a 4 x 4 matrix size:

```
A = ceil(64*randn(4,4)) % generate random
                          data
B = A * A'              % make the matrix to
                          be symmetric
L = chol(B)             % compute cholesky
                          decomposition
B2 = (L' * L)           % reconstruct the
                          original matrix B
A =
      -13          53          41          20
      -19          98          12           9
        2          30         -65          33
        4         -13          61          17
B =
     5059        6113        -441        2100
     6113       10190        2419        -465
     -441        2419        6218       -3786
     2100        -465       -3786        4195
L =
  71.1266     85.9453     -6.2002     29.5248
        0     52.9472     55.7513    -56.7077
        0           0     55.4197     -7.9648
        0           0           0      6.6393
B2 =
     5059        6113        -441        2100
     6113       10190        2419        -465
     -441        2419        6218       -3786
     2100        -465       -3786        4195
```

Selecting the candidate accelerator is easily accomplished with a mouse click on a specific function via the SDSoC environment's GUI.

Let's see how we can obtain an estimation of the performance and resource utilization that we can expect from our application, without going through the entire build cycle.

Figure 3 shows the test bench structure suitable for the SDSoC environment. The main program allocates dynamic memory for all the empty matrices and fills them with data (either read from a file or generated randomly). It then calls the reference software function and the hardware candidate function. Finally, the main program checks the numerical results computed by both functions to test the effective correctness.

Note the use of a special memory allocator called sds_alloc for each input/output array to let the SDSoC environment automatically insert a Simple DMA IP between each I/O port of the hardware accelerator; in contrast, malloc instantiates a Scatter-Gather DMA, which can handle arrays spread across multiple noncontiguous pages in the Physical Address Space. The Simple DMA is cheaper than the Scatter-Gather DMA in terms of area and performance overheads, but it requires sds_alloc to obtain physically contiguous memory.

Selecting the candidate accelerator is easily accomplished with a mouse click on a specific function via the SDSoC environment's GUI. As shown in Figure 4, the routine cholesky_alt_top is marked with an "H" to indicate that it will be promoted to a hardware accelerator. We can also select the clock frequency for the accelerator and for the data motion cores (100 MHz as illustrated in the SDSoC project page of Figure 4).

We can now launch the "estimate speedup" process. After a few minutes of compilation, we get all the cores and the data motion network generated in a Vivado project. The SDSoC environment also generates an SD card image that comprises a Linux boot image

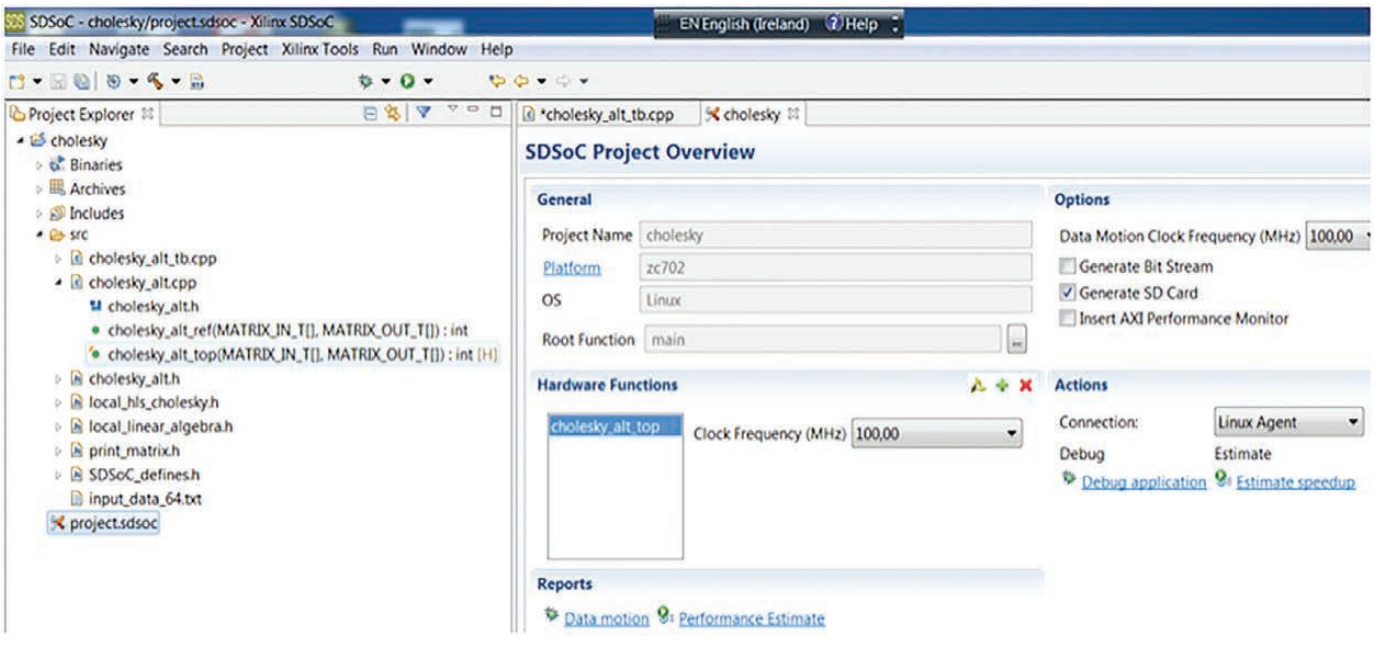


Figure 4 — Setting the hardware accelerator core and its clock frequency from the SDSoC project page

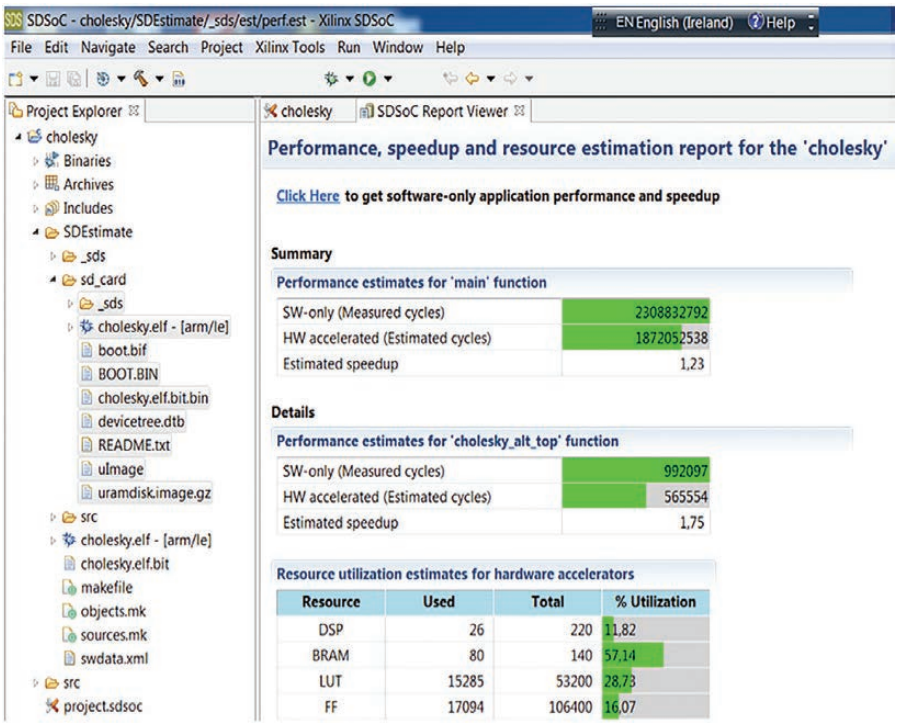


Figure 5 — SDSoC-generated performance, speedup and resources estimation report

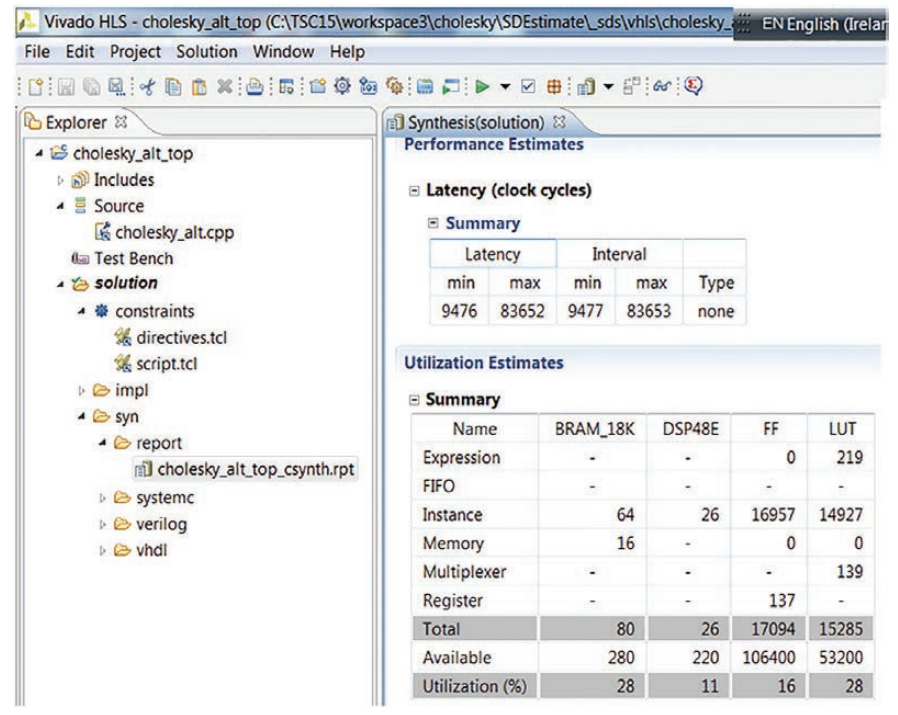


Figure 6: Vivado HLS synthesis estimation report

including the FPGA bitstream and the application binary of the software-only version. We boot from this SD card and run the application on the ZC702 target platform.

Once Linux has booted on the board, we can execute the software-only application, and the SDSoC environment then generates the performance estimation report of Figure 5. We see both the FPGA resources utilization (26 DSP, 80 BRAM, 15,285 LUT, 17,094 FF) and the performance speedup (1.75) of the cholesky_alt_top function if executed in hardware instead of software. We can also see, from the main application point of view, that the overall speedup is lower (1.23) because of other software overhead such as malloc and data transfer. Our complete application is indeed small, focusing mainly on illustrating the SDSoC flow and design methodology; we would need more routines to be accelerated in the PL, but that is beyond the scope of this article.

Using the SDSoC environment, we have generated this information in a few minutes without requiring synthesis and place-and-route FPGA compilation; those processes could take hours, depending on the complexity of the hardware system. Estimations like this one are often enough to analyze the system-level performance of hardware-software partitioning and let users very rapidly iterate a design to create an optimized system.

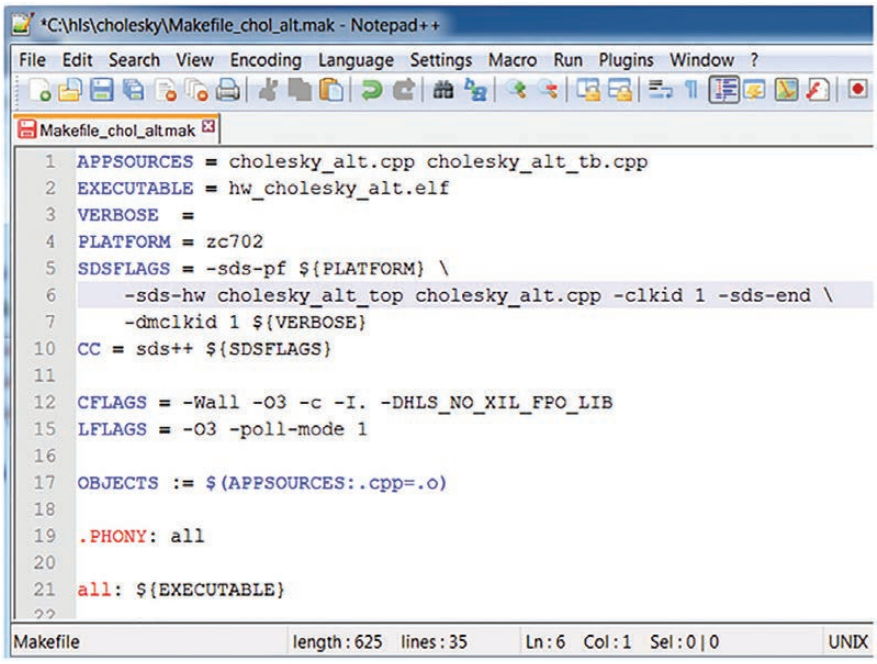


Figure 7 — Makefile for the Release build

UNDERSTANDING THE PERFORMANCE ESTIMATION RESULTS

When the SDSoC environment compiles the application code for the estimate-speedup process, it generates an intermediate directory (_sds in Figure 5) in which it places all intermediate projects (Vivado HLS, Vivado IP Integrator, etc.). In particular, it inserts calls to a free-running ARM performance counter function, sds_clock_counter(), in the original code to measure the execution time of key parts of the program functions. That is why the target board needs to be connected with the SDSoC environment's GUI during the estimate-speedup process. All the numbers reported in Figure 5 are measured with those counters during run-time execution. The only exception is the hardware-accelerated function, which does not exist until after the entire FPGA build (including place-and-route implementation); therefore Vivado HLS computes the hardware-accelerated function's estimated cycles—together with the resource utilization estimates—under the hood, during the effective Vivado HLS Synthesis step.

Assuming the candidate hardware accelerator function runs at F_{HW} MHz clock frequency and needs CK_{HW} clock cycles for the whole computation (this is the concept of latency), and assuming the function takes CK_{ARM} at a clock frequency of F_{ARM} MHz when executed on the ARM CPU, then the hardware accelerator achieves the same performance as the ARM CPU if the computation time is the same, that is, $CK_{HW} / F_{HW} = CK_{ARM} / F_{ARM}$. From this equation, we get $CK_{ARM} = CK_{HW}*F_{ARM} / F_{HW}$. This represents the maximum amount of clock cycles the accelerator can offload from the processor to show any acceleration that results from migrating the function to hardware.

In Figure 6, we report the Vivado HLS synthesis estimation results. Note that the hardware accelerator latency is CK_{HW} = 83,652 cycles at F_{HW} = 100-MHz clock frequency. Since in the ZC702 board we have F_{ARM} = 666 MHz and therefore $CK_{ARM} = CK_{HW}*F_{ARM} / F_{HW}$ = 83,653*666/100 = 557,128, the resultant hardware acceleration is well aligned with the result of 565,554 cycles reported by the SDSoC environment in Figure 5. This is why the SDSoC environment can estimate the number of clock cycles that the accelerator requires without actually building it via place-and-route.

BUILDING THE HARDWARE-SOFTWARE SYSTEM WITH THE SDSOC ENVIRONMENT

Having determined that this hardware acceleration makes sense, we can implement the whole hardware and software system with the SDSoC environment. All we need to do is add the right directives (in the form of pragma commands) to specify, respectively, the FIFO interfaces (due to the sequential scan of the I/O arrays); the amount of data to be transferred at run time for any call to the accelerator; the types of AXI ports connected between the IP core in the PL and the PS; and, finally, the kind of data movers. The following C/C++ code illustrates the applications of those directives. Note that in reality the last

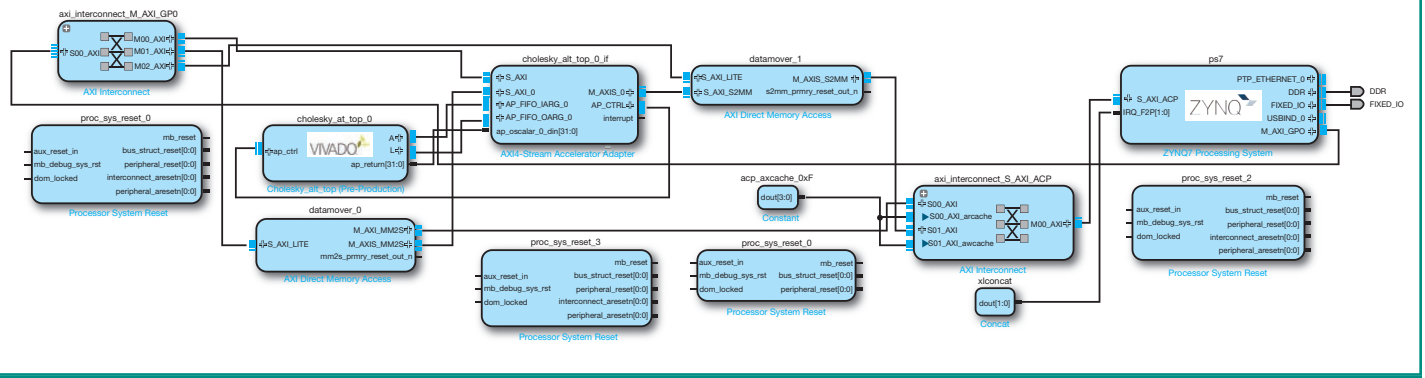


Figure 8 — IP Integrator block-based design done by the SDSoC environment

directive is not needed, because the SDSoC environment will instantiate a Simple DMA due to the use of sds_alloc; we have included it here only for the sake of clarity.

ment calls Vivado IP Integrator in a process transparent to the user (for the sake of clarity, only the AXI4 interfaces are shown). In addition, the SDSoC environ-

```
#pragma SDS data access_pattern(A:SEQUENTIAL, L:SEQUENTIAL) //fifo interfaces
#pragma SDS data copy(A[0:BUF_SIZE], L[0:BUF_SIZE])         // amount of data transf
#pragma SDS data sys_port (A:ACP, L:ACP)                    // type of AXI ports
#pragma SDS data data_mover (A:AXI_DMA_SIMPLE, L:AXI_DMA_SIMPLE) // type of DMAs

int cholesky_alt_top(MATRIX_IN_T  A[ROWS_COLS_A*ROWS_COLS_A],
                     MATRIX_OUT_T L[ROWS_COLS_A*ROWS_COLS_A]);
```

We can build the project in Release configuration directly from the SDSoC environment's GUI, or we can use the Makefile reported in Figure 7 and launched from the SDSoC Tool Command Language (Tcl) interpreter. As is the case with any tool in the Vivado Design Suite, designers can either adopt the GUI or Tcl scripting. To improve the speedup gain, we increase the clock frequency of the hardware accelerator to F_{HW} =142 MHz (set by the -clkid 1 makefile flag).

After less than half an hour of FPGA compilation, we get the bitstream to program the ZC702 board and the Executable Linkable Format (ELF) file to execute on the Linux OS. We then measure the performance on the ZC702 board: 995,592 cycles for software-only and 402,529 cycles for hardware acceleration. Thus, the effective performance gain for the cholesky_alt_top function is 2.47.

Figure 8 illustrates the block diagram of the whole embedded system created when the SDSoC environment reports the Vivado IP Integrator block diagram as an HTML file to make it easy to read (Figure 9). This report clearly shows that the hardware accelerator is connected with the ACP port via a simple AXI4-DMA, whereas the GP port is used to set up the accelerator via an AXI4-Lite interface.

How much time did it take us to generate the SD card for the ZC702 board with the embedded system up and running? We needed one working day to write a C++ test bench suitable to both Vivado HLS and the SDSoC environment, and then we needed one hour of experimentation to get good results from the Linear Algebra HLS Library and one hour to create the embedded system with the SDSoC environment (the FPGA compilation process). Altogether, the process took 10 hours. We estimate that doing all this work manually (step 3 with Vivado IP Integrator and step 4 with Xilinx SDK) would have cost us at least two weeks of full-time, hard work, not counting the experience needed to use those tools efficiently.

After less than half an hour of FPGA compilation, we get the bitstream to program the ZC702 board and the Executable Linkable Format (ELF) file to execute on the Linux OS.

The SDSoC development environment enables the broader community of embedded system and software developers to target the Zynq SoC with a familiar embedded C/C++ development experience. Complete with the industry's first C/C++ full-system optimizing compiler, the SDSoC environment delivers system-level profiling, automated software acceleration in programmable logic, automated system connectivity generation and libraries to speed development. For more information, including how to obtain the tool, visit *http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html*. ■

REFERENCES

1. UG1165, Zynq-7000 All Programmable SoC: Embedded Design Tutorial
2. UG850, ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide
3. UG871, Vivado Design Suite Tutorial: High-Level Synthesis
4. UG948, Vivado Design Suite Tutorial: Model-Based DSP Design using System Generator
5. UG994, Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator
6. UG782, Xilinx Software Development Kit (SDK) User Guide

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
cholesky_alt_top_0	A	A	IN	4096*4	• length: (BUF_SIZE) • sys_port:ACP	S_AXI_ACP:AXIDMA_SIMPLE
	L	L	OUT	4096*4	• length: (BUF_SIZE) • sys_port:ACP	S_AXI_ACP:AXIDMA_SIMPLE
	return	AP_return	OUT	4		M_AXI_GP0:AXILITE:0xC0

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size (bytes)	Paged or Contiguous	Cacheable or Non-cacheable
cholesky_alt_top_0	cholesky_alt_tb.cpp:246:23	A	(BUF_SIZE) * 4	contiguous	cacheable
		L	(BUF_SIZE) * 4	contiguous	cacheable
		ap_return	4	paged	cacheable

Figure 9 — SDSoC connectivity report

Compile, Debug, Optimize

by Jayashree Rangarajan
Senior Engineering Director,
Interactive Design Tools
Xilinx, Inc.
jayr@xilinx.com

Fernando Martinez Vallina
Software Development Manager, SDAccel
Xilinx, Inc.
vallina@xilinx.com

Vinay Singh
Senior Product Marketing Manager,
SDAccel
Xilinx, Inc.
singhj@xilinx.com

Xilinx's SDAccel development environment enables software application design flows for FPGAs.

Xilinx® FPGA devices mainly comprise a programmable logic fabric that lets application designers exploit both spatial and temporal parallelism to maximize the performance of an algorithm or a critical kernel in a large application. At the heart of this fabric are arrays of lookup-table-based logic elements, distributed memory cells and multiply-and-accumulate units. Designers can combine those elements in different ways to implement the logic in an algorithm while achieving power consumption, throughput and latency design goals.

The combination of FPGA fabric elements into logic functions has long been the realm of hardware engineers, involving a process that resembles assembly-level coding more closely than it mimics modern software design practices. Whereas common software design procedures long ago moved beyond assembly coding, FPGA design practices have progressed at a slower pace because of the inherent differences between CPU and FPGA compilation.

In the case of CPUs and GPUs, the hardware is fixed, and all programs are compiled against a static instruction set architecture (ISA). Although the ISAs differ between CPUs and GPUs, the basic underlying compilation techniques are the same. Those similarities have enabled the evolution of design practices from handcrafted assembly code into compilation, debug and optimization design procedures that leverage the OpenCL™ C, C and C++ programming languages common to software development.

In the case of FPGA design, designers can create their own processing architecture to perform a specific workload. The ability to customize the architecture to a specific system need is a key advantage of FPGAs, but it has also acted as a barrier to adopting software development practices for FPGA application development.

Six years ago, Xilinx began a diligent R&D effort to break down this barrier by creating a development environment that brought an intuitive software development design loop to FPGAs. The Xilinx SDAccel™ development environment for OpenCL C, C and C++ enables application compile, debug and optimization for FPGA devices in ways similar to the processes used for CPUs and GPUs, with the advantage of up to 25x better performance/watt for data center application acceleration.

Software designers can use the SDAccel development environment to create and accelerate many functions and applications. Let's look at how the SDAccel environment enables a compile, debug and optimization design loop on a median filter application.

MEDIAN FILTER

The median filter is a spatial function commonly used in image processing for the purpose of noise reduction (Figure 1). The algorithm inside the median filter uses a 3 x 3 window of pixels around a center pixel to compute the value of the center based on the median of all neighbors. The equation for this operation is:

```
  outputPixel[i][j] =
  median(inputPixel[i-1][j-1],  inputPix-
el[i-1][j],  inputPixel[i-1][j+1],
     inputPixel[i][j-1],     inputPixel[i]
[j],     inputPixel[i][j+1],
     inputPixel[i+1][j-1], inputPixel[i+1]
[j], inputPixel[i+1][j+1]) ;
```

COMPILE

After the functionality of the median filter has been captured in a programming language such as OpenCL C, the first stage of development is compilation. On a CPU or GPU, compilation is a necessary and natural step in the software design flow. The target ISA is fixed and well known, leaving the programmer to worry only about the number of available processing cores and cache misses in the algorithm. FPGA compilation is more of an open question: At compilation time, the target ISA does not exist, the logic resources have yet to be combined into a processing fabric and the system memory architecture is yet to be defined.

The compiler in the SDAccel development environment provides three features that help programmers tackle those challenges: automatic extraction of parallelism among statements within a loop and across loop iterations, automatic memory architecture inference based on read and write patterns to arrays, and architectural awareness of the type and quantity of basic logic elements inside a given FPGA device. We can illustrate the importance of these three features with regard to source code for a median filter (Figure 2).

The median filter operation is expressed as a series of nested loops with two main sections. The first section fetches data from an array in external memory called input and stores the values into a local array RGB. The second section of the algorithm is the "for" loop around the getMedian function; getMedian is where the computation takes place.

By analyzing the code in Figure 2, the SDAccel environment understands that there are no loop-carried dependencies on the array RGB. Each loop iteration has a private RGB copy, which can be stored on different physical resources. The other main characteristic that the SDAccel environment can derive from this code is the independent nature of calls to the getMedian function.

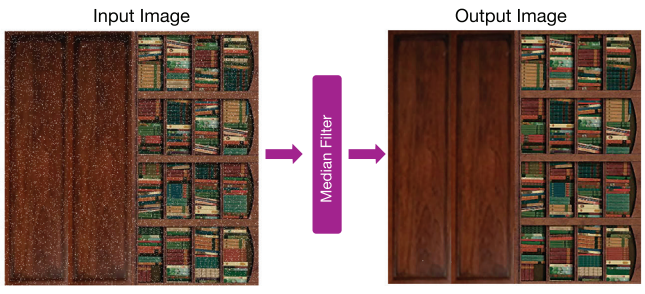


Figure 1 — Median filter operation

```
for (int y=0; y < height; y++) {
    int offset = y * width;
    int prev = offset - width;
    int next = offset + width;

    for (int x=0; x < width; x++) {
      // Get pixels within 3x3 aperture
      uint rgb[SIZE];
      rgb[0] = input[prev + x - 1];
      rgb[1] = input[prev + x];
      rgb[2] = input[prev + x + 1];

      rgb[3] = input[offset + x - 1];
      rgb[4] = input[offset + x];
      rgb[5] = input[offset + x + 1];

      rgb[6] = input[next + x - 1];
      rgb[7] = input[next + x];
      rgb[8] = input[next + x + 1];

      uint result = 0;

      // Iterate over all color channels
      for (int channel = 0; channel < 3;
      channel++) {
       result |= getMedian(channel, rgb);
      }

      // Store result into memory
      output[offset + x] = result;
    }
```

Figure 2 — Median filter code

The version of the algorithm in Figure 2 executes the getMedian function inside a "for" loop with a fixed bound. Depending on the performance target for the filter and the FPGA selected, the SDAccel environment can either reuse the compute resources across all three channels or allocate more logic to enable spatial parallelism and run all channels at the same time. This decision, in turn, affects how memory storage for the array RGB is implemented.

From an application programmer's perspective, the steps described above are transparent and can be thought of as –O1 to –O3 optimizations in the GNU Compiler Collection (GCC).

The printf implementation in the SDAccel environment provides the functionality without consuming additional logic resources.

DEBUG

An axiom of software development is that application compilation does not equal application correctness. It is only after the application starts to run on the target hardware that a programmer can start to discover, trace and correct errors in the application—in other words, debug.

CPU application debug is a well-understood problem, with a multitude of tools from both commercial vendors and the open-source community available to address it. Once again, FPGAs are another story. How does an application programmer debug something that was created to implement the functionality of a piece of code at a given performance target?

The SDAccel development environment addresses this question by borrowing two concepts from the CPU world: printf and GDB debugging.

The printf function is a fundamental tool in the software programmer's toolbox. It is available in every programming language and can be used to expose the state of key application variables during program execution. For CPU devices, this is as simple as monitoring the status of registers. There is no cost in hardware for printf functionality.

In the case of FPGAs, the implementation of printf can potentially consume logic resources that could otherwise be used for implementing algorithm functionality. The printf implementation in the SDAccel environment provides the functionality without consuming additional logic resources. The environment achieves this by separating printf data generation from the decoding and user presentation layers. In terms of hardware resources, the generation of data for printf consumes a few registers—a negligible cost in the register-rich FPGA fabric. Data decoding occurs in the driver to the FPGA. By leveraging the host CPU to execute the data decode and presentation layers for printf, a software programmer can use printf with virtually zero cost in FPGA resources.

The second technique for debugging borrowed from CPUs is the use of tools such as the GNU Project Debugger (GDB) to include breakpoints and single stepping through code. Programmers can use the SDAccel environment's emulation modes to attach GDB to a running emulation process. The emulation process is a simulation of the application-specific hardware that the developer will execute on the FPGA device. Within the context of an emulation process, GDB can watch the state of variables, insert breakpoints and single step through code. From an application programmer's perspective, this is identical to how GDB works on a CPU.

OPTIMIZE

After compiling and debugging, the next step in the software development cycle is to optimize the application. The principles behind application optimization on an FPGA are the same as on a CPU; the difference is in the approach. For a CPU, application code is massaged to fit into the boundaries of the cache and arithmetic units of a processor. In an FPGA, the computation logic is custom assembled for the current application. Therefore, the size of the FPGA and the application's target performance dictate the optimization constraints.

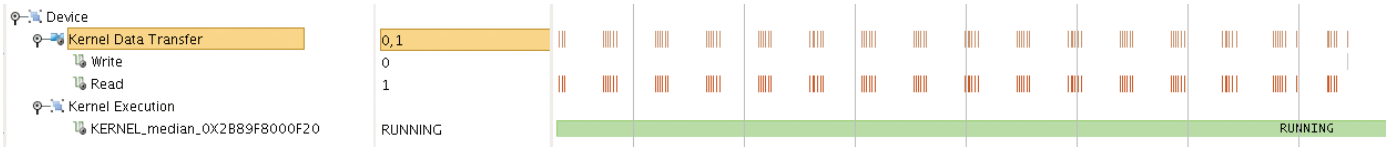


Figure 3 — Memory access transaction trace

Software programmers who use SDAccel can leverage the flexibility of the logic fabric to build high-performance, low-power applications without having to understand all of the details associated with hardware design.

The compiler in the SDAccel environment automatically optimizes the compute logic. The programmer can assist the automatic optimizations by analyzing the data transfer patterns inferred from the code. Figure 3 shows the read and write transactions from the median filter code to the memories for input and output.

Each vertical line in the plot represents a transaction to memory. The green bar shows the duration of media filter function activity. It can be seen from the plot that although the median filter is always active, there are large gaps between memory transactions. The gaps represent the time it takes the median filter to switch from one transaction to the next. Since each transaction to memory accesses only a single value, the gaps between transactions represent an important performance bottleneck for this application.

One way to solve the performance problem is to state burst transactions from external memories to local memories explicitly inside the application code. The code excerpt in Figure 4 shows the use of the async_work_group_copy function employed in OpenCL C kernels. The purpose of this function call is to tell the compiler that each transaction to memory will be a burst containing multiple data values. This enables more efficient utilization of the available memory bandwidth on the target device and reduces the overall number of transactions to memory. In the code of Figure 4, the async_work_group_copy function brings the contents of entire lines from the input image in DDR memory to memories inside the kernel data path.

```
for (int line = 0; line < height; line++) {
    local uint linebuf0[MAX_WIDTH];
    local uint linebuf1[MAX_WIDTH];
    local uint linebuf2[MAX_WIDTH];
    local uint lineres[MAX_WIDTH];
    // Fetch Lines
    if (line == 0) {
      async_work_group_copy(linebuf0,
      input, width, 0);
      async_work_group_copy(linebuf1,
      input, width, 0);
      async_work_group_copy(linebuf2,
      input + width, width, 0);
    }
…
}
```

Figure 4 — Median filter code with explicit burst memory transfers

The memory transaction trace in Figure 5 shows the result of using async_work_group_copy. As Figure 5 shows, the kernel involves a setup time before memory transactions occur that is not present in the original code for the median filter (Figure 2).

The setup time difference has to do with the logic derived from the code. In the original code of Figure 2, the application immediately starts a single transaction to memory and then waits for the data to be available. In contrast, the optimized code of Figure 4 determines whether a memory transaction needs to occur or whether the data is already available in the kernel's local memory. It also allows the generated logic to schedule memory transactions back-to-back and to overlap read and write transactions.

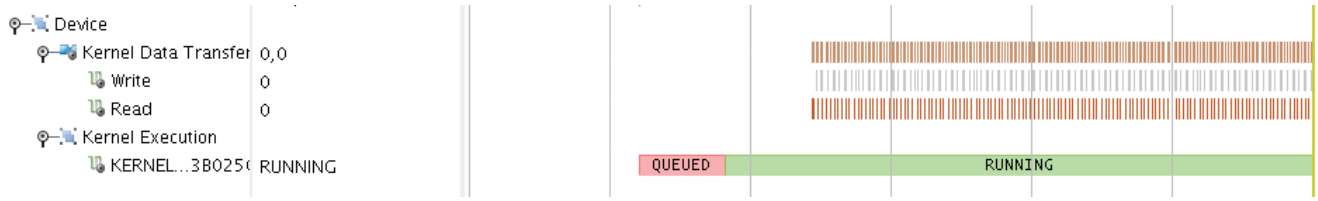
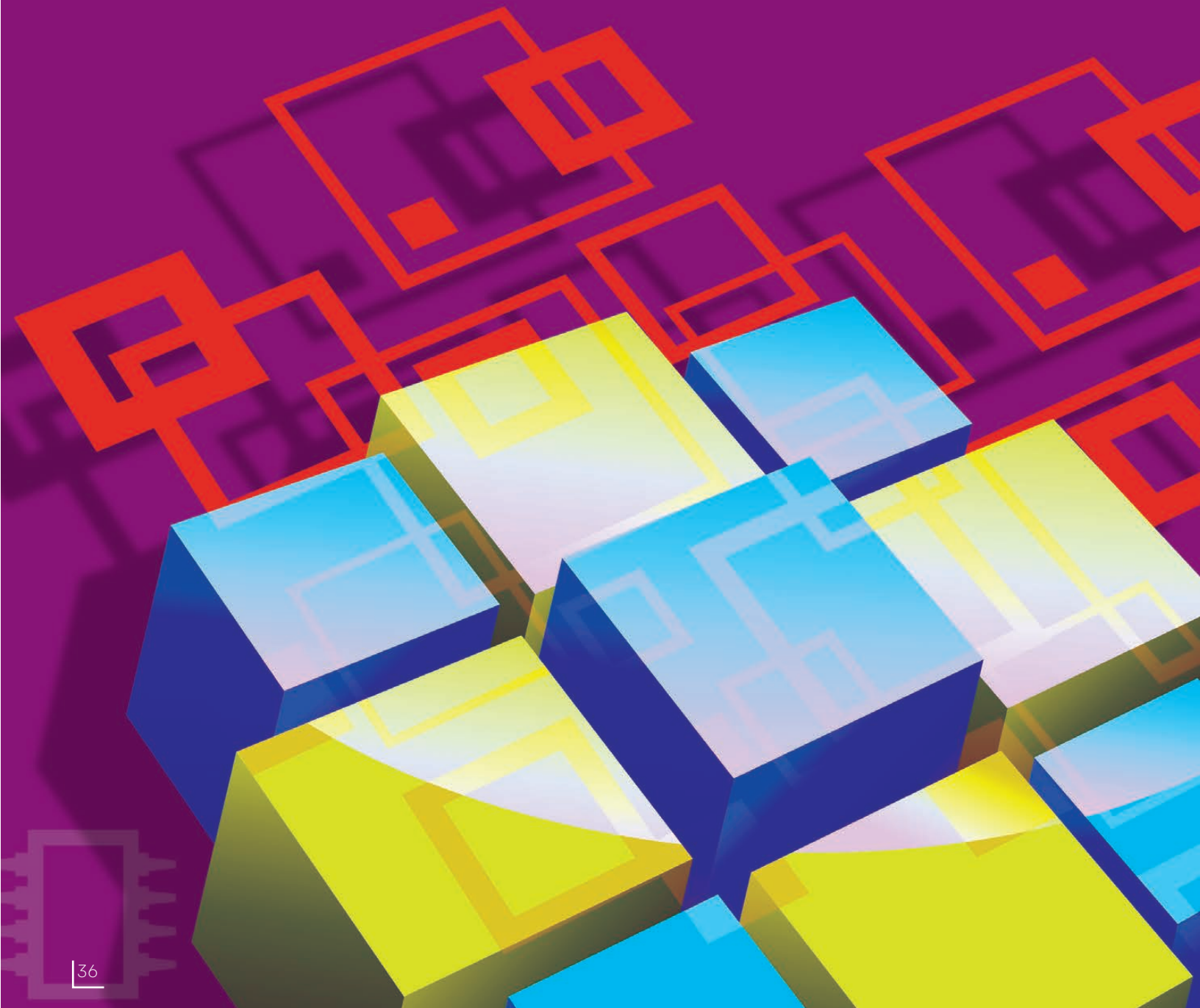


Figure 5 — Memory access transaction trace after code optimization

Whether the final device is a CPU or an FPGA, profiling is an essential component of application development. The SDAccel environment's visualization and profiler capabilities let an application programmer characterize the impact of code changes and application requirements in terms of kernel occupancy, transactions to memory and memory bandwidth utilization.

The design loop created by the operations of compile, debug and optimize is fundamental to software development flows. The SDAccel development environment enables this design loop with tools and techniques similar to the development environment on a CPU, with FPGA-based application acceleration of up to 25x better performance per watt and with a 50x to 75x latency improvement. Software programmers who use SDAccel can leverage the flexibility of the logic fabric to build high-performance, low-power applications without having to understand all of the details associated with hardware design. ■

Developing OpenCL Imaging Applications Using C++ Libraries



by Stephen Neuendorffer
Principal Engineer, Vivado HLS
Xilinx, Inc.
stephenn@xilinx.com

Thomas Li
Software Engineer, Vivado HLS
Xilinx, Inc.
thl@xilinx.com

Fernando Martinez Vallina
Development Manager, SDAccel
Xilinx, Inc.
vallina@xilinx.com

Xilinx's SDAccel development environment leverages the power of preexisting libraries to accelerate application design.

Imaging applications have grown in both scale and ubiquity in recent years as online pictures and videos, robotics, and driver assistance applications have proliferated. Across these domains, the core algorithms are very similar and require a development methodology that lets application developers quickly retarget and differentiate products based on markets and deployment targets.

As a result of those needs, imaging applications typically start as a software program targeting a CPU and employ library calls to standard functions. The combination of software design techniques with readily available libraries makes it easy to get started and to create a functionally correct application on a desktop.

The challenge for the developer lies in optimizing the imaging application for an execution target. By leveraging technology from Xilinx® Vivado® HLS, Xilinx's SDAccel™ development environment makes the use of C++ libraries straightforward for OpenCL™ application developers targeting FPGAs.

SET OF PARALLEL COMPUTATION TASKS

One key characteristic of imaging applications is that they are fundamentally a set of operations on a pixel with respect to a surrounding neighborhood of pixels in space and, for some applications, in time. We therefore can think of an imaging application as a set of parallel computation tasks that a developer can execute on a CPU, GPU or FPGA.

The CPU is always the easiest target device with which to start. The code typically already runs on the CPU before optimization is considered and can leverage the wealth of available libraries. The problem with executing imaging workloads on a CPU is the achievable sustained performance. The overall performance is limited by cache hits/misses and the nontrivial task of parallelization into multiple threads running across CPU cores.

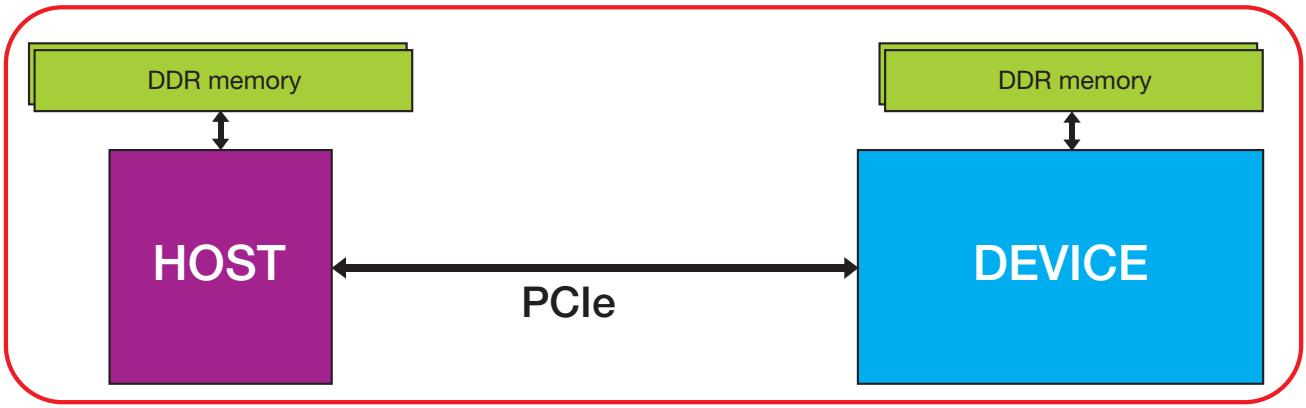


Figure 1 — The basic OpenCL platform contains one host and at least one device.

GPUs hold the promise of much higher performance than CPUs for imaging applications because GPU hardware was purposely built for imaging workloads. Until recent years, the drawback of GPUs for general imaging applications had been the programming model. GPU programming differed from that for CPUs, and GPU models were not portable across GPU device families. That changed with the standardization of programming for parallel systems such as GPUs under the OpenCL framework.

FPGAs provide an alternative implementation choice for imaging workloads. Developers can customize the FPGA logic fabric into workload-specific circuits. The flexibility of the FPGA fabric lets an application developer leverage the performance and power consumption benefits of custom logic while avoiding the cost and effort associated with ASIC design.

As it was for the GPU, one barrier for adoption of FPGA devices has been the programming model. Traditionally, FPGAs have been programmed with a register transfer language (RTL) such as Verilog or VHDL. Although those languages can express parallelism, the level of granularity is significantly lower than what is needed to program a CPU or a GPU. As in the case of GPUs, however, adoption of the OpenCL standard to express FPGA programming in a way that is familiar to software application developers has overcome the programming model hurdle.

OPENCL FRAMEWORK

The OpenCL framework provides a common programming model for expressing data parallel programs. The framework, which has evolved into an industry standard, is based on a platform and a memory model that are consistent across all device vendors supporting OpenCL. A device is defined as any hardware, be it a CPU, GPU or FPGA, capable of executing OpenCL kernels.

The platform in an OpenCL application defines the hardware environment in which an application is executed. Figure 1 shows the main elements of an OpenCL platform.

A platform for OpenCL always contains one host, which is typically implemented on a processor. The host is responsible for launching tasks on the device and for explicitly coordinating all data transfers between the host and the device.

In addition to the host, a platform contains at least one device. The device in the OpenCL platform is the hardware element capable of executing OpenCL kernel code. In the context of an OpenCL application, the kernel code is the computationally intensive part of the algorithm that requires acceleration.

In the case of CPU and GPU devices, the kernel code is executed on one or more cores in the device. Each core is exactly the same per the device specification; this stricture forces the application developer to modify the code to maximize performance within a fixed architecture.

For an FPGA, the SDAccel development environment generates custom cores per the specific computation requirements of the application kernel.

For an FPGA, in contrast, the SDAccel development environment generates custom cores per the specific computation requirements of the application kernel. The application developer thus is free to explore implementation architectures based on the needs of the algorithm to reduce overall system latency and power consumption.

The second OpenCL component is the memory model (Figure 2). This model, which is common to all vendors, defines a single memory hierarchy against which a developer can create a portable application.

The main components of the memory model are the host, global, local and private memories. The host memory refers to the memory space that is accessible only to the host processor. The memories visible to the FPGA (the device) are the global, local and private memory spaces. The global memory space is accessible to both the host and the device and is typically implemented in DDR attached to the FPGA. Depending on the FPGA used on the acceleration board, a portion of the global memory can also be implemented inside the FPGA fabric. The local and private memory spaces are visible only to the kernels executing inside the FPGA fabric and are built entirely inside that fabric using block RAM (BRAM) and register resources.

Let's see how the SDAccel environment leverages OpenCL and C++ libraries for a stereo imaging block matching application.

STEREO BLOCK MATCHING

Stereo block matching uses images from two cameras to create a representation of the shape of an object in the field of view of the cameras. As Figure 3

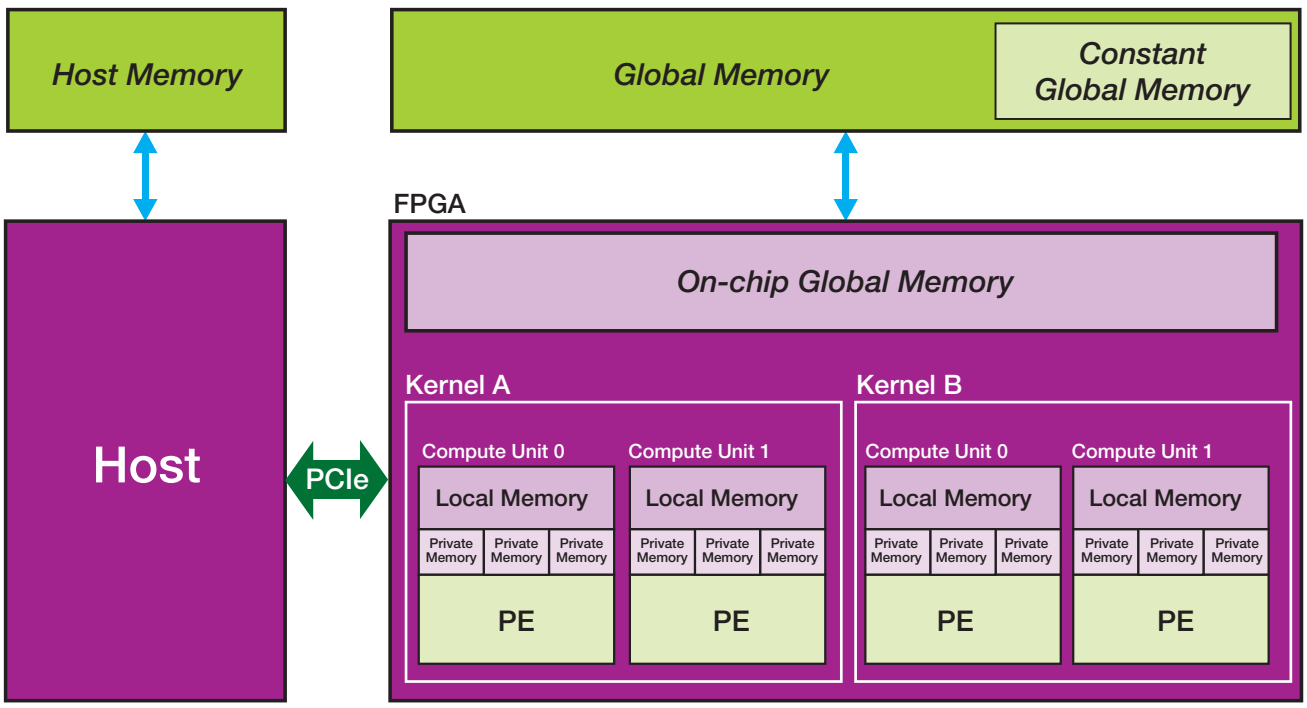


Figure 2 — The OpenCL memory model defines a single memory hierarchy for application development.

The SDAccel development environment leverages technology from Xilinx's Vivado HLS C-to-RTL compiler as part of the core kernel compiler, letting the SDAccel environment use kernels expressed in C and C++ in the same way as kernels expressed in OpenCL C.

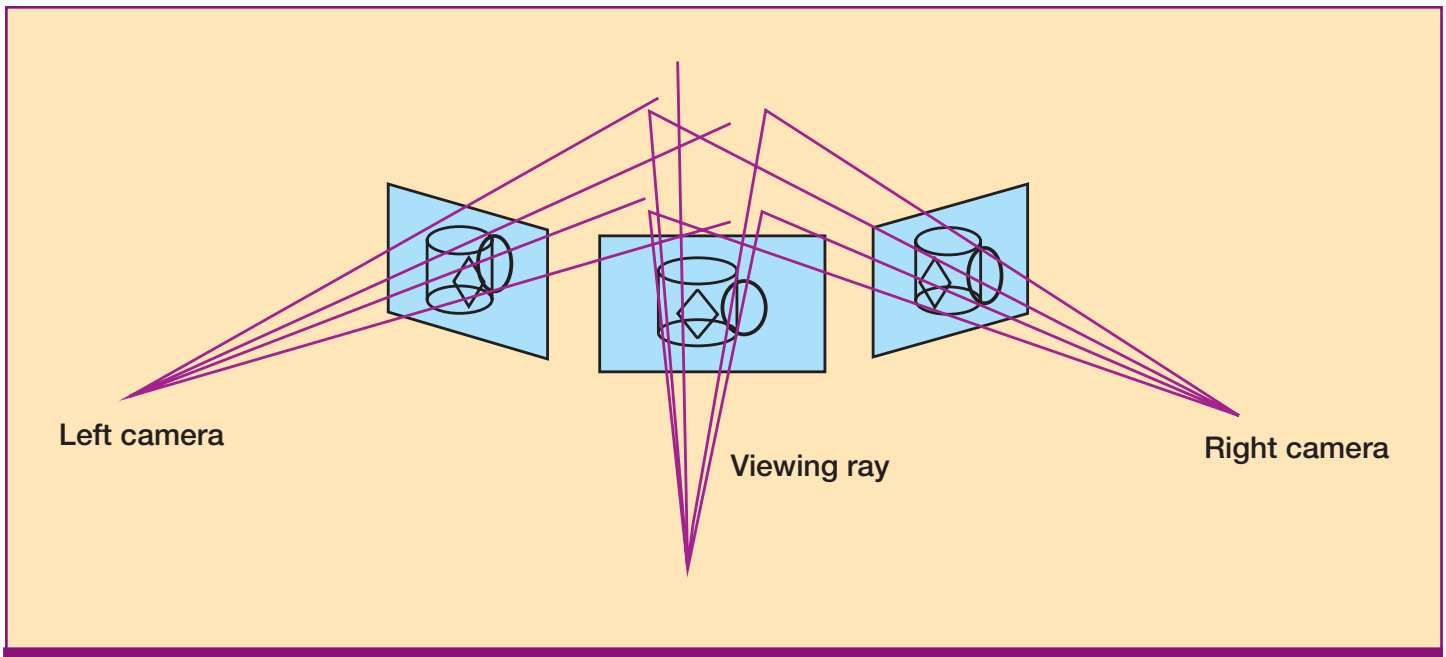


Figure 3 — Conceptual image of multi-camera processing

shows, the algorithm uses the input images of a left and a right camera to search for the correspondence between the images. Such multi-camera image processing tasks can be applied to depth maps, image segmentation and foreground/background separation. These are, for example, all integral parts of pedestrian detection applications in driver assistance systems.

USING C++ LIBRARIES FOR VIDEO

The SDAccel development environment leverages technology from Xilinx's Vivado HLS C-to-RTL compiler as part of the core kernel compiler, letting the SDAccel environment use kernels expressed in C and C++ in the same way as kernels expressed in OpenCL C. Application developers thus can use C++ libraries and code previously optimized in Vivado HLS to increase productivity.

The main code for the stereo block matching application is shown on the next page.

Vivado HLS provides image processing functions based on the popular OpenCV framework. The functions are written in C++ and have been optimized to provide high performance in an FPGA. When synthesized into an FPGA implementation, the equivalent of anywhere from tens to thousands of RISC processor instructions are executed concurrently every clock cycle.

The code for the application uses Vivado HLS video processing functions to create the application. The application code contains C++ function calls to Vivado HLS libraries as well as pragmas to guide the compilation process. The pragmas are divided into those for interface definition and those for performance optimization.

The interface definition pragmas determine how the stereo block matching accelerator connects to the rest of the system. Since this accelerator is expressed in C++ instead of OpenCL C code, the application programmer must provide interface definition

```
void stereobm(
        unsigned short img_data_lr[MAX_HEIGHT*MAX_WIDTH],
        unsigned char img_data_d[MAX_HEIGHT*MAX_WIDTH],
        int rows,
        int cols)
{
#pragma HLS INTERFACE m_axi port=img_data_lr offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=img_data_d offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=img_data_lr bundle=control
#pragma HLS INTERFACE s_axilite port=img_data_d bundle=control
#pragma HLS INTERFACE s_axilite port=rows bundle=control
#pragma HLS INTERFACE s_axilite port=cols bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC2>  img_lr(rows, cols);
    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>  img_l(rows, cols);
    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>  img_r(rows, cols);
    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_16SC1> img_disp(rows, cols);
    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1>  img_d(rows, cols);

    hls::StereoBMState<15, 32, 32> state;

#pragma HLS dataflow
    hls::AXIM2Mat<MAX_WIDTH>(img_data_lr, img_lr);
    hls::Split(img_lr, img_l, img_r);
    hls::FindStereoCorrespondenceBM(img_l, img_r, img_disp, state);
    hls::SaveAsGray(img_disp, img_d);
    hls::Mat2AXIM<MAX_WIDTH>(img_d, img_data_d);
}
```

pragmas that match the assumptions of the OpenCL model in the SDAccel environment.

The pragmas marked with m_axi state that the contents of the buffer will be stored in device global memory. The pragmas marked with s_axilite are required for the accelerator to receive the base address of buffers in global memory from the host.

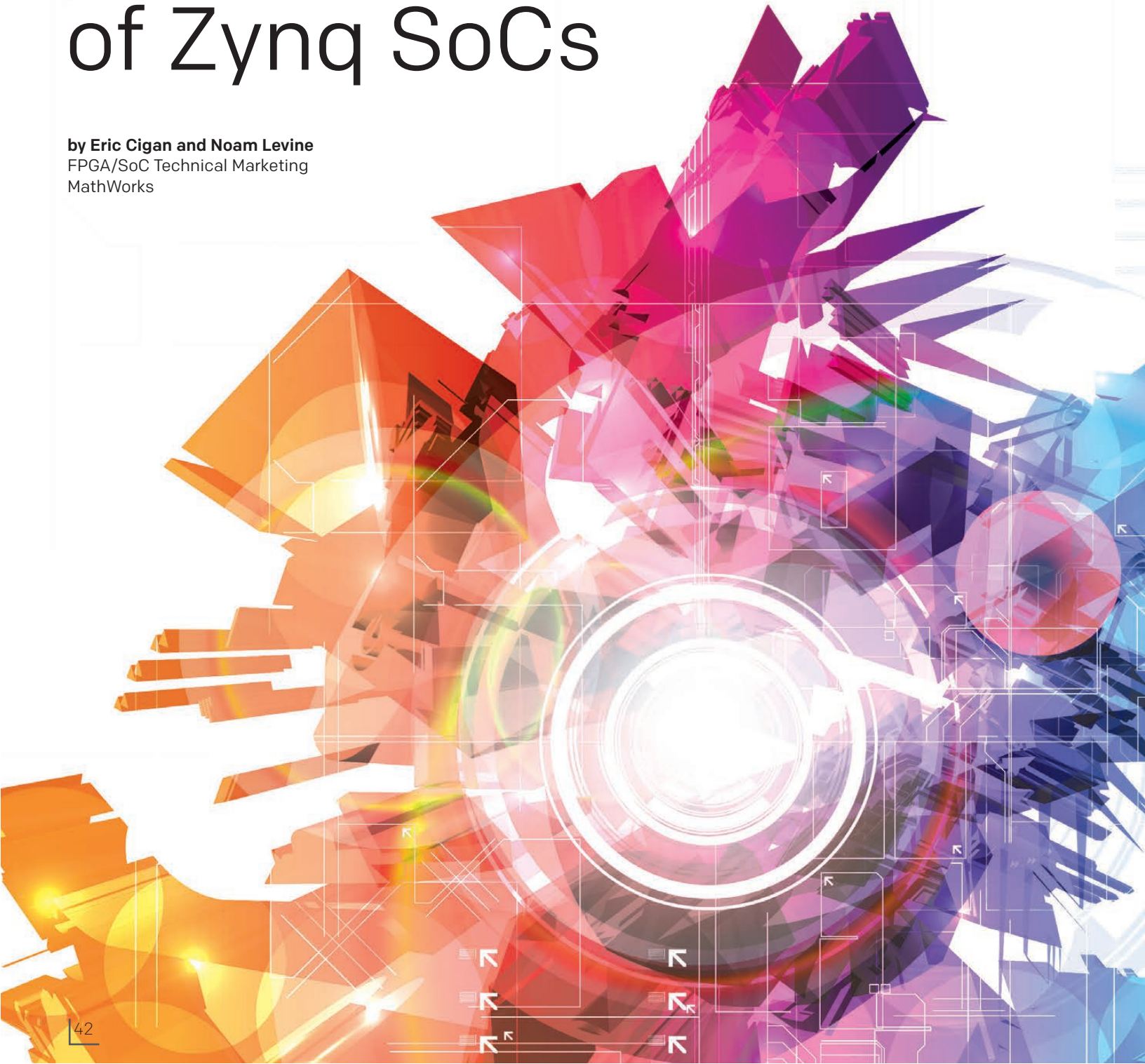
The performance optimization pragma in this code is dataflow. The dataflow pragma yields an accelerator in which different subfunctions can also execute concurrently.

In this accelerator, because of the underlying implementation of the hls::Mat datatype, data is also streamed between each function. This allows the FindStereoCorrespondenceBM function to start operating as soon as the Split function produces pixels, without having to wait for a complete image to be produced. The net result is a more efficient architecture and reduced processing latency relative to sequential processing of each function with full frame buffers in between them.

Imaging applications are a compute-intensive application domain with a rich set of available libraries; the devil is in optimizing the application for the execution target. The SDAccel environment lets developers leverage C++ libraries to accelerate the development of imaging applications for FPGAs programmed in OpenCL. ■

MATLAB and Simulink Aid HW-SW Co-design of Zynq SoCs

by Eric Cigan and Noam Levine
FPGA/SoC Technical Marketing
MathWorks



Model-Based Design workflow lets engineers make design trade-offs at the desktop rather than the lab.

The introduction of the Xilinx® Zynq®-7000 All Programmable SoC family in 2011 brought groundbreaking innovation to the FPGA industry. These devices, with their combination of dual-core ARM® Cortex™-A9 MPCore™ processors and ample programmable logic, offered advantages for a wealth of applications. By adopting Zynq SoCs, designers could reap the benefits of software application development on one of the industry's most popular processors while gaining the flexibility and throughput potential provided via hardware acceleration on a high-speed, programmable logic fabric.

Using MATLAB® and Simulink® from MathWorks®, innovators today can leverage a highly integrated hardware-software workflow to create highly optimized systems. The case study presented here illustrates this model-based workflow.

When Xilinx released the first Zynq SoC in December 2011, designers seized on the idea that they could migrate their legacy, multichip solutions, built from discrete processors and FPGAs, to a single-chip platform. They could create FPGA-based accelerators on the new platform to unclog software execution bottlenecks and tap into an array of off-the-shelf, production-ready intellectual property from Xilinx and its IP partners that would address applications in digital signal processing, networking, communications and more.

The open question was how they would program the new devices. Designers imagining the potential of hardware-software co-design sought integrated workflows that would intelligently partition designs between ARM processors and programmable logic. What they found, however, were distinct hardware and software workflows: conventional embedded software development flows targeting ARM cores, alongside a combination of IP assembly, traditional RTL and emerging high-level synthesis tools for programmable logic.

INTEGRATED WORKFLOW

In September 2013, MathWorks introduced a hardware-software workflow for Zynq-7000 SoCs using Model-Based Design. In this workflow (Figure 1), designers could create models in Simulink that would represent a complete dynamic system—including a Simulink model for algorithms targeted for the Zynq SoC—and rapidly create hardware-software implementations for Zynq SoCs directly from the algorithm.

System designers and algorithm developers used simulation in Simulink to create models for a complete system (communications, electromechanical components and so forth) in order to evaluate design concepts, make high-level trade-offs, and partition algorithms into software and hardware elements. HDL code generation from Simulink enabled the creation of IP cores and high-speed I/O processing on the Zynq SoC fabric. C/C++ code generation from Simulink enabled programming of the Zynq SoC's Cortex-A9 cores, supporting rapid embedded software iteration.

The approach enabled automatic generation of the AMBA® AXI4 interfaces linking the ARM processing system and programmable logic with support for the Zynq SoC. Integration with downstream tasks—such as C/C++ compilation and building of the executable for the ARM processing system, bitstream generation using Xilinx implementation tools, and downloading to Zynq development boards—allowed for a rapid prototyping workflow.

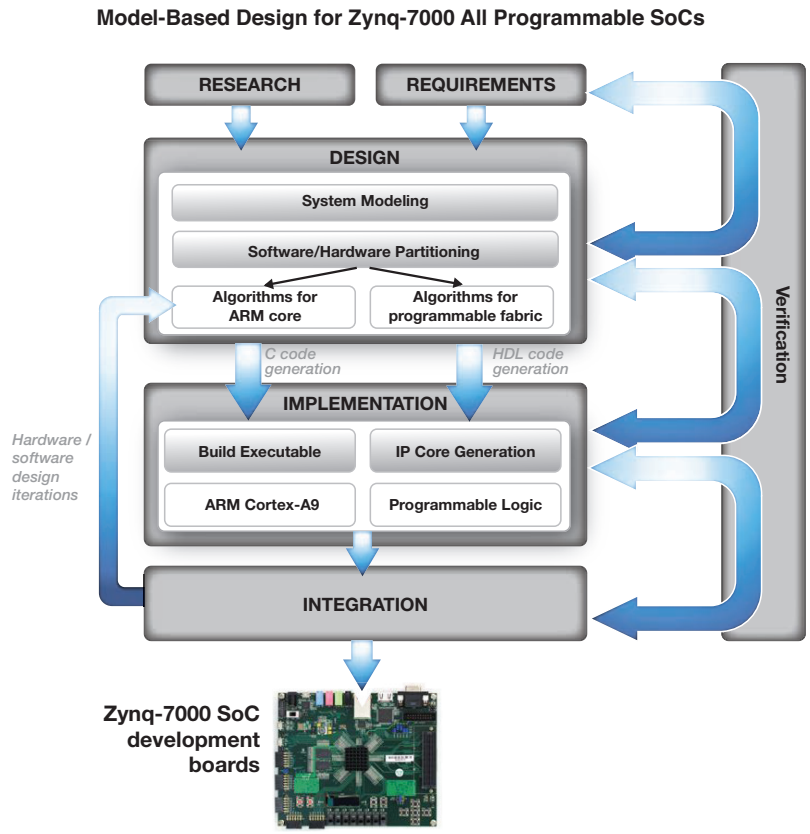


Figure 1 — Designers can create models in Simulink that represent a complete dynamic system and create hardware-software implementations for Zynq SoCs directly from the model.

Central to this workflow are two technologies: Embedded Coder® and HDL Coder™. Embedded Coder generates production-quality C and C++ code from MATLAB, Simulink and Stateflow®, with target-specific optimizations for embedded systems. Embedded Coder has become so widely adopted that when you drive a modern passenger car, take a high-speed train or fly on a commercial airline, there's a high probability that Embedded Coder generated the real-time code guiding the vehicle. HDL Coder is the counterpart to Embedded Coder, generating VHDL or Verilog for FPGAs and ASICs, and is integrated tightly into Xilinx workflows. This mature C and HDL code generation technology forms the foundation of the Model-Based Design workflow for programmable SoCs.

Design teams using Model-Based Design in applications such as communications, image processing, smart power and motor control have adopted this workflow as the means for algorithm developers to work closely with hardware designers and embedded software developers to accelerate the implementation of algorithms on programmable SoCs. Once the generated HDL and C code is prototyped in hardware, the design team can use Xilinx Vivado® IP Integrator to integrate the code with other design components needed for production.

CASE STUDY: THREE-PHASE MOTOR CONTROL

For several reasons, custom motor controllers with efficient power conversion are one of the most popular applications to have emerged for programmable SoCs. Higher-performance, higher-efficiency initiatives are one factor. With electric motor-driven systems accounting for as much as 46 percent of global electricity consumption, attaining higher efficiency with novel control algorithms is an increasingly common motor drive design goal. Xilinx Zynq programmable logic enables precise timing, providing an ideal platform for implementing low-latency, high-efficiency drives.

Another driver is multi-axis control. Ample programmable logic and DSP resources on programmable SoCs open up possibilities for implementing multiple motor controllers on a single programmable SoC, whether motors will operate independently or in combination, as in an integrated motion control system.

Integration of industrial networking IP is a further factor. Xilinx and its IP partners offer IP for integration with EtherCAT, PROFINET and other industrial networking protocols that can be readily incorporated into programmable SoCs.

With electric motor-driven systems accounting for as much as 46 percent of global electricity consumption, attaining higher efficiency with novel control algorithms is an increasingly common motor drive design goal.

To illustrate the use of this workflow on a common motor control example, consider the case of a field-oriented control algorithm for a three-phase electric motor implemented on a Zynq-7020 SoC (details of this hardware prototyping platform are available at *http://www.mathworks.com/zidk*). The motor control system model includes two primary subsystems (Figure 2): a motor controller targeting the Zynq SoC that has been partitioned between the Zynq processing system and programmable logic, and a motor controller FPGA mezzanine card (FMC) connected to a brushless DC motor equipped with an encoder to measure shaft angle.

We can look at hardware-software partitioning in terms of data flow:

- We assign the Velocity Control and Mode Select blocks to the ARM Cortex-A9 processing system because those blocks can run at a slower rate than other parts of the model and because they are the portions of the design most likely to be modified and recompiled during development.
- A Mode Select state machine running on the ARM core determines the motor controller operating mode (for example, open-loop operation or closed-loop regulation). This state machine manages the transitions between the start-up, open-loop control and encoder calibration modes before switching to a closed-loop control mode.
- The encoder sensor signal is passed via an external port to an Encoder Peripheral in the programmable logic and then to a Position/Velocity Estimate block that computes the motor's state (shaft position and velocity).
- A sigma-delta analog-to-digital converter (ADC)

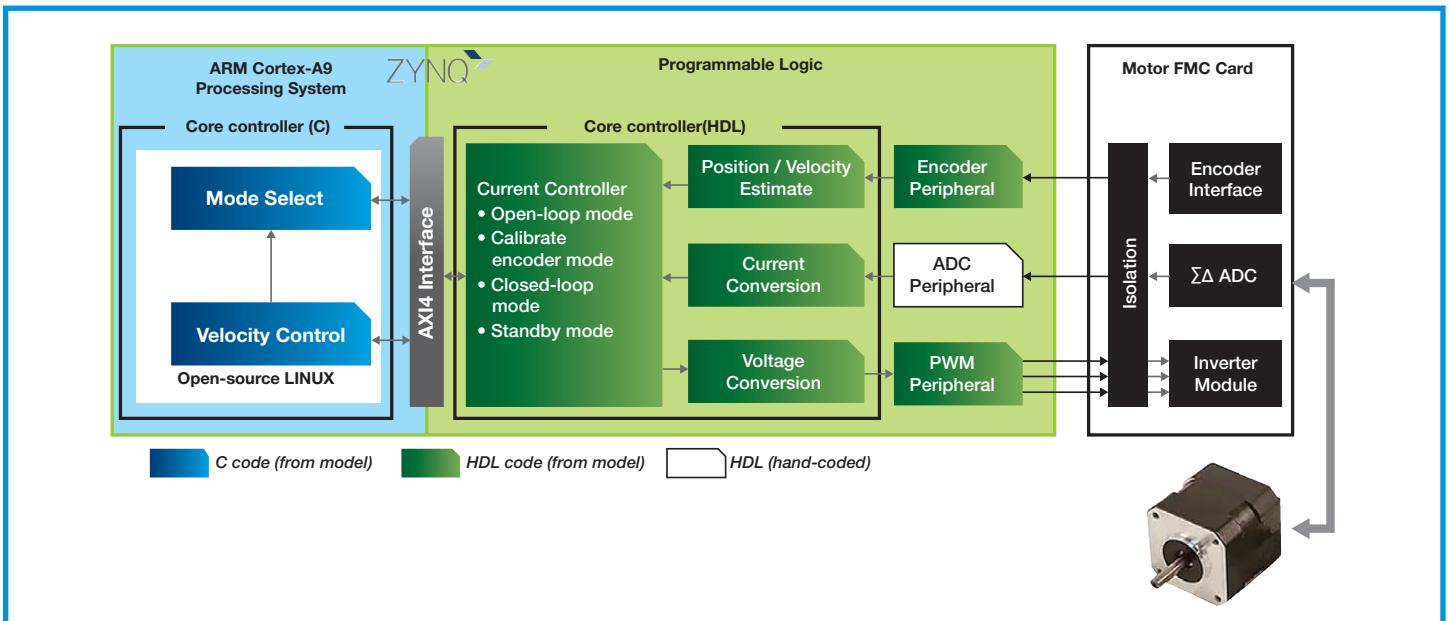


Figure 2 — The motor control system model includes two primary subsystems.

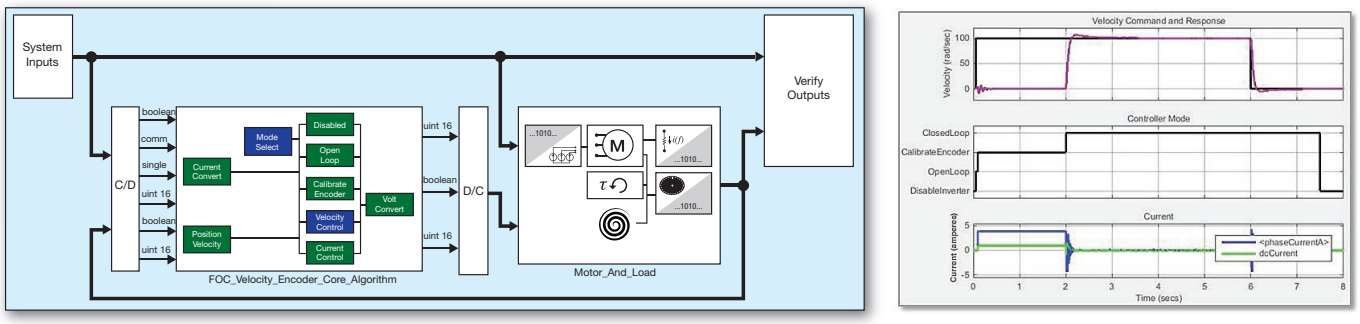


Figure 3 — This control-loop model for a motor control system with simulation results shows the response to a velocity pulse command.

senses the motor current, and a hand-coded ADC Peripheral block processes the current.

- The Current Controller takes the motor state and current, as well as the operating mode and velocity control commands passed from the ARM core over the AXI4 interface, and computes the current controller command. When in its closed-loop mode, the Current Controller uses a proportional-integral (PI) control law, whose gains can be tuned using simulation and prototyping.
- The current controller command goes through the Voltage Conversion block and is output to the motor control FMC via the PWM Peripheral, ultimately driving the motor.

Designers can model the complete system in Simulink (Figure 3).

In Model-Based Design, the system increases to four components in the top-level Simulink model:

- an input model, which provides a commanded shaft velocity and on/off commands to the controller as stimulus;
- a model of the motor control algorithm that will be targeted for the Zynq SoC;
- a plant model, which includes the drive electronics of the FMC, a permanent-magnet synchronous machine (PMSM) model of the brushless DC motor, a model of an inertial load on the motor shaft and an encoder sensor model; and
- an output-verification model, which includes post-processing and graphics to help the algorithm developer refine and validate the model.

In Simulink, we can test out the algorithm with simulation long before we start hardware testing. We can tune the PI controller gains, try various stimulus profiles and examine the effect of different processing rates. As we use simulation, though, we face a fundamental issue: Because of the disparate processing rates typical of motor control—that is, overall mechanical response rates of 1 to 10 Hz, core controller algorithm rates of 1 to 25 kHz and programmable logic operating at 10 to 50 MHz or more—simulation times can run to many minutes or even hours. We can head

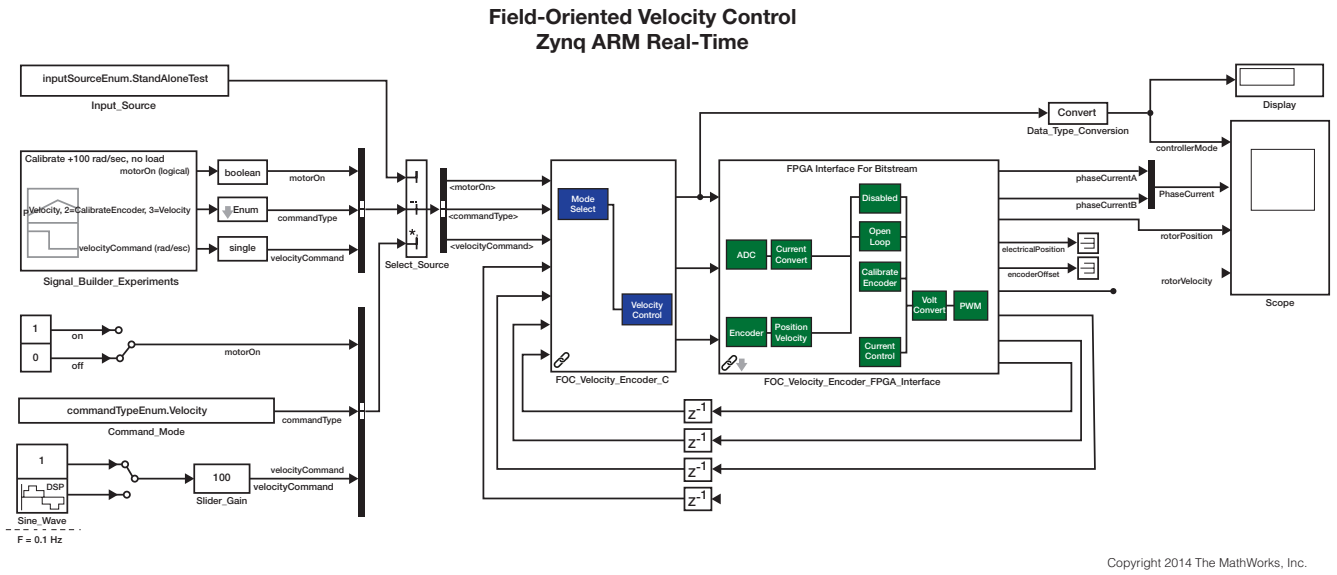
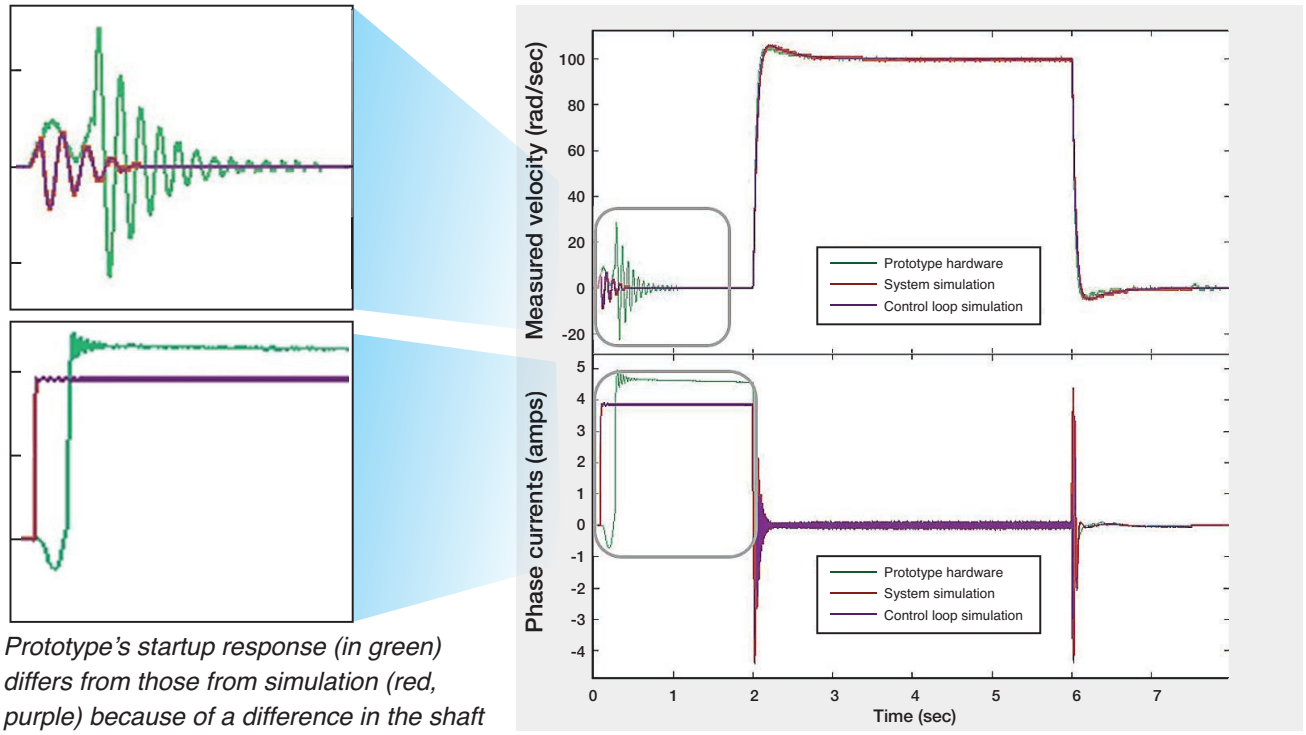


Figure 4(a) — Simulink model for testing prototype hardware



Prototype's startup response (in green) differs from those from simulation (red, purple) because of a difference in the shaft angle at t=0.

Figure 4(b) — Comparison of results from hardware prototype and simulation

Continuous verification between the simulation and hardware environments lets designers identify and resolve issues early in the development process.

off this issue with a control-loop model that uses behavioral models for the peripherals—the PWM, current sensing and encoder processing—producing the time response shown in Figure 3.

After we use the control-loop model to tune the controller, our next step is to prove out the controller in simulation using high-fidelity models that include the peripherals. We do this by incorporating timing-accurate specification models for the C and HDL components of the controller. These specification models have the necessary semantics for C and HDL code generation. With simulation, we then verify that the system with specification models tracks extremely closely to the control-loop model.

Once performance has been validated with the high-fidelity models, we move on to prototyping the controller in hardware. Following the workflow shown in Figure 1, we start by generating the IP core. The IP core generation workflow lets us choose the target development board and walks us through the process of mapping the core's input and output ports to target interfaces, including the AXI4 interface and external ports.

Through integration with the Vivado Design Suite, the workflow builds the bitstream and programs the fabric of the Zynq-7020 SoC.

With the IP core now loaded onto the target device, the next step is to generate embedded C code from the Simulink model targeting the ARM core. The process of generating C code, compiling it and building the executable with embedded Linux is fully automated, and the prototype is then ready to run.

To run the prototype hardware and verify that it gives us results consistent with our simulation models, we build a modified Simulink model (Figure 4a) that will serve as a high-level control panel. In this model, we removed the simulation model for the plant—that is, the drive electronics, motor, load and sensor—and replaced it with I/Os to the ZedBoard.

Using this model in a Simulink session, we can turn on the motor, choose different stimulus profiles, monitor relevant signals and acquire data for later post-processing in MATLAB, but for now we can repeat the pulse test (Figure 3).

Figure 4b shows the results of the shaft rotational velocity and the phase current for the hardware prototype compared with the simulation results. The startup sequence for the hardware prototype differs noticeably from those for the two simulation models. This is to be expected, however, because the initial angle between the motor's rotor and stator in the hardware test differs from the initial angle used in simulation, resulting in a different response as the current control algorithm drives the motor through its encoder calibration mode. When the pulse is applied at 2 seconds, the results from simulation and prototype hardware match almost exactly.

Based on these results, we could continue with further testing under different loading and operating conditions, or we could move on to performing further C and HDL optimizations.

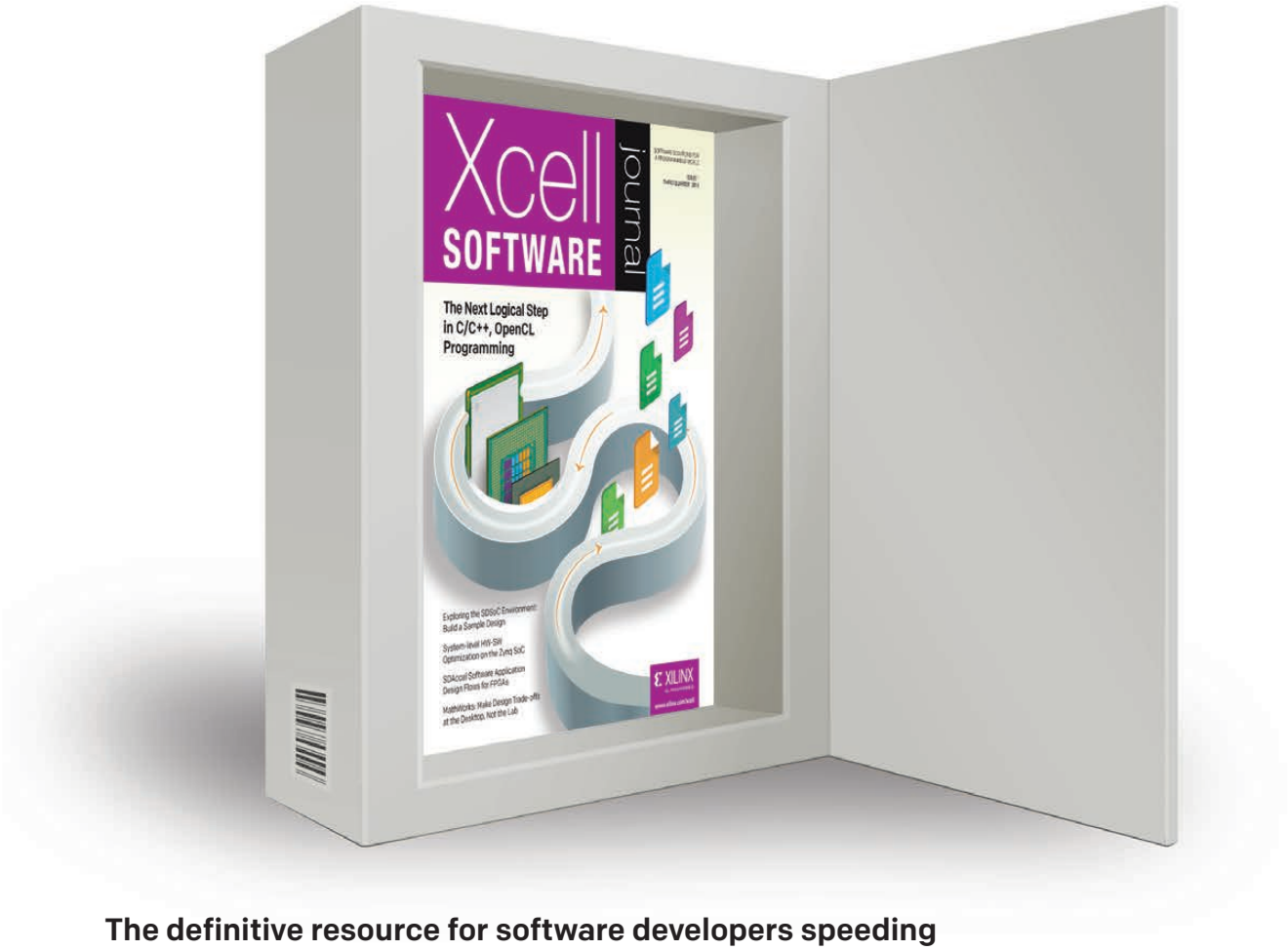
Engineers are turning to Model-Based Design workflows to enable hardware-software implementation of algorithms on Xilinx Zynq SoCs. Simulink simulation provides early evaluation of algorithms, letting designers evaluate the algorithms' effectiveness and make design trade-offs at the desktop rather than in the lab, with a resultant increase in productivity. Proven C and HDL code generation technology, along with hardware support for Xilinx All Programmable SoCs, provides a rapid and repeatable process for getting algorithms running on real hardware. Continuous verification between the simulation and hardware environments lets designers identify and resolve issues early in the development process.

Workflow support for Zynq-based development boards, software-defined radio kits and motor control kits is available from MathWorks. To learn more about this workflow, visit *http://www.mathworks.com/zynq*. ■

This year's best release.

Xcell Publications

Solutions for a Progammable World



The definitive resource for software developers speeding C/C++ & OpenCL code with Xilinx SDx IDEs & devices

The Award-winning Xilinx Publication Group is rolling out a brand new trade journal specifically for the programmable FPGA software industry, focusing on users of Xilinx SDx™ development environments and high-level entry methods for programming Xilinx All Programmable devices.

This is where you come in.

Xcell Software Journal is now accepting reservations for advertising opportunities in this new, beautifully designed and written resource. Don't miss this great opportunity to get your product or service into the minds of those who matter most. Call or write today for your free advertising packet!

For advertising inquiries (including calendar and advertising rate card), contact xcelladsales@aol.com or call: 408-842-2627.

XILINX
ALL PROGRAMMABLE™

Xtra, Xtra

Xilinx® is constantly refining its software and updating its training and resources to help software developers design innovations with the Xilinx SDx™ development environments and related FPGA and SoC hardware platforms. Here is list of additional resources and reading. Check for the newest quarterly updates in each issue.

SDSOC™ DEVELOPMENT ENVIRONMENT

The SDSoC environment provides a familiar embedded C/C++ application development experience, including an easy-to-use Eclipse IDE and a comprehensive design environment for heterogeneous Xilinx All Programmable SoC and MPSoC deployment. Complete with the industry's first C/C++ full-system optimizing compiler, SDSoC delivers system-level profiling, automated software acceleration in programmable logic, automated system connectivity generation and libraries to speed programming. It lets end-user and third-party platform developers rapidly define, integrate and verify system-level solutions and enable their end customers with a customized programming environment.

- SDSoC Backgrounder (PDF)
- SDSoC User Guide (PDF)
- SDSoC User Guide: Getting Started (PDF)
- SDSoC User Guide: Platforms and Libraries (PDF)
- SDSoC Release Notes (PDF)
- Boards, Kits and Modules
- SDSoC Video Demo
- Buy/Download

SDACCEL™ DEVELOPMENT ENVIRONMENT

The SDAccel environment for OpenCL™, C and C++ enables up to 25x better performance/watt for data center application acceleration leveraging FPGAs. A member of the SDx family, the SDAccel environment combines the industry's first architecturally optimizing compiler supporting any combination of OpenCL, C and C++ kernels, along with libraries, development boards, and the first complete CPU/GPU-like development and run-time experience for FPGAs.

- SDAccel Backgrounder
- SDAccel Development Environment: User Guide
- SDAccel Development Environment: Tutorial
- Xilinx Training: SDAccel Video Tutorials
- Boards and Kits
- SDAccel Demo

SDNET™ DEVELOPMENT ENVIRONMENT

The SDNet environment, in conjunction with Xilinx All Programmable FPGAs and SoCs, lets network engineers define line card architectures, design line cards and update them with a C-like environment. It enables the creation of "Softly" Defined Networks, a technology dislocation that goes well beyond today's Software Defined Networking (SDN) architectures.

- SDNet Backgrounder — Xilinx
- SDNet Backgrounder — The Linley Group
- SDNet Demo

SOFTWARE DEVELOPMENT KIT (SDK)

The SDK is Xilinx's development environment for creating embedded applications on any of its microprocessors for Zynq®-7000 All Programmable SoCs and the MicroBlaze™ soft processor. The SDK is the first application IDE to deliver true homogeneous- and heterogeneous-multiprocessor design and debug.

- Free SDK Evaluation and Download ■

Program FPGAs Faster
With a Platform-Based Approach

Take advantage of an integrated hardware and software platform to shorten development cycles and deliver FPGA-enabled technology to market faster. With rugged CompactRIO controllers and LabVIEW FPGA software, you can program and customize Xilinx FPGAs in a more productive environment with a higher level of abstraction–all without knowledge of hardware description languages. Use LabVIEW FPGA's cycle-accurate simulation, built-in functions for I/O, memory management, bus interfaces, and cloud compile capabilities to design, validate, and deploy projects faster.

LabVIEW system design software offers flexibility through FPGA programming, simplifies code reuse, and helps you program the way you think–graphically.

Learn more at ni.com/labview/fpga.

800 453 6202

NATIONAL INSTRUMENTS

MathWorks®
Accelerating the pace of engineering and science

Find it at
mathworks.com/accelerate
datasheet
video example
trial request

GENERATE HDL CODE AUTOMATICALLY

from

MATLAB
and
Simulink

HDL CODER™ automatically converts Simulink models and MATLAB algorithms directly into Verilog and VHDL code for FPGAs or ASIC designs. The code is bit-true, cycle-accurate and synthesizable.

MATLAB® & SIMULINK®