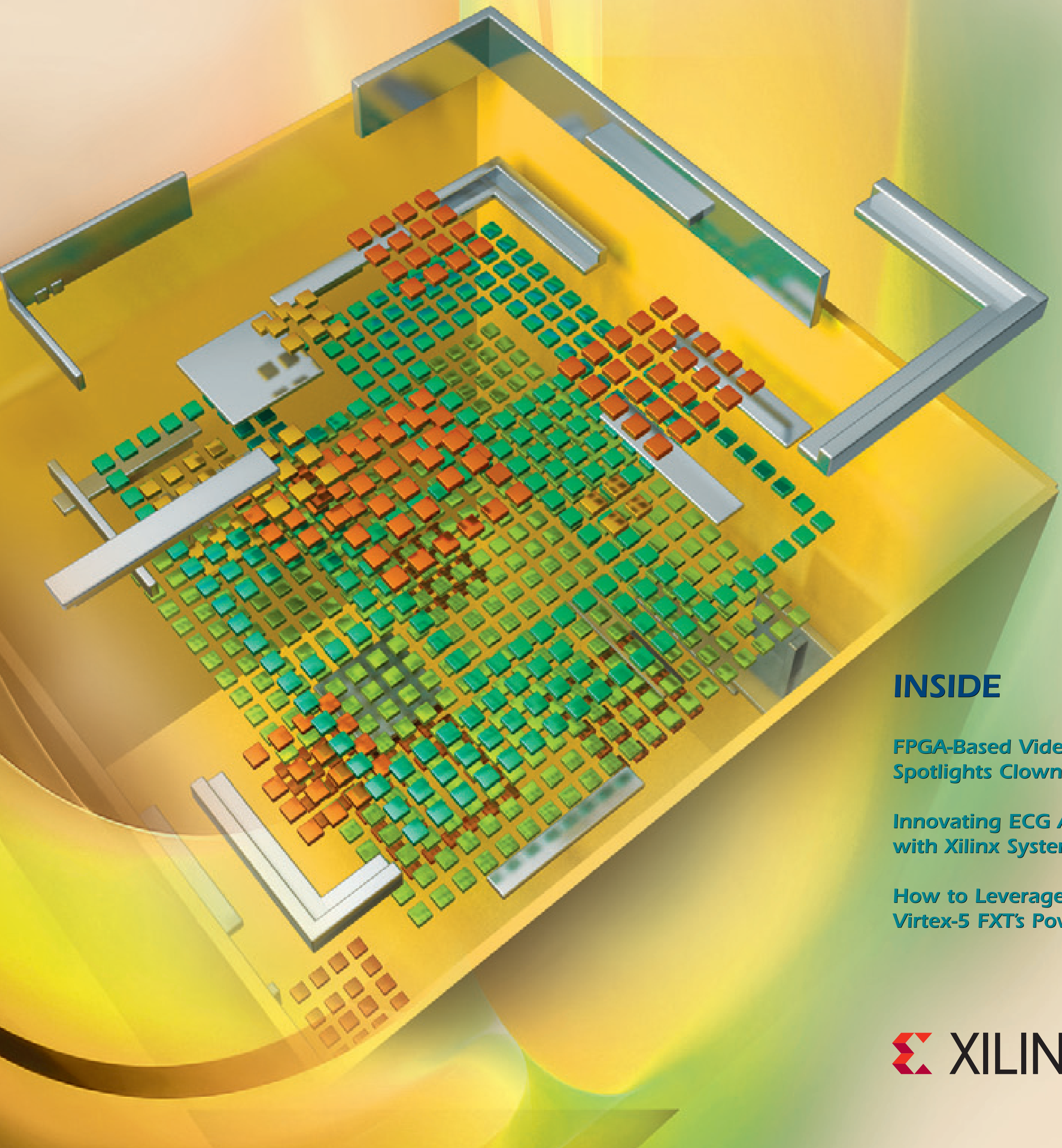
Issue 5
April 2009

Embedded magazine

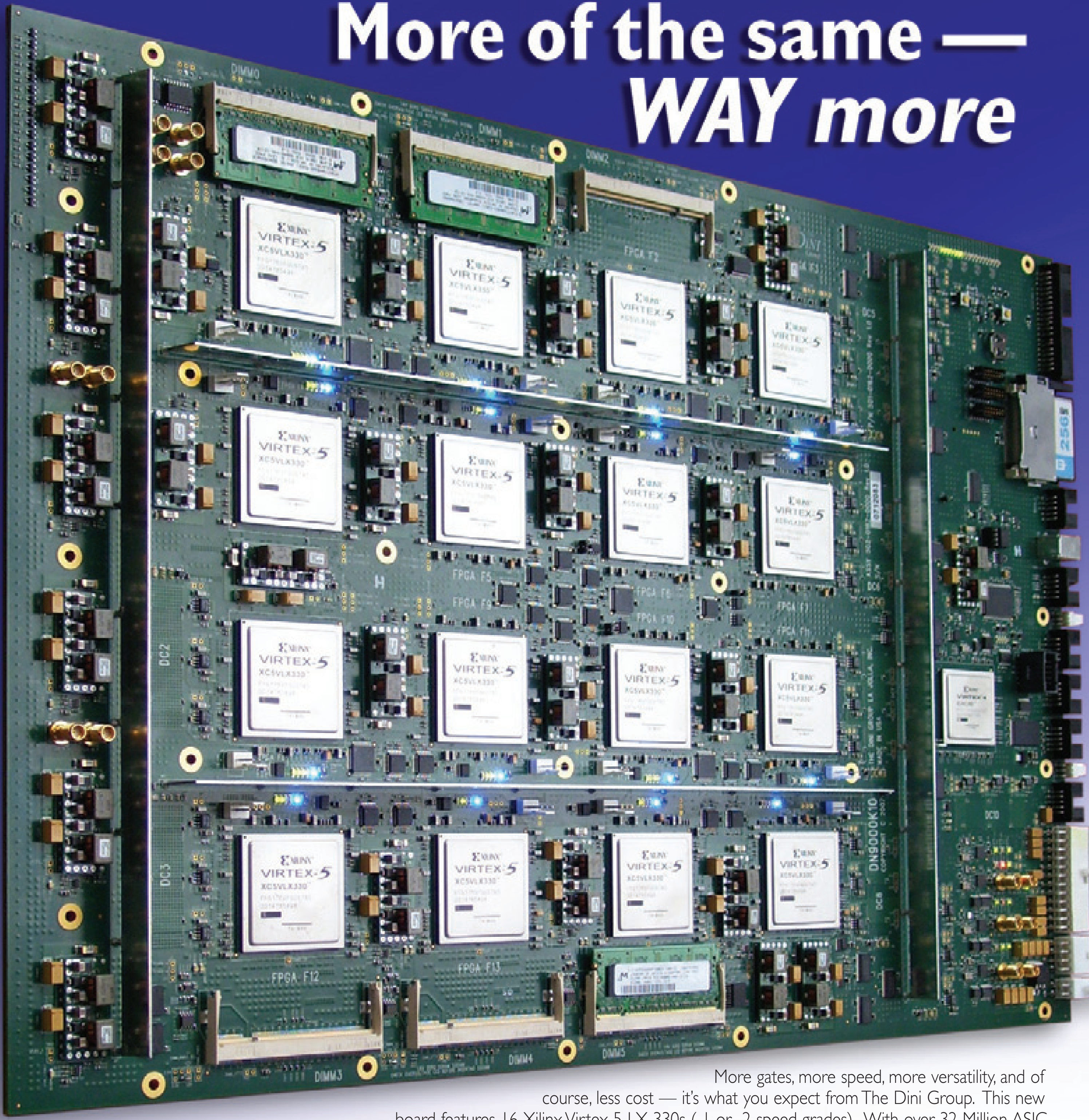
EMBEDDED SOLUTIONS FOR PROGRAMMABLE LOGIC DESIGNS



INSIDE

XILINX®

More of the same — *WAY more*



More gates, more speed, more versatility, and of course, less cost — it's what you expect from The Dini Group. This new board features 16 Xilinx Virtex-5 LX 330s (-1 or -2 speed grades). With over 32 Million ASIC gates (not counting memories or multipliers) the DN9000K10 is the biggest, fastest, ASIC prototyping platform in production.

User-friendly features include:

- 9 clock networks, balanced and distributed to all FPGAs
- 6 DDR2 SODIMM modules with options for FLASH, SSRAM, QDR SSRAM, Mictor(s), DDR3, RLDRAM, and other memories
- USB and multiple RS 232 ports for user interface
- 1500 I/O pins for the most demanding expansion requirements

Software for board operation includes reference designs to get you up and running quickly. The board is available "off-the-shelf" with lead times of 2-3 weeks. For more gates and more speed, call The Dini Group and get your product to market faster.

The DiNI Group

www.dinigroup.com • 1010 Pearl Street, Suite 6 • La Jolla, CA 92037 • (858) 454-3419 • e-mail: sales@dinigroup.com

GET PUBLISHED

WOULD YOU LIKE TO WRITE FOR XCELL PUBLICATIONS? IT'S EASIER THAN YOU THINK.

Interested in adding "published author" to your resume and achieving a greater level of credibility and recognition in your peer community? Consider submitting an article for global publication in the highly respected, award-winning *Xcell Journal*.

The *Xcell* team regularly guides new and experienced authors from idea development to published article with an editorial process that includes planning, copy editing, graphics development, and page layout. Our author guidelines and article template provide ample direction to keep you on track and focused on how best to present your chosen topic, be it a new product, research breakthrough, or inventive solution to a common design challenge.

Our design staff can even help you turn artistic concepts into effective graphics or redraw graphics that need a professional polish.

We ensure the highest standards of technical accuracy by communicating with you throughout the editorial process and allowing you to review your article to make any necessary tweaks.

Submit final draft articles for publication in our Web-based *Xcell* Online or our digital and print *Xcell Journal*. We will assign an editor and a graphics artist to work with you to make your papers clear, professional, and effective.

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

Mike Santarini
Publisher, Xcell Publications
xcell@xilinx.com

Xcell PUBLICATIONS

www.xilinx.com/xcell/

XILINX®

Embedded magazine

PUBLISHER	Mike Santarini mike.santarini@xilinx.com 408-626-5981
EDITOR	Jacqueline Damian
ART DIRECTOR	Scott Blair
DESIGN/PRODUCTION	Teie, Gelwicks & Associates 1-800-493-5551
ADVERTISING SALES	Dan Teie 1-800-493-5551 xcelladsales@aol.com
INTERNATIONAL	Melissa Zhang, Asia Pacific melissa.zhang@xilinx.com
	Christelle Moraga, Europe/Middle East/Africa christelle.moraga@xilinx.com
	Yumi Homura, Japan yumi.homura@xilinx.com
SUBSCRIPTIONS	All Inquiries www.xcellpublications.com
REPRINT ORDERS	1-800-493-5551

Xcell PUBLICATIONS

www.xilinx.com/xcell/

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

Welcome to this Embedded Systems special supplement of *Xcell Journal.*

Every year over the last eight or so years, Xilinx has seen a steady increase in the number of embedded-software designers and DSP algorithm developers using our FPGAs to create new innovations. This rise has its genesis in a few significant factors. First of all, Xilinx has been and continues to be committed to leveraging the latest silicon process technologies and their doubling of transistor counts, in keeping with Moore's Law. The wealth of transistors has allowed us to add advanced processors and ever more DSP slices into our FPGAs, along with programmable logic cells.

The doubling also means that customers can add soft processors like our MicroBlaze™ to their Xilinx designs, and combine them with a growing number of functions that otherwise would have occupied separate chips or other sections of the PCB. This integration reduces the overall bill of materials while delivering a leap in overall product performance, a system power savings and a reduction in the overall physical size of the end products (in turn saving more money).

The next reason is complexity. Today, many systems you want to implement simply outpace the complexity of the devices you previously targeted or the silicon you can afford to design yourself. For a growing number of these advanced systems, a single DSP just won't cut it, a series of DSPs is too cumbersome and the price points and time-to-market risks can't justify an ASIC or ASSP. FPGAs simply are the best choice. As Vin Ratford, our senior vice president of worldwide marketing, puts it, "Once FPGAs were in embedded systems; today they are embedded systems."

All this said, today, if you are an embedded-systems programmer or a DSP algorithm developer who isn't familiar with HDLs, programming an FPGA can seem intimidating. However, over the last eight years Xilinx has steadily acquired a better understanding of the methodologies, software and needs of embedded-software developers and DSP algorithm developers. Through software company acquisitions, internal software and IP development, and partnerships with third-party vendors, we have made great strides in making it easy for FPGA newbies and those unfamiliar with HDLs to create advanced embedded systems with our devices. Mind you, these flows still have lots of room for improvement, but today, embedded-software designers and DSP algorithm developers are making remarkable embedded systems with our FPGAs.

To show you what I mean, in this issue we present three feature articles that demonstrate how your colleagues are using Xilinx FPGAs to continually rewrite what really is the state of the art in embedded-system design.

Our first article, from Xilinx partner Impulse Accelerated Technologies, describes how Impulse, Xilinx and other partners are making it much easier for embedded-software engineers to leverage an emerging class of tools, commonly called ESL tools, to use the Xilinx MicroBlaze soft processor and program their designs into Xilinx FPGAs with minimal understanding of HDLs. In fact, Impulse CEO David Pellerin tells how in 20 hours the company created a demo for an advanced video analytics system that identifies a subject—in this case a clown fish—and tracks it as it moves to different areas of the screen.

Our second article, which first appeared in issue 65 of *Xcell Journal*, is a fascinating piece in which a grad student at the University of Manoa, Hawaii, and his advising professor describe how they implemented an advanced algorithm in a Xilinx FPGA to research next-generation electrocardiogram (ECG) systems. While neither is an HDL expert, they were able to use the Xilinx System Generator tool to successfully implement their algorithm for what potentially could be a life-saving system—literally.

Our third story is derived from an article that first appeared in *Xcell* issue 66. In this fascinating how-to piece, a design team from Missing Link Electronics describes how easy it is to use the APU inside the PowerPC® 440 processor in the Virtex®-5 FXT devices to implement in FPGAs advanced programming once thought the domain of standalone processors or state-of-the-art ASSPs only.

These are just a few of the tales demonstrating how rapidly FPGA technology is advancing and how a growing number of you are leveraging these versatile devices to create next-generation innovations. In fact, Xilinx is stepping up development efforts even more to help you work faster and more intuitively. In February, we announced our Virtex-6 and Spartan-6 Targeted Design Platforms.

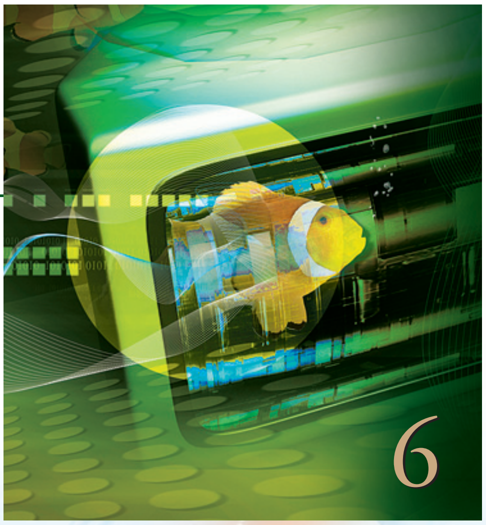
Targeted design platforms are not just state-of-the-art FPGA silicon; they also include the tools, IP, reference designs and methodologies you need to quickly create innovations in your targeted application area. We are adding layers of automation, IP socketization and reference designs for multiple applications, and tailoring tool flows to best suit specific designer skill sets. You'll hear more about our targeted design platforms over the coming months and years.

It's amazing to see what you have created so far. And as targeted design platforms mature, I can't wait to see what you'll come up with next. I encourage you to share your experience and wisdom with your colleagues and fellow *Xcell Journal* readers to further spur creative ideas and future innovations. Feel free to contact me if you have a contribution or even if you just want to chat.

Mike Santarini
Publisher

mike.santarini@xilinx.com
(408) 626-5981

XCELLENCE IN EMBEDDED SYSTEMS DESIGN



Send in the Clown Fish: Implementing Video Analysis in Xilinx FPGAs

High-level design methods combine with Xilinx Video Starter Kit to enable rapid prototyping of FPGA-based object-recognition system.

by David Pellerin
CEO
Impulse Accelerated Technologies
david.pellerin@ImpulseAccelerated.com

As more electronic devices in an increasing number of application areas employ video cameras, system designers are moving to the next evolutionary step in video system technology by adding intelligence or analysis capabilities to help identify objects and people. Advanced production machinery in a factory, for example, may use multiple video analysis systems to monitor machine parts and instantly identify failing components. In such applications, video systems may monitor materials as they move through an assembly line, identifying those that don't meet standards. Surveillance systems may also use advanced video analysis to tag suspicious objects or persons, tracking their movements in concert with a network of other cameras.

Companies and organizations are deploying the first generation of these video analysis systems today in inspection systems, manned or unmanned vehicles, video monitoring devices and automotive safety systems. Designers of these first-generation systems typically implement video-processing algorithms in software using DSP devices, microprocessors or multicore processors. But as designers move to next-generation video applications that are much more fluid and intelligent, they are finding that DSP and standard processors can't accommodate new requirements for video resolutions, frame rates and algorithm complexity.

Digital signal processors certainly do have benefits for video applications, including software programmability using the C language, relatively high clock rates and optimized libraries that allow for quick development and testing. But DSPs are limited in the number of instructions they can perform in parallel. They also have a fixed number of multiply/accumulators, fixed instruction word sizes and limited I/O.

FPGAs, on the other hand, benefit from an arbitrary number of data paths and operations, up to the limit of the device capacity. Larger FPGA devices are capable of performing hundreds or even thousands of multiply operations simultaneously, on data of varying widths.

Because of their advantages in compute-intensive video processing, FPGAs—or combinations of FPGAs and DSPs—have become the favored choice for designers of the most advanced video systems.

At one time, FPGAs were intimidating for algorithm developers untrained in hardware design methods. Programming an FPGA to implement a complex video algorithm required that designers have hardware design skills and a grasp of hardware description languages. But over the last several years, Xilinx and a number of third-party software providers, including Impulse, have created higher-level flows that allow algorithm developers to use FPGAs for even the most complex designs, without requiring substantial hardware design skills.

Thanks to new tools and methods that make it possible to easily use FPGAs for advanced video analysis, Impulse Accelerated Technologies designed a moderately complex, high-definition video-processing application in a matter of days. We used a combination of higher-level design tools, Xilinx video development hardware and Xilinx video reference design examples. This demonstration project, which we called Find the Clown Fish, serves as a model for other, more-complex projects requiring fast bring-up with minimal design risk.

Advanced Video Analysis Requires FPGAs

Complex video-processing applications are often purpose-built. For example, a machine vision technology used in an automated inspection system may require a very specific sequence of video filtering and control logic to identify objects moving down an assembly line. Such a system might determine whether a specific item in a production process, be it a potato chip or a silicon wafer, should be rejected and diverted off the line. In an automotive application, it might be necessary to identify and analyze specific types of objects–road signs, for example–in near real-time.

We designed a moderately complex HD video-processing application in days using higher-level design tools, Xilinx development hardware and reference designs.

In the past there have been significant barriers for software programmers tasked with moving such algorithms out of traditional processors and into FPGAs. Hardware design methods and languages are very different from those used in software. The level of abstraction for FPGAs, using hardware description languages, is much lower than in software design. FPGA development tools have matured in recent years, however, providing software algorithm designers with more-productive methods of prototyping, deploying and maintaining complex FPGA-based algorithms.

Video system designers can develop and deploy their applications in FPGAs by combining multiple high-level methods of design, using a range of available tools and intellectual-property (IP) blocks. Since no one tool or design method is ideal for all aspects of a complex video application, it is best to select the most productive methods for creating different parts of a given video-processing product.

The development hardware is also an important consideration. Well-tested hardware platforms and reference designs will greatly accelerate the design and debugging of complex FPGA-based systems.

Three categories of tools in particular have helped to speed software-to-hardware conversion. Two of them are based on popular software programming languages and environments, while the third makes it easier to manage the increased complexity of FPGA-based systems.

Library-Based Tools Speed Development

MATLAB® and Simulink®, produced by the Mathworks, are popular tools for the development of complex algorithms in many domains. For DSP and video-processing algorithms in particular, they provide a robust set of library elements that designers can arrange and interconnect to form a simulation model.

Tools such as Xilinx System Generator™ extend this capability. System Generator allows designers to take a subset of these elements and use the tool to automatically convert the elements into efficient FPGA hardware.

For example, the developer of a machine vision application could use Simulink in combination with System Generator to quickly design and simulate a pipeline of predefined filters, then deploy the resulting algorithm into the FPGA along with other components for I/O and control.

Xilinx System Generator is a highly productive method for creating such applications, because it includes a wide variety of preoptimized components for such things as FIR filters and two-dimensional kernel convolutions. There are limits, however, to what designers can accomplish using libraries of predefined and only nominally configurable filters.

C-to-FPGA Accelerates Software Conversion

For increased design flexibility, designers can also use C-to-hardware tools such as Impulse CoDeveloper to describe, debug and deploy filters of almost unlimited complexity. Such design methods are particularly useful for filtering applications and algorithms that don't fit into existing, pre-defined blocks. These applications include optical-flow algorithms for face recognition or object detection, inspection systems and image-classification algorithms such as smart vector machine, among others.

C-to-FPGA methods are particularly appealing for software algorithm developers, who are accustomed to using source-level debuggers and other C-language tools for rapid, iterative design of complex systems. Designers can use C not only to express the functionality of the application itself, but also to create a simulated application, for example by tapping into open-source user interface components and widely available image-processing software libraries.

A secondary benefit is that C-language methods enable designers to employ embedded processors within the FPGA itself for iterative hardware/software partitioning and in-system debugging. When designers introduce embedded processors into the application, they can use the C language for both the software running on the processor as well as to describe processor-attached hardware accelerators.

Platform Studio Enables System Integration

The Xilinx ISE®, or Integrated Software Environment, includes Platform Studio, a tool that allows users to assemble and interconnect Xilinx-provided, third-party and custom IP to form a complete system on their target FPGA device. Xilinx and its development-board partners provide board support packages that extend Platform Studio and greatly simplify the creation of complex systems. Reference designs for specific boards such as the Xilinx Video Starter Kit also speed development.

The Platform Studio integrated development environment contains a wide variety of embedded programming tools, intellectual-property cores, software libraries, wizards and design generators to enable fast creation and bring-up of custom FPGA-based embedded platforms. These tools represent a unified embedded development environment supporting PowerPC® hard-processor cores and MicroBlaze™ soft-processor applications.

Finding the Clown Fish in HD Video Streams

To demonstrate how to use these tools and methods effectively in an advanced video application, we decided to create an image-filtering design with a highly constrained schedule, of just two weeks, for showing at the Consumer Electronics Show in Las Vegas. The requirements of our demonstration were somewhat flexible, but we wanted it to perform a moderately complex image analysis, such as object detection, and support multiple resolutions up to and including 720p and 1080i. The system should process either DVI or HDMI source inputs, and support pixel-rate processing of video data at 60 frames per second.

After considering more-common algorithms such as real-time edge detection and filtering, we decided to try something a bit more eye-catching and fun. We decided to "Find the Clown Fish."

More specifically, what we set out to do was monitor a video stream and look for particular patterns of orange, black and gray–the distinctive stripes of a clown fish–and then create a spotlight effect that would follow the fish around as it moved through the scene, thereby emulating the type of object tracking that a machine vision system might perform. The goal was to have a demonstration that would work well with the source video, a clip featuring a clown fish, or perhaps even with a live camera aimed at a fish tank.

For expediency, we decided to start with an existing DVI pass-through filter reference design provided by Xilinx with the Video Starter Kit. This reference design includes a few relatively simple filters including gamma in, gamma out and a software-configurable 2-D FIR filter. An excellent starting point for any streaming-video application, this reference design also demonstrates the use of Xilinx tools including Platform Studio and System Generator.

Within a few hours of receiving the Video Starter Kit from Xilinx, we had a baseline Platform Studio project, the DVI pass-through example, built and verified through the complete tool flow. This reference design served to verify that we had good, reliable setup for video input and display, in this case a laptop computer with a DVI output interface and an HD-compatible video monitor. We then began coding additional video-filtering components using C and the streaming functions provided with our Impulse CoDeveloper and Impulse C compiler.

As seen in the block diagram of the complete video-processing system (Figure 1), a MicroBlaze processor serves as an embedded controller. We inserted the Impulse C detection filter into the DVI video data stream using video signal bus wrappers automatically generated by the Impulse tools.

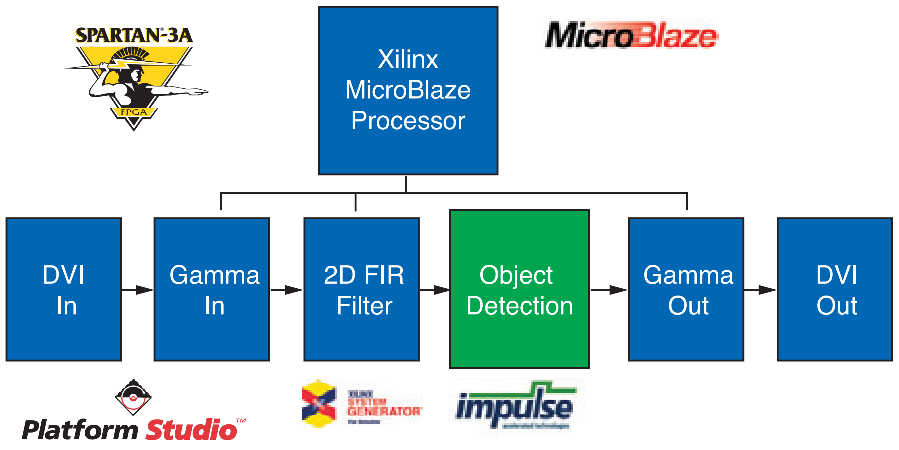


Figure 1 – Impulse Accelerated Technologies implemented this complete video-filtering and object-detection design in a single Spartan-3A FPGA device.

Using C for prototyping dramatically sped up the development process. Over the course of a few days, we employed and debugged many image-filtering techniques involving a variety of algorithm-partitioning strategies. For software debugging purposes, we used sequences of still images (scenes from a video that has an animated clown fish) as test inputs. We interspersed compile and debug sessions using Microsoft Visual Studio with occasional runs through the C-to-hardware compiler in order to evaluate the likely resource usage and determine the pipeline throughput of the various filtering strategies. Only on rare occasions did we synthesize the resulting RTL and route the design to create hardware. The ability to work at a higher level while iteratively improving the detection algorithm dramatically accelerated the design process.

In fact, we performed all of the algorithm debugging using software-based methods, either through examination of the generated BMP-format test image files or using the Visual Studio source-level debugger. At no point did we use an HDL simulator to validate the generated HDL. We used the Xilinx ChipScope™ debugger at one point to observe the video stream inputs and determine the correct handling of vertical and horizontal sync, but otherwise we found no need to perform hardware-level debugging.

Optimizing for Pipeline Performance

A critical aspect of this application, and others like it, is the need for the algorithm to operate on the streaming-video data at pixel rate, meaning the design must process and generate pixels as quickly as they arrive in the input video stream. For our demonstration example, the required steps and computations for each pixel included:

- Unpacking the pixels to obtain the R, G and B values as well as the vertical and horizontal sync and data-enable signals.
- Doing 5 x 5 prefiltering to perform smoothing and other operations.
- Storing and shifting incoming pixel values for subsequent pattern recognition.
- Performing a series of comparisons of saved pixels against specific ranges

Tips, Techniques and Tricks for Programmers

If you are a C programmer experienced with traditional processors, you will need to learn a few new concepts and employ certain coding techniques to obtain the best results when targeting an FPGA.

First, you should use fixed-width, reduced-size integers when possible. For example, when counting scan lines in a frame, there is no benefit to using a standard 16-bit or 32-bit C-language integer data type. Instead, select a data type with just enough bits to represent the maximum value for the counter. C-to-FPGA tools include additional, nonstandard data types for exactly this purpose, as shown below when calculating how far to move the spotlight and adjust its size:

```
co_uint12 diffx;  // Offset of the current pixel
co_uint12 diffy;  // Offset of the current pixel
co_int24 diffx_sq, diffy_sq, diffsum;
. . .
// Calculate if this pixel is in the spotlight
diffx = ABS(x_position - spotlight_x);
diffy = ABS(y_position - spotlight_y);
              diffx_sq = diffx * diffx;
              diffy_sq = diffy * diffy;
              diffsum = diffx_sq + diffy_sq;
if (de_out) { // de_out indicates visible pixels
     if (spotlight_on != 0 && x_position != 0
              && diffsum < spotlight_size) {
          r_out = r_in;   // Pass through
          g_out = g_in;
          b_out = b_in;
     }
     else {
          r_out = (r_in >> 1);  // Dim
          g_out = (g_in >> 1);
          b_out = (b_in >> 1);
     }
}
```

You should also consider refactoring the use of variables and arrays to allow efficient pipelining. Modern C-to-FPGA compilers can schedule parallel operations efficiently, and can take advantage of such FPGA features as dual-port RAM. But there are many cases in which the actual requirements of a given algorithm–the range of input values expected, or input combinations that are known to be impossible–may allow for alternative coding methods that can deliver higher performance.

Also, write or refactor your C code with parallel operations in mind. For example, a common optimization technique in C programming is to reduce the total number of calculations by placing certain operations within control statements, such as if-then-else. In an FPGA, however, it may be more optimal to precalculate values prior to such a control statement, because the FPGA can perform those precalculations in parallel with other statements.

Certain operations–such as very wide or complex comparisons using relational operators–may be easy to code in C using macros, but may result in more logic than expected due to data type promotion. Casts and other coding methods can help you reduce the size of the generated logic and allow for faster clock speeds.

Software programmers can quickly learn these coding techniques, and others like them, by using iterative methods and by paying attention to compiler messages. We employed all of the above techniques when creating our clown fish video analysis demonstration application. – *David Pellerin* and patterns of colors to identify a clown fish stripe.

- Calculating the current and new locations of the spotlight, and moving the spotlight location toward the identified target at a visible rate.
- Calculating the diameter and shape of the spotlight, using simple geometry and a frame counter to create a smooth and steady spotlight effect.
- Filtering pixels by increasing or decreasing the color intensity according to whether a given pixel is within the spotlight radius.

A sample image of the highlighted target (the clown fish) is shown in Figure 2.

To meet the pixel-rate requirement, the design must perform all of these operations for each pixel in the HD video stream at a rate of one pixel for every clock cycle. When processing 720p video at 60 frames/s, this means that the above functions, representing approximately 100 lines of C code in a single automatically pipelined loop, must be performed more than 55 million times each second. We would combine this detection filter with the other filtering components in the system to create the complete application. If we sum up all of the fundamental operations required by all the components (including the two gamma filters, the 2-D FIR filter and the detection and spotlight filter), the design must be able to perform close to 2 billion integer operations per second. This sounds like a lot, but in fact, real-world video-processing algorithms may require many times that much real-time computing.

Creating an efficient pipelined implementation of a complex algorithm is never trivial, but using preoptimized components in combination with C-to-hardware programming has obvious productivity benefits over lower-level, HDL-based methods. Because the algorithm remained expressed in a cycle-independent manner, it took only a small amount of effort to repipeline and reoptimize it after making fundamental changes to the code, for example after adding an entirely new set of computations to handle frame-to-frame behaviors such as expanding and shrinking the spotlight when the clown fish swam in and out of the scenes. Because the C compiler is capable of automatically scheduling operations within a pipelined loop, C programmers are able to focus their energy on higher-, system-level design decisions such as whether to create multiple parallel FPGA processes to solve a complex problem.

We brought up our demo project and got it ready to go with less than 20 hours of actual C coding and debugging. We later spent additional time optimizing the algorithm and adding features, such as a spotlight fade-out effect and support for arbitrary video resolutions.

The complete application includes multiple pipelined filter modules as well as an embedded MicroBlaze processor that we can use to configure parts of the video processing. For example, we can control these filter modules at run-time to modify the brightness levels or perform sharpening or smoothing operations prior to the object-detection and highlighting filter. This is an excellent example of a hybrid hardware/software application that you can implement in a single Xilinx Spartan® FPGA device.

Overall, we created our Find the Clown Fish project in under two weeks, using a combination of available development tools and methods. The use of C language for describing and implementing the detection algorithm greatly reduced the time it took to design and debug, allowing us to explore many alternative approaches to the algorithm, simulate using C-language test fixtures and ultimately try them out in real hardware using the Xilinx Video Starter Kit.

While this example may be only a demonstration, it does suggest a wide variety of other, more-complex video-processing applications that you could develop using the Video Starter Kit. It also shows that software-oriented methods of design can enable significantly faster deployment of machine vision systems.

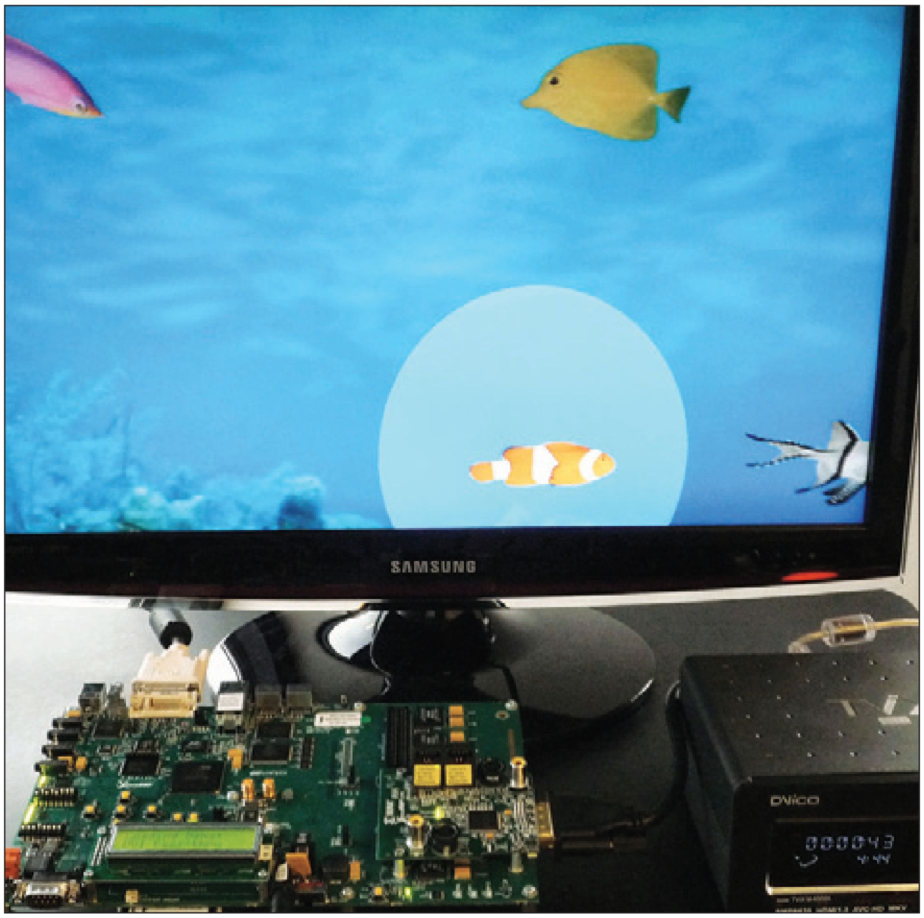
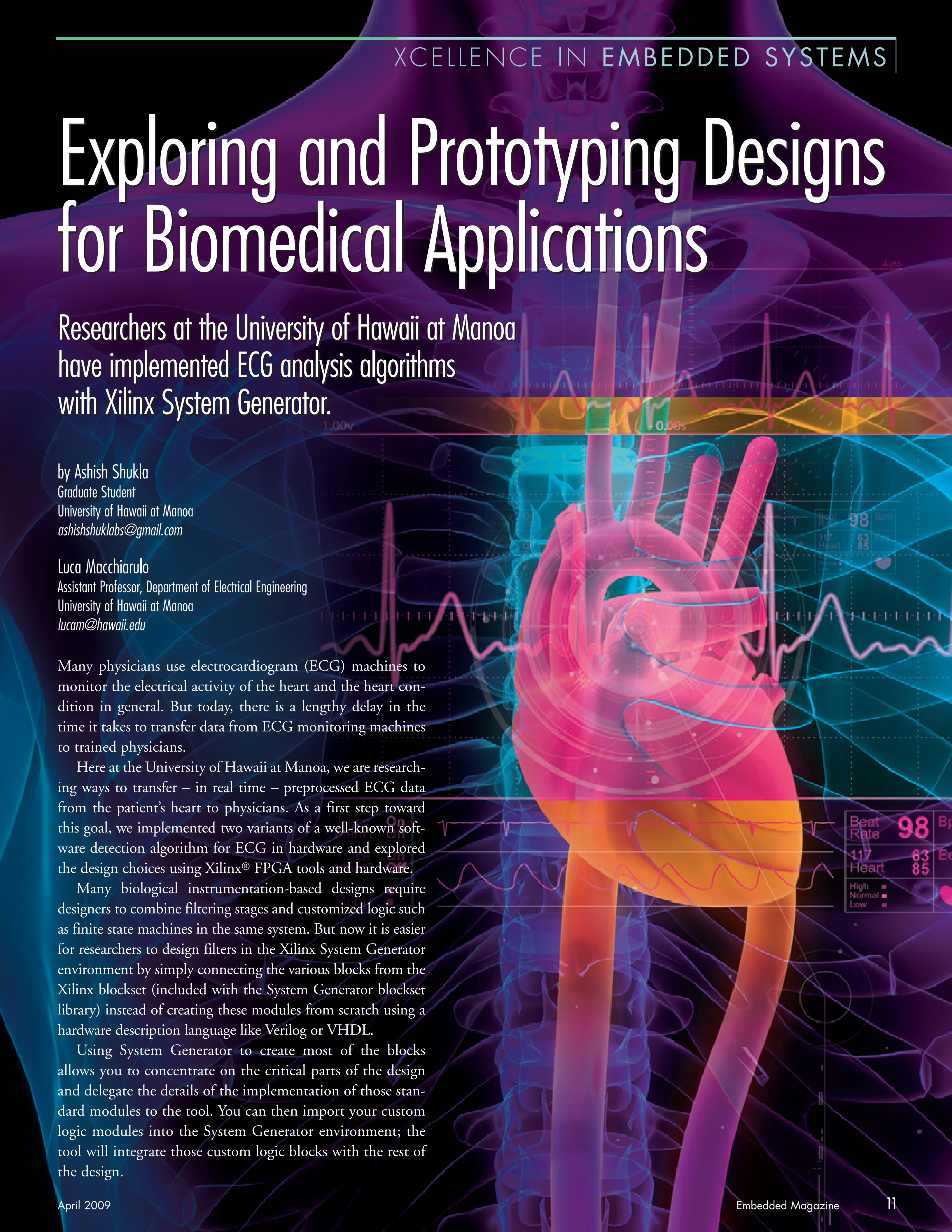


Figure 2 – Test image shows the spotlight effect for a detected clown fish.

Exploring and Prototyping Designs for Biomedical Applications



Researchers at the University of Hawaii at Manoa have implemented ECG analysis algorithms with Xilinx System Generator.

by Ashish Shukla
Graduate Student
University of Hawaii at Manoa
ashishshuklabs@gmail.com

Luca Macchiarulo
Assistant Professor, Department of Electrical Engineering
University of Hawaii at Manoa
lucam@hawaii.edu

Many physicians use electrocardiogram (ECG) machines to monitor the electrical activity of the heart and the heart condition in general. But today, there is a lengthy delay in the time it takes to transfer data from ECG monitoring machines to trained physicians.

Here at the University of Hawaii at Manoa, we are researching ways to transfer – in real time – preprocessed ECG data from the patient's heart to physicians. As a first step toward this goal, we implemented two variants of a well-known software detection algorithm for ECG in hardware and explored the design choices using Xilinx® FPGA tools and hardware.

Many biological instrumentation-based designs require designers to combine filtering stages and customized logic such as finite state machines in the same system. But now it is easier for researchers to design filters in the Xilinx System Generator environment by simply connecting the various blocks from the Xilinx blockset (included with the System Generator blockset library) instead of creating these modules from scratch using a hardware description language like Verilog or VHDL.

Using System Generator to create most of the blocks allows you to concentrate on the critical parts of the design and delegate the details of the implementation of those standard modules to the tool. You can then import your custom logic modules into the System Generator environment; the tool will integrate those custom logic blocks with the rest of the design.

Because noise contamination is an inherent problem in ECG monitoring and we cannot completely remove it, we had to come up with a way to suppress noise contamination.

Automated ECG System Analysis

A heart's natural pacemaker, composed of special self-exciting cells, generates and propagates a polarization and depolarization electrical signal that regulates the proper contraction of the heart muscle. This electrical activity is recorded as an electrocardiogram by a machine called an electrocardiograph, which provides physicians with a wealth of information on heartbeats and the heart's condition in general [1].

The QRS complex – a wave structure that corresponds to the depolarization of ventricles and has a spiked shape in the ECG – is often the most telling waveform found in an ECG signal. The morphology, duration and amplitude of the QRS complex in an ECG signal provides significant information to physicians diagnosing various arrhythmias and other cardiac ailments.

Health professionals can best judge the state of a patient's heart by monitoring the heart's activity (and the QRS complex) during stress or physical activity. Therefore, health professionals often connect ECG monitors to their patients for many hours (generally 24 hours at a time).

Currently, most health professionals use Holter System monitors for this purpose. But Holter System monitors have an inherent battery power and processing power constraint that limits their functionality to data acquisition systems.

A Holter System's heart monitors record the patient's ECG data to flash memory or to a tape attached to the system. A technician then removes the drive from the Holter System and sends it to the lab, where technicians analyze the data. Once the lab technicians complete their analysis, they send the ECG report to the physician. Of course, this means that there is a delay between hooking up the monitor to the patient and getting the ECG data to the patient's physician, thus delaying treatment, possibly with tragic consequences.

Here at the University of Hawaii at Manoa, we are researching how to create ECG analysis systems that will reduce the amount of time it takes to transfer data from the patient's heart to the physician by carrying out the analysis in real time. We hope to achieve this by integrating the ECG analysis system into portable ECG monitors, which physicians can use directly and thus eliminate the technician translation/analysis step.

To aid us in this effort, we have implemented real-time ECG analysis on a Xilinx Spartan™ XC3S500 device in the Spartan-3E Starter Kit, which allows the system to perform this analysis much faster than previous software-based methods. FPGAs offer many advantages for this and other applications because they support rapid prototyping, are less expensive than ASICs at low volume and are quickly reprogrammable.

Furthermore, their fast and efficient testing option, combined with System Generator software, makes them a great choice for algorithm exploration as well as hardware implementation and prototyping.

Implementation of the Algorithm in Hardware

In our research, we are implementing a well-known algorithm by Tompkins and Hamilton for QRS detection in hardware ([2], [3], [4], [5]). To help us with this effort, we took advantage of the fast prototyping features included in the Xilinx toolset.

In our system, the processing of the ECG signal begins with a number of filtering stages through which a signal must pass before the system accurately detects a QRS complex. We divided the design into two main stages. The first stage, or preprocessing stage, comprises four linear filters and one nonlinear filter. The second stage, or peak detection stage, identifies the peak signal in the QRS complex and applies decision rules to qualify a feature as a QRS complex.

Filtering Stage

The ECG recording is very sensitive to even the smallest body movement or noise, such as electrical muscle (myographic) signals and the system's own electrical power-line interference. Because noise contamination is an inherent problem in ECG monitoring and we cannot completely remove it, we had to come up with a way to suppress noise contamination. We achieved this by employing filtering during the preprocessing stage (as in [4], [5]).

To do this, we first pass an ECG signal through an infinite impulse response (IIR) low-pass filter, which suppresses the high-frequency noise in the signal. Then the signal passes through an IIR high-pass filter, which attenuates the P and T waves in the signal and suppresses the DC offset present in the signal.

The low-pass and high-pass filters together form a bandpass filter. It then feeds the output of the high-pass filter to a finite impulse response (FIR) derivative filter, which further emphasizes the QRS complex. This typically presents a more pronounced slope variation compared to the other signal features, making it somewhat easier for observers to identify [1].

The FIR derivative filter step also helps further reduce the noise content of the signal. After filtering, we employ a squaring stage that carries out nonlinear amplification of the QRS complex and makes all data points positive.

Finally, we use the output of the squaring stage as the input to a moving window integrator. Each sample output from this filter is an average of the previous 32 values. We implemented all these filtering stages directly in the Simulink® environment using the System Generator tool. Then we tried out two different approaches for the next stage, which is a moving window integrator.

The first approach involved the connection of a number of register, delay and adder blocks to implement the filter as a

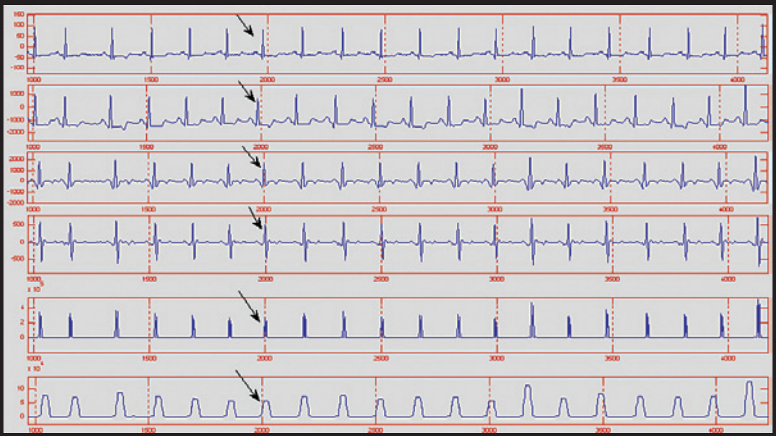


Figure 1 — Input ECG and filtering stages output, from top to bottom: original signal, low-pass output, high-pass output, derivative output, squaring stage output, moving window integrator output. The arrow indicates the position of the QRS complex.

direct Form I structure. With this approach, however, we ended up using 31 adders and an equally large number of delay or register blocks.

We investigated further and later came up with a more resource-efficient structure that uses the block RAM resources in the Spartan-3 FPGA. This final structure uses a small RAM of 32 40-bit words and just two adders and is also very efficient in timing. Figure 1 shows a typical ECG input, with results of the various filtering operations.

Peak Detection Stage

In the peak detection stage, we are trying to find the peak in the output of the moving window integrator. The peak detection process depends on the calculation of the threshold value, and we can locate a peak among the sample values that are greater than the threshold value. But the algorithm does not report a peak until a sample appears in the falling slope of the moving window integrator signal with a value less than half of the peak value.

We use this method because it reduces the number of false peak detections caused by noise. Once the algorithm locates the maximum point of the QRS signal in the moving window integrator output, we search for a peak in an appropriately delayed copy of the output of the high-pass filter and use it as a fiducial point for the position of the QRS complex.

Essentially, we are finding the peak of the QRS complex in the output of the bandpass signal instead of the original signal, because the original signal is highly contaminated by noise. We achieved this using a memory module, which records the previous 60 samples from the high-pass filter and sends a fixed interval of samples

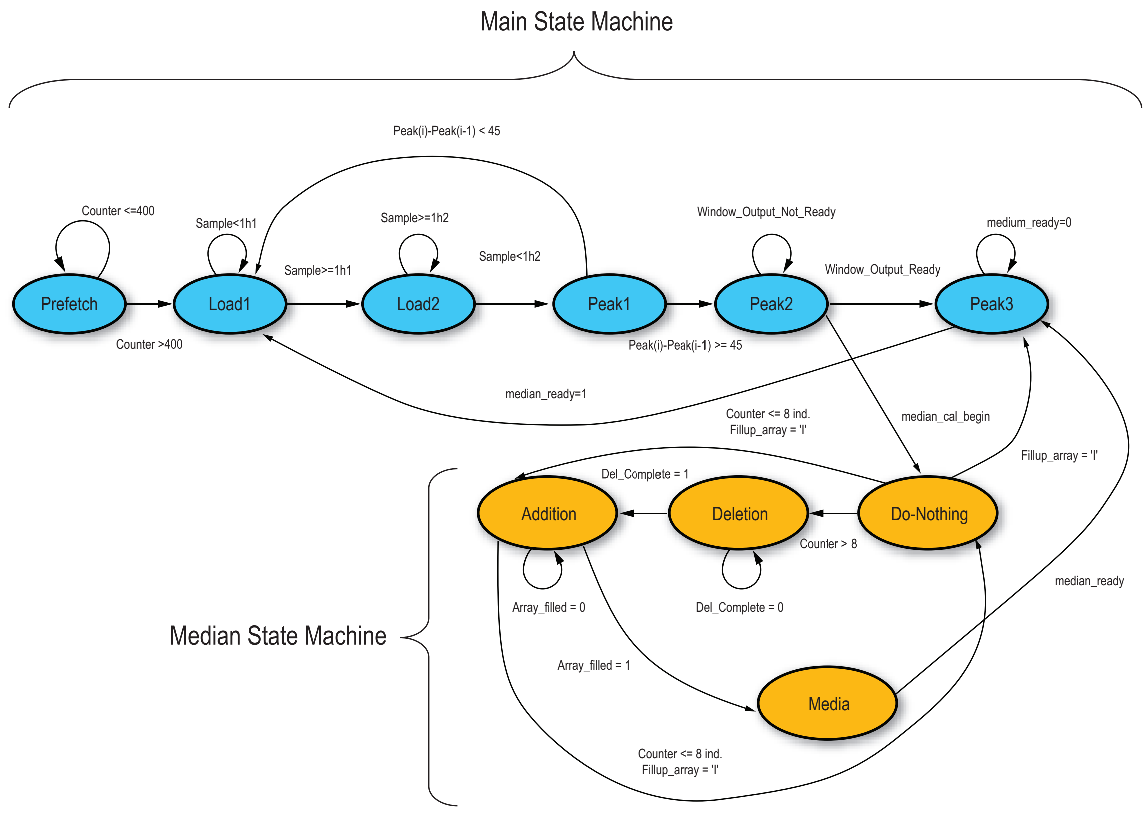


Figure 2 – State machine for the complete design

containing the QRS complex as an input to the window module. The window module finds the maximum QRS signal.

We based the threshold for the next peak detection process in the moving window integrator output on the median of the eight previously detected peaks.

To calculate the running median calculation, the system maintains an updated, sorted list of such peaks. The computation is supervised by a median calculation FSM (called the median state machine) communicating with a separate peak detection FSM (called the main state machine; see Figure 2). The main state machine finds the peak in the moving window output (as discussed); then the median state machine sets up the threshold value for the next detection.

We implemented the entire peak detection stage in VHDL. It contains the memory, window, main and median state machines and the median RAM modules. Later, we imported these VHDL modules into the System Generator environment as black boxes and simulated the design (Figure 3).

Programming and Testing the Hardware Implementation

We programmed the FPGA through the JTAG port. To test the design, we used System Generator's JTAG co-simulation feature, which allowed us to pass data from the Simulink environment through the USB cable to the board (the Xilinx Spartan-3E Starter Kit) containing the downloaded design.

The design implemented in the FPGA processes the data and sends the output back to the MATLAB® environment so that we can compare it to the simulation output. This feature was very convenient and helped us establish the accuracy of the design implementation in hardware. It also allowed us to run tests on complete sets of benchmark ECG data (from *www.physionet.org*) in a fraction of the time we previously needed to test the software version of the algorithm.

Overall, we found System Generator to be very useful for implementing designs for biomedical instrumentation applications. The tool provides easy filter design options and simplifies design implementation by accepting imported custom modules as black boxes. It also provides a fast and efficient way to test a design on hardware, with easy interfacing options between the board and the user's workstation.

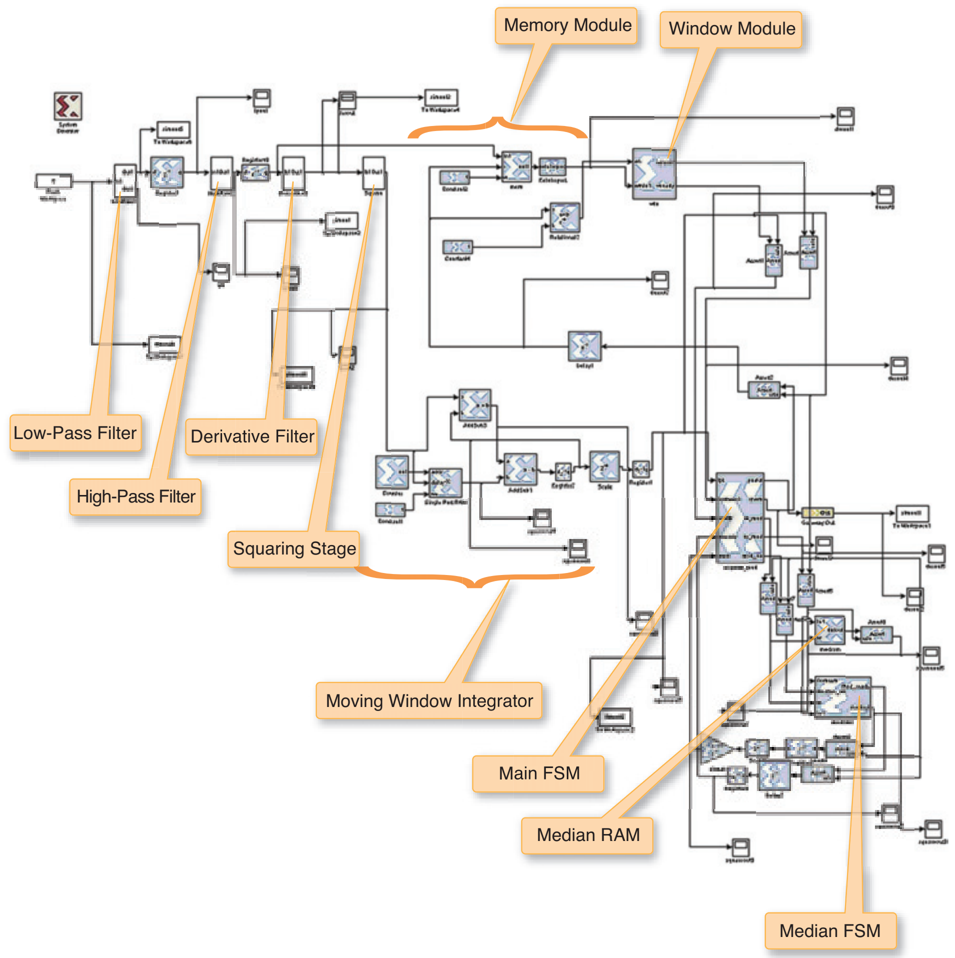


Figure 3 – Complete design in the System Generator environment

Citations

[1] B-U. Kohler, C. Hennig, and R. Orglmeister, "The principles of software QRS detection," *IEEE Engineering in Medicine and Biology Magazine*, vol. 21, no. 1, pp. 42-57, January 2002.

[2] A. Shukla and L. Macchiarulo, "FPGA based ECG Analysis System," *Proceedings of the Sixth IASTED International Conference on Biomedical Engineering*, Austria, pp. 68-72, February 2008.

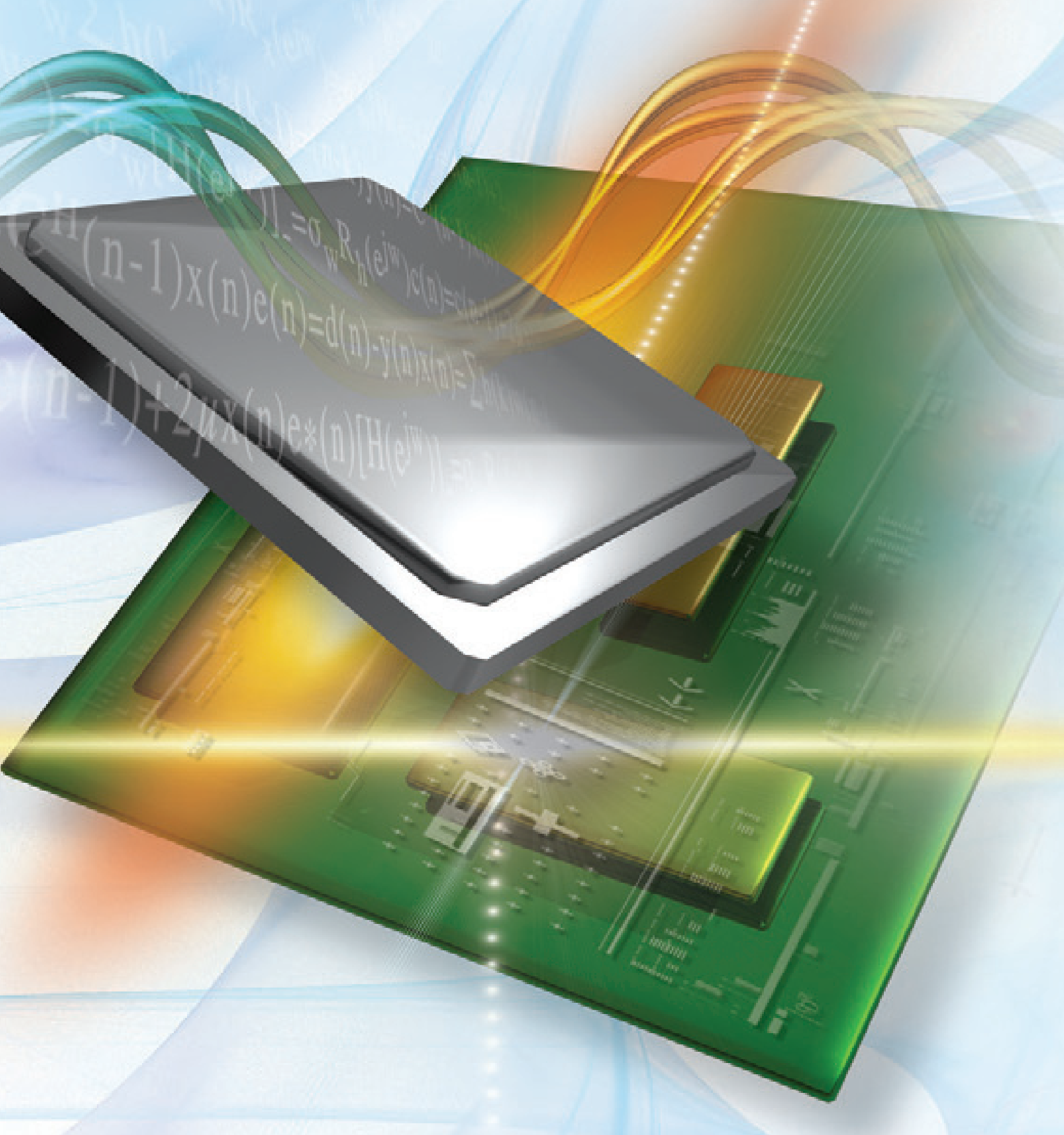
[3] A. Shukla, "Hardware Implementation of real-time ECG analysis algorithms," M.S. thesis, University of Hawaii at Manoa, Honolulu, Hawaii, 2008.

[4] J. Pan and W. Tompkins, "A real-time QRS detection algorithm," *IEEE Transactions on Biomedical Engineering*, vol. BME-32, no. 3, pp. 230-236, March 1983.

[5] P. Hamilton and W. Tompkins, "Quantitative Investigation of QRS Detection Rules Using MIT/BIH Arrhythmia Database," *IEEE Transactions on Biomedical Engineering*, vol. BME-33, no. 12, pp. 1157-1165, December 1986.

Extend the PowerPC Instruction Set for Complex-Number Arithmetic

Using the APU inside a Xilinx Virtex-5 FXT can ease the design of field-upgradable systems.



by Endric Schubert,
Felix Eckstein, Joachim Foerster,
Lorenz Kolb and Udo Kebschull
Missing Link Electronics
endric@missinglinkelectronics.com

Embedded processing systems face a serious technical challenge: how to achieve system upgradability over the product's lengthy life cycle? With rapidly changing standards, designers need a way to quickly modify their designs even after they have deployed their product in the field. But in most cases, it is not sufficient (or even possible) to update only software—many applications also require an increase in compute performance. Yet designing a system with "spare" compute power for later use is neither economical nor technically feasible, since many technology changes are simply not foreseeable.

One solution is to upgrade the compute platform together with the software in such a way that the upgraded system provides sufficient computation power for the additional software processing load. If you build a system with a Xilinx® Virtex®-5 FXT device, you can give your design additional computation power by adding special-purpose compute operations to the PowerPC® processor's Auxiliary Processing Unit (APU).

At Missing Link Electronics, a company working on reconfigurable platforms to link heterogeneously connected embedded systems, we believe the PowerPC processor APU inside the Xilinx Virtex-5 FXT devices is a little gem. It provides embedded-systems designers with the same optimization powers traditionally available only to the "big guys" who build their own custom ASSP devices. Given what's now available to Xilinx users, we think everyone should tap the APU to optimize their designs.

Our project involved extending the instruction set of the PowerPC processor to handle complex-number multiplications. Complex numbers, which are defined as a natural extension of real numbers by means of computation with an imaginary unit, are used in many engineering applications. In multimedia design, for example, complex-number multiplication is useful for the decoding of streaming-media data.

Basics of Extending Instruction Set via the APU

When designers want to optimize an embedded system, they typically do so by looking for ways to extend the instruction set of the microprocessor at the heart of their design. Traditionally, this is the best option when the complexity of the embedded system lies in the software portion of the design. You could also simply put new functionality in the design by adding dedicated hardware blocks.

However, you'll likely find that increasing the instructions holds some great advantages that complement hardware changes, but is somewhat easier for designers to implement. For example, by extending the instructions, you can optimize your design in finer granularity. Also, extending the instruction set typically does not interfere with memory access; thus, it has the potential to optimize the system's overall performance.

Even though individuals, companies and academic researchers have published papers on how to do it, extending an instruction set may seem like a black art to anyone new to this technique. But in reality, it isn't that complex. Let's examine how you can optimize your Virtex-5 FXT design by making some fairly simple additions to the PowerPC processor's instruction set via the APU interface.

In general, to extend the instruction set of an embedded microprocessor, you need to understand that you are modifying both the software and the hardware. First, you are adding hardware blocks to your system to perform specialized computations. These computations execute in parallel in the FPGA fabric rather than sequentially in software. In Xilinx-speak,

Step-by-Step Guide to Using the APU

Here we present detailed information on how the engineers at Missing Link Electronics generated the necessary files for our example design, and show how to use these files to reproduce the results on the Xilinx ML507 Evaluation Platform, which contains a Xilinx Virtex-5 XC5VFX70T device. We also show how to use this design as a starting point for your own APU-enhanced FPGA design.

Step 1: Build Your Coprocessor

Theoretically, you can build almost any coprocessor as long as it fits into your FPGA, but keep in mind that a user-defined instruction (UDI) can transport two 32-bit operands and one 32-bit result per cycle. Our coprocessor for complex-number multiplication is implemented in file `src/cmplxmul.vhd`.

Step 2: Build the FCM Wrapper

To be area-efficient, your coprocessor may need a multicycle behavior similar to ours. Therefore, you will need a state machine to implement a simple handshake protocol between the coprocessor and the Auxiliary Processing Unit (APU). In our example, we did this inside the wrapper "`fcmcmul`," which we implemented in file `src/fcmcmul.vhd`.

Inside the wrapper `fcmcmul`, we instantiated the complex-number multiplication hardware block `cmplxmul`, which becomes the Fabric Coprocessing Module (FCM). Thus, `fcmcmul` provides the interface we need to connect it to the APU. You can find a detailed description of those interface signals in Xilinx document ug200, starting at page 188. The important detail is the timing diagram for "Non-Autonomous Instructions with Early Confirm Back-to-Back" on page 216, which shows the protocol between the APU and the FCM.

Step 3: Connect the FCM with the APU

In general, you can connect your FCM to the APU in two ways: by using the Xilinx Platform Studio (XPS) graphical user interface, or by editing the `.mhs` file. We have found that when cutting and pasting a portion of an existing design into a new one, it is easiest to edit the `.mhs` file. So for this example, we connect the FCM/wrapper and the APU in file `syn/apu/system.mhs`.

We suggest that you do the same. Just copy the section from "`BEGIN fcmcmul`" to "`END`" from our example and paste it into your `.mhs` file.

To make it all work in XPS, you must also provide a set of files in a predefined file/directory structure. In our example, we have called the wrapper block `fcmcmul`; therefore, the file/directory structure looks like this:

```
syn/apu/pcores/fcmcmul/data/fcmcmul_v2_1_0.mpd
syn/apu/pcores/fcmcmul/data/fcmcmul_v2_1_0.pao
syn/apu/pcores/fcmcmul/hdl/vhdl/fcmcmul.vhd
syn/apu/pcores/fcmcmul/hdl/vhdl/cmplxmul.vhd
```

The `.mpd` file contains the port declarations of the FCM. The `.pao` file provides the names of the blocks and files associated with the FCM, and XPS finds the VHDL source files for the coprocessor and the wrapper in the hdl/vhdl directory.

You should replicate and adjust this tree as needed for your own APU-enhanced FPGA design.

Step 4: Hardware Simulation

We have provided the necessary files to test the APU example using ModelSim. As a prerequisite, and only if you have not done this yet, you must generate and compile the Xilinx simulation library. You can do this from the XPS menu **XPS→ Simulation→ Compile**

Simulation Library. Then generate all RTL simulation files for the entire design from the XPS menu **XPS→ Simulation→ Generate Simulation**.

Next, run the RTL simulation to verify your APU design—in particular, the handshake protocol between the APU, the wrapper and your coprocessor.

The simulation shows the two possibilities for the APU delivering the operands in either one or two cycles (as explained in ug200, on page 216). Look for the signals `FCMAPUDONE` and `FCMAPURESULTVALID`.

Step 5: Software Testing

For the complex-number multiplication, we have written a small standalone program, `syn/apu/aputest/aputest.c`, that demonstrates the use of the APU and our coprocessor from a software point of view.

This program configures the APU and defines the UDI. Then it runs a loop to compute the complex-number multiplication using our hardware coprocessor, compares it against the result of a software-only complex-number multiplication and provides a performance analysis.

You must configure the PowerPC APU before it can function properly, in one of two ways: You can either click in XPS and enter the initialization values for certain control registers of the APU, or you can configure the APU directly from the software program that uses the APU. We feel the latter option is more explicit and robust.

In our C source code file, you will find descriptive C macros and function calls to properly initialize the APU. Feel free to copy-and-paste them into your program as needed

Within the loop, we do complex-number multiplication first using the UDI and then using the software macro `ComplexMult`. We use our routines `Start_Time` and `Stop_Time` for performance analysis. The three calls `UDI1FCM_GPR_GPR_GPR` implement the three-cycle hardware complex-number multiplication. We define the C macro `UDI1FCM_GPR_GPR_GPR` in file `syn/apu/ppc440_0/include/xpseudo_asm_gcc.h`, which is a Xilinx EDK-generated file. We implement the C macro `UDI1FCM_GPR_GPR_GPR` via assembler mnemonic `udi1fcm`. Because Xilinx has patched the assembler, this `udi1fcm` mnemonic—though obviously not part of the original PowerPC 440 processor instruction set—is already a proper instruction the APU can handle.

In our test case, `aputest` is the XPS software project that we compiled, assembled, linked and downloaded into the Virtex-5 FXT block RAM for execution by the PowerPC processor.

Step 6: Generate the FPGA Configuration

You can generate the FPGA configuration bit file from the XPS menu **XPS→ Hardware→ Generate Bit Stream**. To save you some time, we have included a bit file for the Xilinx ML507 Development Platform. You can find it in `Syn/apu/implementation/download.bit`.

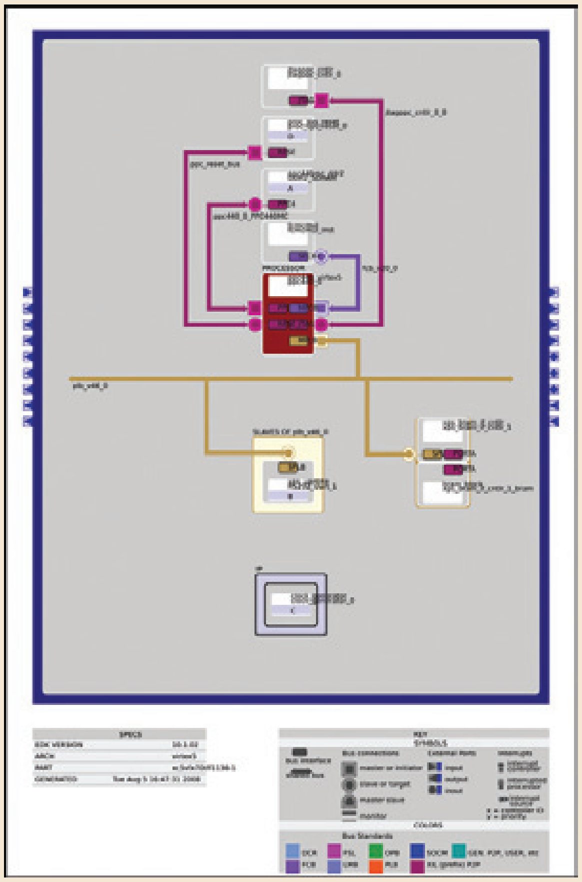


Figure 1 – EDK processor system block diagram

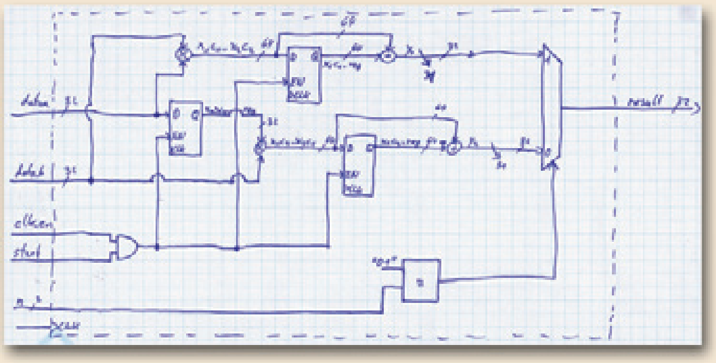


Figure 2 – Complex-number multiplication coprocessor was so easy to design, you can do it on graph paper.

Step 7: Execute the Example Design

Download the FPGA configuration bit file, start the XPS debugger XMD (UART settings are 115200-8-N-1) and view the example design.

The run-time it reports is 4,717 cycles for the software-only design and 1,936 cycles for the UDI hardware-accelerated complex-number multiplication. Thus, the acceleration is approximately 2.4 times, with the complex-number multiplication running with no timing optimizations at 50 MHz. Of course, if we were to use pipelining and increase parallelism, the coprocessor could run much faster, increasing the overall acceleration to five to ten times.

The PowerPC processor APU that resides within the Xilinx Virtex-5 FXT devices allows embedded engineers to accelerate their systems in a very efficient way, by adding special-purpose user-defined instructions for hardware acceleration and coprocessing.

these hardware blocks are called Fabric Coprocessing Modules, or FCMs. You can write FCMs in VHDL or Verilog, and they will end up in the FPGA fabric of the Virtex-5 FXT device. You can connect one or more FCM to the PowerPC processor APU interface.

The next step is to adjust your software code to make use of those additional instructions. You have two options (assuming you are programming in C language). The first is to change the C compiler to automatically exploit cases where the use of the additional instructions would be beneficial. We'll leave this option to the academics and certain folks working on ASSPs.

The second, and more elegant, option is not to touch the compiler but instead to use so-called compiler-known functions. This means that manually, in our software code, we'll call a C macro or a C function that makes use of these additional instructions.

Using either option, we must adjust the assembler so that it supports the new instructions. Fortunately, Xilinx includes a PowerPC compiler and assembler with the Embedded Development Kit (EDK) that already support these additional instructions.

When the PowerPC encounters these new instructions, it quickly detects that they are not part of its own original instruction set and defers the handling of them to the APU. Xilinx has configured the APU to decode these instructions, provide the appropriate FCM with the operand data and then let the FCM perform the computation.

If this is properly done, the software requires fewer instructions when running. Therefore, we can get more compute power out of our design without increasing the CPU clock frequency (which may cause other headaches).

The key reason to use the APU rather than connecting hardware blocks to the microprocessor via the PLB bus is the superior bandwidth and lower latency between the PowerPC processor and the APU/FCM. Another advantage lies in the fact that the APU is independent of the CPU-to-peripheral interface and therefore does not add an extra load to the PLB bus, which the system needs for fast peripheral access.

The APU provides various ways of interfacing between the PowerPC and the FCM. We can use a load-store method or the user-defined instruction (UDI) method. Chapter 12 of the Xilinx User Guide ug200 offers detailed descriptions of these techniques (*http://www.xilinx.com/support/documentation/user_guides/ug200.pdf*).

In our example we'll deploy the UDI method, because it provides the most control over the system, enabling the highest performance. The example design is available for download from our Web site, at *http://www.missinglinkelectronics.com/support/*.

Example Design Description

By adding a UDI, we have extended the PowerPC processor's instruction set to perform complex-number multiplications, a handy optimization for many multimedia decoding systems. The EDK diagram (see sidebar, Figure 1) shows the overall design, including how we connected the complex-number multiplier FCM to the PowerPC processor via the APU, and how software can make use of it.

We picked complex-number multiplication as an example because of its wide applicability in decoding streaming-media data, and because it clearly demonstrates how to make use of the APU by adding a special-purpose instruction.

Complex-number multiplication is defined as multiplying two complex numbers, each having a real value and an imaginary value.

```
(a_R + j a_I, where j*j = -1):

(a_R + j a_I) * (b_R + j b_I) =
(a_R * b_R - a_I * b_I) + j (a_I *
b_R + a_R * b_I)
```

For efficiency, the complex-number multiplication hardware block (**cmplxmul**) performs the multiplication in three stages. This saves hardware resources by using only two multipliers and two adders in this multicycle implementation. Figure 2, also in the sidebar, shows a block diagram (in sketch form) of the complex-number multiplication FCM.

As the VHDL code in **cmplxmul.vhd** demonstrates, we perform the complex-number multiplication in three clock cycles. In file **cmplxmul.vhd** we have implemented the FCM to perform this complex-number multiplication. File **fcmcmul.vhd** provides the FCM/APU interface wrapper to connect our FCM to the APU. As we show in our step-by-step procedure (see sidebar), you can use this wrapper as a template to connect your own FCM to the APU when using the UDI method (the load-store method requires a different interconnect).

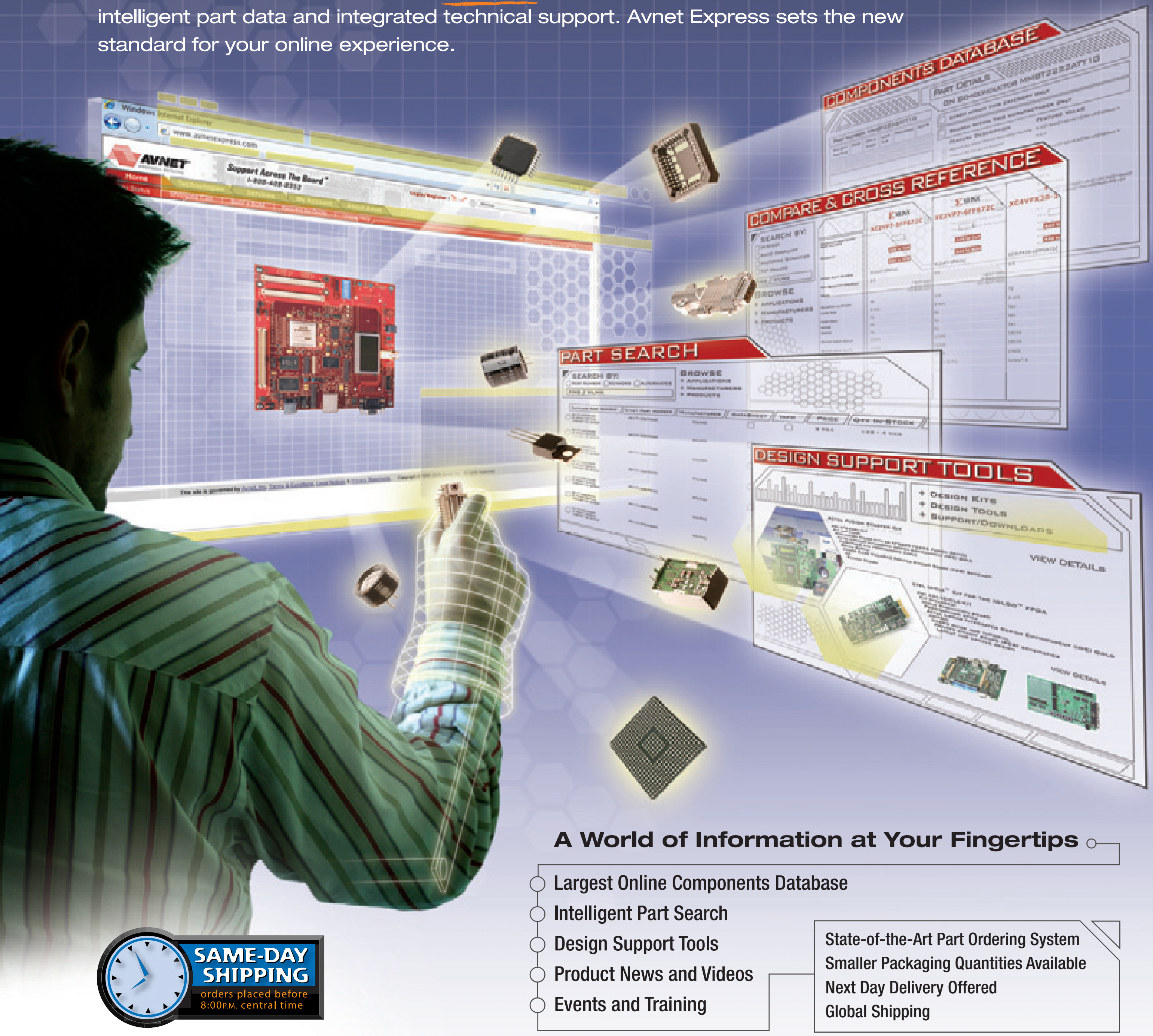
We synthesized our design with Xilinx EDK/XPS 10.1.02 using Xilinx ISE® 10.1.02. We simulated and tested the design with ModelSim 6.3d SE.

The PowerPC processor APU that resides within the Xilinx Virtex-5 FXT devices allows embedded engineers to accelerate their systems in a very efficient way, by adding special-purpose user-defined instructions for hardware acceleration and coprocessing. Using the example design described here as a starting point will show you that mastering the APU is straightforward, and can give your designs a major performance boost without the use of special tools.

You Can Have It ^All^ Now™!

The power of Avnet Express just got better.

Avnet Express connects you to the ***largest*** online components database linked to intelligent part data and integrated technical support. Avnet Express sets the new standard for your online experience.



COMPONENTS DATABASE
COMPARE & CROSS REFERENCE
PART SEARCH
DESIGN SUPPORT TOOLS
SAME-DAY SHIPPING
orders placed before 8:00P.M. central time

A World of Information at Your Fingertips

- Largest Online Components Database
- Intelligent Part Search
- Design Support Tools
- Product News and Videos
- Events and Training

- State-of-the-Art Part Ordering System
- Smaller Packaging Quantities Available
- Next Day Delivery Offered
- Global Shipping

Order your parts today. Online or by phone. Fast.

1.866.AVNET4U www.AvnetExpress.com

AVNET® Express

Accelerating Your Success™

Avnet Green Initiative

Supplier Authorized Distributor

Visit us at the
Embedded Systems Conference 2009
Booth #1420

THIS CHANGES EVERYTHING

Introducing programmability so advanced, it defies logic. As the foundation for Targeted Design Platforms, the new Virtex®-6 and Spartan®-6 FPGAs give you twice the capability at half the power. See why this changes everything at **www.xilinx.com/6**.

VIRTEX 6 SPARTAN 6

XILINX®

PN 2205