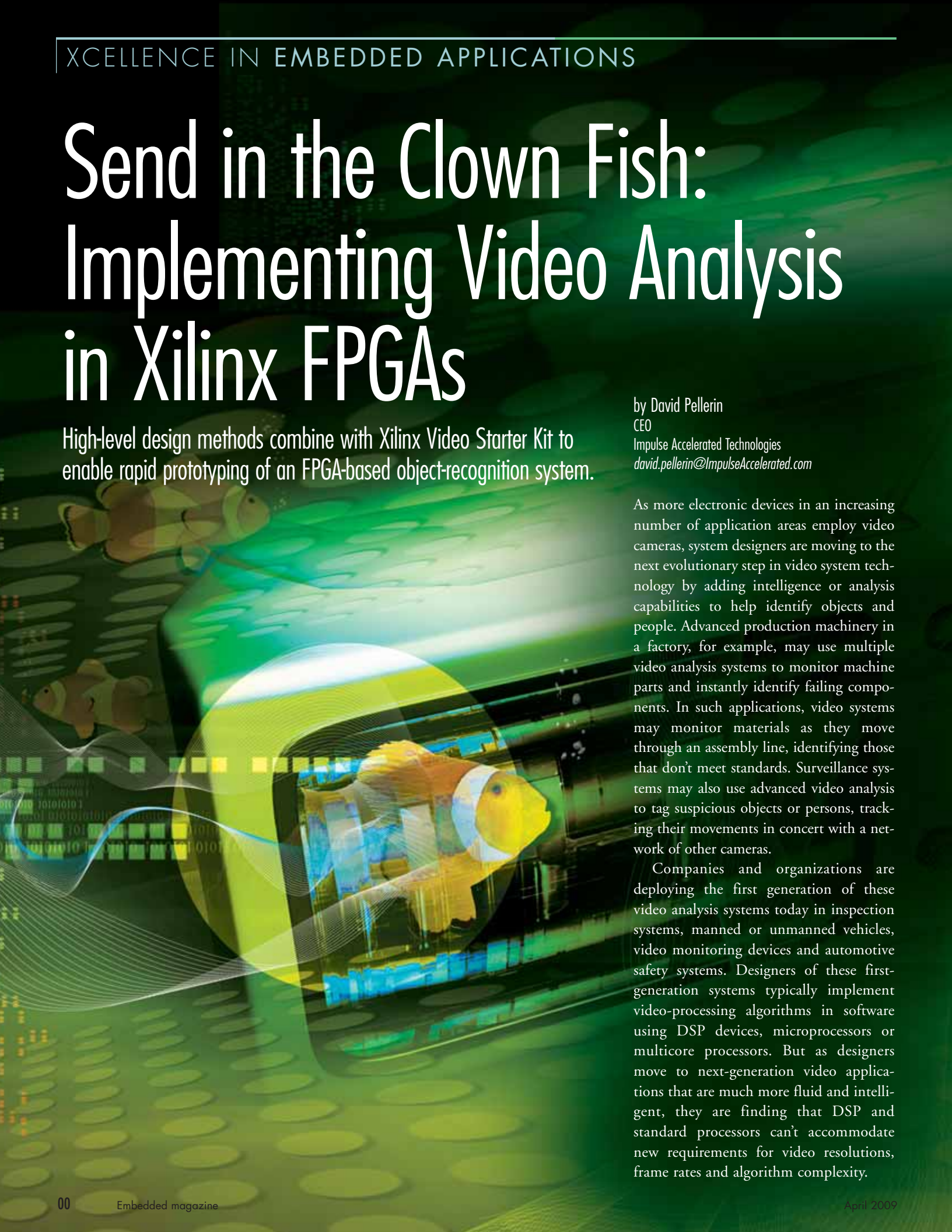
Send in the Clown Fish: Implementing Video Analysis in Xilinx FPGAs

High-level design methods combine with Xilinx Video Starter Kit to enable rapid prototyping of an 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 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.

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.

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 particu-

larly useful for filtering applications and algorithms that don't fit into existing, predefined 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, IP 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.

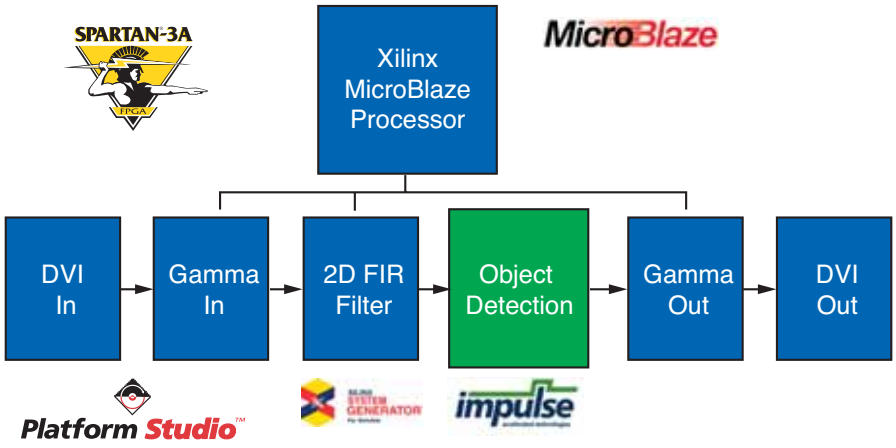


Figure 1 – Impulse Accelerated Technologies implemented this complete video-filtering and object-detection design in a single Spartan-3A FPGA device.

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.

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 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 schemes, 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