Building Automotive Driver Assistance System Algorithms with Xilinx FPGA Platforms

System Generator for DSP is a high-abstraction-level design tool that gives algorithm developers and system architects an efficient path from a Simulink-based algorithmic reference model to an FPGA hardware implementation without any need for HDL coding.

d herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

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

Roberto Marzotto
Design Engineer
Embedded Vision Systems, S.r.l.
roberto.marzotto@evsys.net

Paul Zoratti
Automotive Senior System Architect
Xilinx, Inc.
paul.zoratti@xilinx.com

The emerging market for automotive driver assistance systems (see cover story) requires high-performance digital signal processing as well as low device costs appropriate for a volume application. Xilinx® FPGA devices provide a platform with which to meet these two contrasting requirements. However, algorithm designers unfamiliar with FPGA implementation methods may be apprehensive about the perceived complexity of the move from a PC-based algorithmic model to an FPGA-based hardware prototype. They don't need to be.

A Xilinx tool, the System Generator for DSP, offers an efficient and straightforward method for transitioning from a PC-based model in Simulink® to a real-time FPGA-based hardware implementation. This high-abstraction-level design tool played a central role in a project conducted between engineers at Xilinx and Embedded Vision Systems. The goal of the project was to implement an image-processing algorithm applicable to an automotive lane departure warning system in a Xilinx FPGA using System Generator for DSP, with a focus on achieving overall high performance, low cost and short development time.

Challenges in DA System Development

Automotive driver assistance (DA) system engineers commonly use PC-based models to create the complex processing algorithms necessary for reliable performance in features such as adaptive cruise control, lane departure warning and pedestrian detection. Developers highly value the PC-based algorithm models, since such models allow them to experiment with, and quick-

Copyright © 2008 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

ly evaluate, different processing options. However, in the end, a properly designed electronic hardware solution is necessary to realize economical high-volume production and deployment.

Verifying algorithm performance consistency between deployable target hardware and the software algorithm model can be problematic for many developers. Moving from floating-point to fixed-point calculations (for example, employing different methods for trigonometric functions) can sometimes cause significant variation in the outputs between the reference software algorithm and the hardware implementation model. Further complicating the algorithm performance consistency problem for DA system developers is the fact that the input stimulus is very nondeterministic. For DA systems, which generally rely on inputs from remote sensing devices (cameras, radar and so on), the inputs are the incredibly diverse range of roadway and environmental conditions that drivers may encounter. Engineers may find designing a processing algorithm to adequately address all situations is extremely challenging, and verifying compliance between the software model and its electronic hardware implementation is critical.

For many applications involving image processing, the parallel resources of Xilinx Spartan®-3 FPGA devices deliver higher performance per dollar than VLIW DSP platforms (see our paper in *Xcell Journal* issue 63: *http://www.xilinx.com/publications/xcellonline/xcell_63/xc_pdf/p16-19_63-block.pdf*). However, some system designers still wrongly assume that the only way to program an FPGA is with a hardware description language such as VHDL, which many engineers don't know. But in fact, this is no longer the case. Our design methodology, applying the Simulink modeling tool and the Xilinx System Generator for DSP FPGA synthesis tool, provided not only an easy, efficient way of implementing FPGA designs, but also a means of accelerating algorithm compliance testing with hardware-software co-simulation. Further, you don't have to be familiar with an HDL to use System Generator.

Lane Departure Warning Model Description

The overall function of a lane departure warning (LDW) system is to alert the driver when the vehicle inadvertently strays from its highway lane. A camera in front of the host vehicle captures images of the roadway to identify markings that constitute the lane boundaries. The system continuously tracks those boundaries, as well as the host vehicle's position relative to them. When the vehicle crosses the bounds of the lane, the system issues a warning.

The automotive industry and academic world have widely adopted MATLAB® and Simulink as algorithm and system-level design tools. In particular, Simulink enables automotive algorithm engineers to quickly and easily develop a sophisticated DSP algorithm, thanks to its very high abstraction level and the tool's graphical schematic entry.

Figure 1 shows the top-level block diagram of our LDW system model, designed in Simulink. The green block, labeled Lane Detection, contains an image-preprocessing subsystem, the various stages of which we show in Figure 2. The purpose of the lane detection function is to extract those

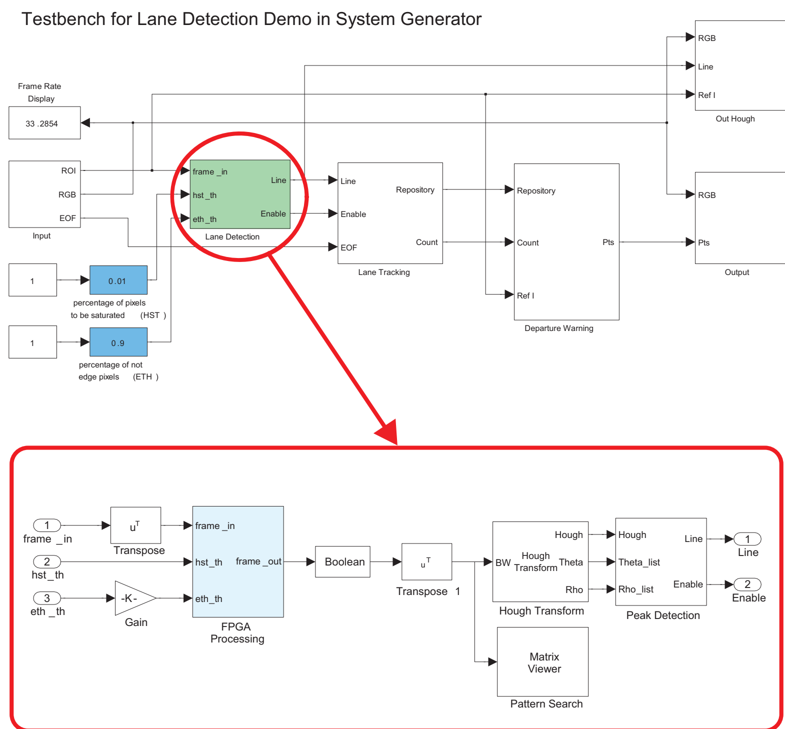

Figure 1 – LDW Simulink high-level block diagram

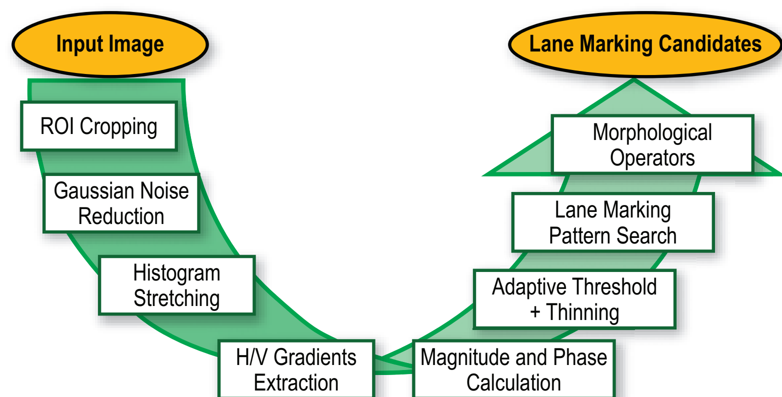

Figure 2 – LDW preprocessing function chain

features of the roadway image most likely to represent lane boundaries.

To improve the performance of the edge detection with respect to noise, the first stage of the pipeline is a 2-D 5x5 Gaussian noise reduction (GNR). The second stage is histogram stretching (HST), a technique developers use to enhance the contrast of the image, exploiting as much as possible the whole gray-level range. The third step, the horizontal/vertical gradient (HVG), enhances those pixels in which a significant change in local intensity is seen. Developers perform HVG by computing the 2-D 5x5 gradients of the image (via the 2-D Euclidean distance).

The edge-thinning (ETH) block determines which points are edges by thresholding the gradient magnitude and applying non-maximum suppression to generate thin contours (one-pixel thick). The lane-marking pattern search (LMPS) acts as a filter, selecting a subset of edge points that display a particular configuration consistent with the lane markings, and removing spurious edge points that arise due to shadows, other vehicles, trees, signs and so on. The last step of the pipeline is a 3x3 morphological filtering (MRP) the system uses for the final cleaning of the lane-marking candidate's map.

In the Simulink model, we have implemented the various stages of the image-preprocessing subsystem using a mixture of Simulink blockset functions and MATLAB blocks. Because of its ability to process large amounts of data through parallel hardware paths, an FPGA is well-suited for the implementation of the lane detection function of our model. Therefore, we targeted this function as the starting point for transitioning the LDW Simulink design to an FPGA.

With this partitioning, the FPGA performs the processing-intensive pixel-level analysis of each frame and reduces the data from a 10-bit gray-scale image to a simple binary image for our downstream processing. For the entire system design, we are targeting an XA Spartan-3A DSP 3400, but we could also fit the system on a smaller 3A DSP 1800 or a 3E 1600.

System Generator Overview

The System Generator for DSP design tool works within Simulink. It uses the Xilinx DSP blockset for Simulink and will automatically invoke the Xilinx CORE Generator™ tool to generate highly optimized netlists for the DSP building blocks. You can access the Xilinx DSP blockset via the Simulink Library browser, which you can, in turn, launch from the standard MATLAB toolbar. More than 90 DSP building blocks are available for constructing a DSP system, along with FIR filters, FFTs, FEC cores, embedded processing cores, memories, arithmetic, logical and bit-wise blocks. Every block is cycle- and bit-accurate and you can configure each of them for latency, area vs. speed performance optimization, number of I/O ports, quantization and rounding.

Two blocks, called Gateway-In and Gateway-Out, define the boundary of the FPGA system from the Simulink simulation model. The Gateway-In block converts the floating-point input to a fixed-point number. Afterwards, the tool correctly manages all the bit growth in fixed-point resolution, depending on the mathematic operation you are implementing during the following functional stages.

Since Simulink is built on top of MATLAB, System Generator allows the use of the full MATLAB language for input-signal generation and output analysis. You can use the From-Workspace and To-Workspace blocks from the Simulink Source and Sink libraries to read an input signal from a MATLAB variable (From-Workspace) or to store a partial result of a signal to a MATLAB variable (To-Workspace). Furthermore, you can set a lot of parameters of the System Generator blocks via MATLAB variables, thus allowing you to customize the design in sophisticated ways, just by updating a MATLAB script containing all such variables (you can assign MATLAB functions to the model and call them back before opening it, or even before starting or after stopping the simulation).

Another important feature of System Generator for DSP is the hardware-software co-simulation. You can synthesize a portion of the design into the target FPGA board (hardware model), leaving the remaining part as a software model in the host PC. That allows you to make an incremental transition from software model to hardware implementation. The tool transparently creates and manages the communication infrastructure via Ethernet and shared memories (between the host PC and the target FPGA device). In such a way, when running a simulation, the part you've implemented in the hardware is really running on the target silicon device, while the software model emulates the rest in the host PC. You can use the shared memories to store, for example, the input image and the generated output image. The Ethernet communication provides enough bandwidth for pseudo-real-time processing. You can find more details in the user manual.

The flexible partitioning between software model and hardware processing, combined with the hardware-software co-simulation capabilities, provides you with a powerful verification tool to measure compliance between the original software-only algorithm and the production-intent hardware implementation. You can use Simulink itself to compare the results of the software processed data to the hardware processed data. This functionality is especially useful in driver assistance applications, where the general system input images are nondeterministic.

Now, let's examine in detail how to model an image-processing algorithm in System Generator for DSP using as an example, for the sake of conciseness, the GNR, which is the first module of the image-preprocessing pipeline.

System Generator Implementation of GNR Function

Random variations in intensity values—aka noise—often corrupt images. Such variations have a Gaussian or normal distribution and are very common among different sensors—that is, CMOS cameras. Linear-smoothing filters are a good way to remove Gaussian and, in many cases, other types of noise as well. To achieve such functionality, we can implement a linear finite impulse response (FIR) filter using the weighted sum of the pixels in successive windows.

Before starting the implementation of the

GNR System Generator block, we realized its behavioral model in MATLAB. It takes only a couple of code lines to implement. First, we have to calculate the kernel, specifying the mask size (5x5 in our case) and the sigma of the Gaussian. Then we can filter the input image by convolution:

```
n_mask = fspecial('gaussian', 5, 0.8);
out_img = conv2(in_img, n_mask, 'same');
```

We can also use this behavioral model to tune the coefficients of the mask by testing the filter on real video data. It can also validate the hardware by verifying that the outputs of the System Generator for DSP subsystem are equal to the output of the MATLAB function, within a specified accuracy, since MATLAB works in floating-point and System Generator in fixed-point arithmetic.

The 2-D GNR module processes the input image in a streaming way (that is, line by line). Figure 3 shows the top-level System Generator block diagram for the entire preprocessing chain as well as the top-level diagram specific to the Gaussian noise reduction function.

The data_in and data_out ports in Figure 3 receive the input stream of pixels and return the filtered stream, respectively. We use the remaining ports for timing synchronization and processing control between adjoining blocks.

We based the internal architecture of the GNR block on two main subsystems, as highlighted by the yellow and blue blocks in the figure. We will drill down into each of these blocks to describe the details of the System Generator design.

We needed the first main subsystem, the line buffer (shown in yellow), to buffer four lines of the input image stream in order to output the pixels aligned 5 x 5. For each input pixel I(u,v), the line buffer returns a 5x1 vector composed by the current pixel and the four previous pixels on the same row, that is [I(u,v-4); I(u,v-3); I(u,v-2); I(u,v-1); I(u,v)]. In Figure 4, we implemented the line buffer block via concatenating two dual-line buffers, each using a dual-port block RAM (a memory resource resident on the FPGA device), a counter

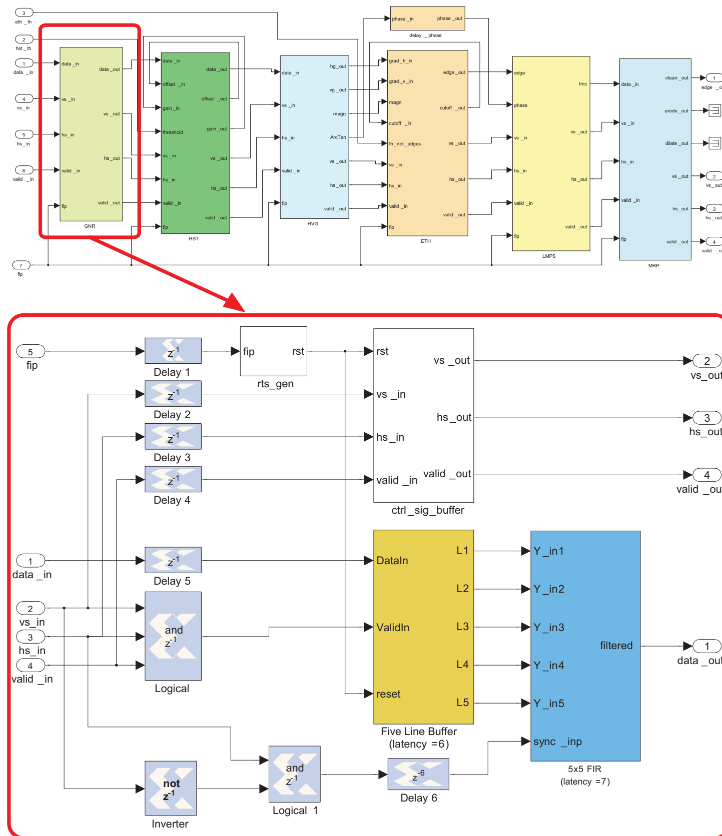

Figure 3 – Top-level preprocessing and Gaussian noise reduction diagrams

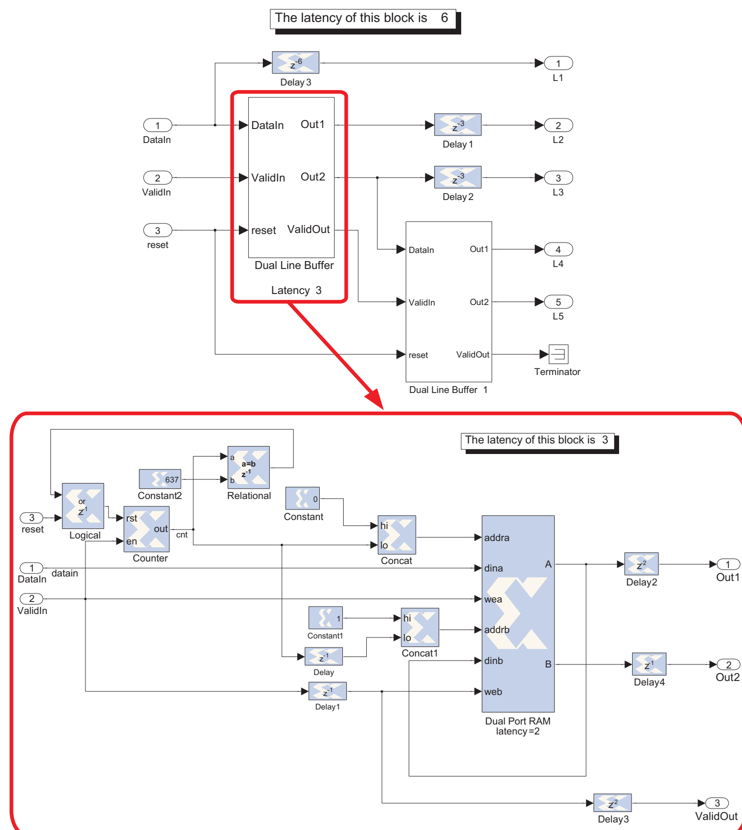

Figure 4 – System Generator implementation of a four-line buffer

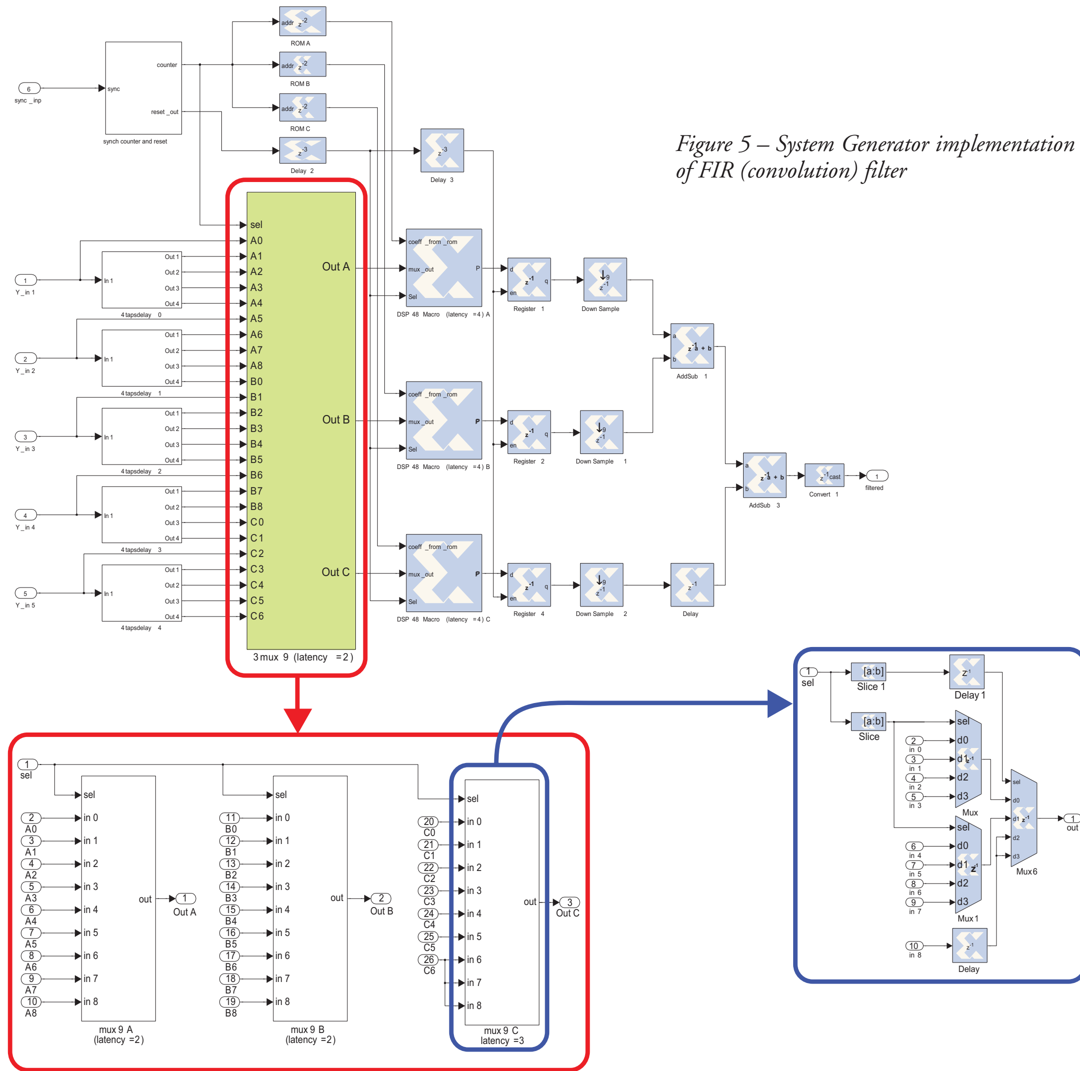

Figure 5 – System Generator implementation of FIR (convolution) filter

for the memory addressing, some simple binary logic and registers to implement appropriate delays.

The blue shading and the Xilinx logo indicate the System Generator primitive blocks, each of which corresponds to optimized synthesizable HDL code. Through this graphical interface, System Generator allows the algorithm developer to easily implement a DSP algorithm in hardware without having any knowledge of HDL coding techniques.

We use the second main subsystem, the 5x5 FIR kernel (shown in blue in Figure 3), to implement the convolution operation. It receives in input a block of 25 pixels from the line buffer block, multiplies them by the 25 coefficients of the Gaussian mask and accumulates the result into a register. We show the details of the implementation in Figure 5. To fit the performance of the target device and to achieve a specified sample rate of 9 million samples per second (MSPS), we chose to use three multiply-accumulators (which we implemented using the FPGA's DSP48 programmable Multiply Accumulator functional units) in parallel. We dedicated each of them, at most, to nine multiplications, via time-division multiplexing (see the larger System Generator blocks in the top diagram of Figure 5). We split and stored the set of 25 coefficients of the mask in three ROMs that we implemented as distributed RAM (another memory resource the FPGA provides).

In our implementation, we chose to fix the coefficients of the mask at compile time. However, we could have easily updated the design to accept these values as real-time inputs. In this way we could dynamically change the run-time intensity of the filtering based on environmental conditions. (Implementation Note: Because of the isotropic shape of the Gaussian, the kernel could be decomposed in two separable masks and the filtering could be obtained as two consecutive convolutions with 1x5 and 5x1 masks along the horizontal and vertical directions. Even though this approach would reduce the FPGA resources required, we didn't adopt it to avoid losing generality and readability of the design.)

Another powerful feature found in System Generator is the ability to apply a preload MATLAB function to customize the design before compilation. By setting parameters of the System Generator module (such as image resolution, FIR kernel values and the number of computational precision bits) and initializing the workspace signals we used during the run-time simulation, we can quickly and easily experiment with different processing

methods and sets of input data. Furthermore, after we completed the simulation, a Stop MATLAB function is called to display the results by computing some useful information and comparing the fixed-point results against the floating-point reference ones. This methodology allows algorithm developers to closely analyze any portion of the hardware implementation and compare it to the original software model to verify compliance.

System Generator FPGA Synthesis Results

Those developing driver assistance systems must implement their designs at a cost level appropriate for high-volume production. The die resources needed to achieve a certain level of processing performance will define the size of the FPGA device they require and, therefore, its cost.

In our lane departure warning preprocessor implementation, we target the XA Spartan-3A DSP 3400, currently the largest device available in the Xilinx automotive product line. We took this approach to support future planned development activities with this model, but analysis of the resources consumed for the preprocessing function clearly shows that this design would fit into a much smaller device.

The following table reports the resources occupied by the GNR block on the XA Spartan-3A DSP 3400 device. The estimation assumes gray-level input images at VGA resolution at a 30-Hz frame rate (which implies an input data rate of 9.2 MSPS).

DSP48s	3	out of	126	2%
BRAMs	3	out of	126	2%
Slices	525	out of	23872	1%

From the perspective of timing performance, the GNR design runs at 168.32-MHz clock frequency and accepts an input data rate of up to 18.72 MSPS.

Total resources needed for the entire lane detection preprocessing subsystem are summarized below:

DSP48s	12	out of	126	9%
BRAMs	16	out of	126	12%
Slices	2594	out of	23872	10%

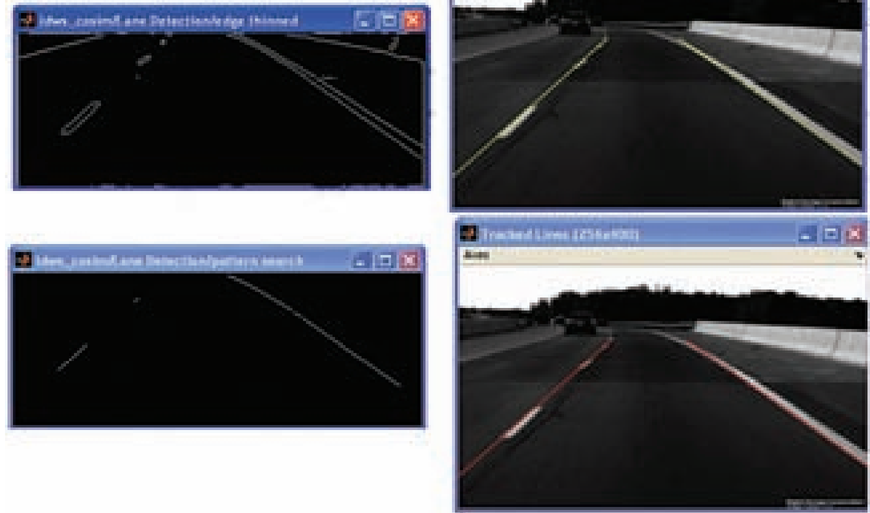

Figure 6 – LDW processing model outputs

The corresponding timing-performance analysis showed a clock frequency of 128.24 MHz, with a maximum input data rate of 14.2 MSPS.

Given these resource requirements, we estimated the preprocessing function would even fit into an XA Spartan-3E 500—roughly one-seventh the density of the XA Spartan-3A 3400A device.

Results and Future Work

Figure 6 provides a sample of the performance of our LDW system, including an FPGA-based image-preprocessing function for lane-marking candidate extraction. You can see the input frame in the two images on the right. The pair of images on the left illustrate the performance of the preprocessing function we implemented in the FPGA. The picture at the top left represents the magnitude of the edge detection function after thresholding. The one at the lower left is taken after the edge-thinning and lane-marking pattern search processes. Clearly, our LDW preprocessor is very effective at taking a roadway scene and reducing the data to only the primary lane-marking candidates. The yellow and red lines, respectively, in the top and lower-right images represent instantaneous and tracked estimations of the lane boundaries based on a simple, straight-line roadway model.

In order to accurately predict the trajectory of the vehicle with respect to the lane boundaries, our future LDW system will use a curvature model. We are currently adopting a parabolic lane model in the object space, assuming the width of the lane is locally constant and is located on a flat ground plane. We can describe a parabolic lane model by creating four parameters incorporating the position, the angle and the curvature. Using a robust fitting technique, it is possible to estimate the four parameters for each frame of the video sequence.

Noise, changing light, camera jitter, missing lane markings and tar strips could weaken the model extraction. To compensate for this information gap and make this phase more robust and reliable, the system needs a tracking stage. Tracking can be done using the Kalman filter in the space of the parameters of the lane model.

The extraction and the tracking of the lane model are the next two stages we will implement in the FPGA soon. For this job we plan to use the AccelDSP™ synthesis tool, another Xilinx high-level DSP design tool. Because it supports a linear algebra library, we can use AccelDSP to implement a four- or six-state Kalman filter.

AccelDSP can also generate gates directly from MATLAB code and we can use it synergistically with System Generator for DSP. Furthermore, the AccelDSP synthesis tool is well suited for feasibility analysis and fast prototyping. It automatically quantizes the original floating-point MATLAB into fixed-point, and it maps the various MATLAB instructions into the FPGA resources. This is probably the easiest-to-use DSP tool that Xilinx provides to driver assistance algorithm designers and system architects.

In short, algorithm designers and system architects working on driver assistance technology can nowadays rely on a highly sophisticated DSP design tool to build their reference algorithmic models, and then easily implement those models into Xilinx FPGA low-cost devices. The result is high quality, high performance and low cost simultaneously.

One crucial feature of System Generator for DSP is the capability to implement a portion of the design into the silicon target device (of a specific board connected via Ethernet) while the remaining part runs on the host PC. Such a hardware-software co-simulation allows easy verification of the hardware behavior while also accelerating simulation speed.

As you can see, we used the Xilinx System Generator for DSP to create an image-preprocessing pipeline for an LDW system. While in this discussion we only revealed some of the details of one of its modules, namely the GNR 2-D FIR filter, the entire lane detection preprocessing function (shown in Figure 2) took only 12 DSP48, 16 BRAM and 2,594 slices of an XA Spartan-3A DSP 3400 device, running at 128.24 MHz with an input data rate of 14.2 MSPS, 50 percent higher than what is needed by VGA image resolution. The whole algorithm design and FPGA implementation required a few weeks of work and did not necessitate the writing of any VHDL code.

We look forward to continuing the project by implementing the extraction and tracking of the lane models in the AccelDSP design tool and then integrating such stages within the System Generator for DSP model. For further detail you can contact any of us by e-mail. (The authors are grateful to professor Vittorio Murino of the computer science department at Verona University for his support and contributions.)

The widest selection
of COTS board-level solutions using Virtex-5 FPGAs
FPE650 - 6U VPX Quad FPGA Processor Board
AD3000/AD1500 - 3.0GSPS or dual 1.5GSPS A/D PMC/XMC
PMC-FPGA05 Virtex-5 LX110 PMC Board
HPE640 & HPE720 -6U VPX PowerPC & FPGA Processor Boards
VPF2 - 6U VXS PowerPC & FPGA Processor Board
MM-6171 - XMC Buffer Memory Node
User programmable Xilinx® Virtex®-5 FPGA signal processors and analog, digital and fiber-optic I/O
A new generation of performance
Single or multiple FPGA solutions
Simple solutions for complex tasks
PMC, XMC, VXS and VPX form factors
Flexibility based on open standards
Commercial and Rugged variants
Easily migrate from development to deployed systems
Libraries and Example Code
Easy to use with head-start time-to-market
Embedded Computing - Data Recorders & Storage - Bus/Protocol Analyzers
For more information, please visit
http://www.vmetro.com/virtex-5 or call (281) 584-0728
VMETRO
innovation deployed