WP452 (v1.0) June 24, 2014

Adaptive Beamforming for Radar: Floating-Point QRD+WBS in an FPGA

By: Luke Miller

The Vivado® HLS tool allows development of fully verified floating-point radar QRD+WBS algorithms for Xilinx® FPGAs in hours, not weeks or months, in a native C environment.

ABSTRACT

The gradual transition of analog signal processing to the digital domain in radar and radio systems has led to advancements in beamforming techniques and, consequently, to new applications. The ability to steer beams digitally with great precision has radically changed the future of commercial and military radar system design.

Adaptive beamforming algorithms have pushed the signal processing world into the use of floating-point arithmetic; this has increased radar capability by allowing creation of multiple simultaneous spot beams for individual real-time target tracking.

The modified Gram-Schmidt (MGS) QR decomposition (QRD) and weight back substitution (WBS), key algorithms for radar DSP, allow a radar to form beams adaptively while suppressing side lobes, noise, and jamming. These applications require a very high number of FLOPS (floating-point operations per second).

Xilinx FPGAs have an orders-of-magnitude advantage in floating-point performance compared to commercial GPU, DSP, and multi-core CPUs.

High-Level Synthesis (HLS), a standard tool in the Xilinx Vivado® Design Suite, supports native C language design. A floating-point matrix inversion algorithm, the heart of adaptive beamforming, can now be coded in native C/C++ or SystemC using Xilinx Vivado HLS. The design described in this white paper was completed in about four hours. Hand-coding a VHDL/Verilog version of this design and verifying it using RTL could take weeks or even months, depending on the skill of the designer.

Using HLS, Xilinx FPGA adaptive beamforming performance and QoR can be increased by orders of magnitude and power consumption greatly decreased compared to existing GPU and multi-core processor designs. This white paper focuses on a complex floating-point, variable-sized MGS QRD+WBS, up to 128x64 in size.

Introduction

Most radars today employ some type of *adaptive digital beamforming*. A high-level block overview of the receiver beamforming concept is illustrated in Figure 1.

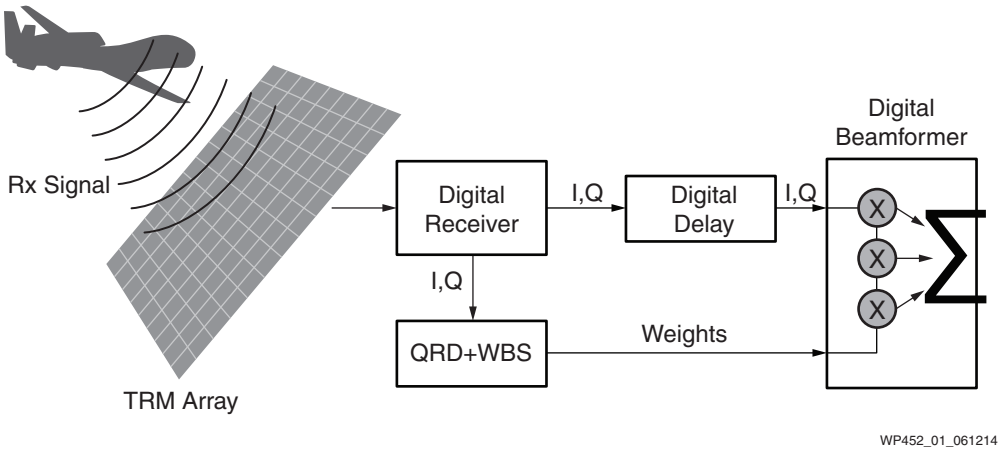

Figure 1: **High-Level Overview of Adaptive Digital Beamforming**

Radar designs can be expected to occupy progressively higher bandwidths, requiring receiving systems to become more reliable at suppressing the effects of noise sources, off-target antenna side-lobe content, interference from hostile jamming signals, and the "clutter" present within the wider passbands characteristic of the newer radar technologies. All this must be done while maintaining directional control of each antenna array element—individually and simultaneously, in real time. Successfully completing these tasks within given time limits is accomplished through element-level processing—that is, digitally processing each antenna element's received signal individually and simultaneously.

A critical part of element-level processing is adaptive digital beamforming. This white paper focuses on the technology of adaptive beamforming and how it can be implemented using Xilinx FPGAs to create a *beam-agile radar system* at reduced cost, complexity, power consumption, and time to market than legacy solutions.

Using the technology and Xilinx components described in this white paper, a beam-agile radar can be created by calculating complex floating point adaptive weights. These weights are based on a subset of complex receive samples buffered from a previous pulse repetition interval (PRI). The challenge of calculating these weights is contained within the need to perform a complex matrix inversion that solves Equation 1 before the next PRI of data is received.

This requires a deterministic, low-latency solution of matrix size that is a function of the radar system requirements. Traditionally, this arithmetic was performed by numerous parallel CPUs to ensure that the floating-point calculations were completed before the next PRI. Given the size, weight, and power (SWaP) constraints of many radar/EW systems, the CPU/GPU approach is not the best option to accomplish these calculations. Xilinx FPGAs can perform highly parallel floating point arithmetic much more efficiently, using less hardware.

The flexibility of the Xilinx FPGAs allow the radar designer to consume vast amounts of data over flexible I/O standards such as JESD204B, SRIO, PCIe®, etc., then calculate the adaptive weights in

one FPGA, in real time. The linear equation to be solved is contained within the QRD+WBS functional block shown in Figure 1, and is expressed mathematically in Equation 1 as:

$$Ax = b \qquad \textit{Equation 1}$$

where

A = the complex matrix, size (m,n) where $m \geq n$; these are receive samples

x = the complex vector being solved for, which becomes the adaptive weights, size (n,1)

b = the desired response or steering vector, size (m,1)

To solve for x, one cannot divide b by A. This is where the modified Gram-Schmidt (MGS) QR decomposition (QRD) is needed, which calculates the matrix inversion. The MGS QRD must use floating-point arithmetic to maintain the precision of the adaptive weight solution.

Vivado HLS Overview

The Vivado® high-level synthesis (HLS) tool uses untimed C/C++ as the golden design source for hardware design, accelerating development significantly through reduction of verification time by orders of magnitude. Targeting algorithms to hardware often requires large input test-vector sets to ensure that the correct system response is being achieved. These simulations can take hours or even days to complete when using RTL and an event-based RTL simulator. When implemented using C/C++, however, simulation can be completed up to 10,000 times faster, finishing in seconds or minutes. The designer can implement these faster verification times to accelerate development by enabling more design iterations per day. This is illustrated in Figure 2.

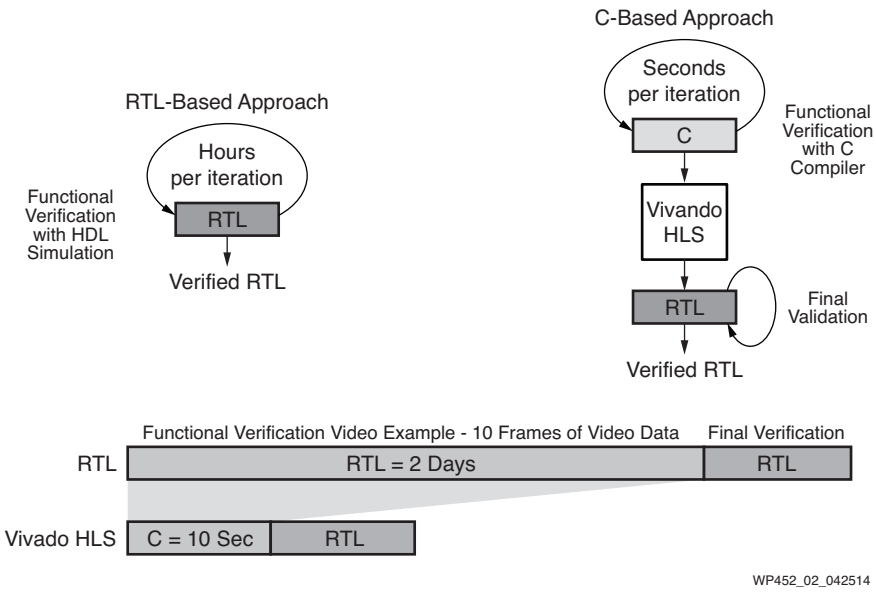

Figure 2: **RTL-Based vs. C-Based Iterative Development Time**

In addition to faster verification, Vivado HLS also supports high-level design exploration that allows users to quickly explore multiple hardware architectures with different area vs. performance trade-offs without modifying the source code. This can be achieved using synthesis directives, such as loop unrolling and pipeline insertion.

While `Math.h` is standard, Vivado HLS ships with a library of linear algebra functions provided in C/C++ that are synthesizable through HLS and optimized for quality of results. These functions have been developed to allow the user to take full advantage of the design exploration features of Vivado HLS, including loop unrolling and pipeline register insertion, that give users the greatest flexibility to generate a hardware architecture that meets their design requirements. The source code for these functions can be modified by the user; the functions serve as working design starting points to provide even greater flexibility in the final implementation. This library includes the following functions and supported data types (see Table 1):

Table 1: **Functions / Data Types Included in Vivado HLS**

Function	Float	Fixed	Complex
Matrix Multiplication	Single, Double	`ap_fixed(16,1)`	Yes
Cholesky Decomposition	Single, Double	`ap_fixed(16,1)`	Yes
QR Factorization	Single, Double	N/A	Yes
SVD	Single, Double	N/A	Yes

The QRD and Weight Back Substitution (QRD+WBS)

The QRD converts the complex matrix *A* into:

$$A = QR \qquad \textit{Equation 2}$$

where:

Q, size *(m,n)* is an orthogonal matrix, which yields the identity matrix when:

$$Q^H Q = I \qquad \textit{Equation 3}$$

Note: Q^H is the complex conjugate transpose for complex systems; Q^T is the transpose for real systems.

R, size (*n,n*) is an upper triangular matrix, or right triangular matrix. It is called triangular as the lower triangle is all zeros. This makes for fast arithmetic when solving for *x*. Substituting the QR relationship back into Equation 1, it follows that:

$$\begin{aligned} QRx &= b \\ Q^H QRx &= Q^H b \\ IRx &= Q^H b \\ Rx &= Q^H b \end{aligned} \qquad \textit{Equation 4}$$

For simplicity, the MGS algorithm is expressed in Octave or MATLAB® code, shown in Figure 3.

```
for i=1:col,
    Q(:,i)=A(:,i);

    for j=1: i -1,
        R(j,i)=Q(:,j)‘ *Q(:,i);
        Q(:,i)=Q(:,i) - (R(j,i)*Q(:,j) );
    end

    R(i,i)=norm(Q(:,i) );
    Q(:,i)=Q(:,i)/R(i,i);
end
```

WP452_03_060914

Figure 3: **The MGS QRD Algorithm in MATLAB or Octave**

Using back-substitution, solve for x. Let $c = Q^H b$.

$$x_j = \frac{1}{R_{j,j}} \left(c_j - \sum_{k=j+1}^{n} R_{j,k}\, x_k \right) \qquad \text{Equation 5}$$

CPU-Based vs. FPGA-Based Beamformer Architectures

Whenever functionality does not fit into a single device, the ramifications can be extreme, causing increases in memory, interfaces, real estate, integration time, cost, and power.

Multiple-CPU Architecture

This is the case when an adaptive beamformer is implemented using the legacy method of employing multiple CPUs. The goal is to beamform 16 channels with QRD+WBS and be solved in approximately 3.5 ms. Not only does the CPU-based design not fit on a single device; it requires much more in-depth system design, qualification, and integration, and consumes massive power.

It is demonstrated in the Results section that each CPU core needs 250 ms to solve a floating-point 128x64 complex QRD+WBS. (This is conservative, as no time is factored in for memory-fetch overhead and scheduling.) Dividing 250 ms by 3.5 ms, 72 cores are needed to solve the QRDs in this implementation. The CPU design also limits the system designer to a very limited set of memory and external interface options.

Six CPU boards (three CPUs per board, four cores per CPU = 72 cores), plus one board for the beamformer master, yields a total of seven boards, each consuming ~200 watts — total power consumption: 1,400 watts. Of course, one could go to a higher CPU clock frequency (consuming even more power), but the CPU solution would still not outperform the FPGA solution. Figure 4 shows a CPU architecture that demonstrates the need for 18 CPUs.

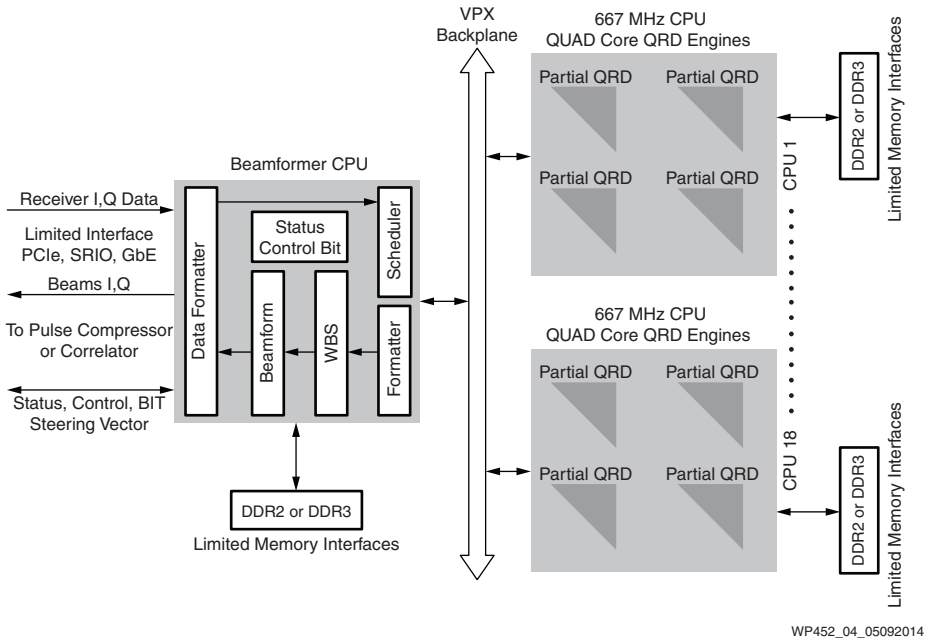

Figure 4: **Adaptive Beamformer Architecture Using 18 CPUs (72 Cores)**

Xilinx Single-FPGA Architecture

The Xilinx FPGA solution fits into one Xilinx Virtex®-7 FPGA very easily. The one-FPGA solution resides on a circuit board that also hosts external memory and other ancillary functions; total power consumption by the single board is about 75 watts. Compared to using a VPX chassis to host over 18 CPUs to solve the same problem (as just described in Multiple-CPU Architecture), it is clear that the CPU solution is no match for the FPGA in terms of power consumption.

The basic FPGA solution is shown in Figure 5; its implementation using Xilinx Vivado® HLS is discussed in greater detail in the section immediately following.

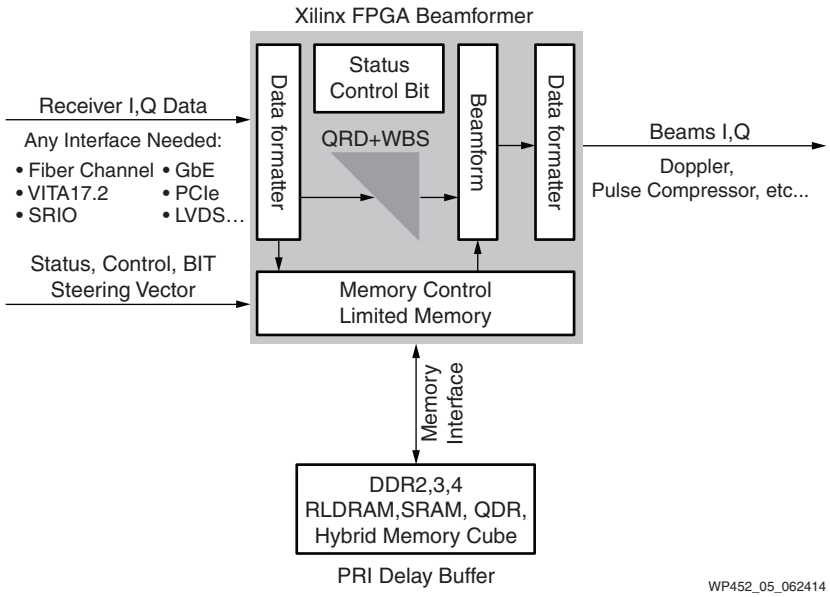

Figure 5: **Adaptive Beamformer Architecture Using a Single Virtex-7 FPGA**

Implementation of MGS QRD+WBS Using Vivado HLS

Vivado HLS allows any algorithm to be written in C/C++ or SystemC; it creates a flexible, portable, scalable core that can be used for the Xilinx Vivado tool flow targeting all Xilinx FPGAs. The design flow is shown in Figure 6.

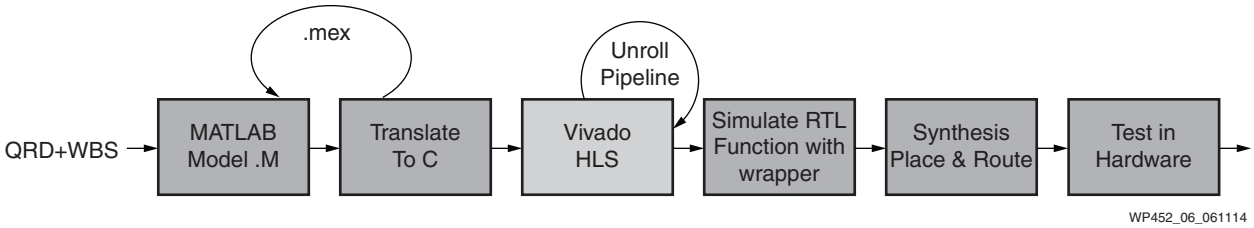

Figure 6: **Adaptive Beamformer Design Flow Using HLS**

Since MATLAB can compile C/C++, called a MEX (MATLAB Executable), the user can call the C/C++ code instead of the equivalent MATLAB function. That means there is only one master model and code, which greatly reduces the design for test and integration challenges. Design time, when compared to writing VHDL/Verilog code by hand, is reduced by orders of magnitude, normally from months to days. This reduction of design time comes from two areas:

- **The design's latency and math are verified with C/C++ simulation runs from an executable, which takes only a few seconds.** Gate-level RTL simulation is 10,000 times slower, and is iterative. Thus, errors are found, need to be corrected, and then re-simulated in RTL. This cycle can cause even the simplest designs to erode precious cost and schedule. The design from Vivado HLS is correct the first time. System integration time is accelerated, and design errors are caught much earlier in the design phase.
- **The design is captured in C/C++ or SystemC models; this means the design is always portable, flexible, and scalable.** The designer is not locked down to a particular FPGA or family. The user can immediately trade the FPGA space and see what an optimum fit for the design looks like without guessing, because the output of the HLS tool reports the FPGA's clock, latency, and resource usage. This feature set is most powerful when a company is competing in a Request for Proposal (RFP). Since the design is in C/C++, the user can even explore using a Zynq®-7000 AP SoC.

Initial Pass Results

The output from the Vivado HLS tool is displayed in just a few seconds, as shown in Figure 7:

•Summary of timing analysis

-Estimated clock period (ns): 6.65

•Summary of overall latency (clock cycles)

-Best-case latency: 3

-Average-case latency: 1746691

-Worst-case latency: 13466627

Area Estimates

•Summary

	BRAM_18K	DSP48	FF	LUT	SLICE
Component	_	24	2524	3607	_
Expression	_	_	0	1489	_
FIFO	_	_	_	_	_
Memory	72	_	0	0	_
Multiplexer	_	_	_	1918	_
Register	_	_	2337	_	_
ShiftMemory	_	_	_	_	_
Total	72	24	4861	7014	0
Available	2940	3600	866400	433200	108300
Utilization (%)	2	~0	~0	1	0

WP452_07_062414

Figure 7: **MGS QRD+WBS Vivado HLS First-Pass Results**

Note the clock period and worst-case latency. For this particular run, the MGS QRD+WBS has a result in approximately 10 ms. It uses DSP48s and block RAMs. For some radar systems, this latency might be acceptable. If faster solve times are needed, the power of directives are demonstrated in Results from Pass Using Directives.

Results from Pass Using Directives

The use of directives demonstrates the power of Vivado HLS. Simply unrolling some of the FOR loops in the C code by a factor of 16, partitioning the block RAMs, and using the PIPELINE directives, the designer can achieve astounding results, as shown in Figure 8. (As with any design, some of the code needs to be restructured to efficiently utilize all the power of the Vivado HLS tool. This would be the same if the designer were targeting many CPU/GPU cores.) Changing the matrix size is trivial using C/C++, but that is not the case if HDL is used as the design entry.

The solution also highlights the sheer DSP densities of Xilinx FPGAs. The MGS QRD + WBS only uses 392 DSP48 slices. That leaves plenty of room for the Adaptive Beamformer and the rest of the radar DSP processing, such as pulse-compression, doppler filtering, and Constant False Alarm Rate (CFAR).

•Summary of timing analysis

-Estimated clock period (ns): 6.79

•Summary of overall latency (clock cycles)

-Best-case latency: 3

-Average-case latency: 93027

-Worst-case latency: 420163

Area Estimates

•Summary

	BRAM_18K	DSP48	FF	LUT	SLICE
Component	_	392	44457	47081	_
Expression	_	_	0	6559	_
FIFO	_	_	_	_	_
Memory	128	_	0	0	_
Multiplexer	_	_	_	21984	_
Register	_	_	19130	_	_
ShiftMemory	_	_	0	276	_
Total	128	392	63587	75900	0
Available	2940	3600	866400	433200	108300
Utilization (%)	4	10	7	17	0

WP452_08_062414

Figure 8: **MGS QRD+QBS Vivado HLS Results Using Directives**

A partial C code example is shown in Figure 9 to highlight the coding style and directives. If this code were written in C++, the designer could use the built-in complex math libraries. When floating point is not needed, Vivado HLS supports fixed-point arithmetic as well. Since the design is in C/C++, changes to the matrix size are trivial. This is not true if coding is done by hand using VHDL/Verilog. Vivado HLS also supports a plethora of interfaces such as FIFO, block RAM, AXI, and various handshaking. The core produced also has a clock, synchronous reset, start, done, and idle signals for full core control.

```
loop10:for (kk=0; kk<row; kk++) {
    #pragma HLS UNROLL factor=16
    #pragma HLS PIPELINE
    #pragma HLS LOOP_TRIPCOUNT max=128

        qtr[kk%16] = Qr[kk][ii];
        qti[kk%16] = Qi[kk][ii];

        cmult(qtr[kk%16],-qti[kk%16],br[kk],bi[kk],&tr[kk%16],&ti[kk%16]);

        rtr[kk%16] = rtr[kk%16] + tr[kk%16];
        rti[kk%16] =  rti[kk%16] + ti[kk%16];
    }
```

WP452_09_061614

Figure 9: **C Code Example Showing Coding Style and Directives**

Results

The results from the Vivado HLS software targeting a Virtex-7 1140T FPGA are summarized in Table 2. The design is an MGS QRD+WBS core that can support variable rows up to 128 and variable columns up to 64.

Table 2: **Comparison of Virtex-7 FPGA and ARM9 CPU MGS QRD+WBS Solutions**

Parameter	Virtex-7 FPGA	ARM-A9
Programmable in C/C++/SystemC:	Y	Y
Flexible I/O	Y	N
HMC/JESD204b	Y	N
Clock (MHz)	125	667
Latency (ms)	3.3	250
System Power (watts)	75	1,400
System Cost	12.5X lower cost	

The system cost using the Virtex-7 FPGA is ~12.5X lower than the ARM-A9 CPU solution. As shown in Table 2, using the Xilinx Vivado HLS tools, the 128x64 problem was solved in 3.3 ms. The same code was run on an ARM Cortex-A9 at 667 MHz, and it produced a latency of 250 ms. This comparison highlights the fact that Xilinx has a complete programmable logic portfolio that meet a range of design needs, from a low latency solution to a highly integrated Zynq-7000 AP SoC solution using the ARM A9 processors. Using Vivado HLS allows the user to quickly trade the space of device, area, latency and clock frequency without running long RTL simulations. Using C/C++ allows the code to move seamlessly and agnostically from Xilinx programmable logic to a Zynq-7000 AP SoC.

This design used three directives: unrolling, pipelining, and block RAM partitioning. The design took approximately four hours to complete, including running many iterations of algorithm simulation against simulation data. Hand-coding a VHDL/Verilog version of this design and verifying it using RTL *could take weeks or even months*, depending on the skill of the designer.

Conclusion

This white paper has demonstrated that Xilinx FPGAs can be programmed in C/C++ using Vivado HLS, tackling one of the most complex challenges for any radar or wireless system: the complex matrix inversion using the MGS QRD+WBS. The benefits of parallel floating-point execution and native C language development are clear. The HLS advantage of native C design, yielding extremely rapid algorithm coding and testing, gives Xilinx FPGAs the performance and low-power advantage over all FPGA competitor and CPU/DSP/GPU options. The Xilinx 7 series FPGAs (and beyond), combined with advanced Vivado and HLS design flow, excels over the GPU, DSP, and CPU in every area. Readers are encouraged to prove this for themselves and become familiar with the Xilinx FPGA offerings, the Vivado tool flow, and Vivado HLS. For a free trial of Vivado HLS please go to:
http://www.xilinx.com/products/design_tools/vivado/vivado-webpack.htm

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
06/24/2014	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer