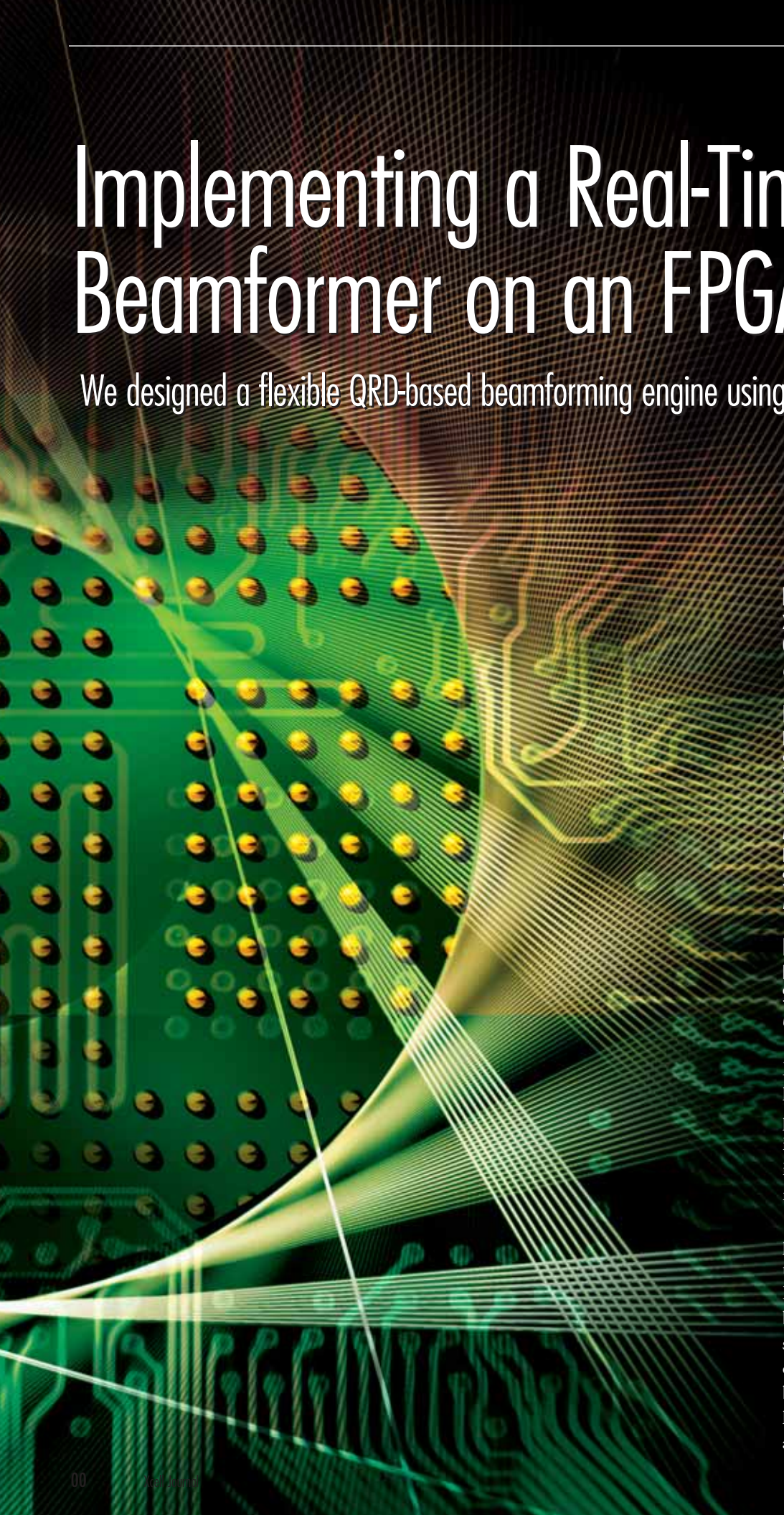
Implementing a Real-Time Beamformer on an FPGA Platform

We designed a flexible QRD-based beamforming engine using Xilinx System Generator.



by Chris Dick
Xilinx Chief DSP Architect
Company: Xilinx
chris.dick@xilinx.com

Fred Harris
Professor
San Diego State University
fred.harris@sdsu.edu

Miroslav Pajic
Engineer
Signum Concepts
miroslav.pajic@signumconcepts.com

Dragan Vuletic
Engineer
Signum Concepts
dragan.vuletic@signumconcepts.com

Most real-world communication systems have a mix of processing elements. For example, application programs, human/machine interface management, and higher networking protocol stack processing are best implemented on a general-purpose processor.

But for high-rate, algorithmically complex data processing – often with hard real-time deadlines – hardware resources like FPGAs are a better match. The interface between the two depends on the circumstance; the FPGA can be a pre-processor, coprocessor, post-processor, or some combination thereof. The trick is to get these heterogeneous systems to interoperate in an elegant fashion.

In this article, we'll describe the development of a flexible, optimized, adaptive beamforming engine that you can easily control through software. The DSP-intensive tasks run on the FPGA, while the command and control run on an external processor. The beamforming engine is a compact QR decomposition (QRD-based circuit) with a novel construction. The interface between the engine and the host processor is implemented by the shared memory abstraction in the Xilinx® System Generator design flow.

MVDR Beamformer

Adaptive beamforming is the application of adaptive filters to spatial signal processing. Time series collected from uniformly spaced array elements are weighted and summed to form a signal component from a selected direction of arrival while suppressing signal components from other directions of arrival (Figure 1). When the directions of arrival of the undesired signal components are unknown or vary with time, the filter weights must be adaptively adjusted to steer nulls to their directions. The adaptation process is performed subject to a constraint that the steering vector has unity gain in the signal direction. The steady state weights of such a beamformer form the minimum variance distortionless response (MVDR) from the array elements.

For reasons of numerical robustness and computational complexity, a common method for computing the required weight vector without directly inverting the correlation matrix is based on QR decomposition; this is the approach adopted here. For details of the procedure, consult "Adaptive Filter Theory" by Simon Haykin.

The QRD Matrix Inversion Process

The QRD process is formed by a sequence of two operators: the unitary rotations that convert complex input data to real data and associated angle-and-element combiners that individually annihilate the selected elements of the input data set. The QRD process is most compactly represented in Figure 2's signal flow diagram. This representation is the systolic array realization of the QRD least-squares solution processor.

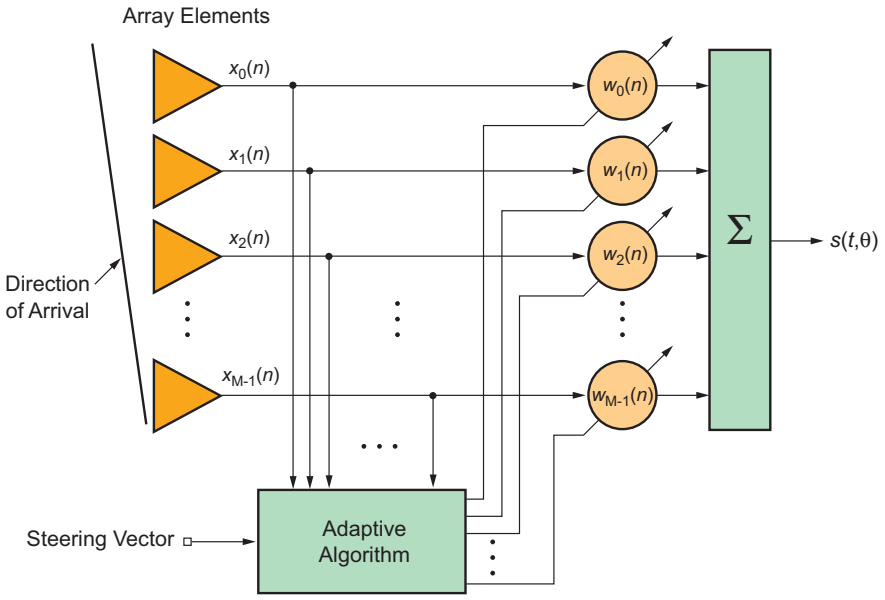


Figure 1 – Adaptive beamformer structure

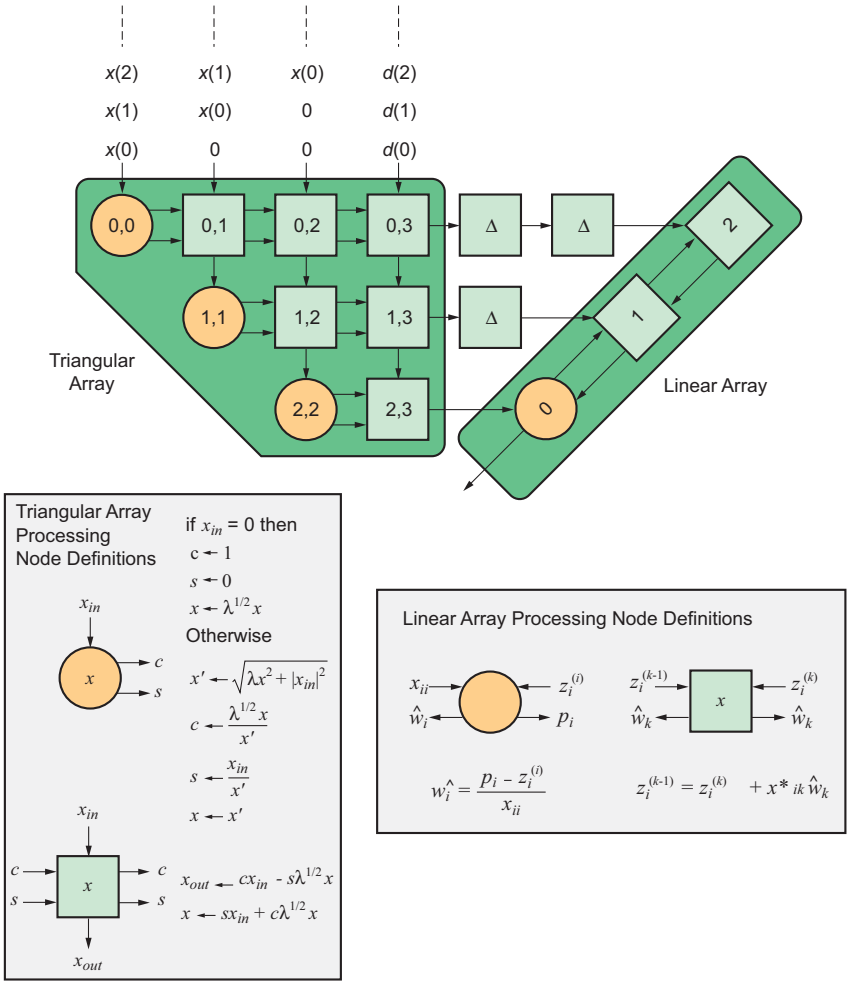


Figure 2 – Systolic array implementation of QRD matrix inversion for a 3 x 3 array

The array contains three types of processing cells: boundary cells, internal cells, and output cells. The boundary cells perform the "vectoring" operation on complex input samples to nullify their imaginary parts and form rotation angles used by internal cells. The internal cells perform Givens rotations of the input values by the angles passed from the boundary cells to annihilate the non-upper-triangular entries of the transformed data matrix. The output cells in the linear array process the elements of the upper triangular array to perform the required back substitution that produces the beamformer weights.

FPGA Implementation of QRD

Our goal was to produce a compact QRD FPGA implementation. The design comprises a single boundary, internal and back substitution cells. The systolic array in Figure 2 is folded onto this set of processing resources. The boundary cell is required to compute two angles. The first angle

$$\Phi = \arctan(\Im(x_{in})/\Re(x_{in}))$$

transforms the complex input samples presented to the boundary cell input port into real-valued data. The transformation that forces the imaginary component of xin to 0 must be applied to all elements in the same row associated with the boundary cell; this operation is one of the tasks performed by the internal cells. Now that the data in the leading position of two adjacent rows are real-valued, a second angle is formed as

$$\Theta = \arctan(x_{in}e^{-j\Phi}/x)$$

which is used to annihilate a term of the input data set in an ordered manner that eventually produces the upper-right triangular matrix R. The arithmetic employed in the boundary cells could be realized in hardware by literally implementing the equations indicated in Figure 2. This would require hardware support for performing square roots and divisions. Although these circuits are commonly implemented in FPGA hardware, we sought alternative methods for computing the required angles that had a lower resource cost over direct and obvious implementation.

One well-known and relatively simple method for computing angles is the vectoring mode of the Coordinate Rotation Digital Computer (CORDIC) algorithm. The CORDIC algorithm is an iterative procedure capable of computing a rich set of mathematical functions. The elemental operations required in the CORDIC algorithm are addition, subtraction, bit-shift, and table lookup. All of these functions are efficiently supported by FPGA architectures such as the Virtex™ series of devices from Xilinx, so the vectoring mode of the algorithm is a good candidate for the foundation of QRD processor boundary cells. As shown in Figure 3, two CORDIC engines are in use in the boundary cell: one for computing ϕ and the other for computing θ.

The CORDIC algorithm is iterative in nature, with each iteration refining the angle estimate by approximately 1 bit of precision. For a process employing an N-iteration CORDIC process, a new output is generated every N clock cycles and a new set of operands presented every N clock cycles. To increase the throughput of the boundary cell, we employed a fully parallel, or unrolled, architecture for the CORDIC (not shown here). After the initial start-up latency of the circuit is absorbed, the initi-

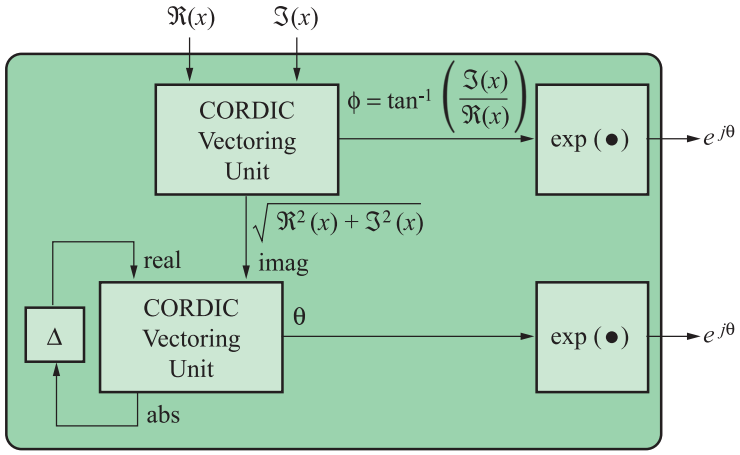


Figure 3 – Boundary cell architecture based on two vector-mode CORDIC processing engines

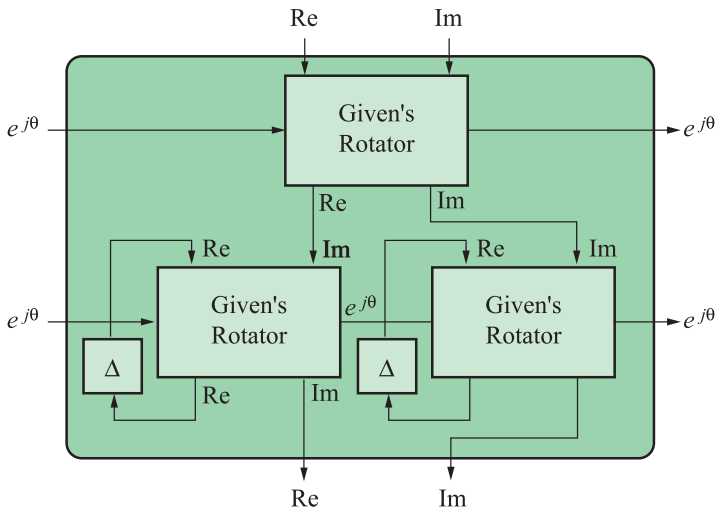


Figure 4 – Systolic array internal cell architecture employing three MAC-based Givens rotation engines

ation and completion rate of the cell is one new input/output per clock cycle.

Each data element x_{in} entering an internal cell (Figure 4) in row m must be rotated by the angle ϕ computed by the boundary cell for the m^{th} row:

$$\begin{bmatrix}\Re(\upsilon)\\ \Im(\upsilon)\end{bmatrix} = \begin{bmatrix}\cos(\phi) & -\sin(\theta)\\ \cos(\phi) & \sin(\theta)\end{bmatrix}\begin{bmatrix}\Re(x_{in})\\ \Im(x_{in})\end{bmatrix}$$

One option that has been commonly used for the rotation task in QRD processors is the rotation mode of the CORDIC algorithm. An alternative is to simply implement the rotation in the obvious manner using multiply accumulate (MAC) functional units; this is the approach adopted in our implementation. The target FPGA technology for the design is the Virtex-4 FPGA. These devices have a vast array of embedded MAC units referred to as DSP48 slices.

The DSP48 slice supports a rich set of opcodes – capable of being updated on a per-clock-cycle basis – that define the arithmetic operation computed by the tile during a given clock cycle. The four multiplications implied in the preceding equation are folded onto a pair of DSP48 slices, with each DSP48 slice computing one of the two output terms $\Re(\upsilon)$ and $\Im(\upsilon)$. Two clock cycles are required to compute the two output terms. Each DSP is supplied with a unique opcode for each clock period. Consider computing the term $\Re(\upsilon)$. During the first clock period, the product $\cos(\Phi)$ x $\Re(x_{in})$ is computed and stored in the DSP48 product or *p* register.

During the second clock cycle $\sin(\Phi)$ x $\Im(x_{in})$ is formed and subtracted from the value in the *p* register to generate the final output term. A similar sequence of computations is performed to produce $\Im(\upsilon)$. Using the DSP48 embedded blocks rather than a CORDIC-based approach for the internal cell reduces the latency of this phase of the computation and also minimizes the amount of FPGA logic fabric (look-up tables [LUTs] and registers) required for the implementation. Table 1 provides a breakdown of the area for the major functional units in the QRD implementation, along with the total area of the design.

The $\cos(\Phi)$, $\sin(\Phi)$, $\cos(\Theta)$, and $\sin(\Theta)$ terms required by the internal cells are computed using a simple LUT that maps the angles Φ and Θ computed by the vectoring units in the boundary cell to their corresponding sine and cosine. Linear interpolation is applied to the output samples of the LUT to increase the accuracy of the mapping from angle to amplitude, while keeping the LUT itself constrained to a single block RAM.

The row and column dimensions of the input array for the QRD processor can be dynamically adjusted at runtime by writing the new dimensions to control registers that are part of the FPGA control plane.

Table 2 provides timing information for several configurations of the input data set.

Functional Unit	LUTs	FFs	DSP48 Slices	Block RAM	Slices
Boundary Cell	2,145	2,057	3	1	1,266
Inner Cell	216	329	6	0	176
Back Substitution	2,862	3,286	4	1	1,932
QRD Total	5,411	5,916	13	6	3,530

Table 1 – FPGA resource utilization for folded QRD and back substitution array

M	N	Cycles for Triangularization	Cycles for Back Substitution	Total Cycles	Time (μs) for 250-MHz Clock
3	3	792	147	939	3.76
8	3	2,112	147	2,259	9.04
5	5	2,540	255	2,795	11.18
9	5	4,572	255	4,827	19.31
7	7	5,656	371	6,027	24.11
10	7	8,080	371	8,451	33.80
9	9	10,476	495	10,971	43.88
11	9	12,804	495	13,299	53.20
10	10	13,630	560	14,190	56.76

Table 2 – Execution time for the triangularization and back substitution phases of the FPGA QRD implementation for an M x N matrix.

Design Flow

Our QRD implementation uses the Xilinx System Generator for DSP model-based design flow. In addition to providing a natural development environment for developing FPGA signal-processing implementations, System Generator has a rich set of features that support the development of heterogeneous applications comprising not just the FPGA element but a processor. The processor could be the embedded PowerPC™ 405 hard IP block, the MicroBlaze™ soft-processor core, or a processor external to the FPGA.

The beamformer developed for this project was partitioned between the host PC and the FPGA platform. In our implementation the host application running on the PC might be considered more of an element of the beamformer verification process (test bench), but the host applica-

tion could be as arbitrary and complex as required by the task at hand.

Our beamformer host application is a MATLAB script (m-code) that simulates the sensor array for the beamforming network. The script simulates a dynamic target and generates the samples of the far-field radiation pattern for the moving target. The samples of the electric field at each sensor are generated in MATLAB and forwarded to the FPGA QRD processor. A new estimate of the beamformer weight vector is produced and returned to the MATLAB environment for further processing.

In this case, the additional processing involves plotting the polar radiation pattern for the updated complex valued weight vector. Note that the host application does not necessarily have to be associated with the MATLAB environment; the application could be a program written in C, for example.

An interesting element of the beamformer application is the management of the interface between the host application, running on a PC in this case, and the QRD process executing on the FPGA platform. System Generator provides a suite of shared memory library objects (ROM, RAM, FIFO) that abstracts virtually all of the details of the processor/FPGA interface and enables the host software and FPGA hardware to be somewhat insulated from each other (Figure 5).

Each new update of the beamformer is essentially a three-step procedure:

1. New input samples from each antenna element, as generated by the MATLAB host application, are forwarded to the QRD engine in situ on the FPGA.
2. The QRD process is triggered.
3. The new weight vector is returned from the FPGA to the host.

The shared memory library modules and associated application programmer interface transform the transfer of data between the FPGA and the host PC into simple assignment statements based on name/space references in MATLAB (or C). For example, the new weight vector w, resident in the MATLAB workspace, is updated with the new beamformer coefficients, FPGAWeights, as computed by the FPGA QRD process using the simple assignment w = FPGAWeights. (FPGAWeights is the name assigned to a shared-memory buffer in the System Generator description of the QRD engine.)

The management of this type of host processor/FPGA interaction by the System Generator framework makes the development of heterogeneous applications straightforward, rapid, less error-prone, and enables an FPGA accelerator engine (like the QRD module in this case) to be easily ported between different hardware platforms without needing to modify the FPGA source code – the System Generator model itself.

The interface abstraction supports transactions between the host application and the System Generator source model, as well as the host application and the final design running on the FPGA platform. This latter element significantly contributes to the validation process of the software and hardware (FPGA) dimensions of the system, as both components can be brought online rapidly using the shared memory abstraction.

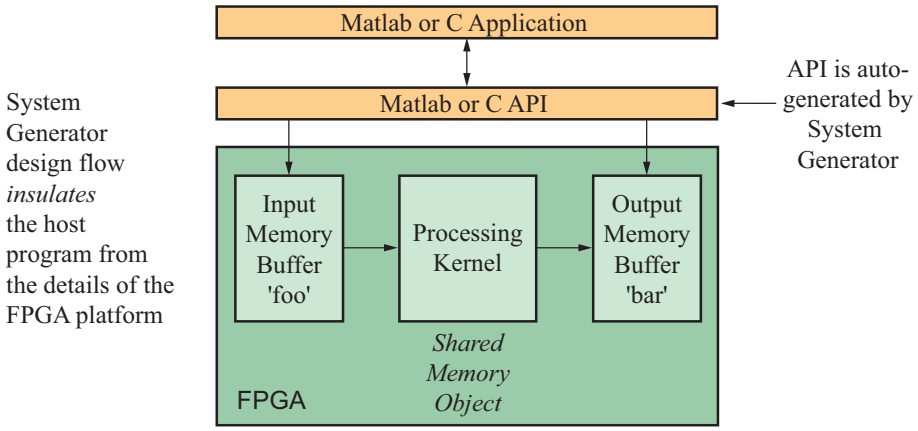


Figure 5 – Hardware/software abstraction enabled by the System Generator shared memory library elements. The host application performs transactions with the FPGA storage elements using simple name/space references – in this case to the memories named "foo" and "bar."

Conclusion

In this article, we've described the FPGA implementation of a flexible QRD processor that enables the run-time definition of the input matrix dimensions. The design employs a mixture of CORDIC-based processing (array boundary cell) and MAC-based (array internal cell) arithmetic that is well matched to the computational resources of an FPGA like the Xilinx Virtex-4 family.

All of the boundary- and internal-cell processing were projected onto a single boundary cell functional unit and internal cell functional unit; however, it should be noted that the abundant resources of FPGA platforms support the realization of a fully parallel systolic array, should the throughput requirements of the target application demand extremely high performance.

The System Generator programming environment enables the rapid development of heterogeneous systems (processors and FPGAs) while insulating programmers from the frequently complex and error-prone programming associated with hardware/software partitions.

This work was performed by the Xilinx Advanced Systems Technology Group (ASTG), the R&D organization within the Xilinx DSP Division, together with our partner organizations Signum Concepts and San Diego State University.