FPGA IMPLEMENTATION OF AN OFDM PHY

Chris Dick (Signal Processing Group, Xilinx Inc.San Jose USA, chris.dick@xilinx.com)
fred harris (CUBIC Signal Processing Chair, College of Engineering, San Diego State University, San Diego CA USA, fred.harris@sdsu.edu)

ABSTRACT

Orthogonal frequency division multiplexing (OFDM) based communication is increasingly being used in environments that exhibit severe multipath. While there are ASSP solutions for many common (e.g. 802.11a) and emerging standards, many communication systems, for example a military software radio, demand flexibility. The arithmetic requirements of an OFDM system can be very demanding. Even the ubiquitous 802.11a WLAN system has arithmetic requirements in the billions-of-operations per second region and cannot be satisfied even by high-end DSP microprocessors. This paper reports on the FPGA implementation of an OFDM transceiver. In addition to the FFT based modulator and demodulator, receiver synchronization and channel estimation is discussed. The FPGA resource requirements of the various sub-systems is reported and the design methodology employed for system design, verification and FPGA implementation is described.

1. INTRODUCTION

Orthogonal frequency division multiplexing (OFDM) is shaping up to be the communication technology of choice for many communication environments spanning the commercial and military sectors. However, even the garden variety 802.11a OFDM WLAN (wireless local area network) standard requires arithmetic resourcing that far exceeds state-of-the-art configurable DSP technologies like high-speed DSP processors. Field programmable gate arrays [3] (FPGAs), with their highly parallel architecture are able to capitalize on the inherent parallelism of the various algorithms that are used in communication technologies like OFDM. While device technology and intellectual property libraries are important for enabling high-performance and reduced product development cycles, increasingly, FPGA vendors are placing an emphasis on design flows that allow communication and signal processing engineers work in the language of the problem, rather than the language of the chip designer, e.g. VHDL or Verilog.

This paper provide a high-level overview of the FPGA implementation of certain aspects of OFDM physical layer processing. FPGA implementations of the modulator, demodulator, packet detector and fine timing estimation algorithms are described. The use of a high-level design tool called *System Generator for DSP*^{TM} [4] is highlighted during the course of the paper.

2. DESIGN FLOW

The Xilinx *System Generator*™ *for DSP* [4] tool suite was employed to implement the OFDM transceiver physical layer processing. System Generator is a visual dataflow design environment based on The Mathworks Simulink® [6] visual modeling tool set. This programming interface allows the system developer to work at a suitable level of abstraction from the target hardware platform, and use the same model not only for simulation and verification, but for FPGA implementation. System Generator blocks are bit- and cycle-true behavioral models of FPGA intellectual property components, or library elements. The library based approach results in design cycle compression in addition to generating area efficient high-performance circuits. Together with model features such as datatype propagation and the extensive virtual instruments that are part of the Simulink libraries, the environment facilitates rapid design space exploration together with powerful mechanisms for model debugging.

A large amount of arithmetic is performed in the process of acquiring and demodulating and OFDM symbol. Simulation time for this class of problem is an issue for conventional HDL (hardware description language) simulators as well as Simulink based design flows. To accelerate the simulation process, the

3. PHYSICAL LAYER SIGNAL PROCESSING

3.1. Modulator/De-modulator

The heart of an OFDM modulator and demodulator are the inverse FFT (IFFT) and FFT respectively. 802.11a WLAN systems employ a 64-point transform with 52 of the sub-carriers actually used for carrying user data from a BPSK, QPSK, 16-QAM or 64-QAM alphabet. The symbol rate for 802.11a systems is 20 MSym./sec. The OFDM symbol period is 4 μs., with 3.2 μs of this interval occupied by the 64-point FFT symbol and the additional 0.8 μs used for the cyclic prefix [1]. Among the many library elements in the

System Generator block set are multiple FFT implementations (System Generator version 6.1). A radix-4 based FFT was used for our implementation, this design requires 192 clock cycles to complete a transform – ignoring the cost of initializing the FFT datapath pipeline. While not essential, it is convenient to use an FPGA processing clock that is an integer multiple of the symbol rate (20 MHz). A 100 MHz master clock was selected, which results in a 64-point (I)FFT transform time of 1.92 µs, which is well within the requirements of a transform completion rate of 1 transform every 4 µs. There is actually ample time for the one FFT engine to be time division multiplexed between the OFDM transmitter and receiver. Simulation of our OFDM transceiver was performed at baseband. In a complete implementation the OFDM data would typically be up-sampled and transposed to a digital IF (intermediate frequency). The frequency domain input data to the modulator (IFFT) was supplied in a bursty manner (synchronous with the 100 MHz clock), and the resulting time series from the transform was similarly generated in a bursty fashion. The FFT symbol time-series was delivered to on-chip dual-port block memory [3]. One memory port is synchronized with the IFFT result bus, with the second port running at the 20 MHz symbol rate. The cyclic prefix is inserted by virtue of a simple address sequencer that reads out the final 25% (16 samples) of the FFT symbol and pre-appends this data to the 64 element FFT symbol to generate an 80-sample sequence that is delivered to the channel. The transmitter memory is double buffered in order to support simultaneous data transmission and IFFT operation.

3.2. Synchronization

There are many challenging synchronization tasks to address in an OFDM-based communication system. In fact, this is frequently the aspect of the system that distinguishes implementations, and the algorithms involved are more often than not proprietary in nature. Prior to performing channel estimation equalization and demodulation, OFDM symbol timing must be acquired. The approach to timing estimation will be different for broadcast and packet switched networks. Here we will consider a random access packet switched system similar to that employed in 802.11a networks.

The receiver does not know when a packet starts, and so the first synchronization task is packet detection. Once a packet has been detected the remaining synchronization functions include course and fine timing recovery and carrier recovery.

Figure 1 shows the structure of the IEEE 802.11a standard preamble. The 10 short preambles (A1-A10) are identical 16-sample duration sequences. The cyclic prefix (CP) is a 32-sample sequence and the long preambles (C1 and C2) are identical 64-sample sequences. As indicated in the figure, the various fields are used for packet detection, automatic gain control (AGC), diversity selection, coarse and fine frequency offset estimation, fine symbol timing estimation and channel estimation.

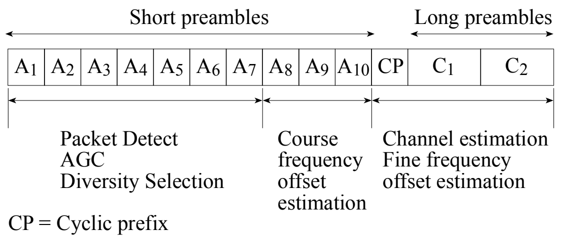
Short preambles
Long preambles
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 CP C1 C2
Packet Detect
AGC
Diversity Selection
Course frequency offset estimation
Channel estimation
Fine frequency offset estimation
CP = Cyclic prefix

Figure 1: IEEE 802.11a standard preamble.

3.2.1 Packet Detection

The packet detector is based on the Schimdl and Cox [2] delay and correlate algorithm commonly used for acquiring symbol timing. As shown in The decision statistic is computed as

$$M(n) = \frac{|P(n)|^2}{(R(n))^2} \tag{3}$$

, the algorithm is essentially a sliding window correlator combined with an energy detector used to normalize the decision statistic and hence guard against fluctuations of the input signal power level.

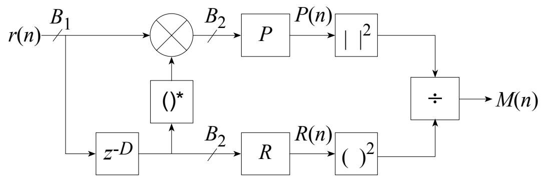

Figure 2: Schimdl and Cox delay and correlate algorithm.

The sliding window P computes a cross-correlation between the input signal and a version of the input signal delayed in time by one short preamble interval - $D = 16$ samples in this case. The second sliding window R is used to compute the received signal energy in the cross-correlation interval. The cross-correlation $P(n)$ and autocorrelation $R(n)$ are calculated according to Eq. (1) and Eq. (2) respectively.

$$P(n) = \sum_{m=0}^{L-1} r_{n+m} r^*_{n+m+D} \tag{1}$$

$$R(n) = \sum_{m=0}^{L-1} r_{n+m+D} r^*_{n+m+D} \tag{2}$$

The decision statistic is computed as

$$M(n) = \frac{|P(n)|^2}{(R(n))^2} \tag{3}$$

FPGA architectural features like the *shift register logic 16* (SRL16) primitive found in the Virtex-II and Virtex-II Pro [3] series of Xilinx devices contribute to producing an efficient and compact FPGA implementation. Figure 3 provides a high-level view of basic component that is used to construct an FPGA – the logic slice [3]. There are many tens-of-thousands of these elemental units available in a single device. Without going into details, the slice basically consists of two lookup-tables (LUTs), two flip-flops (FFs), and additional circuitry for performing high-speed arithmetic. The LUTs are multi-functional components that can be used for computing logic equations, configured as user-application 16x1 RAM or ROM (referred to as *distributed memory*), or used as SRL16 elements. All of these modes are extremely useful for signal processing applications.

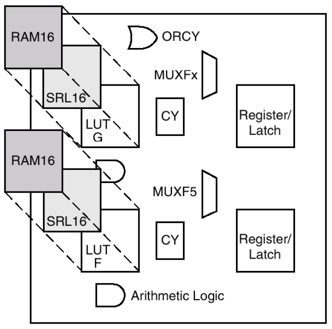

Figure 3: Virtex-II (Pro) logic slice – high-level view.

Functionally, the SRL16 can be viewed as a series arrangement of 16 flip-flops with a dynamically programmable tap point, as shown in Figure 4.

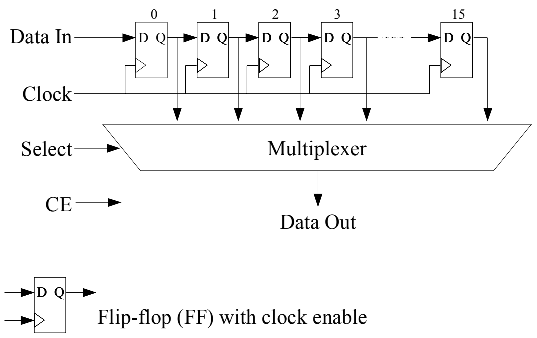

Figure 4: Functional view of the *SRL16* LUT configuration.

The RAM and ROM configurations can be used for storing filter coefficients or data vectors in a signal processing system. The utility and versatility of the SRL16 configuration may not immediately be obvious with respect to signal processing applications, but this unique aspect of Xilinx FPGAs is extremely powerful for building very efficient time-division multiplexed hardware that, for example, can be used to process multiple channels of data. An example is the pulse shaping and up-sampling of the in-phase (I) and quadrature (Q) components of the baseband signal in a transmitter shaping filter or receiver matched filter.

As highlighted in [2] $P(n)$ and $R(n)$ can be calculated iteratively. It is useful to observe that the cascaded integrator comb (CIC) filter shown in conveniently implements the iterations.

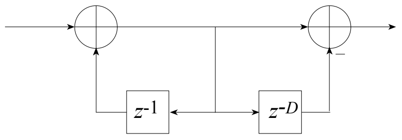

Figure 5: CIC filter used for computing the $P(n)$ and $R(n)$.

SRL16s were used extensively for implementing the packet detector. A delay equal to one short preamble (16 samples in this case) is required to compute the cross-correlation (Figure 2). The CIC filters for computing $P(n)$ and $R(n)$ similarly require a 16-sample delay in the differentiator section of these filters. Using the node precisions indicated in Figure 2, and recalling that that the input sequence is complex valued, $2 \times D \times B_1 + 4 \times D \times B_2$ bits of storage are needed. An obvious implementation might use the slice FFs to resource this storage. In our implementation $D = 16$, $B_1 = 16$ and $B_2 = 16$, so 1536 FFs would be required support the delays. This equates to 1536/2=768 logic slices. If the SRL16 configuration of a LUT is engaged, the slice requirement is reduced to 48, which is 6.25% the area of a flip-flop based implementation. The SRL16 LUT configuration is easily targeted from hardware description languages like VHDL and Verilog, in addition to the visual programming environment *System Generator for DSP* as the *addressable shift register* library component.

To compute the decision statistic $M(n)$ (Eq. (3)) a division is required. There are many techniques for implementing a divider, in our implementation the linear mode of the CORDIC algorithm [5] was employed. The procedure for computing y_0 / x_0 is outlined in Eq. (4).

Given an iteration counter i, the operand register y and an additional state register z are updated at each iteration using conditional additions (subtractions) and logical shifts. These functions are particularly simple and compact to implement in an FPGA. Each iteration contributes approximately one additional bit of precision to the result.

Thus the approach permits a simple mechanism for making a tradeoff between hardware cost and numerical precision.

$$
\begin{aligned}
& i = 0 \\
& d_i = \mathrm{sgn}(y) \\
& x_{i+1} = x_i \\
& y_{i+1} = y_i - d_i x_i 2^{-i} \\
& z_{i+1} = z_i + d_i 2^{-i} \\
& i = i + 1
\end{aligned}
\qquad (4)
$$

As highlighted in Section 2, System Generator is a particularly productive design environment that, amongst other things, enables rapid design exploration. Key nodes in the design can be monitored and analyzed (with Matlab m-file scripts for example) to compute performance metrics that determine if the design satisfies specified requirements.

The ability to invoke Matlab functions at various stages of a simulation is extremely powerful. For example, matlab functions can be invoked to define the precision of a node in the signal flowgraph based on parameters in the Matlab workspace. This approach could be used to correctly size the accumulator in a filter to preclude arithmetic overflow based on the integration interval, regressor vector precision and the filter coefficient precision. Datatype propagation in System Generator can also be considered a mechanism for modifying model characteristics in response to performance (design) requirements. Matlab also provides mechanisms to modify the *structure* of a model in response to system parameters.

While the Simulink graphical block editor is commonly used for schematic capture of a system, it appears less widely appreciated that Simulink models can be constructed *programmatically* through a Matlab API that supports block and signal instantiation, customization, deletion, and other construction methods [6]. It is particularly productive and efficient to use the Matlab API to customize Simulink models in situ. Simulink supports block-specific callback functions during model initialization, the start of simulation, at every simulation step, and when parameters are changed (there are in fact many others). By judicious invocation of the Matlab API, the topology of a Simulink model can be customized during the initialization of a subsystem. This allows the user to customize a model in ways normally considered impossible in a graphical environment.

This approach was used for the realizing the CORDIC divider in the packet detector, making it particularly simple to perform rapid design iterations based on re-definitions of every aspect of the packet detector. In fact, the entire packet detector is easily defined is a masked-subsystem that permits a graphical interface to specify construction of the ocomplete module. Of course this same approach is also useful in other parts of the system, for example, the long preamble correlator. Figures 6 and 7 show two examples of the CORDIC divider with 7 and 10 iterations respectively.

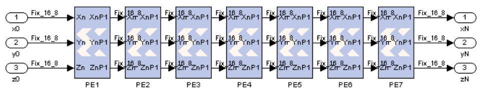

Figure 6: 7-PE CORDIC divider implemented in System Generator. The Matlab API is used to construct the graph at model execution time.

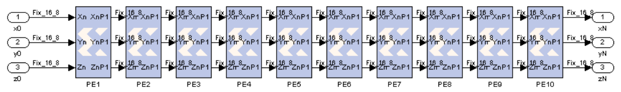

Figure 7: 10-PE CORDIC divider implemented in System Generator. The Matlab API is used to construct the graph at model execution time.

To move between the two models does not require the manual addition (deletion) of arcs or blocks, the complete sub-system is generated using the Matlab API referenced earlier. The point here of course is that this mechanism is extremely valuable in terms of accelerating design turns. Another, probably obvious, use of the approach is to produce modules that are parameterized in every sense of the term, which in turn can be considered a useful method for incorporating design reuse within or across organizations.

Using 16-bit input samples, and maintaining 16-bit precision at all nodes within the correlator, the FPGA implementation consumes 12 embedded multipliers and 462 logic slices.

3.2.2 Long Preamble Correlator

In WLAN systems the preamble is known at the receiver. This allows the use of a simple cross-correlation algorithm for acquiring symbol timing. After the packet detector has provided an estimate of the starting time for an OFDM transmission, the symbol timing can be resolved to sample-level precision by cross-correlating between the received sequence and a local version of the preamble. In our implementation a cross-correlation is performed between the received signal and the long preamble sequence. Using a signaling rate of 20 MHz, and recalling that both the received signal and long preamble are complex valued time-series, the arithmetic requirements to support the correlator is a little over 5 MOPs, where a MOP is assumed here to include all of the operations for computing one output sample – data addressing and arithmetic (multiply-accumulate (MAC)). To place this value in context, it is almost all of the real-time cycles of a state-of-the art instruction-set based DSP microprocessor: and we have not even begun to consider resourcing the carrier recovery, channel estimation and equalization, demodulation, sample clock frequency compensation and forward error correction components of the receiver.

One useful observation that can be exploited to implement a compact FPGA implementation (small slice count, minimal number of embedded multipliers) is that the cross-correlation can be performed using the sign of both the input sequence and the locally stored reference template. That is, a *clipped cross-correlator* is more than adequate for acquiring symbol timing. This completely removes the need for using any of the FPGA embedded multipliers in this case. Figures 8(a) and (b) provide a comparison between a full-precision and clipped correlator implementation respectively. The correlation peaks, indicating the two long pre-ambles are clearly evident in both cases. Of course the hardware cost of the clipped correlator is substantially less than that of the full-precision correlator.

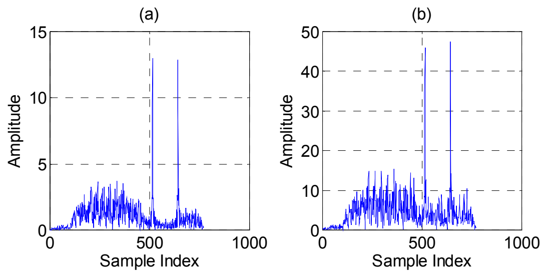

Figure 8: Long preamble correlator output: (a) Full precision datapath. (b) Clipped correlator using only sign of the input samples and 1-bit reference template.

The long preambles $C1$ and $C2$ are identical 64-sample sequences. While the data is presented to the long preamble correlator at a rate of 20 MHz, it is useful to run the correlator itself at the 100 MHz clock that is available in the receiver – recall that the FFT used in the demodulator is clocked at this higher rate. The correlator is decomposed into a number of shorter length sub-correlations, with the output of each of these processing elements (PEs) combined using a binary tree to form the final result. Each sub-correlator PE will be responsible for computing 100/20=5 terms of the final result. This requires $\lceil 64/5 \rceil = 13$ such PEs. The correlation PE is shown in Figure 9. Distributed memory is used to store the sign of 5 elements of the long preamble. During each 50 ns interval a 5 term inner-product is computed between 5 1-bit precision correlation coefficients read from memory and 5 2-bit regressor vector elements stored in an SRL16-based shift register. Two bits are required to represent ± 1 using a two's complement representation. The 1-bit precision correlation coefficients are effectively encoded in the control-plane of the correlator PE since they are directly connected to *add/sub* control port of the accumulator (de-cumulator) in the processing engine.

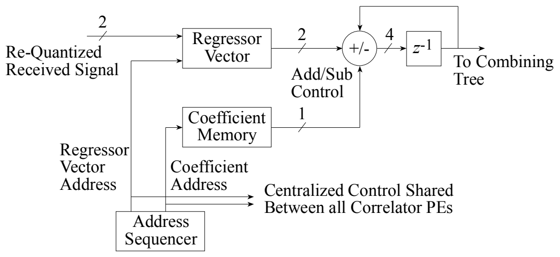

Figure 9: One cell of the long preamble correlator.

It is interesting to note that were it not possible to employ a clipped correlator in the implementation $4 \times 13 = 46$ (a non trivial number) embedded multipliers would have been required. This is an example of one of the many value propositions of FPGA signal processing: datapath *right sizing*. In other configurable signal processing technologies, like DSP processors, every aspect of the datapath is pre-determined at device fabrication time. This includes the type, number and connectivity of the functional units. If the native datapath of the processor is, say, 32-bits, 32-bits will be used to implement the equivalent of 1- or 2-bit arithmetic as is the case in our current example. The FPGA approach to computing is like having a desktop silicon foundry with a turn around time measured in minutes or hours instead of months or years as it is for many complex ASICs and DSP processors.

Once final performance metric to comment on relates to memory bandwidth. One high-level view of FPGA signal processing is as a technology that is essentially a highly parallel memory array with distributed processing resources – or vice-versa. The on-chip memory bandwidth can reach many tera-bytes/second in some cases. The long preamble correlator is a simple example of a parallel memory subsystem at work. During each processing clock interval (100 MHz) each correlator PE reads 3-bits of information – the 1-bit reference template sample and the 2-bit re-quantized input sample stream. There are 13 such PEs operating concurrently in each of 4 correlator segments (complex data, complex template). Therefore, the read memory bandwidth is $4 \times 3 \times 13 \times 100e^{6} = 15.6$ Giga-bits/second or 1.95 Giga-bytes/second. And this is achieved with an effective FPGA memory footprint of about 100 slices – and remember, there are many tens-of-thousands of slices in a current generation FPGA.

The complete long preamble correlator consumes 1100 slices and two embedded multipliers. The two multipliers are used to compute the magnitude-squared of the complex correlation sequence whose output is subsequently processed by a peak detector.

The 100 MHz processing clock used here has its roots in the earlier decisions on the OFDM signaling rates (20 Msym./sec.) and single butterfly (I)FFT architecture. In fact, the correlator will support a clock frequency of 200 MHz. This is equivalent to a computation rate of 10.4 GOPs/sec. Here of course an "OP" is referenced to the elemental computation of the clipped correlator PE.

3.3. Channel Estimation and Equalization

802.11a compliant WLAN systems are burst communication systems. The preamble sequence is used not only to detect a transmission and acquire timing as discussed earlier, but it is used as a channel probe. The channel is assumed to remain static over the whole burst, so that once the channel is estimated, the inverse of the channel response can be employed in the receiver for channel equalization. If required, certain sub-carriers may be reserved as pilot channels that carry specially constructed sequences that permit the receiver to track the channel in addition to other time varying parameters such as carrier frequency offset.

The received signal after the FFT in the demodulator is

$$Y(k) = C(k)X(k) + Z(k) \tag{5}$$

where k is the sub-carrier index, and $C(k), X(k)$ and $Z(k)$ $k = 0,\ldots,63,$ are the complex channel response, transform of the long preamble sequence and noise vectors respectively. Since the long preamble is known at the receiver, a simple channel estimate can be obtained as

$$\hat{C}(k) = \frac{Y(k)}{X(k)} \tag{6}$$

Since the two long preambles are identical, averaging can be used to reduce the error in the estimate. Of course the averaging can be performed (average of *C1* and *C*2) before the demodulation process since the DFT (discrete Fourier transform) is a linear operation. The long preamble is designed as a wideband signal that will excite the channel across all frequencies of interest. A frequency domain representation of the long preamble is stored locally in block memory [3] at the receiver. Both the numerator and denominator in Eq. (6) are complex valued quantities. Using simple arithmetic, the channel estimate $\hat{C}(k)$ can be computed as a real scalar divided by a complex value. This is turn is implemented in hardware as two real-valued divisions. The divisions were implemented using the same CORDIC divider structure described earlier for packet detection. Once the channel estimate, or inverse more precisely, has been computed it is stored in memory and applied in the frequency domain via a complex multiplication to each element of the demodulated (FFT output) data sequence. This complex product is implemented in the obvious manner using the embedded multipliers in the Virtex-II (Pro) FPGAs.

The channel estimator and frequency domain equalizer occupy 776 logic slices, 2 block memories and 10 embedded multipliers

4. CONCLUSION

FPGA signal processing offers system designers many new and exciting possibilities for implementing signal processing based applications like communication systems. This technology enables the construction of a datapath that precisely matches the computation and memory access requirements of an algorithm. In addition, and unlike other configurable technologies such as DSP processors, only the correct, or minimum, number of bits are used to represent signals at each point in the computation graph – so called *datapath right sizing*. Since the device personality is held as a configuration *bitstream* in static RAM (SRAM), modifications, functional extensions and bug fixes can be easily applied – even after the system has been deployed in the field. For example, a network like the Internet could be employed to supply new FPGA configuration data to remote equipment.

This paper has described the implementation of several aspects of an OFDM PHY which could be considered as one personality of either a commercial or military software defined radio. In addition to describing the signal processing algorithms in the PHY, the design flow employed to produce the implementation, *System Generator for DSP*, was highlighted. It is estimated that using this approach to implementation reduced the design, development and verification process from many months down to a few short weeks.

5. REFERENCES

[1] J. Heiskala and J. Terry, *OFDM LANS: A Theoretical and Practical Guide,* Sams Publishing, 2002.

[2] T. M. Schmidl and D. C. Cox, "Low-Overhead, Low Complexity [Burst] Synchronization for OFDM," *IEEE International Conference on Communications,* Vol. 3., pp. 1301-1306, 1996.

[3] Xilinx Inc., Virtex-II Pro Platform FPGAs, http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs

[4] Xilinx Inc., System Generator for DSP, http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=system_generato

[5] Yu Hen Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing", *IEEE Signal Processing Magazine*, pp. 16-35, July 1992.

[6] The Mathworks, Inc., *Using Simulink*, 2002.