Software-Programmable Digital Predistortion on the Zynq SoC

Xilinx’s Zynq All Programmable SoC and Vivado HLS tool provide an efficient and flexible way to implement wireless digital front-end applications.

by Baris Ozgul
Research Scientist
Xilinx, Dublin, Ireland
baris.ozgul@xilinx.com

Jan Langer
Research Scientist
Xilinx, Dublin, Ireland
jan.langer@xilinx.com

Juanjo Noguera
Research Scientist
Xilinx, Dublin, Ireland
juanjo.noguera@xilinx.com

Kees Vissers
Distinguished Engineer
Xilinx, Inc.
kees.vissers@xilinx.com
Digital predistortion (DPD) is an advanced digital signal-processing technique that mitigates the effects of power amplifier (PA) nonlinearity in wireless transmitters. DPD plays a key role in providing efficient radio digital front-end (DFE) solutions for 3G/4G basestations and beyond. A generic DPD system consists of a predistorter that compensates for the nonlinearity effects prior to the input of the PA, and an estimator on the feedback path from the output of the PA, which updates the predistorter coefficients to reflect the possible changes in operational characteristics. Based on the modulation type, power amplifier technology and transmission bandwidth, the DPD solution can differ from system to system. Hence, it is worthwhile to provide a flexible design methodology for DFEs that facilitates the implementation and integration of new DPD coefficient estimation algorithms.

Modern FPGAs are a promising target platform for the implementation of flexible wireless DFE solutions, including DPD. The Xilinx® Zynq™-7000 All Programmable SoC, which integrates the programmable-logic fabric with a multicore ARM® processor system, provides an especially interesting new option. The Zynq-7000 All Programmable SoC enables the partitioning of functionality among hardware and software components to increase the overall system performance. As shown in Figure 1, this All Programmable SoC is capable of implementing all the required functions of a DFE in a single chip.

Our team created a software-programmable design flow to implement the DPD coefficient estimators on a Zynq-7000 device using a methodology that allows the flexible partitioning of the functionality among the hardware and software components, depending on the complexity of the estimation algorithm in use. We achieved a significant productivity increase thanks to the Vivado™ high-level synthesis (HLS) tool, which allowed the implementation and verification of our hardware design at the software level.

SOFTWARE-PROGRAMMABLE DESIGN FLOW ON ZYNQ

The Zynq-7000 SoC is a hybrid computing platform that consists of two major parts. First, there are two embedded ARM Cortex™-A9 processors operating at up to 1 GHz and their support infrastructure, including a cache hierarchy, memory controllers and I/O peripherals. This section in itself represents a complete programmable embedded platform that is fit for use without any FPGA programming. The two ARM processor cores come with a single-instruction, multiple-data (SIMD) extension called NEON that provides a 128-bit-wide data path.

Second, the Zynq-7000 devices also contain an area of programmable logic that represents a conventional FPGA. The major advantage of this device is the high bandwidth available between the FPGA and the embedded processors by means of a multitude of AXI4 communication ports. In this way, it is possible to develop software for the processor and, as necessary, offload compute-intensive tasks into the programmable logic.

Figure 1 – Programmable wireless digital front end on the Zynq All Programmable SoC
Figure 2 illustrates a short overview of our proposed design flow. The first step is no different from a typical software design flow. It involves implementing the application in pure software, where the results can be tested and verified thoroughly.

Next, a profiling step reveals the bottlenecks of the application, such as compute-intensive subfunctions that need hardware acceleration. This so-called hardware/software partitioning involves not only the selection of functions to accelerate, but also the decision on an adequate communication infrastructure such as DMA transfers vs. memory mapping. Additionally, some changes to the software are necessary in order to call the hardware accelerator instead of the original C function.

In a traditional design flow, an experienced hardware designer implements the functions to be accelerated in a hardware description language like VHDL or Verilog. With the availability of the Xilinx Vivado HLS tool, we replaced the manual hardware implementation with an automatic step that uses the original software functions to generate corresponding hardware accelerators. The conversion requires several incremental manual refinement steps that include adding directives to the code or even restructuring the algorithm in order to obtain an efficient hardware implementation. During this process, the code is still executable as software, so that the designer can use the original test and verification environment. This is a big advantage over the traditional hardware design flow.

After the completion of the refinement step, the Vivado HLS tool generates a hardware accelerator implementation that meets the design constraints—for example, the required clock frequency and the amount of hardware resources. Next, the integration step requires the instantiation of communication components (for example, DMA) to enable the interaction between the hardware accelerator and the processor. Finally, system synthesis generates a bitstream to program the programmable logic.

HIGH-LEVEL SYNTHESIS FOR PROGRAMMABLE LOGIC

HLS tools raise the level of abstraction for designs in the programmable logic, and make the time-consuming and error-prone register-transfer-level (RTL) design tasks transparent. These tools take as their input a high-level description of the specific algorithm to implement and generate the RTL design for the target hardware accelerator.

Modern HLS tools accept as their input untimed C/C++ descriptions, from which they interpret the sequential semantics of the input/output behavior and the architecture specifications. Based on the C/C++ code, compiler directives and target throughput requirements, these tools generate high-performance pipelined architectures. Furthermore, they enable automatic pipeline stage insertion and resource sharing to reduce hardware resource utilization.

We have adopted the overall hardware design flow seen in Figure 3. The first step in this flow is restructuring a reference C/C++ code that the designer could have derived from a MATLAB® functional description. Here, restructuring means doing modifications in the original code (which is typically coded for clarity and ease of conceptual understanding rather than for optimized performance) to turn it into a format more suitable for the target processing engine. This is similar to rearranging an application’s code to have more efficient performance on a DSP processor. When targeting FPGAs, this restructuring might involve, for example, rewriting the code so it represents an architecture specification that can achieve the desired throughput, or rewriting to make efficient use of specific FPGA features such as embedded DSP macros.

The functional verification of the implementation code uses traditional C/C++ compilers (gcc, for example) and reuses C/C++ level testbenches...
developed for the verification of the reference code. In addition to the implementation code, constraints and compiler directives (e.g., pragmas inserted in the code) are the other important input of the HLS tool. Two essential constraints are the target FPGA family (that is, the technology) and the target clock frequency. Naturally, both of these factors will have an effect on the number of pipeline stages in the generated architecture. The designer can apply different types of directives to different sections of the code. For example, there are directives for loop unrolling. As another example, there are directives to limit the instances of specific functions or operations in order to minimize the corresponding FPGA resource utilization.

The HLS tool takes all these inputs (the implementation C/C++ code, constraints and directives) to generate an RTL output and to report the throughput of the generated architecture. If the generated architecture does not meet the required throughput, you can modify the implementation C/C++ code or the directives, or both. If the architecture meets the required throughput, then you can use the RTL output as the input to the Xilinx Vivado or ISE®/EDK tools. The reporting of final achievable clock frequency and number of FPGA resources used occurs only after you have run logic synthesis and place-and-route. If the design does not meet timing or the FPGA resources are not as expected, you should modify the implementation C/C++ code or the compiler directives.

It is worth noting that this is an iterative design flow, so the implementation code can go through different types of restructuring until the design requirements are met. A key concept to keep in mind is that the C-level verification infrastructure is reused to verify any change to the implementation. In this way, you do not carry out the verification at the register-transfer level, avoiding time-consuming RTL simulation and hence, contributing to a reduction in the overall development time.
SYSTEM MODEL FOR DPD

High peak-to-average power ratio (PAPR) is a major problem of the non-constant envelope signals (for example, wideband code-division multiple access and orthogonal frequency-division multiple access signals) widely adopted in 3G/4G and emerging wireless systems. Due to high PAPR and PA nonlinearity, the transmitted signals get distorted during transmission. This distortion typically results in a growth of out-of-band spurious emissions. A straightforward solution to this problem is to back off the PA input so as to keep it in the linear operating range of the PA. However, the main disadvantage of this approach is the inefficient use of the PA, which results in a higher cost than required for the same output power. Another solution is to use digital predistortion. DPD negates the nonlinearity effects of the PA and increases efficiency.

As Figure 4 shows, a DPD system consists of a predistorter employed prior to the amplification and a parameter estimator on the feedback path from the output of the PA. (Please note that this illustration is an algorithmic view, which excludes the digital-to-analog and analog-to-digital converters at the PA input and output, respectively, as well as the RF circuitry in between.)

The parameter estimator computes the coefficients of the predistorter based on the samples of the PA input and output. To separate the PA behavior from the additional analog hardware effects, the PA output \( y_0 \) is aligned prior to the parameter estimation. The aligned PA output \( y \) matches the amplitude, delay and phase variations of \( z \).

The predistorter and parameter estimator rely on a memory model that is used to describe the nonlinearity effects of the PA. For wideband DPD applications, it is quite common to employ models based on the Volterra series, which is widely used for the approximation of nonlinear systems.

We consider the efficient implementation of alignment and a least-square (LS) estimator in Figure 4. The DPD system employs these blocks to update the predistorter coefficients when there are major changes in the signal characteristics or power dynamics. The autocorrelation matrix computation (AMC) block in Figure 4 is the most computationally complex function in our LS estimator design. It relies on a Volterra-series-based model and generates the inputs to the coefficient compute (CC) block, which estimates the predistorter coefficients.

Unlike the predistorter filter in hardware, the alignment and the LS estimator blocks do not operate at the sample rate, since they are not in use unless the DPD system updates the predistorter coefficients. Here, our main goal is to reduce the overall coefficient update time. In this way, the DPD solution reacts faster to changing conditions, leading to more-effective predistortion correction. Furthermore, faster updates enable the support of more-complex DPD solutions, using a larger number of active coefficients. With shorter update times, it is also possible to run the same design multiple times in a serial fashion, in order to update predistorter coefficients of different data paths. This approach makes it possible to implement efficient DPD solutions for multiantenna basestations.

Let’s take a closer look at the details of our software-programmable design flow, which facilitates the implementation of efficient DPD coefficient update solutions for modern wireless transmitters.

SOFTWARE IMPLEMENTATION FOR DIGITAL PREDISTORTION

During the software development process, we used a test environment that reads the \( z \) and \( y \) samples in...
Figure 4 from a reference vector and writes a set of coefficients. Subsequently, we compared the coefficients to a reference implementation written in MATLAB that visualizes the difference between both sets of coefficients. We performed the software profiling in two levels. First, we ran the software on a standard x86 server and obtained the profiling results, instrumented the subfunctions of interest with calls to the global CPU timer of gprof software profiling tool, indicating that the AMC block is the bottleneck of the application. It consumes 97 percent of the overall update time, making it a prime candidate for hardware acceleration.

Prior to profiling, we expected the solver used for coefficient computation in Figure 4 would consume a larger part of the update time, because in contrast to the other functionality it was performing double-precision floating-point operations. However, the ARM’s floating-point unit solved the task very efficiently.

Before actually implementing a hardware accelerator for the AMC block, we examined potential software optimization possibilities. The SIMD NEON engine of the ARM processor has a 128-bit-wide data path. Since the AMC algorithm works on 64-bit fixed-point data types, the NEON engine can carry out two parallel computations, as Figure 6 illustrates. Instead of using low-level assembly instructions to access the NEON engine, the compiler provides a set of function-like wrappers for the instructions. These wrappers, which are called intrinsics, provide type-safe operations, while allowing the compiler to automatically schedule the C variables to NEON registers.

Applying the intrinsics in the C code results in a speed-up factor of two. Furthermore, during the NEON operations, the normal ARM processor is free and can continue processing simple non-NEON instructions like loop conditions and pointer increments, while the NEON engine runs in parallel.

**HARDWARE IMPLEMENTATION FOR DIGITAL PREDISTORTION**

To improve the overall parameter update time, we implemented an AMC accelerator using the Vivado HLS tool based on the design flow in Figure 3. Our accelerator’s programmable configurations support a number of different predistorter coefficients and allow the flexible selection of nonlinear terms in the Volterra-series-based model. Hence, it is possible to support several DPD configurations using the same AMC accelerator. In addition, you can make new changes in the existing C++ code and in the compiler directives to generate a brand-new accelerator in a much shorter time than if you were doing a hand-coded RTL design. Let’s take a closer look at some specific examples of code rewriting and compiler directives that we used for the AMC accelerator.
It’s possible to rewrite the C/C++ code to more efficiently utilize specific FPGA resources and, hence, improve timing and reduce area. There are two very specific examples of this type of optimization: bit-width optimizations and efficient use of embedded DSP blocks (DSP48s). For example, the standard approach uses built-in C/C++ data types (such as short, int) in the reference C/C++ code, whereas the actual design may require fixed-point data types having word lengths that are not integer multiples of the byte size. Here, the Vivado HLS tool supports C++ template classes that can represent integer data types with an arbitrary bit width. For our AMC accelerator, we leveraged these template classes, hence reducing FPGA resources and minimizing the impact on timing.

The snippet of C++ code in Figure 7 is a good example of code rewriting, bit-width optimization and the efficient use of DSP48s. The example focuses on complex multiplication, which we widely used in our AMC accelerator. A standard complex multiplication carries out four real multiplications, and requires the use of four different multipliers in a fully pipelined implementation. However, we show in Figure 7 that we can achieve the equivalent functionality by rewriting this code to use three multipliers (employing fewer DSP48 blocks), at the expense of three additional pre-adders and a 1-bit increase in the multiplier word length. In this example, the Vivado HLS tool generates three 33 x 32-bit multipliers, each giving a 65-bit result.

The original reference C++ code for the complex multiplication uses four multipliers, each multiplying two 32-bit numbers. Please note that the original code uses built-in C/C++ data types and the 32-bit inputs should be cast to 64-bit integer to avoid loss of information (because the result is a 64-bit integer). However, based on this code, the HLS tool generates four 64 x 64-bit multipliers, which are clearly much more expensive in terms of DSP48s. On the other hand, by using the C++ template classes in Figure 7 for the data types, the C++ code functionality works fine without any casting, while the Vivado HLS tool generates only three 33 x 32-bit multipliers, each using four DSP48s.

Our main reason for implementing an AMC accelerator was to reduce the time that it took to run the AMC algorithm, and hence to improve the total time to update predistorter coefficients. For this purpose, we pipelined the loops in the C++ code and also applied loop unrolling when possible. Using the Vivado HLS tool, we verified that the most time-consuming loop in our implementation was the matrix computation loop. We unrolled this loop by a configurable factor that we predefined as a C macro in the compiler options. Depending on the unrolling factor, the designer can choose between better resource sharing and shorter computation time.

Figure 8 shows how we employed the configurable unrolling factor (UNROLL_FACTOR) and the Vivado HLS directives for loop unrolling at the software level. In the matrix computation loop, we called the CMULT32 function in Figure 7, using three multipliers for complex multiplication. In Figure 8, the HLS directive on line 3 enables the
Performance results corroborate that our software-programmable design flow allows the implementation of several DPD designs, trading off faster predistorter coefficient update times.

sharing of 3xUNROLL_FACTOR multipliers in the case of loop parallelization. The directive on line 8 partitions the array that stores the matrix values by the unrolling factor, to avoid parallel access problems. Lines 15 and 16 show how we applied the loop pipelining and unrolling directives, respectively. Furthermore, Vivado HLS allows you to specify the resource to implement a variable in RTL. For example, we mapped the Marray variable to dual-port RAM on line 11.

INTEGRATING HARDWARE AND SOFTWARE COMPONENTS

The communication protocol between the processor system and hardware accelerator is the Advanced eXtensible Interface (AXI). The second and most-recent version of AXI is AXI4. Our design uses the Xilinx AXI interconnect to transfer data from the Zynq processing system (ARMs) to the hardware accelerator. Given the bandwidth requirements of this application, we don’t need DMA transfers. As Figure 9 shows, we used the AXI interconnect core to link AXI4-Lite masters to slaves. AXI4-Lite is a lightweight, single-transaction memory-mapped interface. When connected to AXI4-Lite slaves, the AXI interconnect core stores the transaction IDs and restores them in the response transfers. Furthermore, it controls the transactions and does not propagate any illegal transaction to the AXI4-Lite slave.

The AXI FIFOs in Figure 9 are for the input and output data samples of the accelerator, which are using AXI4-Stream interfaces of the accelerator. The AXI4-Lite slave on the accelerator is for the AMC configuration parameters. We generated all the streaming and AXI4-Lite interfaces for the accelerator at the software level, using Vivado HLS.

PERFORMANCE RESULTS ON ZYNQ-7000 SOC

Figure 10 illustrates the total DPD coefficient update time at different design stages. Our target was the Zynq-7000 SoC ZC706 board using a Zynq 7045 device. The target clock frequency that we used in Vivado HLS to generate the AMC accelerator was 250 MHz. However, it is possible to increase the target clock frequency constraint in Vivado HLS in order to generate faster accelerators. Our implementation with Vivado HLS is quite efficient, resulting in 3 percent area utilization.

In Figure 10, the update time for the software-only solution using the original code is around 1.250 seconds, with the AMC as the main bottleneck for the application. After some code optimization for the AMC, we obtained a speed-up factor of about 2. NEON optimizations added another speed-up factor of 2. After accelerating the AMC in the programmable logic, we achieved a 70x speed-up for this block. We have tested our designs successfully on the target device.

Our accelerated design will be preferable in the high-complexity DPD applications going forward. For example, in a multi-antenna basestation, it is possible to run an accelerated design more than once in a serial fashion (computing predistorter coefficients for a different antenna each time). The accelerated design becomes more feasible if the number of predistorter coefficients increases, as well. For low-complexity DPD applications, our software-only design using NEON optimizations can be sufficient.

Performance results corroborate that our software-programmable design flow allows the implementation of several DPD designs, trading off faster predistorter coefficient update times. Our design flow supports the flexible partitioning of functionality among hardware/software components and facilitates the implementation of efficient DPD solutions for modern wireless transmitters.

We would like to thank the Xilinx DPD Engineering Team for their valuable comments and cooperation.

First Quarter 2013 Xcell Journal