SOFTWARE-PROGRAMMABLE DIGITAL PRE-DISTORTION ON NEW GENERATION FPGAS

Baris Ozgul (Xilinx, Dublin, Ireland; baris.ozgul@xilinx.com); Jan Langer (Xilinx, Dublin, Ireland; jan.langer@xilinx.com); Juanjo Noguera (Xilinx, Dublin, Ireland; juanjo.noguera@xilinx.com); Kees Vissers (Xilinx, San Jose, CA, USA; kees.vissers@xilinx.com)

ABSTRACT

In this paper we present a software programmable design flow that facilitates the implementation and integration of efficient digital pre-distortion (DPD) solutions on the leading-edge FPGAs. In addition to software programmability, another key contribution of this design flow is the flexible partitioning of functionality among the hardware and software components, depending on the complexity of the DPD parameter estimation algorithm in use. We have applied ARM-specific optimizations to the software implementation and used Vivado High-Level Synthesis (HLS) tool as the design tool for the programmable logic. We present a comprehensive study reporting the overall system performance when exploring the partitioning of the functionality among hardware and software. For low-complexity algorithms, we show that a software-only solution is applicable after carrying out the ARM-specific optimizations. For higher-complexity algorithms, we use Vivado HLS to accelerate the time-consuming blocks in the programmable logic, leading to a 5X speed up in the overall algorithm execution time.

1. INTRODUCTION

In third and fourth generation (3G/4G) wireless systems and beyond, the application of techniques such as non-constant envelope modulations, MIMO processing and carrier aggregation plays a key role in meeting the target requirements for the spectral efficiency, bit-error rate (BER), cell capacity and throughput [1],[2]. However, such techniques also result in many practical challenges in the air interface, which require the use of more sophisticated and flexible digital radio front-end (DFE) architectures in the wireless base-station. For example, one of the major issues in practice is the high peak-to-average power ratio (PAPR) of non-constant envelope signals [3]. Due to high PAPR and power amplifier (PA) non-linearity, the transmitted signals get distorted. The distortion typically results in a growth in the out-of-band (OOB) power of the signal, causing adjacent channel interference, and increases the BER.

Digital Pre-Distortion (DPD) is an advanced signal processing technique which mitigates the signal distortion mentioned above by inverting the non-linearity effects of the PA [3]. A generic DPD system consists of a pre-distorter 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. The estimator updates the pre-distorter parameters to reflect the possible changes in the operation characteristics. Based on the modulation type, power amplifier technology and transmission bandwidth, the effective DPD solution can differ. Hence, it is worthwhile to provide a flexible design methodology for DFEs that facilitates the implementation and integration of new DPD parameter estimation algorithms.

Modern Field Programmable Gate Arrays (FPGAs) are a promising target platform for the implementation of a flexible architecture for efficient DPD solutions. Furthermore, there are several studies showing that FPGAs could achieve 100X higher performance and 30X better cost-performance than traditional DSP processors in several signal processing applications [4]. However, the key barrier for the widespread adoption of FPGAs in signal processing algorithms was the traditional hardware-centric design-flow and tools. That is, the traditional use of FPGAs requires significant hardware design experience. Recently, new generation FPGAs that integrate the programmable logic fabric with industry-standard embedded processors have become available [5]. These leading-edge platforms enable the partitioning of functionality among hardware and software components to increase the overall system performance. Furthermore, high-level synthesis (HLS) tools have become available as the design tools for FPGAs, which increase the design productivity and reduce the development time, while producing very competitive Quality of Results (QoR) [4].

This paper describes a flexible and software-programmable design flow for the implementation of the DPD parameter estimation algorithms on the leading-edge FPGAs. The two key contributions of this methodology are: 1) flexible partitioning of the functionality among the hardware and software components, depending on the complexity of the algorithm in use, 2) productivity increase

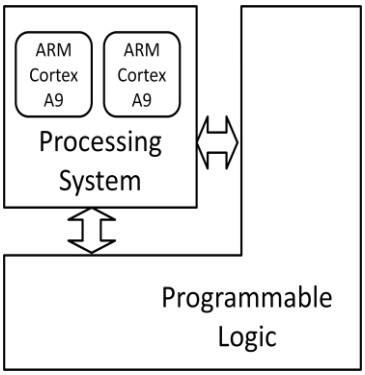

Figure 1: Programmable Zynq platform

thanks to the design tools that allows the implementation and verification of the design at the software level. We have employed the 128-bit single instruction multiple data (SIMD) extension for the ARM processors, called NEON, to optimize the software implementation. Furthermore, we have used Vivado HLS, formerly called AutoESL, as the design tool for the programmable logic.

The remainder of this paper is organized as follows. In Section 2, we give information about our target FPGA platform [5] and describe our software-programmable design flow. In Section 3, we present the system model for a generic DPD solution and specify the parts to be implemented for DPD parameter estimation. The details of our software and hardware implementations and the related optimizations are explained in Section 4. In Section 5, we present a comprehensive evaluation of the overall system performance when exploring the partitioning of functionality among the hardware and software components. It is shown that our software-only solution is able to support low-complexity DPD parameter estimation algorithms. For higher-complexity algorithms, our flexible design flow allows the hardware acceleration of time-consuming blocks in the programmable logic, where we use the Vivado HLS tool to generate the necessary hardware accelerators. The conclusions can be found in Section 6.

2. TARGET PLATFORM AND SOFTWARE-PROGRAMMABLE DESING FLOW

2.1 Software Design Flow on the Target Platform

In recent years, embedded processors and programmable logic devices merged into one chip, enabling better interaction between the two and subsequently a finer grained partitioning between hardware and software. As a recent example of such a platform, the Zynq platform from Xilinx [5] is a hybrid computing platform that is shown in a simplified version in Figure 1. It consists of two major parts. First, there are two embedded ARM Cortex A9 processors and their support infrastructure, including a cache hierarchy,

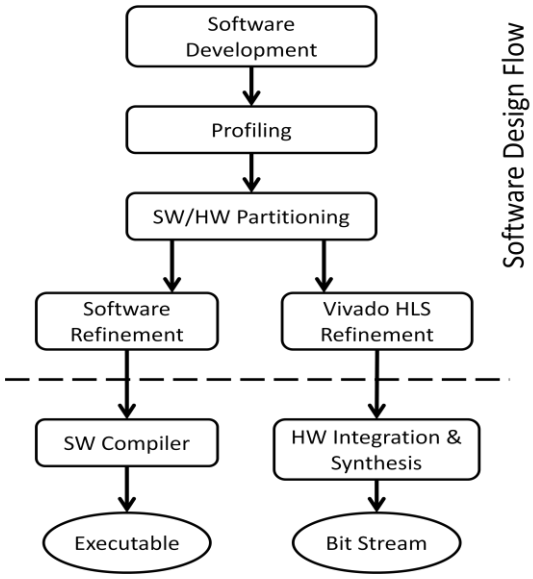

Figure 2: Design Flow

memory controllers and I/O peripherals. This in itself represents a complete programmable embedded platform that can be used without any FPGA programming. The two ARM processor cores come with a SIMD extension called NEON that provides a 128-bit wide data path.

Secondly, the Zynq devices contain area of programable logic. This area represents a conventional FPGA. The major advantage of this platform is the close interaction between FPGA and embedded processors by means of a multitude of AXI4 communication ports. This way, it is possible to develop software for the processor and as necessary off-load compute-intensive tasks into the FPGA logic.

The following paragraph gives a short overview of our proposed design flow for Zynq as illustrated in Figure 2. The first step is no different from a typical software design flow and consists of the implementation of the application in pure software. The result of this task can be thoroughly verified and tested. Next, a profiling step can reveal the bottlenecks of the application, e.g. compute-intensive sub-functions that need hardware acceleration.

This so called hardware/software partitioning involves not only the selection of functions that should be accelerated but also the decision on an adequate communication infrastructure such as DMA transfers versus memory mapping or the order of the data packets to be transferred. Additionally, some changes to the software are necessary in order to call the hardware accelerator instead of the original C function. The whole integration step involves some manual work, but is to a large extent assisted by the tools of the design flow.

In a traditional design flow those functions are implemented in a hardware description language like VHDL

or Verilog by an experienced hardware designer.With the availability of industrial HLS tools like Xilinx Vivado HLS the manual hardware implementation can be replaced by 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 will still be executable as software, so that the original test and verification environment can be used. This is a big advantage over the traditional hardware design flow.

After the refinement is completed, the HLS tool can generate a hardware accelerator implementation that matches the design constraints, e.g. the required clock frequency and the amount of hardware resources. The accelerator design flow using the HLS tools is explained in more detail in Section 2.2.

Next, the integration step requires the instantiation of communication devices that are needed to enable the interaction between the hardware accelerator and the processor. Finally, a system synthesis step generates a bit stream to program the FPGA logic.

2.2 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 *untimed* C/C++ descriptions as their input, 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.

The overall hardware design flow adopted in this paper is shown in Figure 3. The first step in this flow is *restructuring* a reference C/C++ code which could have been derived from a MATLAB functional description. Here, restructuring means doing modifications in the original code 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. The *functional* verification of the implementation code is using traditional C/C++ compilers (e.g., gcc) and reusing C/C++ level test benches developed for the verification of the reference code. In addition to the implementation code, *constraints* and compiler *directives*

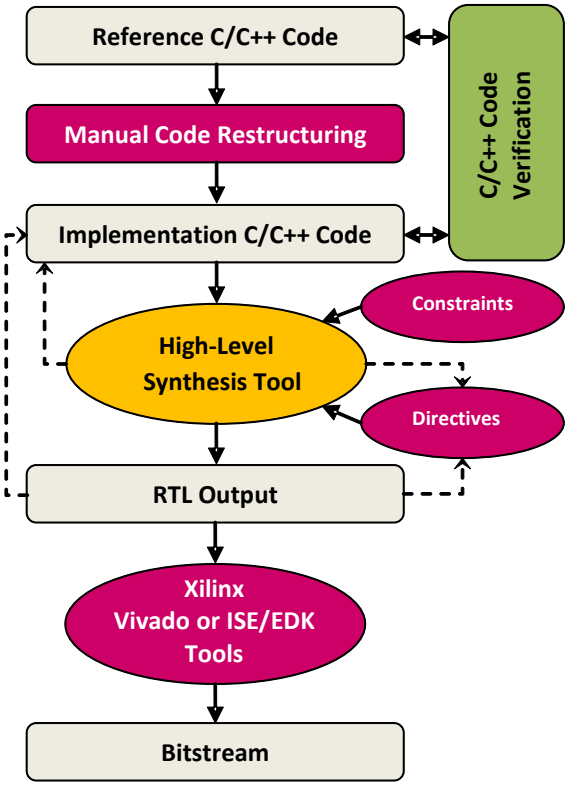

Figure 3: High-level Synthesis for Programmable Logic

(e.g., *pragmas* inserted in the code) are the other important input of the HLS tool. Two essential constraints are the target FPGA family (i.e., technology) and target clock frequency, which obviously have an effect on the number of pipeline stages in the generated architecture. Different types of directives can be applied to different sections of the code. For example, there are directives that are applied to loops (e.g., loop unrolling), while other directives can be applied to arrays (e.g., to partition an array into smaller arrays based on the unrolling requirements). 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 tools take all these inputs (i.e., the implementation C/C++ code, constraints and directives) to generate an RTL output and to report the throughput of the generated architecture. If the required throughput is not met, the designer can modify the implementation C/C++ code and/or the directives. If the generated architecture meets the required throughput, then the RTL output is used as the input to the Xilinx Vivado or ISE/EDK tools. The final achievable clock frequency and number of FPGA resources used is reported only after running logic synthesis and place&route. If the design does not meet timing or the FPGA resources are not as expected, the designer should

modify the implementation C/C++ code and/or the compiler directives. It is worth noting that this is an iterative design flow where the implementation code can go through different types of code restructuring until the design requirements are met. A key concept to keep in mind is that the C-level verification infrastructure is re-used to verify any change to the implementation. In this way, verification is not carried out at the RTL level, avoiding time-consuming RTL simulation and hence, contributing to the reduction in the overall development time.

3. SYSTEM MODEL FOR DPD

High PAPR is a major problem of the non-constant envelope signals (e.g., wideband code division multiple access and orthogonal frequency division multiple access signals), which are widely adopted in 3G/4G and emerging wireless systems. Due to high PAPR and PA nonlinearity, the transmitted signals get distorted in practice. This distortion typically results in a growth of the 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 higher cost than required for the same output power. Another solution is using DPD, which negates the nonlinearity effects of the PA and increases the efficiency.

As shown in Figure 4, the DPD system consists of a *pre-distorter* employed prior to the amplification and a *parameter estimator* on the feedback path from the output of the PA. Please note that the illustration in Figure 4 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 pre-distorter based on the samples of the PA input and output. In order to separate the PA behavior from the additional analog hardware effects, the PA output $y_o(n)$ is *aligned* prior to the parameter estimation. The aligned PA output $y(n)$ matches the amplitude, delay and phase variations of $z(n)$.

The pre-distorter and parameter estimator rely on a memory model that is used to describe the non-linearity effects of the PA. For wideband DPD applications, it is quite common to employ *Volterra series* based models [3]. The most general form of non-linearity with M-tap memory is represented using Volterra series as

$$z(n) = \sum_{k=1}^{K} z_k(n) \quad (1)$$

where

$$z_k(n) = \sum_{m_1=0}^{M-1} \cdots \sum_{m_k=0}^{M-1} h_k(m_1, \cdots, m_k) \prod_{l=1}^{k} y(n-m_l) \quad (2)$$

is the contribution of the kth order Volterra kernel h_k and the input $y(n)$. As shown in (1) and (2), the Volterra series method results in a very complex expression. In order to have a more practical representation, simpler models have been derived by selecting only some of the Volterra products in (2). For example, the *memory polynomial* model is one of the well-known simplified models [3], denoted as

$$z(n) = \sum_{k=0}^{K-1} \sum_{m=0}^{M-1} a_{km}\, y(n-m)|y(n-m)|^k \quad (3)$$

where all the non-diagonal Volterra series terms are set to zero and the non-zero coefficients for the diagonal terms are represented by a_{km}. In (3), the second input term depends only on the magnitude of the signal $|y(n-m)|$, where k denotes the magnitude order. The model in (3) conforms to the boundary conditions since it reduces to a linear time-invariant system when the signal magnitude is small. PAs are also linear for small signals.

The memory polynomial method has been proven to effectively model the actual PAs under typical operating conditions. Nevertheless there have been more generalized models derived to achieve even better performance [3]. The following model includes both diagonal and off-diagonal terms:

$$z(n) = \sum_{k\in K_d} \sum_{m\in M_d} a_{km}\; y(n-m)|y(n-m)|^k + \sum_{k\in K_o} \sum_{m\in M_o} \sum_{r\in R_o} \hat{a}_{kmr}\; y(n-m)|y(n-m-r)|^k \quad (4)$$

where K_d an M_d are the index arrays for the diagonal terms, and K_o, M_o and R_o are the index arrays for the off-diagonal terms composed of a signal and a lagging magnitude. Index arrays allow the selection of the delay taps and the magnitude powers over a given range; rather than implementing all the delay taps and powers as in (3), requiring all the coefficients a_{km}. In vector form, equation (4) can be rewritten as

$$z(n) = \mathbf{U}_n\, \mathbf{A} \quad (5)$$

where $\mathbf{A}$ is the parameter vector which incorporates the active diagonal and off-diagonal coefficients a_{km} and $\hat{a}_{kmr}$, and $\mathbf{U}_n$ is a row vector of active diagonal and off-diagonal terms of signal and magnitude products in (4).

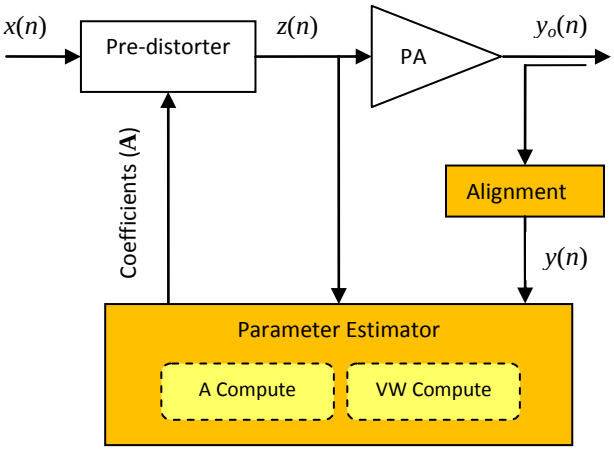

Figure 4: Algorithmic view of DPD

The parameter estimator finds the coefficients in **A**, which are then used by the pre-distorter. The PA inputs $z(n)$ and the aligned PA outputs $y(n)$ are captured to be able to estimate the coefficients. After capturing L samples of z(n) and L + additional samples of $y(n)$ (number of additional samples is based on the delay range adopted in (4)), the relation in (5) can be expressed in matrix form as

$$\mathbf{Z} = \mathbf{UA} \tag{6}$$

where $\mathbf{Z} = [z(n)\ z(n-1) \cdots z(n-L+1)]^T$, **A** is the vector to be estimated, $\mathbf{U} = [\mathbf{U}_n^T\ \ \mathbf{U}_{n-1}^T\ \cdots\ \mathbf{U}_{n-L+1}^T]^T$, $\mathbf{U}_n$ is the row vector in (5) and $(.)^T$ is the transpose operation. The least squares solution to (6) can be found by multiplying each side with $\mathbf{U}^H$ (i.e., the conjugate transpose of **U**), leading to

$$\mathbf{W} = \mathbf{VA} \tag{7}$$

where $\mathbf{W} = \mathbf{U}^H\mathbf{Z}$ and $\mathbf{V} = \mathbf{U}^H\mathbf{U}$. Given that the number of active coefficients is N_A, **W** is a vector of (N_A x 1) and **V** is a matrix of size (N_A x N_A). The solution to (7) can be found as

$$\mathbf{A} = \mathbf{V}^{-1}\mathbf{W}. \tag{8}$$

The *VW computation* module in Figure 4 is for the computation of **V** and **W** in (7). The solution in (8) is found by the *A computation* module.

In this paper, we consider the efficient implementation of alignment and parameter estimation blocks (i.e., *colored* blocks in Figure 4). In a DPD system, these blocks are employed to update the pre-distorter coefficients when there are major changes in the signal characteristics and/or power dynamics. Hence, unlike the pre-distorter, they do not operate at the sample rate. Here our main goal is to reduce the overall *coefficient update time*. In this way, the DPD solution reacts faster to the changing conditions, leading to more effective pre-distortion correction. Furthermore, faster updates enable the support of more complex DPD solutions, using larger number of active coefficients N_A. With shorter update times, it is also possible to run the same design multiple times in a serial fashion, in order to update pre-distorter coefficients of different data paths. This approach allows the implementation of efficient DPD solutions for multi-antenna base-stations.

In the next section, we give the details of our software programmable design flow, which facilitates the implementation of efficient DPD coefficient update solutions for modern wireless transmitters.

4. SOFTWARE AND HARDWARE IMPLEMENTATION FOR DPD

4.1 Software Implementation for DPD

The DPD coefficient update software was targeted as a stand-alone solution running on one of the two ARM processors at 800 MHz. Without an operating system, the application has direct access to all hardware devices. This enables a very accurate profiling step with deterministic results. In this subsection, we first give an overview over the test environment we employed. Then we present the profiling step and the conclusions evolved from that and finally we propose a software optimization based on the ARM's NEON engine that can significantly reduce the update time.

During the software development process a test environment has been used that reads the *z(n)* and *y(n)* samples from a reference vector and writes a set of coefficients. Subsequently, the coefficients are compared to a reference implementation written in Matlab that visualizes the difference between both sets of coefficients. The software profiling has been performed on two levels. First, the software was run on a standard x86 server and gprof was used to get a first estimate of the expected bottlenecks. Second, the software ran on the ARM processor and, depending on the gprof results, the interesting sub-functions have been instrumented with calls to the global CPU timer of the Zynq platform. This timer runs at half the CPU frequency, hence giving excellent resolution with the overhead of only a few cycles.

Table 1: Results of initial profiling

Function	Time	%
Alignment	**29.62ms**	**7.9**
VW computation	**336.40ms**	**89.9**
A computation	**8.09ms**	**2.2**

The profiling results of the three main function blocks running on the ARM processor are given in Table 1. From

the table it can be seen that the VW computation is the bottleneck of the application. It consumes 90% of the overall update time, making it a prime candidate for hardware acceleration. Before profiling, we expected the solver used for A computation in Figure 4 would consume a larger part of the update time, because in contrast to the other functionality it is performed in double precision floating point. However, the ARM's floating point unit solved the task very efficiently.

Before actually implementing a hardware accelerator for the VW calculation, potential software optimization possibilities have been examined. The SIMD NEON engine of the ARM processor has a 128-bit wide data path. Since the VW computation works on 64-bit fixed-point data types, two parallel computations can be carried out in parallel. In Figure 5, one step of the VW, the computation of the W vector is presented in both pure C code and with the ARM NEON instructions applied. 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 are called *intrinsics* and they provide type-safe operations, while allowing the compiler to automatically schedule the C variables to NEON registers. Furthermore, every compiler for ARM processors is required to define the macro __ARM_NEON__ in case the target machine has a NEON engine. That way, a single source code can be used for both NEON and non-NEON implementations.

```
#ifndef __ARM_NEON__
void computeW (CINT64 *W,
               CINT32 const *u,
               CINT16 s, int N_A)
{
  for (int i = 0; i < N_A; ++i) {
    // conjugated complex multiplication
    W[i].real += (int64_t)u[i].real*s.real
               + (int64_t)u[i].imag*s.imag;
    W[i].imag += (int64_t)u[i].real*s.imag
               - (int64_t)u[i].imag*s.real;
  }
}

#else // defined __ARM_NEON__
void computeW (CINT64 *W,
               CINT32 const *u,
               CINT16 s, int N_A)
{
  // load s.real and negate second lane
  // that we avoid subtraction below
  int32x2_t sr;
  sr = vset_lane_s32(s.real,sr,0);
  sr = vset_lane_s32(-s.real,sr,1);

  int32x2_t si = vdup_n_s32(s.imag);

  for (int i = 0; i < N_A; ++i) {
    // load operands
    int64x2_t w = vld1q_s64((int64_t *)(W+i));
    int32x2_t ur = vld1_s32((int32_t *)(u+i));
    int32x2_t ui = vrev64_s32(ur);

    // conjugated multiplication
    w = vmlal_s32(w,sr,ur);
    w = vmlal_s32(w,si,ui);

    // store result
    vst1q_s64((int64_t *)(W+i),w);
  }
}
#endif
```

Figure 5: **Use of NEON instructions in the C code.**

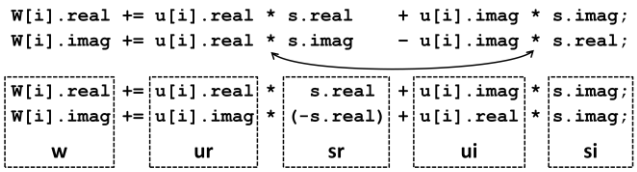

Figure 6: Visualization of the NEON registers for computing W.

The purpose of the code is the scalar multiplication of complex number *s* with the conjugate of complex vector *u* and adding the result to complex vector *W*. In Figure 6, a visualization of the NEON registers is displayed. It is derived from the non-NEON C code. The two parallel operations compute the real and imaginary part of *W* concurrently. The operands for the imaginary part are swapped to allow a more efficient loading of the NEON registers inside the loop.

After applying the intrinsics in the C code, we expect a speed-up factor of about two, because two operations are calculated in parallel. Additionally to this speed-up, there can be a benefit in the fact that 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.

4.2 High-Level Synthesis for DPD

As shown in Table 1, the so-called VW computation in Figure 4 is the most time-consuming task of the parameter estimation process. In order to improve the overall parameter update time, we have implemented a *VW accelerator* using the Vivado HLS tool. The VW accelerator takes the $z(n)$ and $y(n)$ samples as its inputs, constructs the **U** matrix of size (L x N_A) in (6) and, then computes the **W** vector of size (N_A x 1) and **V** matrix of size (N_A x N_A) as its outputs. Our accelerator has a programmable architecture such that it supports a different number of active coefficients N_A and allows the flexible selection of diagonal and off-diagonal terms in the generalized memory model as in (4). Hence, it is possible to support different nonlinear models using the same VW accelerator. In addition to using the design flow in Figure 3, the designer can make new changes in the existing C++ code and/or the compiler

directives to generate a brand-new accelerator in much shorter time than required by a hand-coded RTL design. In the remainder of this section, we give specific examples of code re-writing and compiler directives that we have used for the VW accelerator. Furthermore, we present productivity results where we compare final FPGA resources and the development time.

As shown in Figure 3, functional verification of the design is done in the C level. We have slightly modified the existing DPD parameter estimation code for our C++ test bench to validate our implementation code for VW computation.

The designer can 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 optimizations: 1) bit-width optimizations; and 2) efficient use of embedded DSP blocks (i.e., DSP48s). For example, the reference C/C++ code is normally written using built-in C/C++ data types (e.g., short, int), whereas the actual design can be based on fixed-point data types having word-lengths which are not integer multiples of the byte size. Here the HLS tool supports C++ template classes that can represent integer data types with arbitrary bit-width. For VW computation, we have leveraged the use of 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 to show code re-writing, bit-width optimization and the efficient use of DSP48s. The example is focusing on the complex multiplication, which is widely used by the VW computation. A standard complex multiplication carries out four real multiplications, which requires the use of four different multipliers in a fully-pipelined implementation. However, as shown in Figure 7, equivalent functionality can

```
1:  typedef struct {
2:      ap_int<32>   real;
3:      ap_int<32>   imag;
4:  } CINT32;
5:
6:  typedef struct {
7:      ap_int<64>   real;
8:      ap_int<64>   imag;
9:  } CINT64;
10:
11: CINT64 CMULT32(CINT32 x, CINT32 y)
12: {
13:     CINT64 res;
14:     ap_int<33> preAdd1;
15:     ap_int<33> preAdd2;
16:     ap_int<33> preAdd3;
17:     ap_int<65> sharedMul;
18:     preAdd1 = (ap_int<33>)x.real + x.imag;
19:     preAdd2 = (ap_int<33>)x.imag - x.real;
20:     preAdd3 = (ap_int<33>)y.real + y.imag;
21:     sharedMul = x.real * preAdd3;
22:     res.real = sharedMul - y.imag * preAdd1;
23:     res.imag = sharedMul + y.real * preAdd2;
24:     return res;
25: }
```

Figure 7: Optimized complex multiplication

```
1:  void vw_accelerator_top(
2:       ap_axiu<BW,x,y,z> InData[INSIZE],
3:       ap_axiu<BW,x,y,z> OutData[OUTSIZE],
4:       int N_A, int L,…)
5:  {
6:#pragma AP allocation instances=mul limit=3 operation
7:       <function body>
8:  }
```

Figure 8: Directive to limit instances of operations

be achieved by rewriting this code rather to use *three* multipliers (to employ less DSP48 blocks), at the expense of three additional pre-adders and one-bit increase in the multiplier word-length. In Figure 7, we show the multiplication of two complex numbers with 32-bit real and imaginary parts, using three multipliers. As it can be seen between lines 18-20, the C++ template class is used to declare 33-bit pre-adders. Furthermore, based on lines 21-23, the HLS tool generates three 33x32-bit multipliers, each giving a 65-bit result.

The original reference C++ code for the complex multiplication is using 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 casted 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 64x64-bit multipliers, which are obviously 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 HLS tool generates only three 33x32-bit multipliers, each using four DSP48s.

The "`CMULT32`" function in Figure 7 is inlined and used in different parts of the code. Here it is feasible to limit the instances of multiplications to three, to be able to share the same multipliers throughout the code. The snippet of C++ code in Figure 8 shows how this can be achieved by adding a compiler directive.

The main reason to implement a VW accelerator is to reduce the time used for VW computation and, hence, to improve the overall DPD parameter estimation time. For this purpose, the loops in the C++ code need to be pipelined and also unrolled if possible. Using the HLS tool, we have verified that the most time-consuming loop in our implementation is the V computation loop. We have unrolled this loop by a configurable factor which is 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. The compiler directives that control the loop unrolling are shown in Figure 9. On line 6, based on the unrolling factor, instances of the multiplication operation can be set more than three, compared to Figure 8. This is because three multipliers cannot be shared in the case of loop parallelization. Furthermore, on line 9, we partition the array

```
1:  void vw_accelerator_top(
2:       ap_axiu<BW,x,y,z> InData[INSIZE],
3:       ap_axiu<BW,x,y,z> OutData[OUTSIZE],
4:       int N_A, int L,…)
5:  {
6:#pragma AP allocation instances=mul
  limit=3*UNROLL_FACTOR operation
7:        <function body>
8:      CINT64 Varray[VSIZE];
9: #pragma AP array_partition variable=Varray cyclic
factor=UNROLL_FACTOR dim=1
10:       <function body>
11:     label_compute_V:
12:     for (int i=0; i < VSIZE; ++i)
13:     {
14:#pragma AP pipeline
15:#pragma AP unroll factor=UNROLL_FACTOR
16:          <loop body>
17:     }
18:       <function body>
19: }
```

Figure 9: Loop unrolling

that stores V values (mapped to block RAMs after RTL synthesis) by the unrolling factor, to avoid access problems. Lines 14 and 15 show how the pipelining and unrolling directives are applied, respectively.

Our implementation uses single input single output data interface, for simplicity in the system architecture. The input and output data parameters are declared as arrays. In the input data array, complex capture samples $z(n)$ and $y(n)$ are ordered in an interleaved fashion. The output data array contains the N_A complex elements of **W** and, consecutively, the complex elements in **V**. The HLS tool supports a C-level macro that turns these arrays into high-speed streaming interfaces. Furthermore, different type of macros can be used to assign a low-throughput memory-mapped communication interface to the top-level configuration parameters, such as N_A or L. A snippet of the C++ code that shows the use of these macros is available in Figure 10. As shown in lines 2 and 3, the input and output data arrays are declared using a template structure. Here the template parameters allow the designer to set the bit-width of different fields in the streaming protocol at the time of

```
1:  void vw_accelerator_top(
2:       ap_axiu<BW,x,y,z> InData[INSIZE],
3:       ap_axiu<BW,x,y,z> OutData[OUTSIZE],
4:       int N_A, int L,…)
5:  {
6:     // Streaming interfaces
7:     AP_BUS_AXI_STREAMD(InData, BUS_INDATA);
8:     AP_BUS_AXI_STREAMD(OutData, BUS_OUTDATA);
9:     // Memory-mapped interface
10:    AP_INTERFACE_REG(N_A,        ap_none);
11:    AP_INTERFACE_REG(L,          ap_none);
12:    …
13:    AP_BUS_AXI4_LITE(N_A,        AXIlite);
14:    AP_BUS_AXI4_LITE(L,          AXIlite);
15:       <function body>
16: }
```

Figure 10: Example of accelerator interface

variable declaration. For example, "BW" is a previously-defined integer constant to set the bit-width of the streaming data samples. Likewise, "x", "y" and "z" are used to set the bit-width of other fields in the streaming protocol.

In Figure 11, we show how the VW accelerator design generated using the Vivado HLS tool evolves over time. Please note that the first two weeks of the development time is spent for the integration of the Vivado HLS-generated accelerator to the processor subsystem. The accelerator used during this time is based on the original reference C++ code with minor modifications. The C++ test bench for functional verification was also set up during the same time frame. Figure 11 shows that the actual accelerator optimization process takes less than a week. For example, the massive reduction in the number of DSP48s is based on the bit-width optimizations, code rewriting for complex multiplication and the application of compiler directives to limit the multiplication instances, as discussed earlier (see Figures 8 and 9). The bit-width optimizations also played an important role in reducing the number of flip-flops (FFs). The results obtained in the end of acceleration optimization phase are very similar to the hand-coded RTL results.

As discussed earlier, V computation is the most time-consuming part of the VW accelerator. Hence, during our accelerator exploration, we have implemented faster accelerators by unrolling the V loop. The last two data points in Figure 11 are for V loop unrolling by 2 and 4. The changes applied for V loop unrolling in Figure 9 confirm the size increase in Figure 11. For example, multiplication instances for the complex multiplier need to be increased by the unrolling factor. As a result, the number of DSP48s needed for the complex multiplication increases by the same factor. The accelerator exploration using Vivado HLS takes only a few days, whereas the traditional RTL design

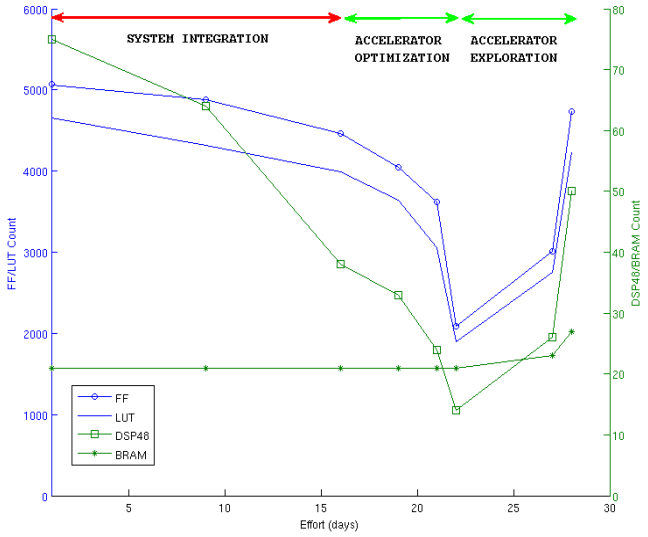

Figure 11: Design evolution vs. development time

requires much longer time. In the case of Vivado HLS, the effort can be as simple as changing a few compiler directives as shown in Figure 9.

Our implementation code in this paper allows different configurations for VW computation, each supporting a different value for maximum number of pre-distorter coefficients. Please note the number active coefficients N_A in (6) is a run-time parameter of the accelerator, as shown in Figure 10 (see lines 4 and 10), which is bounded by the maximum number of coefficients. We use a C macro in the compiler options for the configuration selection, which defines the maximum number of coefficients. For simplicity in the presentation, Figure 11 shows the accelerator results for the first configuration only, supporting the least number of coefficients. However, at the end of the design process, we have generated the accelerators for the other two configurations as well. All accelerators are successfully running on the Zynq board.

4.3 Integration of Hardware and Software Components

The communication between the processor subsystem and hardware accelerator is based on the Advanced eXtensible Interface (AXI) protocol [6], which is part of ARM AMBA, a family of micro controller buses first introduced in 1996. The second and most-recent version of AXI is AXI4, which is included in AMBA 4.0 released in 2010. Our design uses the Xilinx AXI interconnect core IP [6], which is able to connect one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. As shown in Figure 12, we have used the AXI interconnect core to connect AXI4 Lite masters to slaves. The AXI4 Lite is a light-weight, 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.

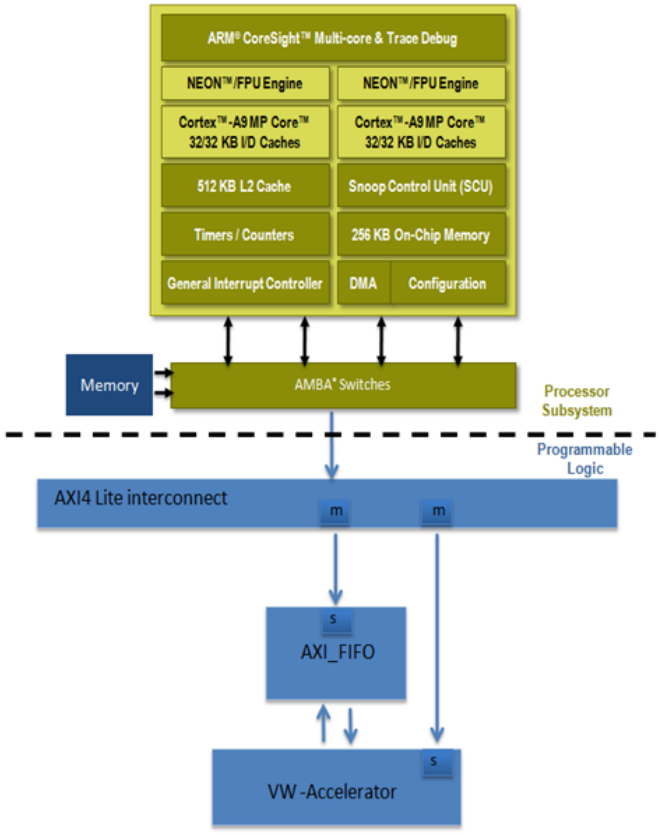

Figure 12: Integration of processor subsystem with the hardware accelerator

The AXI FIFOs in Figure 12 are for the input and output data samples of the accelerator, which are connected to the corresponding AXI4 stream interfaces of the accelerator generated as shown in Figure 10. The AXI4 Lite slave on the accelerator is for the VW configuration parameters. The AXI4 Lite interface for the accelerator is generated in the C level, as illustrated in Figure 10.

5. SYSTEM PERFORMANCE

In this section, we compare the coefficient update times for three specific VW configurations, discussed at the end of Section 4.2. These three configurations will be called C1, C2 and C3 from now on, which are ordered according to the increasing computational complexity. Here C1 denotes the simplest configuration, supporting the lowest value for the maximum number of pre-distorter coefficients. The more complex architectures correspond to the DPD solutions with increasing maximum number of coefficients. Compared to C1, C2 and C3 support 1.5X and 2.3X more coefficients, respectively.

There are *four* different designs implemented for each of these configurations: 1) Software-only design optimizing VW computation using NEON instructions (VWNeon), 2) Software + VW accelerator (VWAx1), 3) Software + VW accelerator unrolling the V computation loop by two (VWAx2), 4) Software + VW accelerator unrolling the V computation loop by four (VWAx4).

All the designs are implemented based on the software-programmable flow explained in Section 2. In designs VWAx1, VWAx2 and VWAx4, only the Alignment and A Computation blocks (see Figure 4) are running on the processer subsystem since the VW computation is carried out by the hardware accelerator. Unrolling of the V computation loop is achieved using compiler directives, as discussed in Section 4.2 (see Figure 9).

Target clock frequency for our designs is 166 MHz. The area results in Figure 11 have also been obtained at 166 MHz. This clock frequency meets our current system requirements. However, it is possible to increase the target

clock frequency constraint in Vivado HLS, in order to generate faster accelerators.

Processing times for different configurations and designs are compared in Table 2. Here the results for the original software implementation of C1 (previously shown in Table 1) are also included, to compare them to the results for "C1/VWNeon". The total update time for "C1/VWNeon" meets our current timing requirements thanks to the NEON optimizations. However, depending on the transmitter specifications, the accelerated designs for C1 (with/without loop unrolling) can also be preferable to use. For example, in a multi-antenna base station, it is possible to run an accelerated design more than once (computing pre-distorter coefficients for a different antenna each time), and attain an update time similar to 254.64 ms.

For more complex DPD solutions, the designs "C2/VWAx1" and "C3/VWAx2" result in update times comparable to "C1/VWNeon". Hence, based on our current requirements, it is more feasible to use these designs for C2 and C3. This is because they use less hardware resources compared to the accelerated designs with further loop unrolling. However, further loop unrolling can be beneficial under different requirements, as in the case of multiple antennas discussed above.

Table 2: Comparison of processing times

Architecture /Design	Align. (ms)	VW (ms)	A (ms)	Total Time (ms)
C1/original SW	29.62	336.40	8.09	374.11
C1/VWNeon	**29.62**	**216.93**	**8.09**	**254.64**
C1/VWAx1	29.62	63.14	8.09	100.85
C1/VWAx2	29.62	41.20	8.09	78.91
C1/VWAx4	29.62	32.34	8.09	70.05
C2/VWNeon	29.62	449.90	15.06	494.58
C2/VWAx1	**29.62**	**130.95**	**15.06**	**175.63**
C2/VWAx2	29.62	80.61	15.06	125.29
C2/VWAx4	29.62	58.65	15.06	103.33
C3/VWNeon	29.62	1004.5	61.72	1095.89
C3/VWAx1	29.62	292.39	61.72	383.73
C3/VWAx2	**29.62**	**167.92**	**61.72**	**259.26**
C3/VWAx4	29.62	110.72	61.72	202.06

In Table 2, it is shown that the accelerated designs using an unrolling factor of four can result in up to 5X improvement in coefficient update times, compared to NEON-optimized solution (e.g., "C3/VWAx4" vs. "C3/VWNeon"). For all configurations, it is also worth noting that unrolling V computation loop by four results in less significant time improvement, compared to unrolling by two (e.g., "C3/VWAx4" vs. "C3/VWAx2"). This is because the time spent for the other functions of the VW accelerator becomes more dominant in the total update time.

6. CONCLUSIONS

In this paper, we have presented a software-programmable design flow for DPD targeting new generation FPGAs. Our design flow allows the flexible partitioning of functionality among hardware and software components, and increases the productivity by reducing the time for implementation and system integration. We have used the ARM NEON instructions to optimize the software implementation and employed Vivado HLS as the HLS tool for the programmable logic. By taking into account three different DPD architectures, we have implemented several designs trading off faster pre-distorter coefficient update times versus the size of the design. We have tested our designs successfully on the target platform. Our flexible design flow facilitates the generation of effective DPD solutions for modern wideband and multi-antenna transmitters.

6. ACKNOWLEDGMENT

The authors would like to thank Vincent Barnes, Dave Fraser, Hemang Parekh, Colin Stirling and Chris Dick from Xilinx, Inc. for their valuable comments and cooperation.

8. REFERENCES

[1] E. Dahlman *et al.*, *3G Evolution: HSPA and LTE for Mobile Broadband*, 2nd ed., Academic Press, 2008.

[2] D. Astély *et al.*, "LTE: the evolution of mobile broadband," *IEEE Communications Magazine*, vol. 47, pp. 44-51, 2009.

[3] D.R. Morgan *et al.*, "A generalized memory polynomial model for digital predistortion of rf power amplifiers," *IEEE Transactions on Signal Processing*, vol. 54, pp. 3852-3860, 2006.

[4] Berkeley Design Technology, Inc., "High-Level Synthesis Tools for Xilinx FPGAs," White paper 2010. [Online] Available: http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf

[5] Xilinx, Inc., *Zynq-7000 Extensible Processing Platform Product Brief*. [Online] Available: www.xilinx.com/publications/prod_mktg/zynq7000/Product-Brief.pdf

[6] Xilinx, Inc., *AXI Reference Guide UG761 (v13.1)*, 2011.