WP437 (v1.0) August 29, 2013

Wireless Base Station ZUC Block Cipher Implementation on Zynq-7000 AP SoC

By: Matt Ruan, Harvest Guo, Leon Qin

Data security and privacy are major concerns for wireless subscribers because their personal information is transmitted over the air and is accessible to anyone. Modern wireless communication systems address this problem by using state-of-the-art cryptographic algorithms, such as ZUC, which has been recently integrated into the 3GPP LTE EEA-3/EIA-3 confidentiality protection methods. In the absence of any commercially available hardware accelerator, however, the ZUC computational load imposes new design challenges to the LTE layer 2 protocol stack processors in base stations; the designer either needs to find a way to optimize the software to offer sufficient computation power for ZUC, or add an FPGA adjacent to the processor for acceleration. The former method is unlikely to meet the throughput requirement, while the latter is not cost-effective.

Together, the Xilinx® Zynq®-7000 All Programmable SoC (AP SoC) and Vivado® Design Suite provide a joint hardware/software development platform that addresses these challenges. The platform directly converts a standard's reference *C* code into an accelerator running in programmable logic and an AXI interconnect to the processor system. This new methodology facilitates hardware/software system function partitioning for advanced performance, and the optimized joint hardware/software design flow provides markedly increased productivity.

Introduction

In wireless communications, user data is carried by a radio signal that propagates over the air, accessible to anyone with suitable radio hardware and software resources. Consequently, data security and privacy are often major concerns for wireless subscribers.

Modern wireless communication systems address these problems by using state-of-the-art cryptographic algorithms and comprehensive authentication mechanisms. These techniques greatly diminish an unauthorized listener's ability to decipher the contents of a conversation—hampering any security breach for a fairly long period of time. Only a brute-force method can attack an unbroken security algorithm, which means that one has to try all 2^{128} keys to decipher the message. It requires prohibitively high complexity, and even for the fastest computer in the world, it would take years to do so.

The 3rd Generation Partnership Program (3GPP) Long Term Evolution (LTE) specification [Ref 1] integrates the latest advances in terms of security. A thorough overview of the subject can be found in "Network Access Security in Next-Generation 3GPP Systems: A Tutorial" [Ref 2]. The LTE Access Network, which is also referred to as the Evolved Universal Terrestrial Radio Access Network (EUTRAN), hosts a number of protocol layers:

- Physical (PHY)
- Medium Access Control(1)
- Radio Link Control (RLC)
- Packet Data Convergence Protocol (PDCP)
- Radio Resource Control (RRC)

During the establishment of a wireless connection, security keys are first generated by the authentication and key agreement (AKA) procedure at the base station and terminal. Access stratum confidentiality is then achieved through ciphering and integrity-checking operations in the PDCP layer. A simplified block diagram of the ciphering and deciphering procedures is shown in Figure 1.

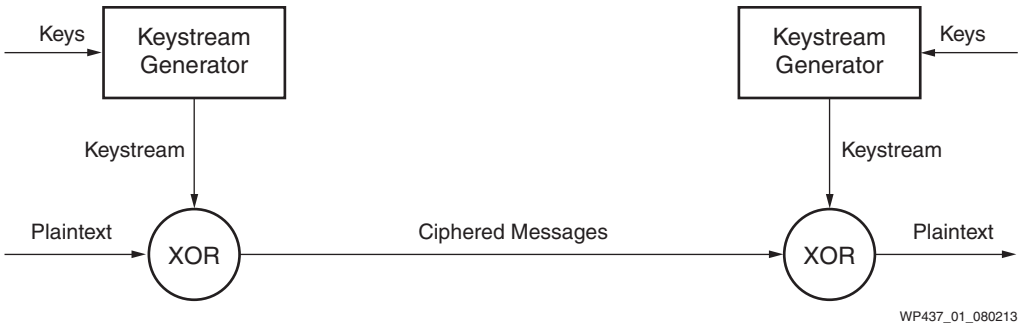

Figure 1: **Ciphering and Deciphering Procedures**

On the transmitter side, the plaintext is bit-wise XORed, with the keystream generated by a predefined algorithm using a specific key. The ciphered message is sent over the

1. The "MAC" acronym usually applied to this protocol layer is reserved in this white paper for the data entity "Message Authentication Code" (described in detail herein); to avoid confusion, "MAC" is purposefully omitted as an acronym for "Medium Access Control layer."

air and can only be decoded if the receiver knows the correct key for the generation of an identical keystream. Figure 2 shows a block diagram of the integrity protection operation, where a Message Authentication Code (MAC) is sent along with the user message such that, at the receiver side, an identical MAC can be computed from the received message with the known key. When the computed authentication code does not match the received code, either the message has been altered by the communication channel, or another key has been used at the transmitter, meaning that the message is not intended for that particular user. In both cases, the received message is discarded to preserve the network's safety.

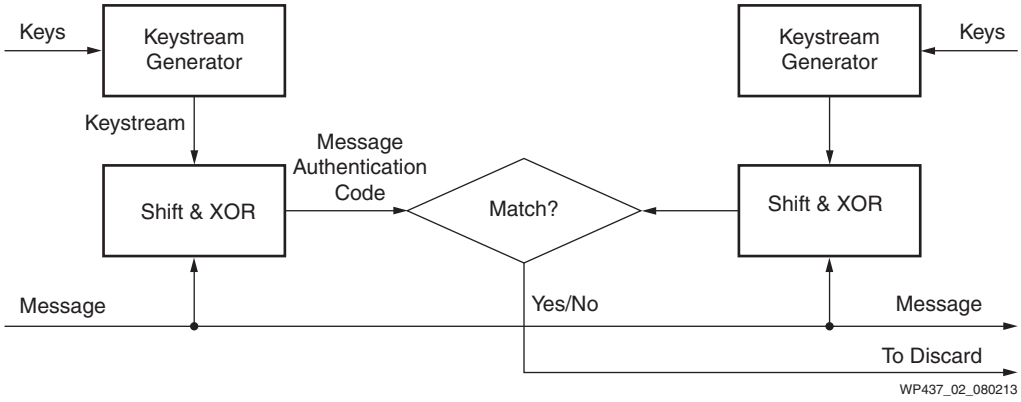

Figure 2: **Diagram of Integrity Protection**

The effectiveness of the ciphering and integrity protection operations largely depends on the keystream generation algorithm, which needs to guarantee that the:

- Keystream can only be generated by the correct key
- Possibility of guessing the key from the ciphered message is almost zero

Three keystream generation methods have been adopted by the 3GPP LTE standard [Ref 1]: SNOW 3G [Ref 3]; AES [Ref 4]; and ZUC [Ref 5]. The corresponding encryption and integrity protection solutions are referred to as EEA-1, EEA-2, EEA-3, and EIA-1, EIA-2, EIA-3, respectively. Having more than one cryptographic option provides enhanced security, which maintains protection in the event that any of the algorithms ever becomes vulnerable.

The ZUC-based EEA-3 and EIA-3 algorithms [Ref 6] were proposed by the Chinese Academy of Sciences and adopted by 3GPP recently; consequently, support for EEA-3/EIA-3 is likely to be mandatory for LTE wireless infrastructures as deployed in China. Unfortunately, since these solutions are relatively new to the standard, they are not yet supported by any PDCP accelerator on the market. An external FPGA or stand-alone co-processor is typically required to provide EEA-3 and EIA-3 functionality to the Layer 2/3 processors. This increases system cost and power consumption.

Xilinx recently released the Zynq-7000 AP SoC, which integrates two ARM® Cortex™-A9 processors and multiple peripherals, tightly coupled with the programmable logic for complex system development. The processor system allows a pure software solution to run in real time for rapid functional testing and profiling. The computationally heavy tasks are further offloaded to an accelerator running in the programmable logic and connected to the processor system via the AXI interconnect ports [Ref 7]. Developers are therefore able to select and configure IP and peripherals on an as-needed basis without wasting processing capabilities or power. Additionally, Xilinx provides a high-level synthesis (HLS) tool called Vivado HLS [Ref 8] that can

directly convert *C/C++* code into RTL. This tool eliminates the need for manual RTL coding, shortens the RTL functional verification phase, and thus maximizes the productivity of both software and hardware engineering teams.

The example in this white paper starts with EEA-3 and EIA-3 reference *C* code attached to the 3GPP specification [Ref 5][Ref 6]. A software solution is first quickly built for the processor cores of the Zynq-7000 device. Vivado HLS is then used to convert the reference *C* code into an EEA-3/EIA-3 accelerator. The accelerator runs in the programmable logic and directly reads the data from a DDR memory via the AXI interconnect. After the processing is completed, the result is automatically written back to the DDR memory without any intervention of the processor.

With only simple optimization techniques, the automatically generated accelerator can achieve a much higher throughput than that of a dedicated ARM core. The purpose of this example is to illustrate the new joint hardware/software design methodology, which takes advantage of the unique architecture of the Zynq-7000 AP SoC.

ZUC Algorithm Overview

As shown in Figure 3, the ZUC algorithm accepts a 128-bit key and an initialization vector as inputs, and it outputs a keystream to be used for ciphering and integrity protection operations. The key and initialization vector are loaded into the linear feedback shift register (LFSR) during the initialization phase; the LFSR is clocked 32 times with the output of the non-linear function *F()* being taken as feedback. In working mode, the LFSR is updated without taking any external input, and the *F()* function output is used for the calculation of the output keystream Z. The most common operations in the *F()* function are circular shifting and modulo additions, which mimic linear polynomial operations. A non-linear sub-function *S* is defined in *F()* to further randomize the output. The details of the algorithm can be found in *Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification* [Ref 5], and implementation on a Xilinx Virtex®-5 FPGA is reported in *An FPGA Implementation of the ZUC Stream Cipher* [Ref 9].

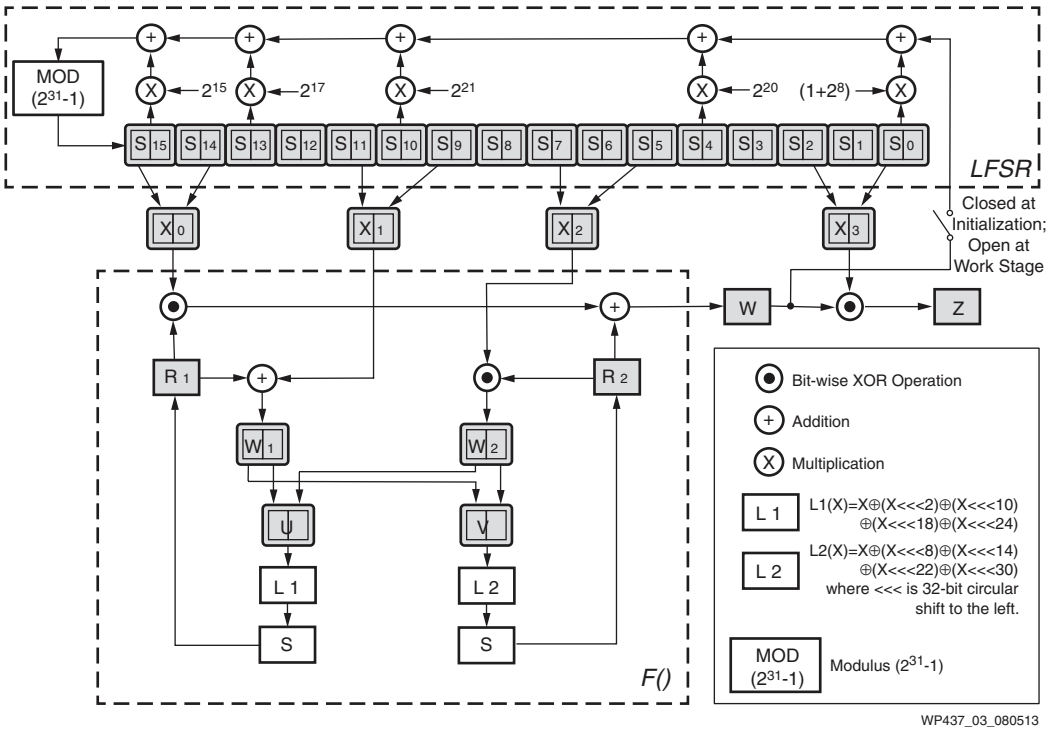

Figure 3: **Block Diagram of the ZUC Keystream Generator**

Zynq-7000 All Programmable SoC Overview and Development Flow

A high-level block diagram of the Zynq-7000 AP SoC is shown in Figure 4. Each ARM Cortex-A9 processor has two 32 KB built-in level-1 caches for instructions and data, respectively. Another 512 KB on-chip level-2 cache is shared by the two processors. The snoop control unit (SCU) maintains the coherency of the caches in the two processors. The interconnection between the processor system and programmable logic is achieved through nine distinct AXI ports. The S_AXI_GP slave ports are typically used by the programmable logic that needs to access the processor system peripherals, while the M_AXI_GP master ports are mainly used by the processor system to access the register maps built in the programmable logic. The S_AXI_HP slave ports provide an efficient way for the programmable logic to access an external DDR memory or the 256 KB on-chip memory, while, for latency sensitive applications, the accelerator coherent port S_AXI_ACP offers direct accesses to the caches via the SCU. For a more detailed description of the Zynq-7000 AP SoC, refer to the *Zynq-7000 All Programmable SoC Technical Reference Manual* [Ref 7].

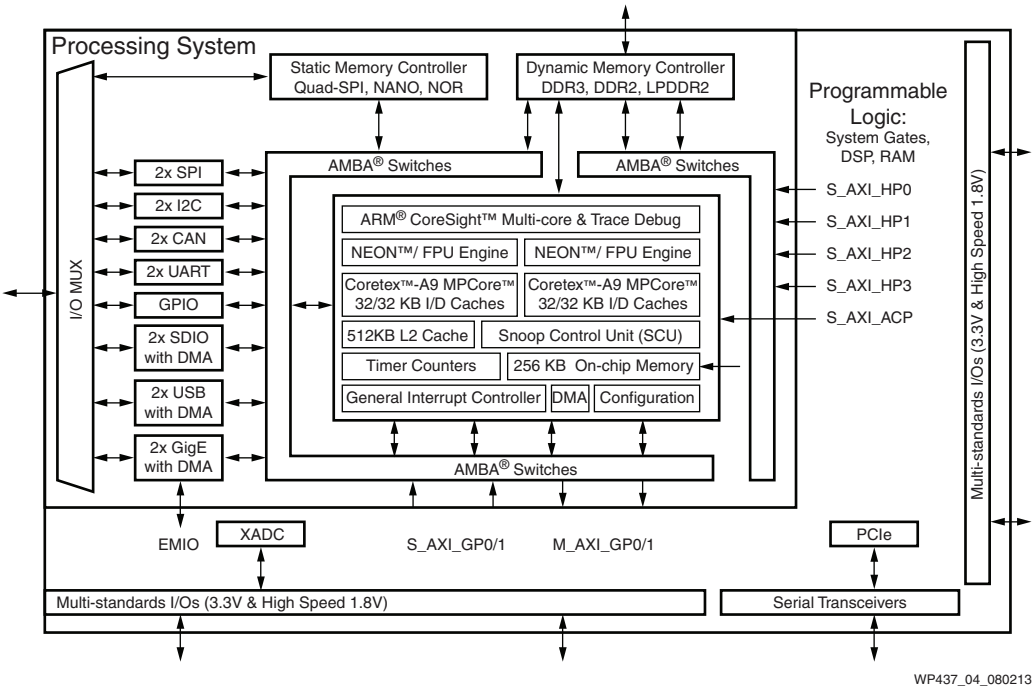

Figure 4: **Simplified Block Diagram of the Zynq-7000 AP SoC**

System Hardware Definition

The development flow for the Zynq-7000 AP SoC consists of two major steps. The system hardware definition step defines the various components of the system and how they are connected together. The Xilinx Platform Studio (XPS) [Ref 10] is used to configure the main hardware parameters of each component and IP:

- Clock source and frequency
- AXI interconnect ports and base addresses
- Connection to interrupt controllers
- Connection to external input/output pins

For Xilinx IP cores, including those generated by Vivado HLS, the interconnections and bus configurations are fully automated and customizable. After the processor system has been defined with XPS, Xilinx PlanAhead™ [Ref 11] can automatically generate a top-level RTL wrapper that can be synthesized by the ISE® or Vivado tools. The resulting BIT file is finally downloaded to the Zynq-7000 AP SoC, which then behaves as a fully customized application-specific device under software control.

Software Development

Software development, the other major step, usually takes longer and requires more effort than hardware development. To address this, Xilinx SDK [Ref 10] automates the most tedious and error-prone software tasks, such as coding low-level hardware drivers and board support packages (BSP). The tool reads the hardware description files produced by XPS and automatically generates the required BSP files and low-level drivers for the hardware components implemented in the Zynq-7000 device. Refer to *Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT) — A Hands-On Guide to Effective Embedded System Design* [Ref 12] to see how a simple "Hello world" program can be generated by just a few mouse clicks.

The system hardware definition and the software development can actually be parallelized and conducted iteratively. For example, a first version of the accelerator might support only a low data rate with limited functionality, yet be functional enough for initial software development. While the hardware team is further optimizing the accelerator, the software team can work on system software coding and testing with the *initial* accelerator. By the time a fully functional accelerator is available, the software is ready for final integration and testing.

In case there is a need to further offload the processor using additional accelerators implemented in programmable logic, the hardware system definition can be readily modified and a new software application program interface (API) developed. The above incremental steps do not involve any hardware change outside of the Zynq-7000 AP SoC.

EEA-3 and EIA-3 Software Solution without Hardware Acceleration

It is quite straightforward to build a pure software EEA-3/EIA-3 solution from the reference *C* code provided with the standard [Ref 5][Ref 6]. Testing of the software solution has been performed on the Xilinx ZC706 evaluation board [Ref 13], featuring a Zynq-7000 XC7Z045 device, with clock rates for the ARM core and external DDR3 memory set to 733 MHz and 533 MHz, respectively. A simple XPS project — without any hardware acceleration — has been created for the ZC706 board, and the "Hello world" SDK project is used as the starting point for software development. Along with the EEA-3/EIA-3 specification [Ref 6], a number of golden test vectors are provided. These golden test vectors have been used to validate the solution and also make sure there are no compatibility issues with the *C* compiler.

A specific testbench has been developed to measure the achievable data throughput in batch mode. The analysis has been performed on a large number of random test vectors with random parameters used for the EEA-3/EIA-3 algorithm, including the keys. The generated data is stored on external DDR memory in the format described in Table 1. The EEA-3/EIA-3 functions are then run to process this data; after the functions return, the throughput is simply estimated from the total number of message bits processed over the time elapsed.

Table 1: **Test Vector Storage Format in DDR Memory**

Address Offset	Data
0	Key[31:0]
1	Key[63:32]
2	Key[95:64]
3	Key[127:96]
4	Count[31:0]
5	Zeros[26] & Bearer[4:0] & Direction
6	Length[31:0]
7	Data[31:0]
…	…
6+L (L = ceil(Length/32))	Data [Length-1: (L-1)*32] and Zeros [32*L – Length]

The Xilinx SDK tool offers multiple compilation options that can significantly affect performance results. By default, compilation optimization is OFF to ease debugging. After the functional validation of the code is completed, however, it is recommended that the optimization option be turned ON to improve software performance through function in-lining, loop unrolling, and more aggressive use of the ARM-A9 registers. Once enabled, these optimizations are fully automatic.

The throughput results are summarized in Table 2 for the EEA-3/EIA-3 pure software solution, *without* and *with* compilation optimization.

Table 2: **EEA-3/EIA-3 Software Solution Test Results with 733 MHz ARM Clock**

Data Block Length (bit)	Test Data Size (Mb)	*Without* Compilation Optimization		*With* Compilation Optimization	
		Elapsed Time (ms)	Throughput (Mb/s)	Elapsed Time (ms)	Throughput (Mb/s)
EEA-3 (Software Only, No Hardware Accelerator):					
256	250.5	29,050	8.6	11,641	21.5
4,096	509.0	12,341	36.1	5,753	88.5
65,536	535.0	10,404	45.0	4,859	110.1
16,777,216	520.1	9,945	45.6	4,777	108.9
EIA-3 (Software Only, No Hardware Accelerator):					
256	250.5	55,670	4.5	18,660	13.4
4,096	509.0	57,323	7.8	18,879	27.0
65,536	535.0	57,515	8.1	18,466	28.9
16,777,216	520.1	54,876	8.3	17,922	29.0

Data block length is the length of data for each EEA-3/EIA-3 operation. For each block length, a large number of test data blocks are generated to ensure sufficient averaging in the results. These results show that:

- The throughput increases with the data block length. The reason: For short blocks, a larger portion of time is spent on the initialization stage of the ZUC keystream generator without producing any valid output.
- Enabling the compilation optimization yields a throughput improvement factor of roughly 2.5X.
- With an ARM Cortex-A9 core completely dedicated to the cryptographic tasks, the maximum achievable data throughputs for EEA-3 and EIA-3 are 110 Mb/s and 29 Mb/s, respectively. This is lower than the typical requirement for small cells, where the aggregated downlink and uplink data rate is at least 150 Mb/s. Some hardware acceleration is typically required to offload the processor.

EEA-3 and EIA-3 Hardware Accelerator Architecture

To get the most benefit from the hardware accelerator, the intervention of the processor during EEA-3/EIA-3 operations must be minimized — for example, so that it needs only to specify the input and output memory addresses and whether EEA-3 or EIA-3 operation is to be performed. The accelerator is then responsible for reading the data from external memory, processing it for EEA-3 or EIA-3 as instructed, and storing the results. The EEA-3 outputs are written back to the external memory at the address provided by the processor, while the EIA-3 MAC calculation result is made available to the processor from an internal register in programmable logic.

Two different interfaces need to be implemented by the accelerator, one to control the external memory access and the other to manage the register map. The high-performance AXI slave port (S_AXI_HP) has dedicated connections to the DDR3 memory controller and is therefore ideal for external memory access. The programmable logic register access can be performed at moderate speed, and the general purpose master port (M_AXI_GP) is an appropriate choice. Detailed information about the AXI interconnect ports can be found in the Zynq-7000 technical reference manual [Ref 7].

A block diagram of the EEA-3/EIA-3 accelerator is provided in Figure 5. Using this architecture, the EEA-3 and EIA-3 functionality is reduced to a few register accesses (from the processor perspective), which can be easily supported with negligible system load.

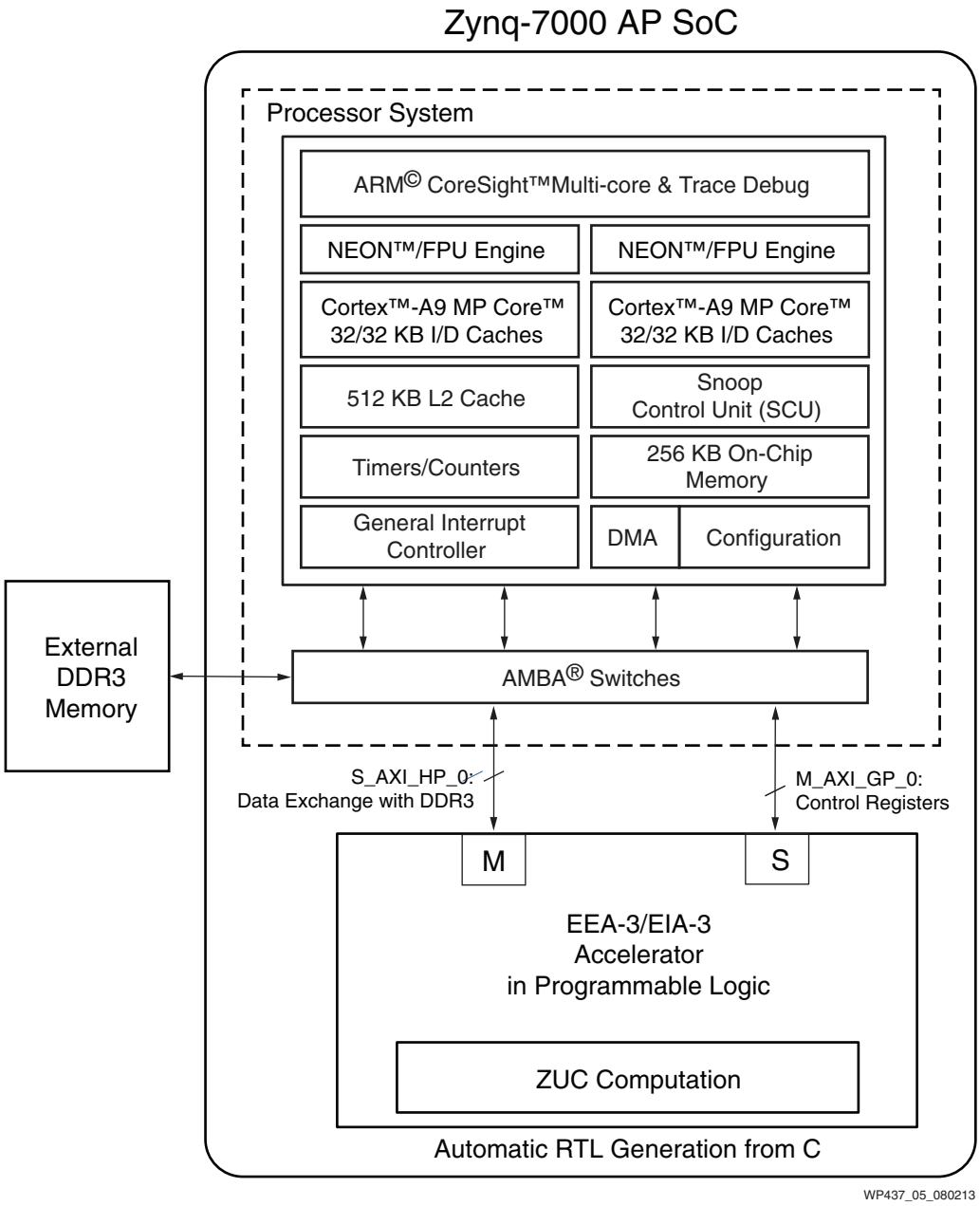

Figure 5: **Block Diagram of the EEA-3/EIA-3 Accelerator in Zynq-7000 AP SoC**

EEA-3 and EIA-3 Hardware Accelerator Implementation Example

The EEA-3/EIA-3 accelerator is described in *C* programming language and synthesized into RTL with Vivado HLS, without any manual RTL coding. The original EEA-3 and EIA-3 reference *C* code is taken as the baseline for the implementation. However, some modifications are required to make it suitable for hardware realization.

Redefine the Interface

Three AXI interface adapter cores are defined in Vivado HLS: AXI4LiteS, AXI4M, and AXI4Streaming. Control registers are recommended to connect to an AXI slave interface via the AXI4LiteS adapter, while the AXI4M adapter allows access to AXI address space from programmable logic. For streaming data access, the AXI4Streaming adapter is the most efficient choice. In the case of the EEA-3/EIA-3 accelerator, the AXI4M and AXI4LiteS adapter cores are selected for memory and register accesses, respectively. In *C* code, a pointer to a 32-bit integer is defined as the input to the main function, and some synthesis directives specify the mapping of the pointer to an AXI bus master adapter core in the programmable logic.

The accelerator needs to know the memory read/write start addresses before executing a cryptographic task. These are implemented as 32-bit integer input arguments to the EEA-3/EIA-3 function; synthesis instructions map the variables to an AXI4LiteS adapter core in programmable logic. The EIA3 return value — i.e., the MAC calculation result — can be bundled into the same AXI slave port.

Vivado HLS synthesizes the *C* code into an IP block called "Pcore," which includes all the required interfacing logic for integration with the processor system. When importing the Pcore, the XPS tool automatically identifies the AXI interfaces to generate the appropriate interconnections. Only the AXI bus base addresses need to be specified by the user via the XPS graphical user interface (GUI).

Change from Block-Wise to Word-Wise Processing

The original *C* code uses a buffer of the same size as the data block to save the generated keystream before processing the data. A more efficient approach is to compute successive 32-bit key words and adopt a word-based data processing approach. This can be realized by breaking the loop in the ZUC main function and creating a new function, which outputs one 32-bit key for each function call.

Merge EEA and EIA Functions

The accelerator is shared by both EEA-3 and EIA-3 operations, so significant area savings can be achieved by merging the code of these two functions.

With these modifications, the reference *C* code can be efficiently synthesized into RTL with Vivado HLS. No in-depth hardware knowledge is required to optimize the reference *C* code; hence, this can be readily performed by a software engineer. In addition to RTL, the Vivado Design Suite generates low-level drivers for the main APIs needed for application software development, including register accesses, interrupt handling, and start/stop status controls.

The following shows some exemplary *C* code to start the EEA-3/EIA-3 processing; when the processing finishes, the return value can be read from the programmable logic:

```
// start the processing
XEeaeia3_hw_Start(&pcore);

// wait until it finishes
while (!XEeaeia3_hw_IsIdle(&pcore));

// get the returned value
u32 mac_eia = XEeaeia3_GetReturn(&pcore);
```

The EEA-3 processed data has been saved to DDR by the time execution of the above code completes; the EIA-3 MAC output is available in the variable **`mac_eia`**.

The baseline accelerator uses no additional directives beyond those required for interface definition. All the HLS synthesis options are the default values, and the target clock frequency is set to 100 MHz. Since the accelerator implementation result indicates an F_{MAX} as high as 180 MHz, the clock frequency for the accelerator and interconnect logic is set to 150 MHz so that full performance is achieved after integration with the processor system.

Detailed implementation results are given in Table 3.

Table 3: **Baseline Accelerator Implementation Results on 7Z045T-2**

	Slice	FF	LUT	DSP	BRAM18K	F_{MAX}
Accelerator Core Only	846 (1.5%)	2,147 (0.5%)	2,786 (1.3%)	0	5 (0.5%)	180 MHz
With AXI Interconnection	1,811 (3.3%)	3,652 (0.8%)	4,495 (2.1%)	0	5 (0.5%)	160 MHz

The baseline accelerator takes only a small portion of the total available programmable logic resources in the 7Z045 SoC.

Some simple optimizations can be performed on the baseline accelerator:

1. **Change of memory access to burst mode**

   The baseline code uses 32-bit word-based access to the external memory, which is highly inefficient. A much better solution is to use burst mode by allocating a small buffer for data pre-fetching. The size of the buffer represents a trade-off between hardware resources and overall throughput. After some experimentation, the sizes of read and write buffers were set to 2K and 1K bits, respectively.

2. **Optimization of function in-lining**

   In the *C* programming language, a function can be called in multiple places of a program with different parameters. This complicates the interface and limits the parallelism of function execution. Alternatively, multiple instances of the function can be created to avoid register sharing; using this approach simplifies the interface.

   This optimization approach is referred to as *in-lining*. By default, Vivado HLS automatically determines whether or not a function should be in-lined to achieve an appropriate balance between resource utilization and performance (speed). However, the user can optionally take direct control of function in-lining by issuing specific directives. For example, the non-linear function *F()* in the ZUC reference code is used in both the initialization and working stages. In this design context, Vivado HLS by default instantiates the function twice to produce a higher degree of parallelism, a feature that is not really advantageous here. Fortunately, the tool can be forced to instantiate the function only once.

3. **Optimization of resource mapping for *S* function**

   The *S* function is part of the non-linear function *F()*. It consists of two sub-functions known as *S1* and *S2*; they both take 8-bit inputs and output 8-bit values. These functions are implemented as 2,048-bit read-only memories (ROMs). As the latency of the *F()* function has a direct impact on the data throughput, it is better to use distributed memory rather than block RAM for the implementation. Synthesis directives are inserted into the *C* code to ensure the array variables are mapped to dual-port look-up table (LUT)-based ROMs by Vivado HLS.

4. Parallelization of EIA-3 Shift-and-XOR Operation

The EIA-3 calculation uses the keystream as a string of input bits. For all the locations where the message bits are equal to 1, key words of 32 bits are extracted from the keystream and XORed together. In the reference *C* code, this is realized by a shifter followed by a 32-bit XOR unit. This is also the architecture of the baseline accelerator. To improve the throughput, it is desirable to parallelize the shift-and-XOR operations of the EIA-3 function in programmable logic.

As a trade-off between area and throughput, four shift-and-XOR operations are grouped together such that only eight clock cycles are required to process each 32-bit word. A block diagram of the EIA-3 calculator is shown in Figure 6, where **`Z[n:n+63]`** are the ZUC output bits from **`n`** to **`(n+63)`**, **`M[n]`** is message bit **`n`**, and **`T[n]`** is the 32-bit computed MAC up to bit position **`n`**. Since the proposed architecture takes just one clock cycle to compute **`T[n+3]`** from **`T[n-1]`**, the processing speed is four times that of a serial implementation.

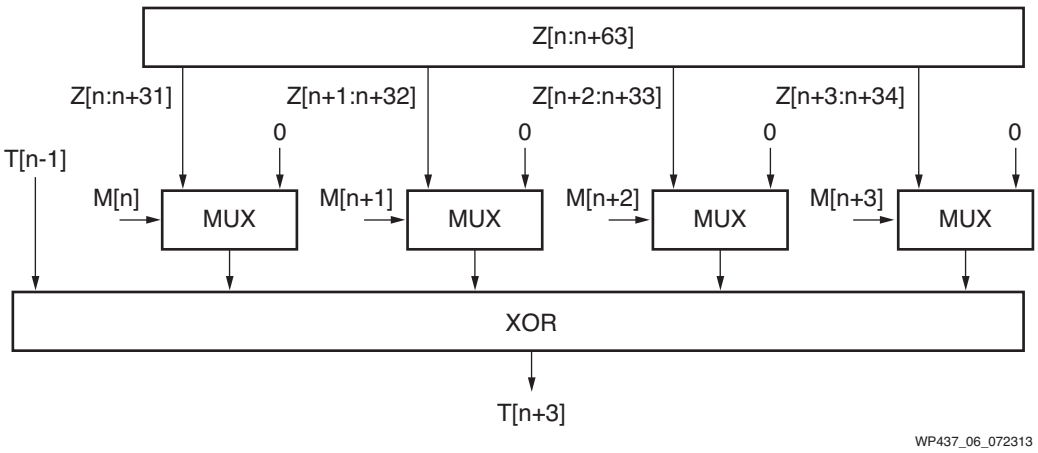

Figure 6: **Parallelized EIA-3 Calculator**

Table 4 summarizes the logic resource requirements of the accelerator using the above-mentioned optimization techniques. The default HLS synthesis options are used and the target clock frequency is set to 100 MHz. For the integration of the accelerator with the processor system, the target clock frequency is increased to 150 MHz. Though the F_{MAX} for the accelerator is only 141 MHz, the integrated system met the timing constraint of 150 MHz using the default PlanAhead implementation options.

Table 4: **Baseline Accelerator Implementation Results on 7Z045T-2**

	Slice	FF	LUT	DSP	BRAM18K	F_{MAX}
Accelerator Core Only	975 (1.8%)	2,374 (0.5%)	2,920 (1.3%)	0	2 (0.2%)	141 MHz
With AXI Interconnection	1,826 (3.3%)	3,856 (0.9%)	4,513 (2.1%)	0	2 (0.2%)	154 MHz

Compared to the implementation results of the baseline accelerator shown in Table 3, the optimization improves the accelerator performance without increasing the hardware resources. Because the *S* functions are realized in look-up tables instead of block RAMs, the number of block RAMs is reduced from five to two at the cost of only a slight increase in LUT usage.

The EEA-3/EIA-3 accelerator occupies only a small portion of total programmable logic resources available in the 7Z045T-2 SoC; hence, it is feasible to instantiate multiple accelerators for higher data throughput. The scalability of programmable logic therefore offers an advantage over conventional hardware accelerators, which are capable only of a fixed throughput.

Further optimization of the *C* code is still possible to improve throughput performance. However, this would require a more in-depth understanding of the processor peripherals and is beyond the scope of this white paper.

Performance Results

The batch-mode testbench developed for the software solution can be reused for the validation of the programmable logic based accelerator. The Zynq-7000 All Programmable SoC can actually be viewed as a software/hardware co-simulation platform that runs thousands of times faster than a pure software-based simulation, resulting in a significant reduction in verification time.

Table 5: **EEA-3/EIA-3 Programmable Logic Accelerator Solution Test Results**

Data Block Length (bits)	Test Data Size (Mb)	Baseline Version		Optimized	
		Elapsed Time (ms)	Throughput (Mb/s)	Elapsed Time (ms)	Throughput (Mb/s)
EEA-3 (with Hardware Accelerator):					
256	250.5	10,374	24.2	9,290	27.0
4,096	509.0	9,363	54.4	3,500	145.5
65,536	535.0	9,074	59.0	2,650	201.9
16,777,216	520.1	9,334	55.7	2,511	207.1
EIA-3 (with Hardware Accelerator):					
256	250.5	19,057	13.1	10,828	23.1
4,096	509.0	26,365	19.3	5,849	87.0
65,536	535.0	26,908	19.9	5,056	105.8
16,777,216	520.1	26,106	19.9	4,844	107.4

The EEA-3/EIA-3 accelerator generated by Vivado HLS passed the functional validation without any error on the Xilinx ZC706 evaluation board. The measured data throughputs are summarized in Table 5 and compared in Figure 7. Observations include:

- The *baseline* accelerator achieves 50 Mb/s throughput for EEA-3 and 19 Mb/s for EIA-3. This result is better than what is achieved by the software-only solution, in which an entire ARM processor is dedicated to the cryptographic task in debug mode. Essentially, it means that the baseline accelerator serves the purpose of offloading the cryptographic work from the processor and that its performance is adequate for initial software development.
- The accelerator, with some simple optimizations, achieves 200 Mb/s throughput for EEA-3 and 100 Mb/s for EIA-3. This performance is *much* better than software-only solution and fulfills the requirement of a small cell, which typically sees a throughput around 150 Mb/s.

- When *small* block sizes are used, it can be seen that the costs of header processing and processor/accelerator handshaking is quite high; as a result, the benefits provided by the hardware accelerator are significantly lower compared to the benefits obtained using *large* block sizes. Using a larger block size dramatically increases the performance benefit provided by the hardware accelerator.

From these metrics, therefore, it can be concluded that both the baseline and optimized hardware accelerators serve their purpose of offloading cryptographic work from the processors, resulting in a desirable increase in throughput. These benefits are illustrated in Figure 7.

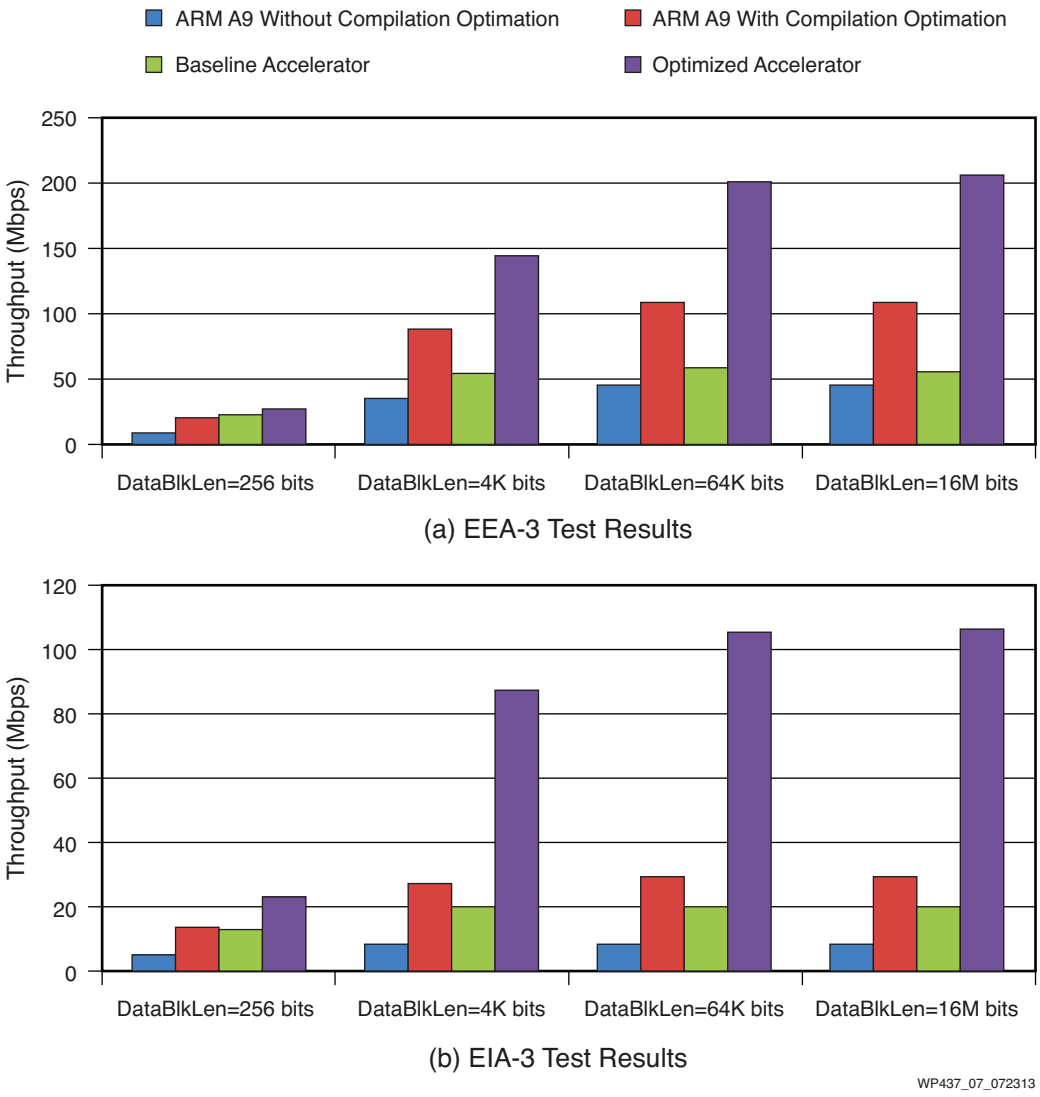

Figure 7: **Test Results: Performance Comparison of EEA-3/EIA-3 Solutions**

Summary

The Xilinx Zynq-7000 All Programmable SoC provides a flexible platform for a software-oriented development methodology where hardware accelerators can be easily generated from C code and automatically connected to the processor system via internal AXI interfaces. The platform offers programmability for evolving technologies like LTE and new standards including, but not limited to, the ZUC algorithm. This white paper illustrates the advantages of Zynq-7000 All Programmable SoC- based design methodology using the Vivado Design Suite, which facilitates an optimal hardware/software partitioning of the system functionality for better performance.

Call to Action

- Read the Zynq-7000 All Programmable SoC User Guides [Ref 7][Ref 12].
- Read the Vivado HLS User Guide and Tutorial [Ref 8]
- Download the ZUC standards [Ref 5][Ref 6] and build an EEA-3/EIA-3 accelerator on a Zynq-7000 All Programmable SoC
- Extend the methodology introduced by this paper to other 3GPP air interface cryptographic methods — e.g., EEA-1/EIA-1 [Ref 3] and EEA-2/EIA-2 [Ref 4].

References

1. 3GPP TS 33.401, *3GPP System Architecture Evolution (SAE); Security Architecture*, Release 12, v12.7.0, Mar 2013.
2. C.B. Sankaran, "Network Access Security in Next-Generation 3GPP Systems: A Tutorial," in *IEEE Communications Magazine*, Feb 2009, page 84-91.
3. 3GPP TS 35.215, *Confidentiality and Integrity Algorithms UEA2 & UIA2; Document 2: SNOW 3G Specification,* Release 11, Sep 2012.
4. NIST: *Advanced Encryption Standard (AES) (FIPS PUB 197),* 1997.
5. ETSI/SAGE Specification, *Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification,* v1.6, Mar 2011.
6. ETSI/SAGE Specification, *Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 Specification*, v1.6, Mar 2012.
7. Xilinx UG585, *Zynq-7000 All Programmable SoC Technical Reference Manual,* v1.6, Jun 2013.
8. Xilinx UG902, *Vivado Design Suite User Guide: High-Level Synthesis,* v2013.2, Jun 2013.
9. P. Kitsos, N. Sklavos, and A.N. Skodras, "An FPGA Implementation of the ZUC Stream Cipher," in the proceedings of the 14th Euromicro Conference on Digital System Design, 2011, page 815-817.
10. Xilinx UG683, *EDK Concepts, Tools, and Techniques — A Hands-On Guide to Effective Embedded System Design*, v14.6, Jun 2013.
11. Xilinx UG632, *PlanAhead User Guide*, v14.3, Oct 2012.
12. Xilinx UG873, *Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT) — A Hands-On Guide to Effective Embedded System Design*, v14.6, Jun 2013.
13. Xilinx UG954, *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide*, v1.2, Apr 2013.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
08/29/13	1.0	Initial Xilinx release.

Notice of Disclaimer