2016 IEEE International Parallel and Distributed Processing Symposium Workshops

# A 16-bit Reconfigurable Encryption Processor for $\pi$-Cipher

Mohamed El-Hadedy*, Hristina Mihajloska†, Danilo Gligoroski‡, Amit Kulkarni§, Dirk Stroobandt§ and Kevin Skadron¶
*Coordinated Science Laboratory, University of Illinois, 1308 West Main st, Urbana, IL-61801, USA.
Email:hadedy@Illinois.edu
†Faculty of Computer Science and Engineering, Cyril and Methodius University, Skopje, Macedonia.
Email:hristina@finki.ukim.mk
‡Telematics Department, The Norwegian University of Science and Technology, Trondheim, Norway.
Email:danilog@item.ntnu.no
§ELIS department, Computer Systems Lab, Ghent University, Ghent, Belgium.
Email:{Amit.Kulkarni, Dirk.Stroobandt}@UGent.be
¶Department of Computer Science, University of Virginia, Charlottesville, Virginia, USA.
Email:skadron@virginia.edu

***Abstract*—This paper presents an improved hardware implementation of a 16-bit ARX (Add, Rotate, and Xor) engine for one of the CAESAR second-round competition candidates, $\pi$-Cipher, implemented on an FPGA. $\pi$-Cipher is a nonce-based authenticated encryption cipher with associated data. The security of the $\pi$-Cipher relies on an ARX based permutation function, which is denoted as a $\pi$-function. The proposed ARX engine has been implemented in just 266 slices, which includes the buffers of the input and the output. It can be clocked at 347 MHz. Also, in this paper, a message processor based on the proposed ARX engine is introduced. The message processor has been implemented in 1114 slices and it can be clocked at 250 MHz. The functionality of the proposed ARX engine was verified on the Xilinx Virtex-7. The new design of the ARX engine allows for almost four times speedup in performance while consuming only 17% larger area than previously published work. We extend our message processor implementation by using parametrized reconfiguration technique after which an area reduction of 27 slices is observed.**

***Keywords*-FPGA; Authenticated encryption; CAESAR; Cryptographic competitions; $\pi$-Cipher; TLUT; micro-reconfiguration; parameterized configuration;**

## I. Introduction

Cryptography is essential to the modern IT society. In 2013, the National Institute of Standards and Technology NIST funded a new Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [1] to identify a portfolio of authenticated ciphers that offer advantages over the current AES-GCM and are suitable for widespread adoption as a next-generation standard. In addition to security considerations, availability of an efficient *hardware* implementation will be a factor in the full CAESAR selection. A popular way to construct simple operations and fast cryptographic primitives is the so-called ARX design, where the construction only uses Additions (A $\boxplus$ B), Rotations (A $\lll$ r), and Xors (A $\oplus$ B). These operations are very simple and can be implemented efficiently in software or compactly in hardware. As a proof of concept two of the SHA-3 finalists, Blake [2], [3] and Skein [4], follow this design strategy, also MD/SHA family [5] [6] are referred to as ARX, stream ciphers such as Salsa20 [7] and ChaCha [8], and block ciphers, such as TEA [9] and HIGHT [10]. In the ongoing CAESAR competition, few of the candidates that passed in the second round are ARX based. One of them is $\pi$-Cipher, which we use to drive the design of a generic ARX crypto-processing architecture that can therefore support a variety of crypto-ARX primitives. In this paper, we introduce the first implementation of the $\pi$-function with 16-bit words using the new compact ARX engine. Although the introduced custom ARX engine has less flexibility compared to prior work [11] [12] [13], the proposed engine allows for almost four times speedup in performance while consuming only 17% larger area than previously published work [11]. The message processor, $\pi$-function, and ARX engine for 16-bit version of $\pi$-Cipher are implemented and evaluated in an FPGA using a Xilinx Virtex-7. In order to optimize the implementation of the processor on the FPGA, we make use of parameterized configuration technique [14] that optimizes the key generation module and contributes in the reduction of the resource utilization of the processor.

The rest of the paper is organized as follows: in Section II, we present a detailed description on $\pi$-Cipher. In Section III, we present the ARX engine architecture that creates cryptographic primitives followed by the description of the $\pi$-function IV. The $\pi$-function encapsulates the ARX engine and together with key generator forms the message processor that is presented in Section V. In Section VI, we present the hardware implementation and the results of the encryption processor and discuss more them. In Section VII we briefly describe the parameterized configuration tool flow along with the improvements in the results followed by we conclude in Section VIII.

A part of this work was done while Dr. Mohamed EL-Hadedy was a Research Associate with the Department of Computer Science at the University of Virginia, 85 Engineer's Way, P.O.Box 400740, Charlottesville, Virginia

/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.27

Table I: An algorithmic description of the ARX operation $*$ for 4-tuples of 16–bit words $(X * Y)$.

**Input:** $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$ where $X_i$ and $Y_i$ are 16–bit variables.
**Output:** $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ where $Z_i$ are 16–bit variables.
**Temporary 16–bit variables:** $T_0, \ldots, T_{11}$.

$$
\begin{array}{lcl}
T_0 & \leftarrow & ROTL^1(const_1 + X_0 + X_1 + X_2); \\
T_1 & \leftarrow & ROTL^4(const_2 + X_0 + X_1 + X_3); \\
T_2 & \leftarrow & ROTL^9(const_3 + X_0 + X_2 + X_3); \\
T_3 & \leftarrow & ROTL^{11}(const_4 + X_1 + X_2 + X_3); \\
T_4 & \leftarrow & T_0 \oplus T_1 \oplus T_3; \\
T_5 & \leftarrow & T_0 \oplus T_1 \oplus T_2; \\
T_6 & \leftarrow & T_1 \oplus T_2 \oplus T_3; \\
T_7 & \leftarrow & T_0 \oplus T_2 \oplus T_3; \\
 & & \\
T_0 & \leftarrow & ROTL^2(const_5 + Y_0 + Y_2 + Y_3); \\
T_1 & \leftarrow & ROTL^5(const_6 + Y_1 + Y_2 + Y_3); \\
T_2 & \leftarrow & ROTL^7(const_7 + Y_0 + Y_1 + Y_2); \\
T_3 & \leftarrow & ROTL^{13}(const_8 + Y_0 + Y_1 + Y_3); \\
T_8 & \leftarrow & T_1 \oplus T_2 \oplus T_3; \\
T_9 & \leftarrow & T_0 \oplus T_2 \oplus T_3; \\
T_{10} & \leftarrow & T_0 \oplus T_1 \oplus T_3; \\
T_{11} & \leftarrow & T_0 \oplus T_1 \oplus T_2; \\
 & & \\
Z_3 & \leftarrow & T_4 + T_8; \\
Z_0 & \leftarrow & T_5 + T_9; \\
Z_1 & \leftarrow & T_6 + T_{10}; \\
Z_2 & \leftarrow & T_7 + T_{11};
\end{array}
$$

## II. $\pi$-CIPHER

$\pi$-Cipher is a nonce-based authenticated encryption cipher with associated data. This cipher is a parallelizable and incremental, sponge-based design. It is designed to accommodate words and blocks with different sizes, and different security levels [15], [16]. $\pi$-Cipher's design is based on several canonical cryptographic concepts but has some intrinsic new features. The encryption/authentication operation of $\pi$-Cipher can be described in five phases:padding, initialization, processing the associated data, processing the secret message number (SMN) and processing the message. In all of them a main role in the security and design perspective has the permutation function, denoted as $\pi$-function. It is an ARX-based permutation function that consists of three rounds, while each round consists of eight ARX operations blocks, denoted as $*$ operations. Every $*$ operation has as input two 4-tuples of $\omega$-bit words ($\omega$ = 16, 32, 64) and performs in total 52 ARX operations on them. An algorithmic description of the $*$ operation is given in Table I. The $*$-operation operates with 8 constants $(const_1, \ldots, const_8)$ consuming $8 \times \omega$ bits of memory.

One round of the $\pi$-function uses two consecutive transformations on the input string chunks $(I_1, \ldots, I_N)$. A generic description of the algorithm for one round of the $\pi$-function is given in Table II. This round is sequentially repeated three times. For every round, different pairs $(C_1, C_2)$ of the round constants are used. The total memory space that is occupied by them is $8 \times 4 \times \omega$ bits.

Table II: A generic algorithmic description of one round of the $\pi$-function

**Input:** $I_1, \ldots, I_N$ and $C_1, C_2$ where $I_i$ are input string chunks (4-tuples of 16-bit words) and $C_1$ and $C_2$ are round constants (4-tuples of 16-bit words).
**Output:** $J_1, \ldots, J_N$

$J_1 = C_1 * I_1$
For $i = 2$ to $N$ do
    $J_i = J_{i-1} * I_i$
$J_N = J_N * C_2$
For $i = N - 1$ downto 1 do
    $J_N = J_N * J_{N+1}$

More details about the $\pi$-Cipher can be found in the official documentation of the cipher [15].

## III. ARX ENGINE ARCHITECTURE

Because of the nature of the $\pi$-function, to process two inputs $X$ and $Y$ with the transformations $\mu$ and $\nu$ independently, the ARX engine consists of a dual core processor, with the cores running in parallel. Each core has a 64-bit buffer and receives the data from one 16-bit input port. Also it has sixteen read ports, where each port is controlled by 2-bit address bits. The total width of the address port is 32 bits, with each bit coming from the control unit as it is shown in Figure 1. Once the data is written on the reading ports, they are then processed by four 16-bit adders. Next, the results of the adders are processed by the 16-bit rotator unit. Every core needs to do XOR operations before outputs the result. The XOR Bank is also controlled by the signals from the control unit, and it is responsible for mixing the output from the Rotator unit. The results from the processor's cores are sent to the other four ripple-carry 16-bit adders, and after that stored into the 64-bit buffer FIFO. Once the *Z* buses data are stored in the FIFO, the control unit sets its flag *Arx_flag* high. This is for denoting that the ARX engine has processed the data and it is ready to receive new data from the input ports.

### A. Adder

The 16-bit ARX engine relies on using eight 16-bit adders to process the data that comes out of the buffers, and other four 16-bit adders that calculate the final result, as shown in Figure 1. Each four-input port in the 16-bit adders consist of three 16-bit ripple-carry adders. The first two adders are used to add the buffer results, and the last adder is used to sum the two results from the previous adders. All adders in the engine are controlled by several control bits from the control unit.

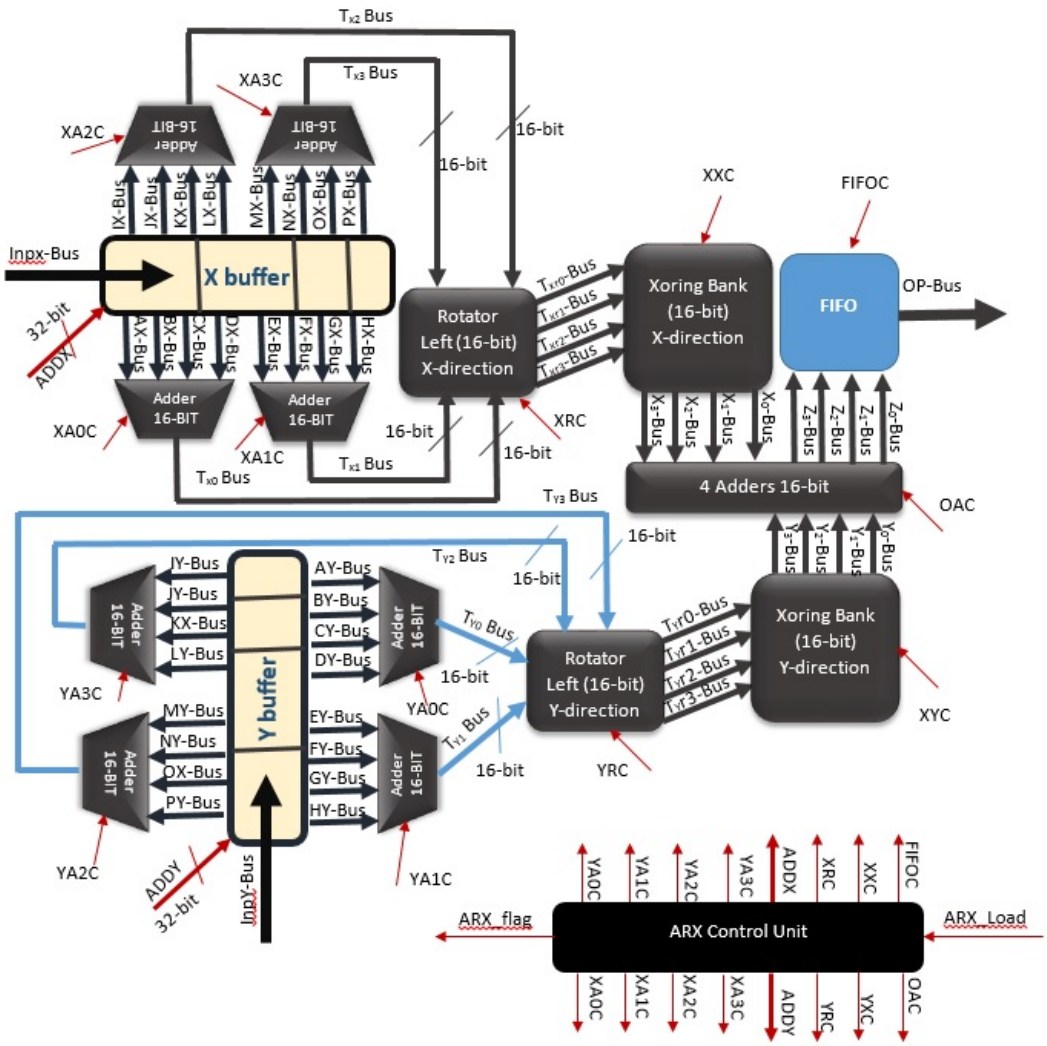



Figure 1: A 16-bit ARX Engine

### B. Rotator

The rotator is responsible for left rotating the adder's outputs by different rotation values based on the algorithmic description of the $\pi$-Cipher [15].

### C. XOR Bank

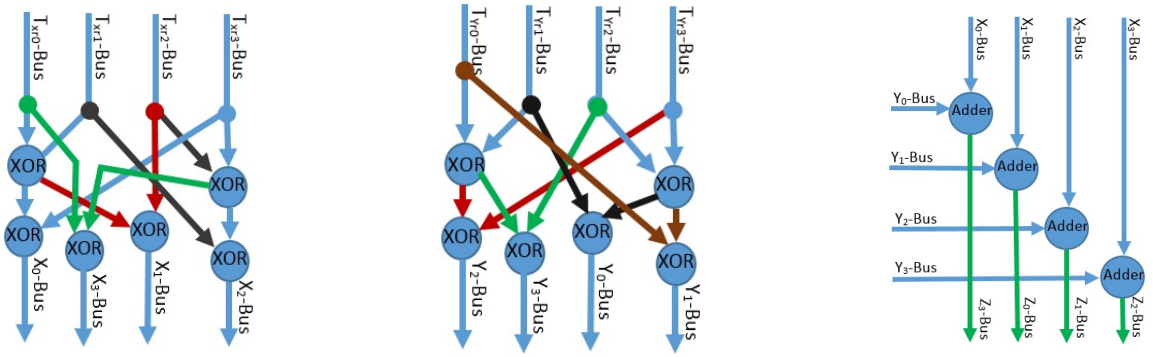
The XOR banks, as shown in Figure 2a and Figure 2b, are intermediate stages between the rotators outputs and the final stage of the engine, to maximize the diffusion of the bits [15]. The output data of both XOR banks are added to each other by using four 16-bit parallel ripple-carry adders, as shown in Figure 2c. Next, the adders output data is stored in the FIFO.

### D. ARX Control Unit

The ARX control unit has been built based on a Moore finite state machine. It consists of six sequential states, which are controlling the operation from the buffers to the adding bank stage. The buffer state consists of two counters, one for receiving the data from the input ports and store it in the buffer, and the other counter is used to read the stored data on the buffers reading ports. This operation is followed by rotators state, which controls the several parallel left rotation operations from the buffer side to the XOR bank. Once the rotators state is completed, the XOR state starts to control the four 16-bit parallel XOR operation. At the end of the state there is an internal signal, which it initiates the FIFO state, that controls the storing process on the FIFO. In total, the ARX engine takes seven cycles to execute the input data sets.

## IV. The $\pi$-function

The ARX engine introduced in Section 3 is used to implement the $\pi$-function. As shown in Figure 3, a $\pi$-function core




(a) X-direction (b) Y-direction (c) Adding Bank

Figure 2: XOR & Add Banks for both processors

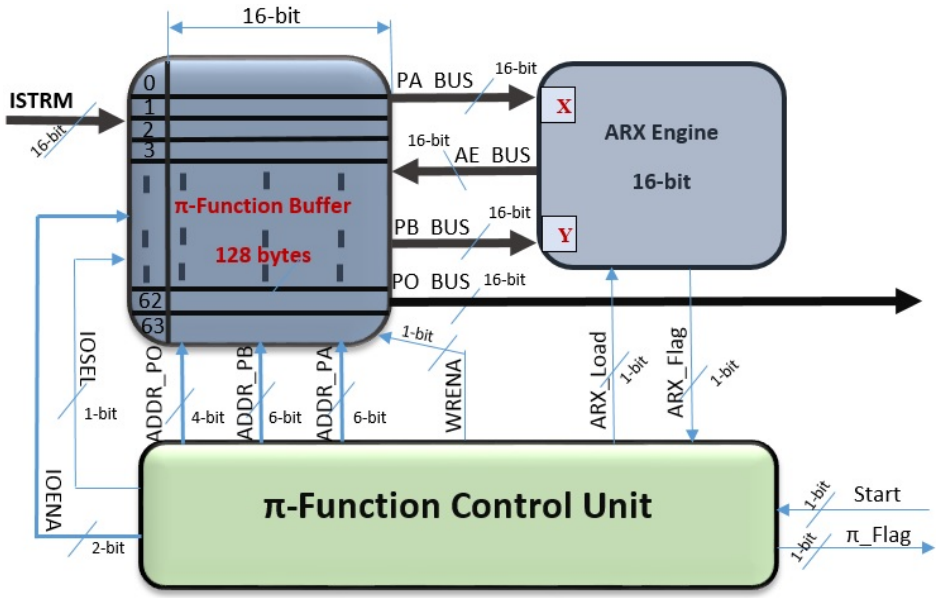



Figure 3: $\pi$-function core

consists of 128 byte memory ($\pi$-function buffer) and it stores the 256-bit input data along with the internal constants (six 64-bit constants). As described in Section 2, the $\pi$-function consists of three rounds, where each round consists of eight ARX engines running sequentially as shown in Figure 4.

The $\pi$-function control unit relies on Moore-style state machine with 32 states; each state consists of two sub-states, one to control the data direction from the buffer to the ARX engine, and another one to control the data flow from the ARX engine to the memory buffer.

### A. $\pi$-function Control Unit

This unit controls the $\pi$-function buffer's input by 1-bit signal, which chooses whether the input comes from the input data set or ARX engine. The $\pi$-function buffer is divided into two major parts; the first part is 96 bytes long, and stores the $\pi$-function internal constants, and the input data sets; the second part is 32 bytes long and it is reserved for intermediate results between the rounds. The $\pi$-function control unit consists of 32 states running sequentially, as described in Section IV. Each state consists of two sub-states. The first one controls the data flow from the input port to the function buffer, while the second one starts by moving from the $\pi$-function buffer to the data ARX engine buffer. This is done by setting the *ARX_Load* signal to high for one cycle. Then, the control unit of the ARX engine will take care of the data processing, until the *ARX_Flag* becomes high. That means the ARX engine has finished the computational stage and it is ready to pass the executed data to the $\pi$-function buffer. Meanwhile, the *ARX_Flag* signal initiates the next state by rising the *WRENA* signal, and choosing the looping path instead of the input path, by rising the IOSEL signal to high. This operation performs four times

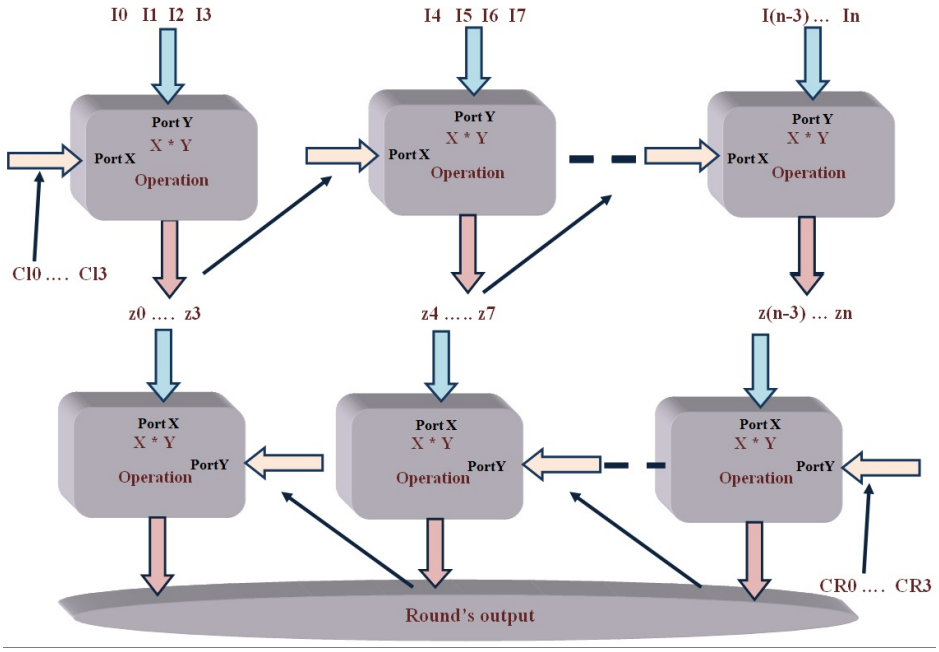



Figure 4: $\pi$-function round

on every round. This will be followed by the exchange in the data direction flow between the ARX engine input ports. The exchange is made based on the mathematical model [15] shown in Figure 3. Using the introduced scheme, the *pi*-function three rounds data takes 675 cycles to complete.

## V. Message Processor

The $\pi$-function core from Section IV is used to implement the message processor. As shown in Figure 5, a message processor consists of the initialization generator (KPIG), Data Bus $2 \times 1$ multiplexer, 16-bit ALU, $\pi$-function core, 16 bytes ciphertext buffer, 16 bytes tag buffer and message processor control unit (MPCU). The MPCU allows the user to choose either receive key, PMN, IS, or all of them together as inputs through the 4-bit PC control signal. While the KPIG is storing the key and PMN, the 16-bit ALU is storing the message in its local 32 byte buffer (the first 64 bytes from the buffer are reserved for the counter, and the rest for the message block).

Considering there is no associated data and SMN in this design, the output of the KPIG is considered as the CIS (Common Internal State) for the message blocks [15], [16].

Once the initialization phase is done, KPIG will send the data to the $\pi$-function core through the Data Bus multiplexer. This operation is controlled by the MPCU through *DBSEL*, *PF_start*, and *Kpmn_flag* signals.

After $\pi$-function, CIS is generated and its copy is stored for further use in the IS buffer of the KPIG.

The 16-bit ALU has a 32 bytes buffer for storing the message block. The first 64-bits of the buffer are reserved for the counter. Another 32 bytes buffer located in the ALU is used for storing the result of the $\pi$-function. After the first invocation of the $\pi$-function the result as a ciphertext is redirected to the Ciphertext buffer, and after the second invocation the result as a tag is stored in the Tag buffer. All this actions are controlled with the signals from the MPCU.

### A. KPIG

The KPIG contains two buffers, each of them is represented with 32 bytes. The first buffer is used to store the key and the public part of the nonce - PMN (public message number). The other one is used to store the result after the initialization phase, CIS value (Common Internal State). The KPIG has eight states that are controlled by 3-bit signals as shown in Table III. The KPIG as a standalone unit has been implemented in just 53 slices and can run at 460 MHz.

### B. ALU

ALU contains two buffers, and each buffer is represented in 32 bytes and arithmetic and logic unit. The rule of the ALU is to xor the message with the selected data from the output of the $\pi$-function or just pass the $\pi$-function's output without any changes based on the ALU_mode value. The ALU as a standalone unit has been implemented in just 118 slices and can run at 408 MHz.

### C. MPCU

MPCU is the control unit of the message processor. It consists of five sequential states based on the Moore's finite state machine. The initial state is used to clear all the control signals and prepare the message processor to receive new

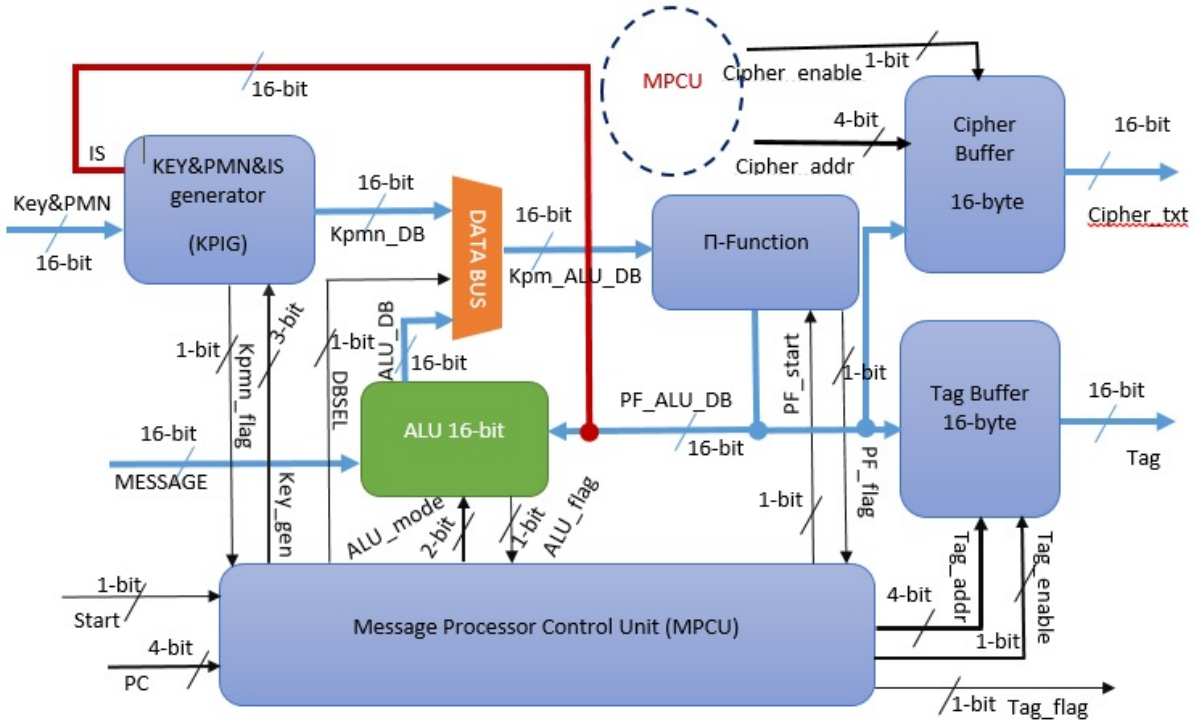



Figure 5: Message Processor

counter and message blocks. Once the start signal is set to high, the load message state and generate key state control both ALU and KPIG units to store the data based on the value of the PC signal. In the automatic mode, the PC value is "0000", which means that the KEY and PMN are received from the input port. For the manual mode, the user can choose whether he wants to upload the KPIG buffers by the key, PMN, IS, or all of them together. When the *kpmn_flag* is set high, the KPIG's output will be processed by the $\pi$-function through the ALU unit. Once the counter is processed by the ALU, the *ALU_flag* control signal will start a new state in MPCU to process the ALU outputs by $\pi$-function. This is followed by another state that change the ALU mode and process the message block by the $\pi$-function. This time when the *PF_flag* becomes high, the tag buffer will store the final data if there are no more parts of the message that need to be processed. When the *Tag_flag* is set high by the MPCU, this means the final data have been started to be written on the Tag data bus.

## VI. Hardware Implementation

The ARX engine, the $\pi$-function core, and the message processor for 16-bit version of $\pi$-Cipher were synthesized for and verified on the Xilinx Virtex-7 architecture specifically, a XC7VX485T-2FFG1761. They have been described on the FPGA platform in VHDL and were synthesized using ISE design suite 14.7.

### A. ARX Performance

$$Throughput = \frac{Number\ of\ input\ bits \times Max frequency}{Number\ of\ clock\ cycles\ per\ block} \quad (1)$$

As discussed in Section II, the $\pi$-function consists of 3 rounds, with each round having eight ARX operation blocks. The throughput of the design is given in Equation 1. The area, clock rate, and throughput of the custom ARX processing units are summarized in Table IV.

### B. ARX Engine Performance

The 16-bit version of the ARX engine has been implemented in 266 slices running at 347 MHz, achieving 4.34 Gpbs on the Xilinx Virtex-7 platform. The input data of the ARX engine takes around seven cycles to be executed, plus four cycles to store the input data in the ARX buffer and one cycle to move the executed data out of the engine. Even though the prior work [11] is more flexible than our proposed custom ARX engine, ours is almost four times faster than the previous implementation, as shown in Table IV. We attribute this to the fact that the previous implementation uses a native 64-bit Arithmetic and Logic Unit (ALU), which we suspect lowers the achievable frequency by increasing the critical path. The improvements we achieved with our introduced ARX engine would certainly decrease the total execution time that is needed to complete all three rounds of the $\pi$-function.

Table III: Description of the KPIG eight states

| key_gen | State | Function |
|---|---|---|
| 000 | Initial | Clear the buffers and internal control signals |
| 001 | GENKEY_ISRD | Store the key, kpmn_DB data bus ← ((Key ‖ PMN ‖ 10*) ⊕ IS) |
| 010 | GENKEY_ISGENRD | Store the key and IS. kpmn_DB data bus ← ((Key ‖ PMN ‖ 10*) ⊕ IS) |
| 011 | GENPMN_ISGENRD | Store the key, PMN, and IS. kpmn_DB data bus ← ((Key ‖ PMN ‖ 10*) ⊕ IS) |
| 100 | GENPMN_ISRD | Store the PMN. kpmn_DB data bus ← ((Key ‖ PMN ‖ 10*) ⊕ IS) |
| 101 | GENKEYPMN_GENISRD | Store the PMN, Key, and IS. kpmn_DB data bus ← ((Key ‖ PMN ‖ 10*) ⊕ IS) |
| 110 | GENKEYPMN_ISRD | Store the PMN and Key. kpmn_DB data bus ← ((Key ‖ PMN ‖ 10*) ⊕ IS) |
| 111 | GENIS | Store IS |

Table IV: The ARX Performance ($\pi$16-Cipher)

| | ref [11] | ARX Engine |
|---|---|---|
| Throughput | 1.2 Gpbs | 4.34 Gpbs |
| Area(Slices) | 227 | 266 |
| Frequency(MHz) | 250 | 347 |
| Throughput/Area (Mbps/slices) | 5.4 | 16.71 |

### C. The $\pi$-function Performance

In this paper, the $\pi$-function has been implemented based on the introduced 16-bit ARX engine in just 971 slices, running at 250 MHz, and achieving 95 Mbps. The total area of the $\pi$-function can be reduced by decreasing the states of the $\pi$-function control unit from 32 to just 16 (total number of ARX engines in each round $\times$ number of rounds). Even though the ARX engine can run at 347 MHz, the $\pi$-function can only run at 250 MHz. The drop in the frequency is due to the additional modules needed, such as the function's buffer and the control unit, which controls the data flow of the function. This might end up increasing the critical path.

### D. Message Processor Performance

In this paper, the message processor was implemented in 1114 slices based on the introduced ARX engine, runs at 250 MHz and achieves 15 Mbps for a processing message length of 128 bits in 2165 cycles. The total area used to implement one message processor is just 1% of the whole FPGA area. That means we can use as many as almost 100 message processors to run 1600 bytes of the message in parallel at 250 MHz, achieving 1.5 Gbps.

## VII. Parameterized Configuration

We use parameterized FPGA configuration technique to implement the message processor as an parameterized application. An application is said to be parameterized if some of its input values change infrequently compared to the rest called parameters. The technique enables us to implement the parameterized application with less FPGA resources (mainly Look Up Tables) compared to the classic static (conventional) implementation. This helps in shortening the critical path of the design and hence it also improves the processor's performance [17].

The tool flow used to generate the parameterized configuration consists of two stages: a *generic stage* and a *specialization stage*. In the generic stage, a parameterized application (or design) described in a Hardware Description Language (HDL) is processed to yield a Partial Parameterized Configuration (PPC) and a Template Configuration (TC) as depicted in Figure 6.

The following tool flow steps explains the generic stage and are adapted from the conventional tool flow [14].

### A. Synthesis

In this step, the HDL design is converted into a network of logic gates. The parameter inputs described in the HDL are annotated by –PARAM and this annotation makes the difference between regular inputs and parameter inputs. The parameter inputs are also a part of the Boolean network of logic gates produced after synthesis.

### B. Technology Mapping

During the mapping stage, the synthesized Boolean network is mapped onto the available resources of the target FPGA architecture such as LookUp Tables (LUTs), DSP blocks and BRAMs while optimization of circuit area and speed (LUTs depth) are being taken into consideration. The conventional mapping tool would map to the static LUTs and hence it would result in the conventional bitstreams after place and route. To generate a parameterized bitstream, authors in [17] change the conventional mapping tool to a tunable version, TMAP, so that the Boolean functions of parameter inputs are mapped to Tunable LookUp Tables (TLUTs). These are virtual LUTs that differ from conventional LUTs in the fact that their lookup entries are defined as the boolean functions of the parameter inputs instead of static ones and zeros. The truth table entries will be reconfigured upon every change in parameter values.

Presently, the parameterization of BRAM and DSP blocks is not yet possible but parameterization of the routing switches called TCONs is established at the virtual FPGA level. However, the practical implementation in commercial FPGAs is yet to be done [18]. The TMAP mapping algorithm is described in [17] and can be integrated with the conventional Xilinx tool flow which is explained in [19].

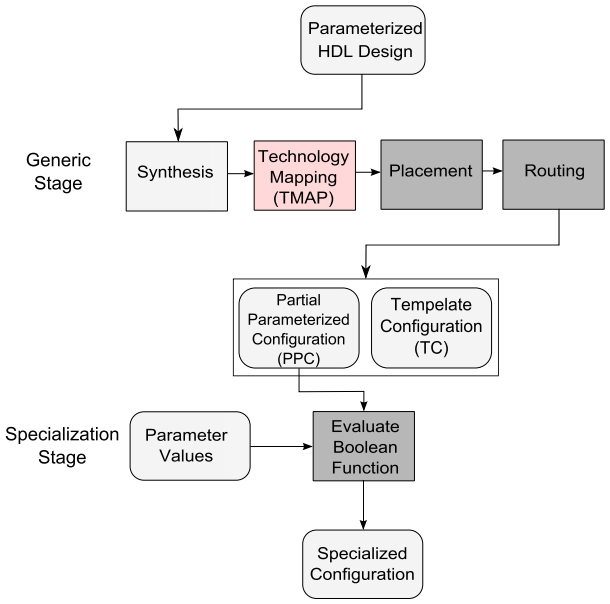



Figure 6: Parameterized Configuration tool flow.

### C. Placement and Routing

In the placement step, the mapped resources are placed or associated to specific blocks of the target FPGA architecture. Extensive optimization is considered so that interconnect wire length and interconnect delay is minimized. The router configures the physical switch blocks to achieve the required interconnect according to the circuit. Since the placement and routing does not depend on parameter inputs or TLUTs, a conventional placer and router can be used.

The final output of the generic stage is the Template Configuration (TC) and Partial Parameterized Configuration (PPC). TC is a static bitstream which contains static ones and zeros, which are used for configuring during the start of the FPGA. The PPC contains sets of multi-output Boolean functions of the parameter inputs. The PPC needs to undergo the specialization stage, along with parameter values to produce an efficient specialized configuration.

In the Specialization stage, the Boolean functions are evaluated for specific values of the parameters thus generating specialized bitstreams. For every infrequent change in parameter values, the Boolean functions are evaluated by a Specialized Configuration Generator (SCG). The SCG can be implemented on an embedded processor such as the PowerPC or the ARM cortex-A9 present within the FPGA core.

The SCG reconfigures the FPGA via a configuration interface called HWICAP, by swapping the specialized bitstreams into the FPGA configuration memory. The HWICAP encapsulates the ICAP primitive (port) of the FPGA and forms a controller that orchestrates the swapping of specialized bitstreams via the interface port ICAP. The bitstreams are accessed in the form of frames, and a frame is defined as the smallest addressable element of an FPGA configuration data. Each frame has its unique frame address that can be used to point to the frame during the reconfiguration. The software to implement the Parameterized Configuration is available as an open source project on GitHub [20].

### D. The HWICAP driver: "$XHwIcap\_SetClbBits$"

The HWICAP supports a reconfiguration driver function called "$XHwIcap\_SetClbBits$" to perform the reconfiguration. The function accepts two crucial function arguments:

1) Location co-ordinates of a TLUT: This information is used to generate the frame address that is used to point to the frame that contains truth table entries of the TLUT.
2) Truth table entries: These are the specialized bits generated after the specialization stage. The TLUT truth table entries need to be overwritten with these specialized bits.

The reconfiguration takes place in 3 major steps:

1) *Read frames*: using the frame address, a set of four consecutive frames containing the truth table entries of a TLUT are read from the FPGA configuration memory.
2) *Modify frames*: the current truth table entries of a TLUT are replaced by the specialized bits, thus the modified frames contain specialized bitstreams.
3) *Write-back frames*: using the same frame address, the modified four frames (specialized frames) are written back to the FPGA configuration memory, thus accomplishing the *micro-reconfiguration*.

*Micro-reconfiguration* is a fine-grain form of reconfiguration tailored to implement parameterized applications.

The reconfiguration time is a major overhead of the parameterized configuration approach. Using the HWICAP, the time taken to reconfigure one TLUT is $230\mu s$. However, with custom reconfiguration controller such as MiCAP and MiCAP-Pro [21] and the techniques used in [22] the reconfiguration time can be effectively suppressed by the factor upto 37.

### E. Parameterized configuration for the message processor

We make use of parameterized configuration technique to implement the message processor with the input: "Key&MN" as a parameter input. The input "Key&MN" is used to generate the encryption key of the message processor. For every change in key input, the TLUTs whose configuration hold the key input values are reconfigured with the new key value. This technique optimizes the Key generator module (KPIG) and therefore reduces utilization area by 27 slices. However, this optimization comes at the cost of reconfiguration time. The results in terms of LUTs resource utilization is tabulated in Table V.

Table V: Parameterized message processor results

| Implementation | LUTs(TLUTs) | Reconfiguration time($ms$) |
|---|---|---|
| Conventional | 9052(0) | 0 |
| Parameterized Configuration | 8942(256) | 600 |

Clearly, we observe a difference of 108 LUTs in the resource area optimization. Since each slice contain 4 LUTs in the Xilinx Virtex-7, the resource optimized in terms of slices is 27. The overall effect on the performance of the message processor was (to be estimated). Since the key input to the key generator module doesn't change frequently, it is worth to accept the performance improvement of the message processor by (to be estimated) at the cost of reconfiguration time of 600 ms.

## VIII. Conclusions and Future Work

This paper presents the design and analysis of the reference implementation of the ARX (Add, Rotate, and XOR) engine of 16-bit version of $\pi$-Cipher on the FPGA platform. $\pi$-Cipher is one of the second-round candidates of the ongoing CAESAR competition for authenticated ciphers. The proposed ARX engine has been implemented in just 266 slices on the Xilinx Virtex-7 platform, achieving a throughput of 4.34 Gpbs at 347 MHz. Comparing the result with the prior work, the introduced ARX engine is almost four times faster. The $\pi$-function, which is the most expensive element from the design, has been implemented as well. In order to optimize the size, the message processor has been implemented using parameterized configuration technique that optimizes key generator module by saving 27 slices at the cost of the reconfiguration time of 600 ms. Therefore, the parameterized configuration helps to investigate the trade-off between the reconfiguration time and the resource utilization. However, the reconfiguration speed can be improved with custom reconfiguration controllers and drivers described in [23]. In the future, we will further optimize the $\pi$-function implementation and the message processor control units in order to decrease the number of the states. This will increase the performance while decreasing the total area of the message processor. We also plan to investigate advantages of implementing the encryption processor on an overlay architecture called Virtual Coarse-Grained Array (VCGRA) [24].

## Acknowledgements

This work was supported in part by NSF grant no. CDI-1124931 and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

## References

[1] D. J. Bernstein, "Caesar: Competition for authenticated encryption: Security, applicability, and robustness," CAESAR web page, 2013, http://competitions.cr.yp.to/index.html.

[2] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "Sha-3 proposal blake," Submission to NIST (Round 3), 2010. [Online]. Available: http://131002.net/blake/blake.pdf

[3] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "Compact implementations of blake-32 and blake-64 on fpga," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 170–177.

[4] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The skein hash function family," Submission to NIST (Round 3), 2010. [Online]. Available: http://www.skein-hash.info/sites/default/files/skein1.3.pdf

[5] N. I. of Standards and Technology, "Secure hash standard (shs), fips pub 180-4," Federal Information Processing Standards Publication, March 2012, 2012. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[6] R. Rivest, "The md5 message-digest algorithm," RFC Editor, United States, 1992. [Online]. Available: https://www.ietf.org/rfc/rfc1321.txt

[7] D. J. Bernstein, "New stream cipher designs," M. Robshaw and O. Billet, Eds. Springer-Verlag, 2008, ch. The Salsa20 Family of Stream Ciphers, pp. 84–97. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68351-3_8

[8] ——, "Chacha, a variant of salsa20," 2008. [Online]. Available: cr.yp.to/papers.html#chacha

[9] D. Wheeler and R. Needham, "Tea, a tiny encryption algorithm," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., 1995, vol. 1008, pp. 363–366. [Online]. Available: http://dx.doi.org/10.1007/3-540-60590-8_29

[10] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee, "Hight: A new block cipher suitable for low-resource device," in *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'06. Springer-Verlag, 2006, pp. 46–59. [Online]. Available: http://dx.doi.org/10.1007/11894063_4

[11] M. El-Hadedy, K. Skadron, H. Mihajloska, and D. Gligoroski, "Programmable processing element for crypto-systems on FPGAs," in *Proc. HEART*, June 2015.

[12] K. Shahzad, A. Khalid, Z. Rakossy, G. Paul, and A. Chattopadhyay, "Coarx: A coprocessor for arx-based cryptographic algorithms," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, May 2013, pp. 1–10.

[13] M. El-Hadedy, D. Gligoroski, and S. Knapskog, "Area efficient processing element architecture for compact hash functions systems on virtex5 fpga platform," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, June 2011, pp. 240–247.

[14] A. Kulkarni, K. Heyse, T. Davidson, and D. Stroobandt, "Performance Evaluation of Dynamic Circuit Specialization on Xilinx FPGAs," in *Proceedings of the 11th FPGAworld Conference*, ser. FPGAworld '14, 2014.

[15] D. Gligoroski, H. Mihajloska, S. Samardjiska, H. Jacobsen, M. El-Hadedy, and R. E. Jensen, "$\pi$-cipher v2," Cryptographic competitions: CAESAR, 2014, http://competitions.cr.yp.to/round1/picipherv2.pdf.

[16] D. Gligoroski, H. Mihajloska, S. Samardjiska, H. Jacobsen, R. E. Jensen, and M. El-Hadedy, "$\pi$-cipher: Authenticated encryption for big data," in *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, vol. 8788. Springer, 2014, pp. 110–128. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11599-3_7

[17] K. Bruneel, W. Heirman, and D. Stroobandt, "Dynamic data folding with parameterizable configurations," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 4, 2011.

[18] E. Vansteenkiste, K. Bruneel, and D. Stroobandt, "A Connection Router for the Dynamic Reconfiguration of Fpgas," in *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ser. ARC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 357–364.

[19] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 964–969.

[20] K. Bruneel, K. Heyse, A. Kulkarni, and D. Stroobandt, "TLUT tool flow based Dynamic Circuit Specialization," 2012. [Online]. Available: https://github.com/UGent-HES/tlut_flow

[21] A. Kulkarni, V. Kizheppatt, and D. Stroobandt, "MiCAP: A custom Reconfiguration Controller for Dynamic Circuit Specialization," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, Dec 2015, pp. 1–6.

[22] A. Kulkarni, T. Davidson, K. Heyse, and D. Stroobandt, "Improving reconfiguration speed for Dynamic Circuit Specialization using Placement Constraints," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–6.

[23] A. Kulkarni and D. Stroobandt, "How to efficiently reconfigure Tunable LookUp Tables for Dynamic Circuit Specialization," *INTERNATIONAL JOURNAL OF RECONFIGURABLE COMPUTING*, vol. 2016, pp. 1–11, 2016.

[24] A. Kulkarni, E. Vasteenkiste, D. Stroobandt, A. Brokalakis, and A. Nikitakis, "A fully parameterized Virtual Coarse-Grained Reconfigurable Array for High Performance Computing Applications," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016)*, May 2016.