Hardware Overhead Analysis of Programmability in ARX Crypto Processing

Mohamed El-Hadedy[*]
mea4c@virginia.edu

Kevin Skadron[†]
skadron@virginia.edu

ABSTRACT

This paper evaluates the area and performance overhead of a programmable cryptographic accelerator specialized to support ARX (Add, Rotate, and Xor) based encryption standards, which are common in symmetric cryptography. This overhead is measured by comparing to a variety of custom ARX implementations optimized specifically for $\pi - Cipher$. This is a new algorithm for authenticated encryption that offers advantages over AES-GCM and is a candidate in the CAESAR competition. The programmable processor is designed to accommodate different word sizes, different block sizes and different security levels. The custom variants require separate versions to support these diverse capabilities. We find that the overhead of the programmability is quite high. For example, we implemented the Programmable Processing Element PPE in 227 slices, achieving a throughput of about 1.2 Gbps/block, regardless of the word size. In comparison, our best custom 64-bit implementation so far requires 445 slices, achieving 3.09 Gbps. This means that two PPEs running in parallel can achieve 75% of the throughput of the custom 64-bit solution, while providing flexibility to support diverse cryptographic standards.

Keywords

FPGA, Encryption, Crypto-systems, CAESAR

1. INTRODUCTION

Cryptography is essential to the modern IT economy. 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 CAESAR selection. Yet algorithm-specific, fixed-logic processing elements cannot support the Babel of cryptographic standards already present today or diverse data-path widths, let alone the rapid innovation in cryptographic standards. Many of these standards are based on ARX (Add, Rotate, and Xor) operations, including such well-known standards as AES and SHA. A generic ARX crypto-processing architecture can therefore support a variety of important encryption standards.

We therefore designed a new, area-efficient programmable crypto-processor, the PPE, tailored for use with any encryption algorithms based on, with an associated, tailored VLIW architecture[2]. The PPE architecture is also designed to support different word sizes, block sizes, and security levels. In this paper, to evaluate the area and performance overhead of the resulting flexibility, we compare the PPE to three different custom implementations of the ARX engine that are optimized for one algorithm, π-Cipher. All different architectures are evaluated in an FPGA implementation using a Xilinx Virtex-7 FPGA. π-Cipher is one of the CAESAR candidates, whose core permutation function is based on ARX operations. This cipher is a parallelizable, sponge-based design. π-Cipher is designed to accommodate different word sizes, different block sizes and different security levels [3, 4].

2. π-CIPHER

π-Cipher is a nonce-based authenticated encryption cipher with associated data. The π permutation is based on ARX operations and uses a two pass counter based sponge component that is denoted as a *triplex component*. More details can be found in the official documentation [3]. The core part of every sponge construction is the permutation function, and the whole security of the primitive relies on it. π-Cipher has an ARX based permutation function which is denoted as the π function[3]. One round of the permutation function consists of eights blocks of the π function (ARX operations). Four of the π function blocks generate the intermediate coefficients from the input message blocks. The other four blocks generate the round results from the intermediate coefficients.

3. ARCHITECTURE OF THE PPE

The proposed PPE is based on a very long instruction word VLIW architecture as shown in Fig. 1. It consists

[*]Dr.Mohamed El-Hadedy is a research associate with the Department of Computer Science at the University of Virginia, 85 Engineer's Way, P.O.Box 400740, Charlottesville, Virginia

[†]Professor. Kevin Skadron is the chair of the Department of Computer Science at the University of Virginia, , 85 Engineer's Way, P.O.Box 400740, Charlottesville, Virginia, USA

HASP '15, June 13 2015, Portland, OR, USA
© 2015 ACM 978-1-4503-3483-9/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2768566.2768574.

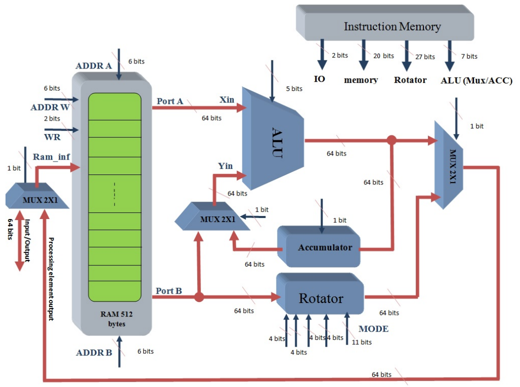

Figure 1: Top level architecture of the programmable processing element

of an Arithmetic and Logic Unit ALU, Rotator block, intermediate 512 byte coefficient memory, and Multiplexers to organize the data traffic from Input/Output port and the feedback from the computational part to the coefficient memory.

A VLIW architecture was chosen as an efficient way to issue multiple operations in each clock cycle. Changes of control-flow (e.g., branches) are not supported (or needed, for ciphers we have examined), one major source of efficiency. We currently create programs by hand; a compiler is future work, but the limited set of operations and the regular structure of the VLIW architecture should make this straightforward.

The PPE proposed here implements the ARX engine; it assumes the input text has already been padded if necessary.

3.1 ALU

Although ripple carry adders RCA are the most area-efficient adders, their worst-case propagation delay can be severe. Many fast adders have been proposed. We use a carry look-ahead adder (CLA), which solves the carry-delay problem by calculating the carry signals in advance, based on the input bits [5]. The Virtex-5 FPGA has LUTs with six inputs and one output; a single LUT can output one bit data from six-bit input data. If this resource is properly used for the circuit design, it is possible to obtain a higher performing circuit. We use these capabilities to build the CLA ALU based on the CLA. The ALU offers three different operations: word XOR, word addition, and subtraction (modulo 2^n). The ALU can process data either from both coefficient memory ports (Port A and Port B) or from xin port and the accumulator register as shown in Fig. 1. In each cycle, the ALU can process 4 data words of 16 bits, 2 data words of 32 bit, or 1 data word of 64 bit, based on the value of the control mode bits For instance, the mode control bits have to be 111 and the operation 01 if we would like to add two 64 bit words.

3.2 Rotator

The Rotator consists of four 16-bit sub rotations, controlled by the finite state machine (FSM) Each 16 bit rotator consists of a 64 x 4 array of 2x1 multiplexers, controlled by four, 4-bit RC control bits. Based on the value of the mode control bits, the rotator can rotate right four 16-bit words, two 32-bit words, or one 64-bit words.

3.3 Coefficient Memory

As shown in fig. 1, the coefficient memory has two output ports 64 bits (Port A and Port B). There are two address ports, ADDRA and ADDRB, to control the value of Port A and Port B. ADDRW is responsible for writing data into the memory through ram_inf. The memory size is 512 bytes.

3.4 Instruction Memory

The dual-ported instruction memory of 448 bytes holds the program that implements the chosen cipher algorithm, expressed as a sequence of VLIW instructions. The instruction memory has two address ports; the first 6-bit port acts as an instruction fetch address. The other 6-bit port specifies a write address, to change the values of the instruction memory. There are two enable bits: one permits read for instruction fetch and the other port is used to change the values of the instruction memory.

This dual-ported design allows a new cipher program to be streamed in *while* the current program progresses, or an early start on a new program while the rest of the program is still loading, or seamless processing of a program that cannot fit entirely into the on-chip memory. The instruction-fetch ADDR works as a pointer moving sequentially and wrapping around. While this processing is happening, the write ADDR is loading the new code. Each instruction consists of 2 bits to control the input/output interface, 20 bits for controlling the coefficient memory, 27 bits for controlling the different sections of the Rotator, and 7 bits for controlling the ALU.

4. ARX CUSTOM PROCESSOR

In this paper, we introduce three different custom hardware implementations for π-Cipher.

The ARX is presented in three different versions based on the data width size. For the 16 bit version, each memory block consists of three 4-byte random-access memories and one 4-byte read-only memory to hold the ARX coefficients. For the 32-bit version, each memory block contains three 8-byte RAMS and one 8 byte ROM to hold the ARX coefficients. For the 64 bit version, each memory block consists of three 16-byte RAMs and one 16-byte ROM to hold the ARX coefficients. Each memory controller is simply a finite state machine controlling two different counters. The first counter is used as a write address to get the input for both directions and located in the memory. Once the data are located, the read counter is used to move the data from the memory to the parallel adders to process the intermediate coefficients based on the adder-controller finite state machines.

4.1 Single Width

The basic ARX architecture consists of two memories, ten 16-bit adders, two Rotators, and two Xoring banks, distributed in two groups. One part is used to calculate the Y equations, and the other part is used to calculate the X equations. Each direction is controlled by a controller, to organize the data flow from the input ports to the output ports.

4.1.1 *ALU (Single-Width Core)*

Once the reading process starts, the adders between the memory and rotators calculate the first intermediate coefficients controlled by the ADDER controller. In the 16-bit version, the controller moves in one cycle per each equation; then rotator starts to calculate the other phase of the equation and hold it in the Xoring bank. The operation will continue until the all four equation have been calculated (see Sec. 2) . Then the Xoring bank starts to calculate final values of Y and X in one cycle. The Y and X directions are running in parallel. The output of X and Y are going to be summed by four 16-bit adders to produce the output. The final controller is responsible to receive a control signal from both Xoring blocks in each direction and generate the output flag, which is used to trigger the further blocks.

4.1.2 *Rotator (Single Width)*

The rotator is receiving data from the adders and sending data to the Xoring bank, based on the ROL controller. Simply, the rotator left-rotates the data coming from the adders (see Sec. 2).

4.1.3 *Xoring bank (Single Width)*

The Xoring bank recieves the data from the Rotator and buffers them one by one, until the four equations have finished their work in the adder section. Once the four values are ready in the Xoring bank, the Xoring operation starts, controlled by the Xoring controller, to produce the final result of each direction. Once the data has been processed by the Xoring Bank in each direction, the combination between both outputs produces the final output of the ARX engine.

4.1.4 *Final Controller (Single Width)*

The final controller, once it has received the xor_flag from the Xoring controller, will generate the output flag, which is used in further blocks in the π-Cipher.

4.2 Double-Width Core

Instead of using three adders and one rotator in each direction to compute the equations as in the single core, we use six adders and two rotators in each direction. This decreases execution cycles for a given amount of work. However, this increases memory output ports to 8 instead of 4 ports.

4.3 Quad-Width Core

Instead of using six adders and two rotators in each direction to compute the equations as in the double core, we use twelve adders and four rotators in each direction. This increases memory output ports to 16 instead of 8 ports.

5. HARDWARE IMPLEMENTATION

Both the custom ARX engines and the proposed PPE 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.

5.1 FPGA Implementation of PPE

The PPE has been implemented in just 227 slices, including the 512-byte coefficient memory, 448 byte instruction memory, and the computational unit (rotator and ALU), and can be clocked at 250 MHz. The slice logic distribution and utilization of the PPE on Virtex 7 show the area used for implementing the PPE is just 2% of the XC7VX485T-2FFG1761 device.

5.2 π-Cipher on PPE

$$Throughput = \frac{Number\ of\ input\ bits \times Maxfrequency}{Number\ of\ clock\ cycles\ per\ block} \tag{1}$$

As discussed in Section 2, the π-function consists of 4 rounds, and each round has eight ARX operation blocks. The throughput of the design is given in Equation 2. For 64-bit operation, implementing each operation block costs only 371 bytes in the instruction memory and 384 bytes in the coefficient memory, requiring 54 cycles to process 256 input bits (64 X 4) per block, achieving a throughput of 1.17 Gbps/block at 250 MHz for the 64-bit version. For the 16-bit version, each block costs 91 bytes in the instruction memory executed in 13 cycles to process 64 input bits, achieving a throughput of 1.2 Gpbs/block at 250 MHz. For example, implementing one full round of the π-function (64-bit version) with eight blocks used 512 bytes for storing the coefficients and intermediate values, and the instruction memory will be reprogrammed eight times. As mentioned before, the reconfiguration time is overlapped with the processing time, so for one round of π-function on PPE, the overall data throughput is 150 Mbps. The area, clock rate, and throughput of the custom ARX processing units is summarized in Tables 1 and 2.

6. PERFORMANCE AND OVERHEAD COMPARISON

In this section, we compare the performance of the different implementations of the different word-size versions of π-Cipher. The comparison is made between the PPE and three custom implementations, based on performance, area, throughput, and throughput/area. Table 1 shows the comparison between the PPE, single-, double-, and quad-width implementations of the ARX engine. Even though the custom hardware implementations are much more efficient than the PPE, the PPE still has two advantages. It is a programmable element which can be used to implement different algorithms based on the ARX paradigm; and it can support different word sizes. The word size (16, 32, or 64) is important, because the π-Cipher equations change as a function of the word size. Thus, even though 16-bit word sizes yield the most efficient custom-hardware implementation, in terms of both raw performance as well as area efficiency, supporting 64-bit operation on a 16-bit data-path requires additional effort. The PPE supports this natively. In 64-bit mode, the PPE acts as a single core, using the full data-path width. Similarly, in 16-bit mode, the PPE acts as a quad-width core. All three modes in the PPE are supported, with a total cost of 227 slices. All three modes operate at 250 MHz.

The PPE design we have so far is handicapped because it uses a native 64-bit ALU, which we suspect lowers the achievable frequency by increasing the critical path. Narrower word-sizes are handled by treating the 64-bit data-path as a SIMD unit, ala MMX/SSE. The 64-bit single-wide custom version uses 16-bit adders instead of increasing the arithmetic data-path width. Exploring the impact of ALU width on the PPE's performance is an urgent next step in this analysis.

The closest performance between the PPE and the custom implementations occurs for the 64-bit word size. This is also the fairest comparison for the PPE, which was designed to accommodate up to 64-bit operation. For the area required

Table 1: The ARX Performance (π16-Cipher)

	PPE	Single Width	Double Width	Quad Width
Throughput	1.2 Gpbs	3.57 Gbps	3.68 Gpbs	4.34 Gpbs
Area(Slices)	227	132	154	266
Frequency(MHz)	250	371	324	347
Throughput/Area (Mbps/slices)	5.4	27.69	24.47	16.71

Table 2: The ARX Performance (π64-Cipher)

	PPE	Single Width	Double Width	Quad Width
Throughput	1.17 Gpbs	3.09 Gbps	3.68 Gpbs	4.22 Gpbs
Area(Slices)	227	445	447	634
Frequency(MHz)	250	254	243	245
Throughput/Area (Mbps/slices)	5.28	7.13	8.4	6.8

in our 64-bit custom solution, the PPE can be duplicated, using two PPEs running in parallel to process the X and Y directions in ARX engine concurrently. A small amount of bridge logic will be needed between the intermediate memories to keep both computational core in the PPEs connected and combine the final results. We expect this overhead to be negligible; quantifying this overhead is in progress. So in almost exactly the same area as the single-width 64-bit custom core, we can achieve about 2.3 Gbps, or 75% of the 64-bit custom-core's throughput, but with greater flexibility.

7. CONCLUSIONS AND FUTURE WORK

In the paper, we presented different hardware implementations of the ARX engine. The first implementation was based on a VLIW programmable processing element, specialized for ARX ciphers. Then, using π-Cipher as a case study, we evaluated the overhead of this programmable ARX processor by comparing its area and performance (on a Xilinix Virtex-7 FPGA) to custom implementations of the ARX engine specifically optimized for π-Cipher. In the custom implementations, we introduced three different hardware implementations to explore trade-offs between area and speed. These implementations are the first hardware implementations of the the π-Cipher 16 and 64 bit versions.

The chief value of the PPE is its support for any ARX algorithm. The increasing diversity in encryption algorithms requires many computer systems to be able to accommodate this diversity. However, our results show the overhead of the PPE to be quite high, especially for narrow (16) bit-width word sizes, e.g. for embedded systems. However, our current version of the PPE was designed with a wide data-path to accommodate up to 64-bit word sizes without the need for multiple clock cycles per word. If only 16-bit support is needed, the PPE can be shrunk considerably, and the frequency most likely improved, while retaining the ability to support diverse ARX protocols. For 64-bit word sizes, two PPEs can be used together to achieve 75% of the throughput of a single-wide 64-bit custom implementation, using about the same area. While still a significant area and performance overhead, 25% will in many cases be an acceptable price to pay for the flexibility to support multiple encryption protocols.

Acknowledgements

This work was supported in part 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.

8. 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] M. El-Hadedy, K. Skadron, H. Mihajloska, and D. Gligoroski, "Area Programmable Processing Element for Crypto-Systems on FPGAs," in *Proceedings of the International Symposium on High-Efficient Accelerators and Reconfigurable Technologies, HEART2015*, June 2015.

[3] D. Gligoroski, H. Mihajloska, S. Samardjiska, H. Jacobsen, M. El-Hadedy, and R. E. Jensen, "π-cipher v1," Cryptographic competitions: CAESAR, 2014, http://competitions.cr.yp.to/caesar-submissions.html.

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

[5] S. Kao, R. Zlatanovici, and B. Nikolic, "A 240ps 64b carry-lookahead adder in 90nm cmos," in *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*, Feb 2006, pp. 1735–1744.