CoARX: A Coprocessor for ARX-based Cryptographic Algorithms

Khawar Shahzad[1], Ayesha Khalid[1], Zoltán Endre Rákossy[1], Goutam Paul[2]*, Anupam Chattopadhyay[1]
[1]MPSoC Architectures, RWTH Aachen University, Germany
khawar.shahzad@rwth-aachen.de
{khalid,rakossy,anupam}@umic.rwth-aachen.de
[2]Department of CSE, Javavpur University, Kolkata, India
goutam.paul@ieee.org

Abstract

Cryptographic coprocessors are inherent part of modern System-on-Chips. It serves dual purpose - efficient execution of cryptographic kernels and supporting protocols for preventing IP-piracy. Flexibility in such coprocessors is required to provide protection against emerging cryptanalytic schemes and to support different cryptographic functions like encryption and authentication. In this context, a novel crypto-coprocessor, named CoARX, supporting multiple cryptographic algorithms based on Addition (A), Rotation (R) and eXclusive-or (X) operations is proposed. CoARX supports diverse ARX-based cryptographic primitives. We show that compared to dedicated hardware implementations and general-purpose microprocessors, it offers excellent performance-flexibility trade-off including adaptability to resist generic cryptanalysis.

Categories and Subject Descriptors

B.7[**Integrated Circuits**]: [General,Types and Design Styles]; C.3[**Special-purpose and Application-based Systems**]: [Microprocessor/ microcomputer applications, Real-time and embedded systems]

General Terms

Design, Performance

Keywords

ARX, Cryptography, Coprocessor, CGRA

密码算法从操作形式的分类

1. Introduction

Security is an essential part of today's information systems. In order to guarantee the privacy of a user, cryptographic algorithms are intrinsic for a wide range of application domains, e.g., wireless transmission, multimedia and image processing. Additionally, cryptographic protocols are employed in modern SoCs to prevent IP piracy threats [28]. To ensure efficient execution of cryptographic algorithms ranging from private and public key encryptions to authentication schemes, cryptographic accelerators are increasingly being used. These accelerators are implemented as dedicated hardware with little flexibility [17] or as a flexible processor with algorithm-specific customizations [32, 12]. Flexibility in the accelerator is an important design dimension for several reasons. *Firstly*, it provides a common implementation supporting different cryptographic operations required by a standard. For example, the communication standards GSM, 3GPP, Bluetooth, IEEE 802.11 and ISO/IEC 29192 recommends usage of cryptographic algorithms belonging to classes of stream ciphers, block ciphers, message authentication codes (MAC) as well as public key cryptography. *Secondly*, it can continuously protect against evolving cryptanalytic methods by adopting appropriate design changes. For example, to increase the resistance against cryptanalysis, a proposed countermeasure against attack on Threefish-512 requires only some additional operations [21]. Moreover, throughput requirements often require designers to switch between different versions of the same algorithm.

Despite a number of a design proposals over flexible cryptographic accelerators, flexibility during architectural design is restricted to determination of common operators and it is rarely explored from the algorithm designers' perspective. The idea of combining computational workloads under a class that captures a pattern of computation and communication is presented at [4] and also promoted by Intel Recognition-Mining-Synthesis (RMS) view [14].

However, these constructions are not always sufficient to include the algorithm designers' knowledge. For example, in symmetric key cryptography, certain classes can be identified, e.g., ARX functions [35], Fiestel networks [24], Substitution-Permutation networks [19] and sponge constructions [9]. Such classes provide a template or a construction method, which can be employed to design a cryptographic algorithm. Generic architecture templates or flexible coprocessors for these classes have not been proposed earlier. This paper makes a first attempt into such design by proposing a coprocessor for ARX-based cryptographic algorithms.

The term AXR (later renamed to ARX) was coined by Ralf-Philipp Weinmann [35]. The algorithmic simplicity, efficient implementations in software, absence of timing attacks of ARX cipher contribute to their popularity [26]. Further, it is shown in [21, Section 5] that A, R and X operations are *functionally complete* in the sense that any function can be implemented with these three operations. A suitable sequence of ARX operations may lead to a secure cryptographic primitive. Therefore, design of a generic ARX architecture is of prime importance which, to the best of our knowledge, has not been attempted before the current work. Cryptanalysis of ARX based ciphers has been extensively reported in literature [21, 25, 23, 26] though, no major cryptanalytic breakthrough to threaten the security is till date known. As a result, diverse ARX-based algorithms made to the final rounds of recent design competitions for symmetric-key cryptosystems [3] and hash functions [2].

To begin with, we select five prominent ARX-based algorithms. Three of those, namely, HC-128 [36], Salsa20 [7] and ChaCha [8] belong to stream cipher category and two, namely, BLAKE [6] and Skein [15] are hash functions. A commercial high-level processor design environment [1] is chosen for our design flow. We performed a detailed design space exploration before making each design decision. The final coprocessor implementation, termed CoARX, is synthesized with a standard cell library for obtaining performance results.

*Corresponding Author. This work was done in part while he was visiting RWTH Aachen, Germany as an Alexander von Humboldt Fellow.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC '13, May 29 - June 07 2013, Austin, TX, USA.
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00.

The performance is benchmarked against published implementation results for each of the algorithms on dedicated hardware and flexible processors. The advantage of algorithmic view in the architecture is demonstrated with intuitive design alterations for improved resiliency. In summary, our contributions are as following.

- A detailed design space exploration for a novel programmable ARX coprocessor.
- Mapping of 5 common ARX algorithms on the coprocessor and enhancements of the design with algorithm-specific protections and optimizations.
- Detailed performance results and benchmarking against dedicated as well as programmable implementations.

2. Related Work

Acceleration efforts of various ARX cryptographic algorithms are reported on various platforms such as ASICs, FPGAs, GPUs and GPPs. Inclusion of custom instruction set extensions on 16-bit microcontrollers was also proposed for SHA-3 finalists (including Skein and BLAKE) [12] based on their implementation studies [20]. Section 5 compares the acceleration efforts of the algorithms under consideration when mapped on GPPs, customized microcontrollers and multicore systems (GPUs and IBM cell architecture). Due to the absence of a single VLSI implementation flexible enough to map any ARX algorithm, we consider implementation of individual algorithms. We refer below the reported ASIC implementations that standout in throughput. The reader is referred to Appendix A for a summary of FPGA implementations of the algorithms.

BLAKE: Out of the various dedicated hardware implementations of hash function BLAKE [6, 16, 22, 17], the most noteworthy in terms of throughput comes from the authors of the original algorithm [17]. A throughput of 20 Gbps for 8G-BLAKE-512 on 90nm CMOS technology while consuming 128 kGE area is reported.

Skein: Tillich *et al.* [33] reports area and throughput results of the implementation of Skein hash function using a 180nm standard cell library. Walker *et al.* used a 32nm standard cell technology and claimed to achieve a 5X throughput improvement over Tillich's work [34], after an appropriate scaling. More recent skein-512-256 implementation results claim a throughput of 3 Gbps with 66 KGates on 130nm CMOS technology [16] and 6.7 Gbps with 43 KGates on 90nm CMOS technology [22].

HC-128: The only ASIC based implementation for HC-128 proposed at [11] reports a throughput of 22.88 Gbps while consuming an area of 12.65 kGE with 21 KBytes of dual ported RAM [11] using 65nm standard cell library.

Salsa20 and ChaCha: Various implementations of the two stream ciphers are presented at [18], with a reported peak throughput of 6.5 Gbps using 40 KGates of area for Salsa20 and ChaCha.

From the algorithm point of view, ARX-based cryptography received close scrutiny of the cryptographers and cryptanalysts. General [25, 21] and individual cryptanalysis [5] for ARX-based algorithms is performed. Software toolkit for performing differential cryptanalysis on ARX-based constructions is proposed at [27].

3. Design Space Exploration

The microarchitecture of CoARX is decided by a simple observation that, all ARX-based algorithms perform a mix of addition (A), rotation (R) and xor (X) operations in different order in each round of the algorithm. Each of the stream ciphers HC-128, Salsa20 and ChaCha has an internal state that is randomized by a secret key and at every round of keystream generation the state is updated using ARX operations. The hash functions BLAKE and Skein compress the input message into fixed length output string and the operation again happens through several iterations each involving the basic ARX operations. For detailed description of the algorithms, one may refer to [36, 7, 8, 6, 15] (we give a summary of the algorithms in the Appendix B). One should note that the order of ARX operations, the number of rounds, the word-lengths of variables that are processed vary from one algorithm to another and hence poses a difficult challenge for generic architecture design.

To have the maximum flexibility and performance, the design need to support 3^3 different functional units for the triple operations. The same flexibility can be obtained from a reconfigurable architecture with these three operators. The algorithms also exhibit data-oriented computing pattern thereby, justifying the selection of Coarse-Grained Reconfigurable Architecture (CGRA) [13]. CGRAs can be implemented with wide range of design choices. In the following the rationale of specific CGRA design choices are elaborated, considering all the structural elements in bottom up fashion.

Functional Units Arrangement: Any number of functional units (A, R and X) may be arranged in any order inside a Processing Element (PE). We consider three architectures, showing an increasing trend of complexity and efficiency, allowing up to one, two or three operations per cycle as shown in Fig. 1(a), Fig. 1(b) and Fig. 1(c) respectively. They were modeled using a high-level synthesis environment [1] and synthesized with Synopsys Design Compiler using 90nm technology with maximum achievable clock frequency as reported in the Table 1.

PE_1 allows one operation per cycle and consequently, requires a single write port to the register file. The second architecture, PE_2 is designed to exploit the fact that a 64-bit A and R blocks in hardware contribute almost equally to the critical path while the contribution of X is significantly less. Hence the decrease in the clock frequency, compared to PE_1 is not significant. All the three units may be used separately, like PE_1, or in a non-overlapping simultaneous combination, e.g., AX and R or A and RX. PE_3 is more flexible, allowing all possible non overlapping combinations of one, two and three operations in a single cycle, e.g., AR and X or AXR or XR and A. The flexibility is achieved at the expense complex interconnects and a layer of multiplexers before each FU. Consequently, PE_3 has the least operating frequency compared to other architectures.

The round operations of each of the algorithms are mapped to the three architectures considered and cycle count and functional units used per cycle are calculated. For facilitating the critical choice amongst these architectures the following metrics are evaluated.

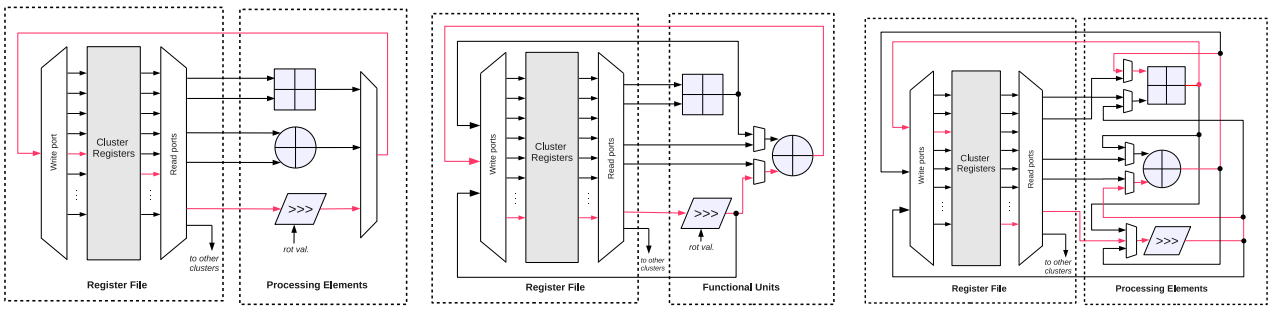
- Computational time is calculated from the clock frequency and the number of cycles for each algorithm.
- Resource Utilization (RU) gives the average utilization of functional units in the cluster grid. It is computed as,

$$RU_{algo} = \sum_{i=0}^{N} \sum_{j=0}^{C} \sum_{k=0}^{M} \frac{U_{ijk}}{(N * C * M)} \quad (1)$$

  Where C is the number of clusters, M is the number of FUs inside a cluster (3 in PE_2) and N is the total number of cycles. $U_{ijk} = 1$ if the k^{th} FU in j^{th} cluster is participating in the i^{th} cycle of the algorithm, otherwise it is 0.

The RU of PE_1 is worst since it can not exceed 33.3% and only one operation can be performed in a single cycle. Consequently the mapping of ARX algorithms requires a large number of instructions. On the other extreme, PE_3 shows the best RU figures. However, its multiplexers and read/write ports affect the critical path, resulting in relatively worse execution time compared to PE_2 in few cases as given in Table 1. Furthermore, the large configurability options of PE_3 requires a large configuration memory word. PE_2 is a compromise of complexity and flexibility. Due to its reasonable RU and computational time comparable to that of PE_3, we pick PE_2 to be our design choice. Note that, the choice of number of registers is also influenced by the choice of FU arrangement, i.e., more intermediate storage is needed for PE_1 than for PE_3 or for PE_2.

Register File and Read/Write ports: The FUs as well as the local registers of clusters are 64-bit wide to support double-word ARX algorithms (Skein and BLAKE-64), however, mapping of 32-bit algorithms (HC-128, Salsa20, ChaCha) remains trivial. The number of



(a) PE_1 (b) PE_2 (c) PE_3

Figure 1. Processing Elements with Different Functional Unit Arrangement

Table 1. Estimated Performance and Resource Utilization Results

	Freq. (MHz)	Possible Operations	BLAKE-512 (128 Bytes)		Skein-512-512 (64 Bytes)		ChaCha (64 Bytes)		Salsa20 (64 Bytes)		HC-128 (264 Bytes)		Average	
			time (ns)	RU %	time (ns)	RU %	time (ns)	RU %	time (ns)	RU %	time (ns)	RU %	time (ns)	RU %
PE_1	1075	A,R,X	431.53	33.33	245.52	33.33	234.36	33.33	234.36	33.33	265.98	15.38	282.35	29.74
PE_2	913	AX,RX	323.96	53.32	131.33	86.07	188.25	50	188.25	50	216.70	27.77	209.70	53.43
PE_3	610	AR,RA,XA,XR ARX,AXRR,XA RAX,XAR,XRA	301.86	77.77	196.87	86.07	150.93	100	150.93	100	324.83	27.77	225.09	78.31

local registers is dictated by BLAKE, which requires 6 registers for the computation of its core function.

Number of Clusters: The choice of number of clusters inside a CGRA influences the throughput and resource utilization, resulting in resource under-utilization when number of clusters is too large and low parallelism and consequently low throughput when number of clusters is too small. Among the target ARX algorithms, BLAKE, Salsa20 and ChaCha consist of up to 4 parallel executions of their core operation. Flavors of Skein can have 2-8 parallel ARX operations, whereas for HC-128, 3 parallel ARX operations can take place. Hence a 4 cluster grid (2x2) is chosen as a trade-off between throughput and resource utilization.

Memory Hierarchy: Inside each cluster, a 2-level memory hierarchy is maintained by a small register file and a large SRAM. The memory holds the plaintext, ciphertext, message hash, Initialization Vector and Key. This dual-ported SRAM has 64 bit word length and has 1024 elements. Its size is dictated by HC-128, that requires two secret S-Boxes, each with 512 elements. Out of the 6 registers in register file, two namely *m0, m1* hold the value to be written or read from port 0 and port 1 of the memory, respectively. Movement of memory values to any other register incurs one cycle latency.

Inter Cluster Interconnects: The number and nature of interconnects between clusters is influenced by the permutation of ARX operations in the subsequent rounds of ARX algorithms. For BLAKE, Salsa and ChaCha each cluster calculates one core function (referred as G function in the algorithms). At the completion of G functions on the columns, these algorithms proceed diagonally for which each cluster requires exactly one value from every other cluster. Consequently, a MESH style interconnect is chosen with 3 incoming and 3 outgoing register values from each cluster. For algorithms requiring more permutation and data exchange between clusters, data availability is ensured by one cycle overhead.

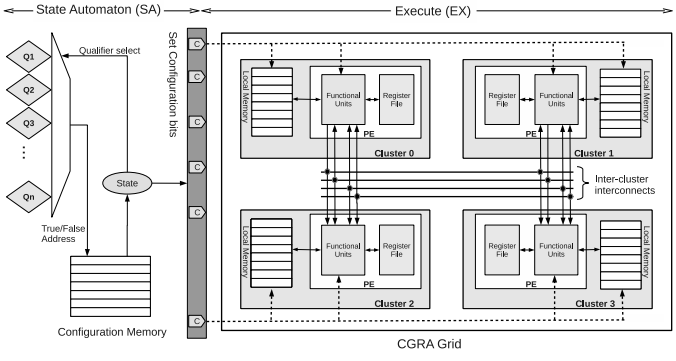


Figure 2. Block Diagram of CoARX

The top-level architecture of the final CGRA-based ARX coprocessor is presented in Fig. 2. The design is distributed over two pipeline stages. *State Automaton* stage fetches the instruction from the *configuration memory*. The instruction is organized as a 4-tuple. A *Qualifier*, a *True* address and a *False* address are the three fields, which handle looping and conditional jumps during the execution. The rest of the instruction are the configuration bits for the CGRA that control the functionality of the clusters in the *Execute* stage.

4. Mapping of the ARX Algorithms

On the proposed CGRA coprocessor, any ARX-based cryptographic algorithm can be mapped. The reconfigurablility is controlled by the instructions in the configuration memory. The configuration of the entire CGRA has 269-bit word length and constitutes 46 words. Configuration words are manually written and are updated to perform a different combination of ARX operation every cycle. For switching the current algorithm running on the CGRA, the configuration memory is updated. Fig. 3 shows a breakdown of a configuration word and a discussion of various fields follows.

Qualifier Control: The iterative nature of the cipher algorithms is handled by a 4-bit *Qualifier*. The selection of next address to be *True* address or *False* address is determined by the qualifier condition. For conditional jumps (e.g., evaluating number of rounds, message/ block size), if the condition is evaluated true then a jump is made to the 8-bit true address or else to the false address. An unconditional jump is evaluated always as the qualifier being true.

Cluster Configuration: The bit fields in the instruction word of configuration memory are used to configure the input multiplexers and the processing elements of the entire CGRA. Each cluster requires 61 bits of configuration word. The dual ported local memory in each cluster requires a 2-bit *cmd* and a 10 bit *address* for 1024 word memory to be written and read from. Register *m0* and *m1* are tied to communicate with port0 and port 1 of the memory, respectively. *FU config.* includes the specification of source and destination registers of the functional units namely A, R and X.

- *misc.* bits: specify if the operation is 32-bit or 64-bit wide. Future extension may include specifying Rotator FU being used as a shifter or not.
- *src. operands*: The multiplexers for each of the two operands of A and X are 8×1, R requires only one operand and a 6-bit rotation value. The rotation value and direction of the 64 bit

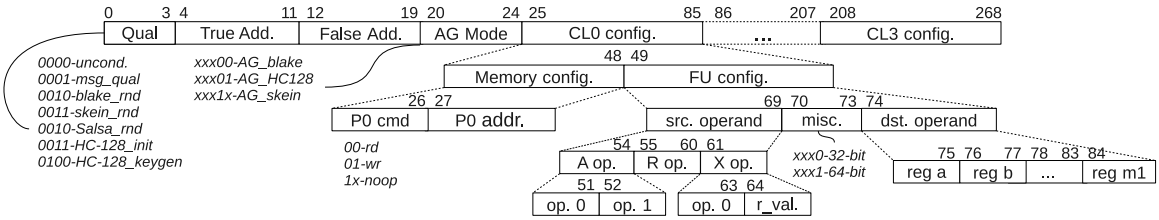


Figure 3. Configuration Word Details

operand is specified by a 6-bit *r_val* field. The operands may include registers from other clusters as well.

- *dst. operand*: For each of the 6 registers in the register file, selection lines for write demultiplexers requires 2-bits. The registers may be updated by the Functional Units output or simply bypassed values.

4.1 Algorithm Specific Modes

Some ARX algorithms require specific modes to be indicated in configuration word to carry out non-ARX operations.

BLAKE Address Generation: For BLAKE, the addresses for loading msg and $constants$ from memory, are not static and they change in each round. One solution for implementation of this scenario is direct loading of data memory in a cluster by msg and $constants$ during initialization phase. But this approach renders the system vulnerable to side channel attacks. Instead a specific address generation mode is defined to carry out BLAKE address generation.

HC-128 Address Generation: Some operations in HC-128 keystream generation are not strictly ARX. They are modulo-512 subtraction, byte processing in h_1 and h_2 functions and counter increment/addition and shift operations in the expansion step of initialization. To cater them, specific hardware blocks are being used.

Skein Key Generation: Skein requires injecting a subkey into the states by adding it with the outputs of MIX functions every fourth round. Skein key generation mode enables hardware block *skein subkey generator* that is implemented in hardware as shift registers, along with adders and is capable of generating one subkey in a single cycle.

Table 2. CoARX Synthesis Results

Area (kGE)			Memory (kBytes)		Freq. (MHz)
Combinational	Sequential	Total	Configuration	Data	(Core)
82.6	12.4	95	1.5	32	700

5. Implementation and Benchmarking

The proposed architecture CGRA is developed using Language for Instruction Set Architectures (LISA) description [1]. LISA can be used to generate RTL description of an architecture along with software tools including Compiler, Assembler, Linker, Profiler and Debugger. At present, the CGRA is programmed with hand written assembly language but the configuration word generation process can be easily automated due to the regular arrangement of FUs in a PE. LISA processor description (5K lines of code) was processed by Synopsys Processor Designer to generate Verilog RTL (38K lines of code). We used Synopsys Design Compiler, topographical mode, Faraday 90nm CMOS technology library for the synthesis of HDL and results are presented in Table 2. The area estimates of sequential and combinational logic is given in equivalent NAND gates. Each of the four clusters have 8 KBytes of dual ported SRAM as local memory. Performance of different ARX algorithms on CoARX is presented in Table 3.

5.1 Comparison with ASICs

Different ASIC implementations considered for comparison are tabulated with their reported throughput in Table 4. For BLAKE, Henzen *et al.* report the ASIC with highest throughput [17] so far. For Skein, results reported for Skein-512-256 with highest throughput are considered [22]. Implementation results for HC-128, taking keystream generation of 0.75 cycles/Byte (same as our implementation) is considered for comparison [11]. For Salsa and ChaCha comparison, we consider the implementation from [18].

For a fair comparison, the estimated throughput of the ASICs of ARX algorithms under consideration has been added in Table 4 after direct technology scaling to 90nm. CoARX delivers throughput in the same order of magnitude. On a closer investigation, some architectural aspects of CoARX, which affects the critical path of the design are as following.

- **Barrel Shifter**: For CoARX, a 64-bit Barrel rotator is used to support flexible rotation amounts. Considered ASICs for comparison (Table 4) use wire routing for fixed rotations instead.
- **Datapath Width**: For catering algorithms with 64-bit datapath and scalability, CoARX datapath is 64-bit wide. Consequently, ASIC implementations of ARX-based algorithms with 32-bit datapath have a smaller critical path and outperform CoARX in throughput. This can be made configurable to reduce the throughput gap.
- **Flexible Interconnects**: To support various possibilities of source operands from local and neighboring clusters registers, the input multiplexers of FUs in each of cluster are 8x1. Similarly, the write ports of the registers may take outputs of various FUs and each has a 4x1 multiplexer for possible inputs. These multiplexers increase the critical path of the design.
- **Unfolding Transformation**: According to the unfolding transformation [20], a design achieves better throughput if it is unfolded, which is the case in most of the ASIC implementations. ASICs for BLAKE implement an entire G function in one cycle [17]. For Skein, eight rounds are unfolded and implemented in one cycle [33]. Salsa20 and ChaCha also use similar kinds of transformations [18]. To incorporate enough flexibility, CoARX utilizes modular functional units and hence does not take advantage of unfolding transformation.

The area of the individual ASIC implementations cannot be compared vis-a-vis the area of CoARX due to its added flexibility. Sum of area of the best-performing ASICs results in 181 kGE, the area of CoARX is 47% less in comparison. A more suitable metric for comparison is the area-efficiency (throughput per area) reported in Table 4. The individual ASICs have significantly higher throughput per area, when compared against CoARX. This is the expected *flexibility gap*. The fact that CoARX implementation is efficient can be shown by considering the same metric for a hypothetical ASIC combining the best implementations. For this measure, the collective area (181 kGE) of individual ASICs is chosen. CoARX performs comparably for all the designs, except BLAKE. The BLAKE implementation reported in [17] performed an efficient round rescheduling technique to reduce the critical path after unrolling multiple subsequent ARX operations. For CoARX, such optimization is not possible since, the critical path is constrained by the operations within a PE.

Fig. 4 compares CoARX throughput of ARX algorithms mapped on it, with Intel M 1600 MHz processor and ARM926EJ-S 1200 MHz processor and easily outperforms these platforms by a factor of 1.6 to 22 times.

In the absence of a CoARX like flexible crypto core, a fair comparison in terms of power is not possible. Power-figures (in mW), in similar process-technology, are available only for some individual ASICs, e.g. BLAKE: 10.84 [22], 15.65 [38], Skein: 17.17 [22], 39.71 [38] and Salsa20: 8.42 [37]. These values are better than (though the in same order of magnitude as) CoARX (Table 4). That is because these ASICs have algorithm-specific optimizations in their design and operate at a clock frequency lower than that of CoARX. Redundancy to

Table 3. Performance of different ARX algorithms on CoARX

	BLAKE-512	Skein-512	HC-128	Salsa20/20	ChaCha/20
Input Block size (Bytes)	128	64	-	-	-
Output Block size (Bytes)	64	64	-	64	64
No of rounds (r)	16	72	-	20	20
Initialization Round Calculation Finalization	5 cycles $21\times r$ cycles 9 cycles	7 cycles r+((r/4)+1)×2 cycles 6 cycles	46241 setups/sec 0.75 cycles/Byte -	4 cycles $17\times(r/2)$ cycles 12 cycles	4 cycles $17\times(r/2)$ cycles 13 cycles
Total (cycles)	350	123	-	186	187
Throughput (Gbps)	2.05	2.91	7.47	1.93	1.92
Power (mW)	83	89	53	71	61
Energy (mJ/Gbit)	41	30	7	33	29

Table 4. Comparison with ASICs (Throughput scaled to 90nm)

Implementation Reference	Area (kGE)	Frequency (MHz)	Throughput (Gbps)		Throughput/Area (kbps/GE)		
			ASIC	CoARX	Individual	Combined	CoARX
4G-BLAKE-512(16 rounds) [17]	79	532	16.5	2.05	208.86	91.16	21.57
Skein-512 [22]	43.13	251	6.73	2.91	156.13	37.18	30.63
HC-128 Parallel keystream [11]	13.66	1670	12.86	7.47	941.43	71.04	78.63
Salsa20 4xS-QR [18]	22.81	365	4.67	1.93	204.64	25.79	20.31
ChaCha 4xS-QR [18]	22.44	366	4.67	1.92	208.19	25.81	20.21

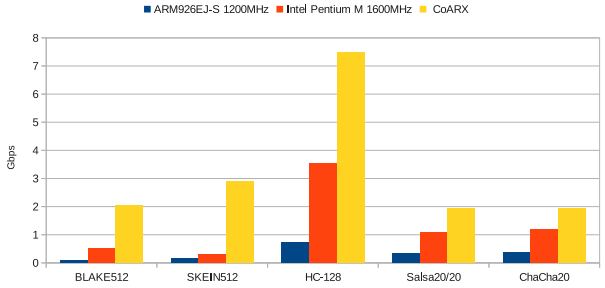


Figure 4. Comparison with GPP and Embedded Processor

achieve flexibility in CoARX design also contributes to higher power figures.

5.2 Comparison with Customized Microcontroller

Constantin *et al.* proposed custom instruction extensions for efficient implementation of all SHA-3 hash function competition finalists including BLAKE and Skein [12]. A 16-bit PIC24 microcontroller is used as a starting point. For BLAKE, the throughput improvement is mainly attributed to custom instructions for rotation and address generation. For Skein, a single custom instruction is added for performing 64-bit left rotate in two cycles. In order to make sure that the device remains usable in resource constrained environments, 2×16-bit barrel shifters have been used to perform 64-bit rotations. The reference microcontroller core occupies an area of 23 kGE. The customizations led to an area overhead of 10%. BLAKE and Skein can be mapped on the above mentioned extended architecture with 155 Cycles/Byte (10.32Mbps throughput) and 158 Cycles/Byte (10.12Mbps throughput) respectively. In terms of throughput and area-efficiency (throughput/area), CoARX easily outperforms this. Furthermore, CoARX offers more flexibility (e.g. different rotation amounts) by focusing on ARX class of algorithms.

5.3 Comparison with Multicore Implementations

Several ARX based cryptographic functions have been undertaken for throughput benchmarking on different multicore architectures, e.g., IBM Cell architecture and GPGPUs [30, 31]. A fair comparison between CoARX and a multicore system is hard, since the throughput scaling of multicore architectures is not linear due to the distribution of storage, varying latency for different storage and limitation of active number of threads per core. Considering these restrictions, multiple parallel data-streams for encryption/authentication are invoked in [30] in order to maximize the usage of the available cores. For Cell architecture, the 4-way SIMD instructions for each Synergistic Processing Elements (SPEs) are exploited for parallelism. With increasing size of the data block, more number of parallel threads are deployed resulting in higher encryption/authentication speed. To provide a fair comparison, the highest achievable throughput (denoted *TP*), the number of active Scalar Processor (SP) cores for GPU and active streams for an SPE are reported in the following Table 5.

Table 5. Performance on Multicore Architectures

	SPE [30]		GPGPU [30, 31]		CoARX
	TP (Gbps)	Streams	*TP* (Gbps)	Active SP	*TP* (Gbps)
BLAKE-32	5.1	4	36.80	240	2.05
Skein-512	1.9	2	22.10	240	2.91
HC-128	-	-	2.26	240	7.47
ChaCha	-	-	42.40	240	1.93
Salsa20	-	-	10.86	48	1.93

Though it is hard to compare multicore architectures' performance with that of CoARX without offering similar scalability, several points can be made. First, CoARX offers 4 parallel ARX operations per cycle matching a 4-way SIMD instruction of SPE and 4 active SPs in GPGPU. Second, the individual SP cores of [30, 31] are synthesized at a frequency of 1242 MHz and 1350 MHz respectively, which are much higher than that of CoARX. Finally, for both BLAKE-32 and Skein-512, the implementation of [30] considers 4 different messages for parallel hashing, fixing the message size suiting the GPGPU requirements. In practice, the message size can vary. We experimented with a single large message for CoARX, allowing any message size as input. Since both BLAKE-32 and Skein-512 involve chaining with the hash value of last message chunk, the internal dependency between the concurrent threads are ignored in [30], resulting in more throughput. Considering the above facts, CoARX outperforms both GPGPU and SPE on a throughput per core basis.

5.4 Flexibility Study: Threefish-512 and Skein-512 Design Variants

Cryptanalysis of existing algorithms and design of new algorithms that resist the known attacks are always at arms race. CoARX is flexible enough to adapt variants and offer resistance against generic attacks. We demonstrate this through a case study with Threefish-512 and Skein-512. Threefish-512, an ARX-based block cipher, has been mapped on CoARX as the building block of Skein-512. A general attack to ARX-based cryptosystems and in particular to Threefish-512 has been reported in [21]. This attack makes a reduced round variant of Skein-512 to be vulnerable. We show that with CoARX, addition of few instructions is sufficient to thwart such attacks. At first, the attack model presented in [21] is studied.

According to Lemma 1 and the attack model in Section 4.2 of [21], the probability of getting a rotational pair for each addition is $2^{-1.415}$ for each addition in MIX and $2^{-0.28}$ for each addition in *subkey* addition part. Since there are a total of $72 \times 4 = 288$ additions in MIX and $18 \times 8 = 144$ additions in *subkey* addition, the probability of the complete attack is given by $2^{(-1.415)\times 288+(-0.28)\times 144} = 2^{-448}$, giving a complexity of 2^{448}. The 512-bit key has 8 words of 64-bits and the three leftmost bits in each key-word are assumed to be known in the attack model, the effective key-length is $8 \times 61 = 488$ bits. Thus, according to the attack of [21], instead of 2^{488} attack complexity of

random guessing, one has 2^{448} attack complexity. To counteract the above attack, we need to increase the number of additions. Suppose, we increase the total number of additions in the MIX by a_1 and the total number of additions in the *subkey* addition by a_2. Then we must have $1.415 \times (288 + a_1) + 0.28 \times (144 + a_2) \geq 488$. Simplifying the above inequality, we obtain

$$1.415 \times a_1 + 0.28 \times a_2 \geq 40. \quad (2)$$

Now, we propose two design variants to achieve the inequality (2). In the first variant, the total number of rounds is increased by a multiple of 4, say, by $4x$. Since we get 8 extra additions from *subkey* addition after every 4 rounds, we have $a_1 = 16x$ and $a_2 = 8x$. Substituting in 2 and simplifying, we get $x >= 1.61$. Since x has to be an integer, we take $x = 2$, meaning that 8 extra rounds are needed. We propose to continue the same *subkey* generation algorithm for those 8 extra rounds. In the second variant, only the number of additions in MIX is increased without increasing the *subkey* additions. This means, $a_2 = 0$. Thus, from inequality (2), we get $a_1 \geq 29$ (after rounding). For symmetry of computation, $a_1 = 36$ is taken and those 36 additions are distributed as follows. Instead of 4 rounds followed by a *subkey* addition, we propose to use 5 and 4 rounds alternately followed by a *subkey* addition. In this way, there will be a total of $5 \times 9 + 4 \times 9 = 81$ rounds interleaved by the usual *subkey* additions. The above two design variants of Skein-512 (Threefish-512) are mapped on the CoARX without much degradation in throughput. The first design variant requires 12 extra cycles ($8 + 2 \times 2$) and the second design variant requires 9 extra cycles causing a throughput degradation of 8.8% and 6.8% respectively. For designing the second design variant additional permutation and rotation constants are also required to perform the 5^{th} alternate round, which are added to the configuration memory. The entire design modification is performed within few hours.

Very recently, a differential power attack has been reported [10] on Skein. The same paper also proposes a countermeasure which does not alter the basic ARX structure and therefore our architecture can be easily adapted to include this as well.

6. Conclusion and Outlook

In the context of flexible and efficient cryptographic accelerators, this paper studies the design of CoARX, a flexible coprocessor for ARX-based algorithms. CoARX shows comparable area-efficiency against dedicated hardware accelerators and significantly higher throughput compared to off-the-shelf processor-based implementations. The algorithmic perspective of flexibility provides unique advantage against general and specific cryptanalysis.

The detailed physical design and studies for general prevention of side channel attacks are on our future roadmap. The idea of class-specific cryptographic accelerator will be further probed for other design primitives like Fiestel networks and sponge-based constructions.

Acknowledgment

We would like to sincerely thank Rishiraj Bhattacharyya for drawing our attention to the class of ARX-based cryptosystems as a common design kernel.

7. References

[1] LISA 2.0. Available at www.synopsys.com/Systems/BlockDesign/ProcessorDev.

[2] SHA-3 Cryptographic Hash Algorithm Competition. Available at http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[3] eSTREAM: the ECRYPT Stream Cipher Project. Available at http://www.ecrypt.eu.org/stream/index.html.

[4] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical report, UCB/EECS-2006-183, University of California, Berkeley, 2006.

[5] J. Aumasson et al. Tuple cryptanalysis of ARX with application to BLAKE and Skein. ECRYPT 2 Hash Workshop, 2011.

[6] J. Aumasson, L. Henzen, W. Meier and R. Phan. SHA-3 proposal BLAKE ver 1.3, 2010. Available at https://www.131002.net/blake.

[7] D. J. Bernstein. The salsa20 family of stream ciphers. Available at http://cr.yp.to/papers.html#salsafamily, December 2007.

[8] D. J. Bernstein. ChaCha, a variant of Salsa20. Available at http://cr.yp.to/papers.html#chacha, January 2008.

[9] G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. Sponge functions. In Ecrypt Hash Workshop 2007. Available at http://csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html.

[10] C. Boura, S. Leveque and D. Vigilant. Side-Channel Analysis of Grostl and Skein. In IEEE Symposium on Security and Privacy Workshops 2012, pages 16–26.

[11] A. Chattopadhyay, A. Khalid, S. Maitra and S. Raizada. Designing high-throughput hardware accelerator for stream cipher HC-128. In IEEE International Symposium on Circuits and Systems 2012, pages 1448–1451.

[12] J. Constantin, A. Burg and F. Gürkaynak. Investigating the potential of custom instruction set extensions for SHA-3 candidates on a 16-bit microcontroller architecture. Cryptology ePrint Archive, Report 2012/050, 2012. Available at http://eprint.iacr.org/2012/050.

[13] A. DeHon. The density advantage of configurable computing. In *Computer*, vol. 33 (4), pages 41–49, 2000.

[14] P. Dubey and S. Engineer. Teraflops for the masses: Killer apps of tomorrow. In Workshop on Edge Computing Using New Commodity Architectures, 2006. Available at http://gamma.cs.unc.edu/EDGE/SLIDES/dubey.pdf.

[15] N. Ferguson et al. The Skein Hash Function Family, Version 1.3. http://www.skein-hash.info/sites/default/files/skein1.3.pdf, October 2010.

[16] X. Guo et al. ASIC implementations of five SHA-3 finalists. In IEEE DATE 2012, pages 1006–1011.

[17] L. Henzen, J.-P. Aumasson, W. Meier and R.-W. Phan. VLSI Characterization of the Cryptographic Hash Function BLAKE. In *IEEE Transactions on VLSI Systems*, vol. 19 (10), pages 1746–1754, 2011.

[18] L. Henzen, F. Carbognani, N. Felber and W. Fichtner. VLSI hardware evaluation of the stream ciphers Salsa20 and ChaCha, and the compression function Rumba. In 2nd International Conference on Signals, Circuits and Systems 2008, pages 1–5.

[19] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2007.

[20] J. K. Kobayashi, Ikegami, S. Matsuo, K. Sakiyama and K. Ohta. Evaluation of Hardware Performance for the SHA-3 Candidates using SASEBO-GII. Available at http://eprint.iacr.org/2010/010.

[21] D. Khovratovich and I. Nikolić. Rotational cryptanalysis of ARX. In Fast Software Encryption 2010, LNCS vol. 6147, Springer, pages 333–346.

[22] M. Knezevic et al. Fair and consistent hardware evaluation of fourteen round two SHA-3 candidates. In *IEEE Transactions on VLSI Systems*, vol. 20 (5), pages 827–840, 2012.

[23] G. Leurent. ARXtools: A toolkit for ARX analysis. In: The Third SHA-3 Candidate Conference. Available at http://www.di.ens.fr/~leurent/arxtools.html.

[24] M. Luby and C. Rackoff. How to Construct Pseudorandom Permutations and Pseudorandom Functions. In *SIAM Journal on Computing*, vol. 17 (2), pages 373–386, 1988.

[25] K. Mckay and P. Adviser-Vora. Analysis of ARX round functions in secure hash functions. PhD thesis, George Washington University, 2011.

[26] N. Mouha. ARX-based cryptography. Available at https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/nicky_mouha_arx-slides.pdf.

[27] N. Mouha, V. Velichkov, C. De Canniere and B. Preneel. Toolkit for the Differential Cryptanalysis of ARX-based Cryptographic Constructions. In Workshop on Tools for Cryptanalysis 2010, pages 125–126.

[28] J. Roy, F. Koushanfar and I. Markov. Epic: Ending piracy of integrated circuits. In IEEE DATE 2008, pages 1069–1074.

[29] A. Shimizu and S. Miyaguchi. Fast data encipherment algorithm FEAL. In EUROCRYPT 1987, LNCS vol. 304, Springer, pages 267–278.

[30] D. Stefan. Analysis and Implementation of eSTREAM and SHA-3 Cryptographic Algorithms. Master's thesis, Cooper Union College, 2011. Available at http://www.scs.stanford.edu/~deian/pubs//stefan:2011:analysis.pdf.

[31] S. Neves. Cryptography in GPUs. Master's thesis, University of Coimbra, 2009. Available at http://eden.dei.uc.pt/~sneves/pubs/2009-sn-msc.pdf.

[32] S. Tillich. Instruction Set Extensions for Support of Cryptography on Embedded Systems. PhD thesis, Graz University of Technology, Austria, 2008. Available at https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=39243.

[33] S. Tillich. Hardware Implementation of the SHA-3 candidate Skein. In Cryptology ePrint Archive Report 2009/159. Available at http://eprint.iacr.org/2009/159.

[34] J. Walker, F. Sheikh, S. Mathew and R. Krishnamurthy. A Skein-512 hardware implementation. 2010. Available at http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/WALKER_skein-intel-hwd.pdf.

[35] R.-P. Weinmann. AXR - Crypto Made from Modular Additions, XORs. In Dagstuhl Seminar 09031, January 2009. Available at http://www.dagstuhl.de/Materials/Files/09/09031/09031.WeinmannRalfPhilipp.Slides.pdf.

[36] H. Wu. The stream cipher HC-128. Available at http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf.

[37] Good, T and Benaissa, M Hardware results for selected stream cipher candidates. In State of the Art of Stream Ciphers, 2007, pages 191–204.

[38] M. Srivastav, X. Guo, S. Huang, D. Ganta, M. B. Henry, L. Nazhandali and P. Schaumont. Design and Benchmarking of an ASIC with Five SHA-3 Finalist

Table 6. FPGA based Implementations of ARX Algorithms

Reference	FPGA Device	Area (CLB / LE)	Block RAMs	Operating Frequency (MHz)	Throughput (Mbps)
Salsa20-sr [F1]	Xilinx Virtex-II 2V250fg256	194 CLB	4	250	38
Salsa20-qr [F2]	Altera Cyclone EP1C20F400C6	2356 LE	-	55	343
Salsa20-sr [F2]	Altera Cyclone EP1C20F400C7	3400 LE	-	40	931
Salsa20-dr [F2]	Altera Cyclone EP1C20F400C8	3510 LE	-	30	1280
Salsa20[F3]	Xilinx Spartan 3 xc3s50pq208-5	1615 CLB	-	23.5	213
ChaCha_config1 [F4]	Xilinx Virtex-6 XC6VLX75T-2	49 CLB	2	362	595, 422, 266
ChaCha_config2 [F4]	Xilinx Virtex-6 XC6VLX75T-2	77CLB	2	316	520, 368, 232
ChaCha_config2 [F4]	Xilinx Virtex-6 XC6VLX75T-2	77CLB	2	345	569, 403, 254
BLAKE-32 [F6]	Xilinx Virtex-3 xc3s50-5	124 CLB	2	190	115
BLAKE-32 [F5]	Xilinx Virtex-3 xc3s50-5	360 CLB	2	135	315
BLAKE-64 [F7]	Xilinx Virtex-6 xc6vlx75t-1	117 CLB	-	274	105
BLAKE-64 [F5]	Xilinx Virtex-6 xc6vlx75t-1	146 CLB	1	189	277
BLAKE-64 [F6]	Xilinx Virtex-5 xc5vlx50-2	108 CLB	3	358	314
Skein [F7]	Xilinx Virtex-6 xc6vlx75t-1	240 CLB	-	160	179
Skein [F5]	Xilinx Virtex-6 xc6vlx75t-1	162 CLB	1	166	34.9
Skein [F8]	Xilinx Virtex-5 xc5v	555 CLB	-	271	237
Skein [F9]	Xilinx Virtex-5 xc5v1x110-3	821 CLB	not specified	119	1610

Candidate. In ECRYPT 2 Hash Workshop, 2011.

APPENDIX

A. FPGA Implementations of ARX Algorithms

Most of the FPGA implementations target high throughput or low area as their design goals. The reuse of functional units in various ARX based cryptographic algorithms has not be exploited for a flexible implementation. The only work that is noteworthy in terms of flexibility for ARX based Skein and BLAKE is reported by Nuray *et al.* [F4]. Their reported FPGA implementations of these hash functions also support the primitive cipher functions on which they are based on; i.e., a unified core for BLAKE and ChaCha (a stream cipher) and one for Skein and Threefish (a tweakable block cipher). For a unified BLAKE and ChaCha coprocessor when mapped on XC6VLX75T-2 using 144 CLBs and 3 Block RAMs the throughput reported for BLAKE-32 and BLAKE-64 was 288 and 255 Mbps respectively [F4]. The same coprocessor could produce keystream for 8, 12 and 20 rounds of ChaCha at 1102, 780 and 492 Mbps respectively [F4]. Skein and Threefish coprocessor consumes relatively more slices on the same device and support Skein-512-512, Skein-256-256 along with various flavors of Threefish i.e., Threefish-256, Threefish-512 and Threefish-1024 [F4].

Due to the absence of any other flexible implementation, this Section summarizes the reported individual FPGA implementations of the 5 ARX based cryptographic algorithms undertaken in CoARX. A comparison of these implementations in terms of throughput and resource utilization of the target device is done in Table 6.

HC-128

HC-128 is an synchronous stream cipher in eSTREAM finalists. Since it belongs to software profile, no FPGA implementations are being reported for it so far.

Salsa20

For Salsa20, the earliest FPGA implementation results were reported by Junjie *et al.*[F1]. They proposed a compact hardware implementation of Salsa20 comprising of a single *QuarterRound* function block. The global clock was 250 MHz, and for the *QuarterRound* block it was 125 MHz. Gaj *et al.* also implemented salsa20 using a single quarter-round architecture in combinational logic and using eight clock cycles to implement the entire *DoubleRound* [3]. Evaluation of various possible architectures for sasla20 namely Salsa20-dr (unrolled *DoubleRound* iterative architecture), Salsa20-sr (single round iterative architecture), Salsa20-qr (*QuarterRound* resource shared iterative architecture) were undertaken for implementation on an Altera device [F2]. These three possibilities provide various design points in area-performance trade-off as specified in Table 6.

ChaCha

ChaCha has not been separately undertaken for an FPGA implementation. Since ChaCha makes the building block of BLAKE hash function, a combined lightweight coprocessor is designed for BLAKE and ChaCha using deep pipelining for high clock frequency [F4]. Various piepline configurations of ChaCha are reported for its 8, 12 and 20 round variants as specified in Table 6.

BLAKE

Kaps *et al.* presented a lightweight implementation of SHA-3 finalists, including Skein and BLAKE on various FPGAs [F5]. The initial state is stored in a four way distributed RAM for ease in access by the 1/2 G-function implemented. For BLAKE-32, the achieved area efficiency is comparable to the work by Beuchat *et al.* [F6]. For BLAKE-64, the area efficiency of the design proposed by Kaps *et al.* [F5] is more than twice than the one reported by Kerckhof *et al.* [F7] on the same FPGA device, due to the use of a 32 bit data width instead of 64 in the former.

Skein

For a lightweight implementation of Skein, Kaps *et al.* employed resource reuse by folding 4 Mix functions into 1 and within the Mix function reused a 32-bit adder to perform 64-bit additions. The adder is also used for key injections [F5]. Consequently the area is reduced but the number of clock cycles increase significantly when compared to other lightweight 64 bit [F7] and 32 bit implementations [F8].

A.1 FPGA Implementations References

[F1] J. Yan, H. M. Heys. Hardware Implementation of the Salsa20 and Phelix Stream Ciphers. In *Canadian Conference on Electrical and Computer Engineering (CCECE)* 2007, pages 1125-1128.

[F2] M. Rogawski. Hardware evaluation of eSTREAM Candidates: Grain, Lex, Mickey128, Salsa20 and Trivium. In *State of the Art of Stream Ciphers Workshop (SASC)* 2007, eSTREAM, ECRYPT Stream Cipher Project Report, volume 25. Available at `http://www.ecrypt.eu.org/stream`.

[F3] K. Gaj, G. Southern and R. Bachimanchi. Comparison of hardware performance of selected Phase II eSTREAM candidates. In *State of the Art of Stream Ciphers Workshop (SASC)* 2007, eSTREAM, ECRYPT Stream Cipher Project Report, volume 26. Available at `http://www.ecrypt.eu.org/stream`.

[F4] N. At, J. L. Beuchat, E. Okamoto, I. San and T. Yamazaki. Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA. In Cryptology ePrint Archive Report 2013/113. Available at `http://eprint.iacr.org/2013/113`.

[F5] J. P. Kaps, P. Yalla, K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung and J. Pham. Lightweight Implementations of SHA-3 Candidates on FPGAs. In *Progress in Cryptology-INDOCRYPT* 2011, pages 270-289.

[F6] J. L. Beuchat, E. Okamoto, T. Yamazaki. Compact Implementations of BLAKE-32 and BLAKE-64 on FPGA. In *International Conference on Field-Programmable Technology (FPT)* 2010, pages 170-177.

[F7] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni and G. de Dormale and F.X. Standaert. Compact FPGA implementations of the five SHA-3 finalists. In *Smart Card Research and Advanced Applications* 2011, pages 217-233.

[F8] B. Jungk. Compact implementations of Grostl, JH and Skein for FPGAs. In *ECRYPT II Hash Workshop* 2011. Available at `www.ecrypt.eu.org/hash2011/proceedings/hash2011_09.pdf`.

[F9] K. Latif, M. Tariq, A. Aziz, A. Mahboob. Efficient hardware implementation of secure hash algorithm (SHA-3) finalist-Skein. In *Frontiers in Computer Education* 2012, pages 933-940.

B. Overview of ARX-based Algorithms

ARX based cryptographic algorithms are limited not just to one class of cryptographic functions. Several hash functions, stream ciphers and block ciphers are based on ARX. A Few noticeable examples include FEAL [29] and Threefish (block ciphers), Salsa20, ChaCha, HC-128 (stream ciphers) BLAKE and Skein (hash functions). Including bitwise boolean functions with ARX includes more hash functions like MD4, MD5, SHA-1 to ARX family. 6 out

of the 14 second-round candidates of NIST SHA-3 hash function competition are ARX based: Blue Midnight Wish, CubeHash, Shabal, SIMD, BLAKE and Skein, out of which two reached final round [2]. Also 2 out of 7 finalists of the eSTREAM project (Salsa20, HC-128) are ARX based. Algorithmic descriptions of the ARX algorithms chosen for mapping on our coprocessor are given below.

B.1 BLAKE Hash Functions Family

BLAKE is a cryptographic hash function chosen as one of the finalists of the SHA-3 hash function competition. It performs well in software as well as in hardware [17]. As per the initial specifications, its has four major variants as given in Table 7.

Table 7. BLAKE HASH functions

Algorithm	Word	Message	Block	Digest	Salt
BLAKE-224	32	$< 2^{64}$	512	224	128
BLAKE-256	32	$< 2^{64}$	512	256	128
BLAKE-384	64	$< 2^{128}$	1024	384	256
BLAKE-512	64	$< 2^{128}$	1024	512	256

BLAKE follows a HAIFA iteration mode which is an improved version of Merkle-Damgård paradigm. The BLAKE compression function takes the following input values:

- an 8-word chaining value, $h = h_0, h_1..., h_7$
- a 16-word message block, $m = m_0, m_1..., m_{15}$
- a 4-word salt, $s = s_0, s_1..., s_3$
- a 2-word counter, $t = t_0, t_1$

Apart from the above mentioned input values, BLAKE uses 16 constants $c = c_0, c_1..., c_{15}$. The BLAKE's compression function has three steps:

- Initialization
- Rounds calculation
- Finalization

In the initialization round, a 4x4 matrix is initialized such that results of initialization vary with varying inputs. Chaining hash value or initialization value, salt, counter, and constants are used in the initialization stage. The input to the initialization process is as follows:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix}$$

After the initialization the initial state is given by:

$$\begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

Where h_i are the initialization or the chaining values, s_i are the salt, c_i are the constants and t_i are the counter values. After the initialization stage, the round stage begins and depending upon the BLAKE variant, the round function iterates for a number of rounds. A round function performs the transformation in the following manner:

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13})$$
$$G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$

Fig. 5 shows a column and diagonal step on which the following transformation is applied:

$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12})$$
$$G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

Where the function $G_i(a, b, c, d)$ is as follows:

$$\begin{aligned} a &= a + b + (m_{\sigma_r(2*i)} \oplus c_{\sigma_r(2*i+1)}) \\ d &= (d \oplus a) \ggg R_1 \\ c &= c + d \\ b &= (b \oplus c) \ggg R_2 \\ a &= a + b + (m_{\sigma_r(2*i+1)} \oplus c_{\sigma_r(2*i)}) \\ d &= (d \oplus a) \ggg R_3 \\ c &= c + d \\ b &= (b \oplus c) \ggg R_4 \end{aligned}$$

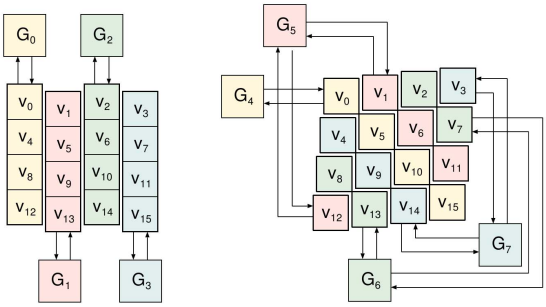


Figure 5. Diagonal and column steps of BLAKE [6]

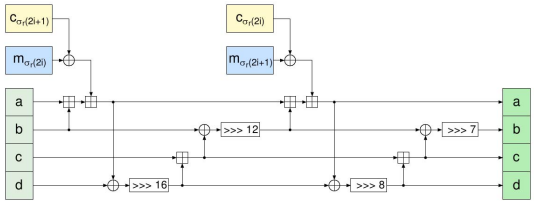


Figure 6. G function of BLAKE hash [6]

Fig. 6 shows a G_i function. Where $\sigma_0, ..., \sigma_{10}$ are ten permutations of the numbers $0, ..., 15$, r denotes the round number, R_i denotes the rotation constants which are dependent on the variants of BLAKE, c_i are the constants and m_i are the messages. The first four invocations of the G function (G_0, G_1, G_2, G_3) can operate in parallel and the next invocations (G_4, G_5, G_6, G_7) can operate in parallel after that. After each round a chaining value h is calculated and it is fed to the new round. After a specified number of rounds (depending on BLAKE variants) the round function is complete and the finalization step starts. The input to the finalization stage is the initial state $v_0, ..., v_{15}$, salt $s_0, ..., s_3$ and the chaining value $h_0, ..., h_7$. The finalization stage performs the following steps:

$$\begin{aligned} h'_0 &= h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\ h'_1 &= h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\ h'_2 &= h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\ h'_3 &= h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\ h'_4 &= h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\ h'_5 &= h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\ h'_6 &= h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\ h'_7 &= h_7 \oplus s_3 \oplus v_7 \oplus v_{15} \end{aligned}$$

Where h'_i denotes the hashed value that can be the final output or the chaining value. To hash a message that has more than 16 words, the message is broken into blocks, each of 16 words. After that the compression function is applied to it, the resultant h'_i is chained to the next iteration of the compression function. The process continues till we reach the end of message. This process is given as:

$$\begin{aligned} &h^0 = \mathbf{IV} \\ &\text{for } i = 0, ..., N-1 \\ &h^{i+1} = \mathbf{compress}(h^i, m^i, s, l^i) \\ &\mathbf{return}\ h^N \end{aligned} \tag{3}$$

where h is the chaining value, N is the number of message chunks each of 16 words, s denotes the salt, and h denotes the output of the compression function.

B.2 Skein Hash Functions Family

Skein is also one of the five finalists of the SHA-3 hash competition [15]. The datapath of Skein is 64 bits, which makes it an excellent candidate to be implemented on 64 bit microprocessors. Threefish, a tweakable block cipher is

at the core of the Skein hash function. Threefish block cipher has three inputs, key, tweak and plain text. The tweak is 128 bits for all block sizes. Based on the internal state size of the cipher, there are three variants of Threefish, Threefish-256, Threefish-512 and Threefish-1024. Threefish can have multiple state sizes and multiple output sizes.

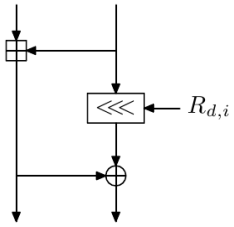


Figure 7. A MIX function of Skein [15]

The compression function of Skein consists of Threefish block cipher in Matayas-Meyer-Oseas (MMO) construction form. Threefish in MMO mode along with tweak specification defines the Unique Block Iteration mode. The UBI mode converts variable length inputs to fixed length outputs. Threefish uses a large number of simple rounds instead of small number of complex rounds. The core of Threefish is a MIX function that is made up of an Addition, Rotation and Xor. Fig. 7 shows a MIX function.

In case of Skein-512, the primary candidate of Skein family, four MIX functions are required to constitute one round of Skein-512. Fig. 8 shows the construction of Skein-512 using MIX functions.

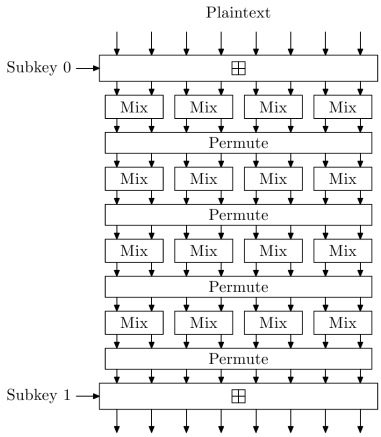


Figure 8. 4 Rounds of Skein-512 Hash [15]

After every four rounds, a subkey has to be injected. The subkey is injected by adding it with the outputs of MIX functions after every fourth round. A subkey is generated using keywords, tweakwords and subkey number in a subkey generator. Skein-256 has two MIX operations in every round and Skein-1024 has 8 MIX operations in every round. Skein-512 has 72 rounds. Fig. 9 shows the key generator for Skein-512.

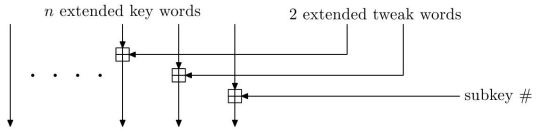


Figure 9. Keygenerator of Skein [15]

In normal hashing mode, Skein has three invocations of the UBI. One invocation is for the configuration, one for the message and the last invocation is for the output transform. Fig. 10 shows Skein in a normal hashing mode. The rotation constants of Skein repeat after every eight rounds and thus VLSI hardware are made, consisting of eight rounds along with two subkey injections.

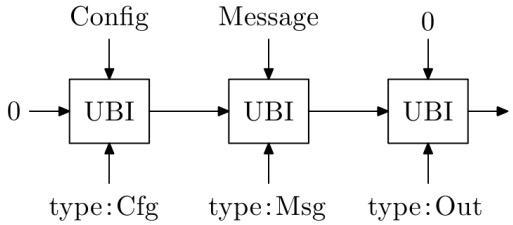


Figure 10. Skein in normal hashing mode [15]

B.3 HC-128 Stream Cipher

HC-128 is one of the finalists of eStream project [36]. The cipher comprises of two secret tables. A non-linear feedback function is used to update the elements inside the tables. During the update, three consecutive steps can be computed in parallel. HC-128 makes use of the following mathematical functions:

$+$: addition modulo 2^{32}.

$\boxminus$: subtraction modulo 512.

$\oplus$: bit-wise exclusive OR.

$\|$: bit-string concatenation.

$\gg$: right shift operator (defined on 32-bit numbers).

$\ll$: left shift operator (defined on 32-bit numbers).

$\ggg$: right rotation operator (defined on 32-bit numbers).

$\lll$: left rotation operator (defined on 32-bit numbers).

Two internal state arrays P and Q are used in HC-128, each with 512 many 32-bit elements. A 128-bit key array $K[0, \ldots, 3]$ and a 128-bit initialization vector $IV[0, \ldots, 3]$ are used, each entry being a 32-bit element. Let s_t denote the keystream word generated at the t-th step, $t = 0, 1, 2, \ldots$.

The following six functions are used in HC-128.

$$f_1(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3),$$

$$f_2(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10),$$

$$g_1(x, y, z) = ((x \ggg 10) \oplus (z \ggg 23)) + (y \ggg 8),$$

$$g_2(x, y, z) = ((x \lll 10) \oplus (z \lll 23)) + (y \lll 8),$$

$$h_1(x) = Q[x^{(0)}] + Q[256 + x^{(2)}],$$

$$h_2(x) = P[x^{(0)}] + P[256 + x^{(2)}],$$

where $x = x^{(3)} \| x^{(2)} \| x^{(1)} \| x^{(0)}$ is a 32-bit word, with $x^{(0)}, x^{(1)}, x^{(2)}$ and $x^{(3)}$ being the four bytes from right to left.

The key and IV setup of HC-128 recursively loads the P and Q array from expanded key and IV and run the cipher for 1024 steps to use the outputs to replace the table elements. It happens in four steps as follows.

Let $K[i+4] = K[i]$ and $IV[i+4] = IV[i]$ for $0 \le i \le 3$.
The key and IV are expanded into an array $W[0, \ldots, 1279]$ as follows:

$$\begin{aligned} W[i] &= K[i], && \text{for } 0 \le i \le 7; \\ &= IV[i-8], && \text{for } 8 \le i \le 15; \\ &= f_2(W[i-2]) + W[i-7] \\ &\quad + f_1(W[i-15]) + W[i-16] + i, && \text{for } 16 \le i \le 1279. \end{aligned}$$

Update the tables P and Q with the array W as follows:
$P[i] = W[i + 256]$, for $0 \le i \le 511$,
$Q[i] = W[i + 768]$, for $0 \le i \le 511$.
Run the cipher 1024 steps and use the outputs to replace the table elements as follows:
For $i = 0$ to 511, do
$P[i] = (P[i] + g_1(P[i \boxminus 3], P[i \boxminus 10], P[i \boxminus 511])) \oplus h_1(P[i \boxminus 12])$;
For $i = 0$ to 511, do
$Q[i] = (Q[i] + g_2(Q[i \boxminus 3], Q[i \boxminus 10], Q[i \boxminus 511])) \oplus h_2(Q[i \boxminus 12])$;

The keystream is generated using the following algorithm.

$i = 0$;
repeat until enough keystream bits are generated
{
    $j = i \bmod 512$;
    if $(i \bmod 1024) < 512$
    {
        $P[j] = P[j] + g_1(P[j \boxminus 3], P[j \boxminus 10], P[j \boxminus 511])$;
        $s_i = h_1(P[j \boxminus 12]) \oplus P[j]$;
    }
    else
    {
        $Q[j] = Q[j] + g_2(Q[j \boxminus 3], Q[j \boxminus 10], Q[j \boxminus 511])$;
        $s_i = h_2(Q[j \boxminus 12]) \oplus Q[j]$;
    }
    end-if
    $i = i + 1$;
}
end-repeat

B.4 Salsa20/r Stream Cipher

Salsa20 is a stream cipher that makes use of hash function in counter mode to generate keystream bits (S) that are mixed with the plaintext (P) by xoring to generate ciphertext (C) [7]. Block size of its inputs and outputs is 64 bytes. The input counter is incremented for every preceding block of plaintext. Salsa20 accepts four types of inputs, each consisting of words of 32 bit granularity. Firstly, an input key, that is either 256-bit ($k_0, k_1, ..., k_7$) or 128-bit ($k_4 = k_0, ..., k_7 = k_3$) in size; further a 64-bit nonce (n_0, n_1) and a 64-bit counter (t_0, t_1); finally four words of pre-defined constants ϕ_i, whose values are dependent upon the key size. These inputs are arranged in a predefined order in a 4x4 vector, called the initialization vector (X) in the rest of the discussion as shown below.

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} \phi_0 & k_0 & k_1 & k_2 \\ k_3 & \phi_1 & n_0 & n_1 \\ t_0 & t_1 & \phi_2 & k_4 \\ k_5 & k_6 & k_7 & \phi_3 \end{pmatrix}$$

The initialization vector is subjected to a series of rounds composed of additions, cyclic rotations and xors, to achieve a random permutation. Originally the number of rounds was set to 20 (Salsa20/r, $r = 20$); however, the version of cipher included in the eSTREAM portfolio was reduced to 12 rounds, for performance reasons.

Let X, S and P be 4x4 arrays of 16-words each enumerated as $(x_0, x_1, ..., x_{15})$, representing initialized vector, Salsa20 keystream and plaintext respectively. Then Salsa20/r function for keystream generation can be represented mathematically as:

$$\begin{aligned} S &= Salsa20_k(X) \\ Salsa20_k(X) &= DoubleRound^r(X) + X \\ DoubleRound(X) &= RowRound(ColumnRound(X)) \end{aligned}$$

Each *DoubleRound* comprises of four *QuarterRounds* performed on the columns of the initialization vector X, followed by four *QuarterRounds* performed on the rows of the output.

$$\begin{aligned} ColumnRound(X) = (y_0, y_1, ..., y_{15}) &= Y \\ (y_0, y_4, y_8, y_{12}) &= QuarterRound(x_0, x_4, x_8, x_{12}), \\ (y_5, y_9, y_{13}, y_1) &= QuarterRound(x_5, x_9, x_{13}, x_1), \\ (y_{10}, y_{14}, y_2, y_6) &= QuarterRound(x_{10}, x_{14}, x_2, x_6), \\ (y_{15}, y_3, y_7, y_{11}) &= QuarterRound(x_{15}, x_3, x_7, x_{11}). \end{aligned}$$

$$\begin{aligned} RowRound(Y) = (z_0, z_1, ..., z_{15}) &= Z \\ (z_0, z_1, z_2, z_3) &= QuarterRound(y_0, y_1, y_2, y_3), \\ (z_5, z_6, z_7, z_4) &= QuarterRound(y_5, y_6, y_7, y_4), \\ (z_{10}, z_{11}, z_8, z_9) &= QuarterRound(y_{10}, y_{11}, y_8, y_9), \\ (z_{15}, z_{12}, z_{13}, z_{14}) &= QuarterRound(y_{15}, y_{12}, y_{13}, y_{14}). \end{aligned}$$

Where each $QuarterRound(a, b, c, d)$ consists of four $ARXrounds$, named so since they comprise of additions (A), cyclic rotations (R) and xoring (X) operations only as given:

$$\begin{aligned} b &= b \oplus ((a + d) \lll 7), \\ c &= c \oplus ((b + a) \lll 9), \\ d &= d \oplus ((c + b) \lll 13), \\ a &= a \oplus ((d + c) \lll 18). \end{aligned}$$

A 4x4 matrix C, representing a 16-word ciphertext block is calculated simply by xoring it with the keystream ($C = S \oplus P$).

B.5 ChaCha20/r Stream Cipher

ChaCha, a variant of Salsa20, was proposed by the author of Salsa20. The main aim of its creation was to improve the diffusion per round in Salsa20 which does not come at the expense of extra operations [7]. Being similar to Salsa20, ChaCha also has reduced round variants. The major differences between Salsa20 and ChaCha are following:

- Initialization matrix
- Inputs to the *DoubleRound* function
- *QuarterRound* function

The initialization matrix in ChaCha differs from Salsa20 in the following manner:

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & n_0 & n_1 \end{pmatrix}$$

Where k_i, ϕ_i, n_i and t_i are key, constants, nonce and counter words each of 32 bit granularity, respectively. The *DoubleRound* calculation in ChaCha operates in the following order:

$$\begin{aligned} ColumnRound(X) = (y_0, y_1, ..., y_{15}) &= Y \\ (y_0, y_4, y_8, y_{12}) &= QuarterRound(x_0, x_4, x_8, x_{12}), \\ (y_5, y_9, y_{13}, y_1) &= QuarterRound(x_1, x_5, x_9, x_{13}), \\ (y_{10}, y_{14}, y_2, y_6) &= QuarterRound(x_2, x_6, x_{10}, x_{14}), \\ (y_{15}, y_3, y_7, y_{11}) &= QuarterRound(x_3, x_7, x_{11}, x_{15}). \end{aligned}$$

$$\begin{aligned} RowRound(Y) = (z_0, z_1, ..., z_{15}) &= Z \\ (z_0, z_1, z_2, z_3) &= QuarterRound(y_0, y_5, y_{10}, y_{15}), \\ (z_5, z_6, z_7, z_4) &= QuarterRound(y_1, y_6, y_{11}, y_{12}), \\ (z_{10}, z_{11}, z_8, z_9) &= QuarterRound(y_2, y_7, y_8, y_{13}), \\ (z_{15}, z_{12}, z_{13}, z_{14}) &= QuarterRound(y_3, y_4, y_9, y_{14}). \end{aligned}$$

Where X is the 4x4 array of 16-words enumerated as $(x_0, x_1, ..., x_{15})$ representing initialized vector.

The third difference between Salsa20 and ChaCha is the structure of the *QuarterRound*.

$$\begin{aligned} a &= a + b \\ d &= (d \oplus a) \lll 16 \\ c &= c + d \\ b &= (b \oplus c) \lll 12 \\ a &= a + b \\ d &= (d \oplus a) \lll 8 \\ c &= c + d \\ b &= (b \oplus c) \lll 7 \end{aligned}$$