CCproc: An Efficient Cryptographic Coprocessor

Dimitris Theodoropoulos†‡, Ioannis Papaefstathiou‡, Dionisios Pnevmatikatos‡[1]

† Computer Engineering,
TU Delft, The Netherlands,
{D.Theodoropoulos}@tudelft.nl

‡ ECE Department,
Technical University of Crete, Greece, GR73100
{ygp, pnevmati}@mhl.tuc.gr

ABSTRACT

In this paper we introduce CCproc, a symmetric-key cryptographic (co)processor with a custom instruction set optimized for cryptographic applications. We study ten popular crypto algorithms, and provide custom solutions for them, while we also offer general support for future encryption algorithms. We design a custom but simple datapath able to execute the proposed instruction set and analyze its performance, proving to be competitive with other general purpose (but not custom) approaches, while having very small implementation cost.

1. INTRODUCTION & MOTIVATION

Cryptography is routinely used to protect the privacy and authenticity of documents and communications, and is applied in large computer systems but also in handheld devices such as cell phones. Encryption algorithms are used by an increasing number of applications, and can consume as much as 95% of a server's processing capacity [14]. Therefore it is desirable to offload the encryption burden to an encryption accelerator. Much work has been done in the area with many software and hardware specific symmetric-key algorithmimplementations (e.g. [1], [3], [7], [11], [13], [15]). Unlike most previous work, we focus on general (co)processor design that will be able to implement many today's popular algorithms but also to efficiently support potential new ones. We propose CCproc, a simple 32-bit co-processor with an extended RISC instruction set and datapath structure. The instruction set includes several algorithm specific instructions, as well as support for general purpose processing.

To design the CCproc instruction set, we first studied 10 popular symmetric-key algorithms: Blowfish [20], Twofish [19], DES [25], AES [8], MARS [6], Serpent [2][15], IDEA [22], RC4 [24], RC5 [18] and RC6 [17]. This group includes the 5 finalists for AES (Rijndael, Twofish, MARS, RC6, Serpent). This way we have covered a representative set of symmetric-key algorithms, and designed a customized ISA that will be able to efficiently implement these and new algorithms. Our work is based on the following observations:

- Symmetric-key algorithms exhibit very small amounts of internal parallelism. The most commonly used mode is CBC (Cipher Block Chaining), where each plaintext block is first XORed with the previous encrypted block before it is encrypted. This operation serializes the entire computation, leaving little hope for speedup using parallelism (thread, ILP, or other type).
- Often, algorithms specify a sequence of operations, i.e. add followed by another add, xor, etc. It is very efficient to have instructions that combine these simple operations.
- We use a 32-bit wide datapath and exploit internal data parallelism in the algorithms.
- Modulo multiplication with a prime polynomial in GF (2^8) is common in symmetric-key algorithms. Our co-processor uses 32-bit values and combines 4 such multiplications in one instruction.
- Rotations and shifts are heavily used in the symmetric-key algorithm field.
- Algorithms perform a fixed number of rounds, resulting in a simple and predictable branch behavior we can exploit using a "loop" instruction to reduce the branch penalty.
- Permutations are only used by DES and Serpent, so we forgo general permutation boxes and offer specific hardware permutation units for each of the two algorithms.
- We offer custom, independent sboxes for each algorithm, removing the need for Sbox initialization. We also pre-compute the combined results of an Sbox followed by an operation, reducing the number of operations needed.

Based on the above observations, we created an instruction set specifically designed to improve processing of symmetric-key algorithms. This instruction set will be analyzed in the next sections along with the general purpose instructions that are included in CCproc's ISA. We first describe the CCproc's instruction set and datapath. Then, in sections 3 and 4, we analyze its ISA and its performance, and discuss related work, and section 5 offers our conclusions and thought for future work.

2. CCPROC BASE ISA AND DATAPATH

In order for CCproc to execute general-purpose programs, it supports all the basic arithmetic and logical operations. We use a simple (MIPS-like) RISC instruction set. We include only two simple conditional branch operations: comparison for (in)equality with zero, allowing for faster implementation. After analyzing the symmetric-key algorithms, we found that most loops iterate for a fixed number of times and we use a "*loop"* instruction. A loop begins with an *ldlc* instruction, which loads loop-count (lc) register the number of iterations. *Loop* takes a single address argument, and each time it is executed the *lc* register is reduced by 1. The remaining instructions such as subroutine calls, stack usage and data memory access are all based on MIPS ISA.

CCProc uses a 5-stage pipelined datapath similar to MIPS R3000. The main differences are: (a)**:** We implement the *"loop"* instruction in the Instruction fetch stage, (b) we implement all cascaded and modulo multiplication instructions in the execution stage, and (c) we perform accesses to Sboxes in the MEM stage. Table 1 shows the instruction formats, where Op (6-bit) is the operation code, s*x* (5-bit) is a source register (x=1..3), A (5-bit)

[1]Dionisios Pnevmatikatos is also with FORTH-ICS

is the algorithm ID, TR (5-bit) is a target register, opt (2-bit) is the options an instruction might have, DA (26-bit) is the destination address and I (16-bit) is the immediate.

Table 1. CCproc Instruction formats

R	Op	s3	s2	A	TR	s1	Opt
J	Op	DA					
I	Op	I[15..12]			TR	s1	I[1..0]

2.1 Double/Cascaded Instructions

Double instructions combine two or more dependent operations. An example is *addadd rd,ra,rb,rc*, i.e. rd = (ra + rb) + rc. A similar approach has also been considered by authors in [21] and in other contexts in [12]. We used the following double instructions in our instruction set: AddXor, XorXor, AddAdd, and Gfmul4. We provided only logical, or arithmetic-logical pairs, since they are very cheap to implement and do not increase the latency noticeably. To support these instructions, CCProc can read up to 3 registers from the RF. *Gfmul4xor* is an instruction that performs the GF(2^8) multiplication modulo a prime polynomial, an operation used in many cryptography algorithms. It performs 4 independent multiplications and combines their results using xor. We used the Karatsuba algorithm that for an 8-bit GF(2^8) multiplier requires about 90 simple gates (with four or fewer inputs) arranged in 15 logic levels [10].

2.2 Algorithm-Specific Instructions

Besides the above double instructions, our study of the crypto algorithms led us to include in CCproc the list of custom instructions listed in Table 2. Below we address each algorithm:

Table 2. Algorithm specific instructions

Algorithm	Instruction	Operation
Blowfish	bld rc,ra,bsboxi	rc←bsboxi[ra], i=1,2,3,4
	bst ra,rc,bsboxi	bsboxi[ra]←rc, i=1,2,3,4
Twofish	tld rc,ra,comi - i=0,1,2	rc←tsbox[ra]
DES	desldp rc,ra,rb	Rc←dessbox[rb&ra]
	desip rc,ra,rb,per - per=IP,IP1	rc←32LSBs [permute(rb:ra)]
		high1←32MSBs [permute(rb:ra)]
	desp rc,ra,rb,per - per=IP, E, PC1, PC2	rc←32LSBs [permute(rb:ra)]
		high1←32MSBs [permute(rb:ra)]
AES	aesld rc,ra,mode – mode=en, dec, rcon	rc←aessbox[ra]
MARS	marsld rc,ra, sel – sel = sbox select	rc←marssbox[ra]
	marslde rc,ra	rc←marssbox[ra]
Serpent	serldel rc,ra,sersboxi	rc←sersboxi[ra], i=0,1,2,3
	serldeh rc,ra,sersboxi	rc←sersboxi[ra], i=4,5,6,7
	serlddl rc,ra,sersboxi	rc←sersboxi[ra], i=0,1,2,3
	serlddh rc,ra,sersboxi	rc←sersboxi[ra], i=4,5,6,7

Blowfish has a very large initialization process of the 18 Subkeys and 4 Sboxes, 256x32 bits each. To avoid accessing CCproc's data main memory we use a separate 1024x32 bit memory, that stores the 4 Sboxes. To access an Sbox we use *bst* (store) or *bld* (load) and specify the Sbox with *bsboxi*=i, (i=1..4). To store a value to Sboxi, we read 2 registers, the first for the Sboxi address and the second for the data to be stored.

Towfish requires a sequence of Sbox operations, so we have pre-computed the entire q_0 and q_1 basic Sboxes and created 2 LUTs with these names. We used separate memories and optimized the layout to reduce the total size. Shown in Figure 1, there are 2 inputs, *IN* and *mode*. *IN* is the 32-bit register that contains 4 bytes that access q_0 and q_1. The column accessed depends on *mode*. The entire result is computed from the Sboxes with 5 instructions: 3 *tld* for each column and 2 for xoring the results.

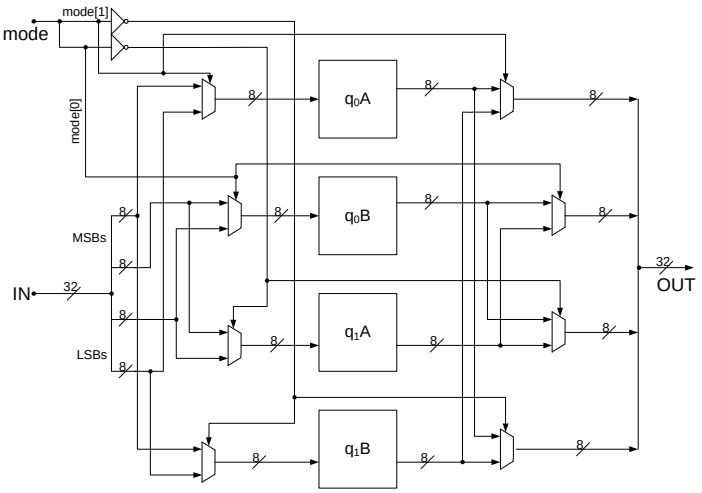

Figure 1. Twofish Sboxes in CCproc.

DES mostly uses permutations and Sboxes, offering a clear target. It needs 8 Sboxes that use 8x6=48-bit for address and they will return 32-bit output. The *desldp* instruction reads 2 registers from RF that contains the 48-bit address. Next, this address is split to 8 6-bit values to access the 8 Sboxes and the 8 4-bit values are concatenated. The last result must pass through the P-permutation and then will be stored to rc. Finally, the d*esp* instruction selects among permutations E, PC-1, and PC-2 and *desip* between IP and IP^{-1}.

AES can take advantage of parallelism and achieve better performance. Besides the *gfmul4xor* instruction, another parallel operation is SubBytes, which substitutes all bytes of other bytes, depending on the AES LookUpTable (LUT). We included the instruction *aesld*, which fetches a 32-bit register, and process the 4 bytes independently, passes them either through 4 LUTs or Rcon depending on the *mode*. LUTs are AES's Sboxes and InvSboxes when it comes for encryption and decryption respectively. Rcon is used for key scheduling.

MARS uses 2 256x8 bit Sboxes (S0 and S1) that are accessed in 2 different ways according to the phase of processing. Our analysis lead to the design of the circuit shown in Figure 2, where *IN*, *mode* and *E* are the 32-bit value ra, sel and E respectively and *OUT* is the result to rc register.

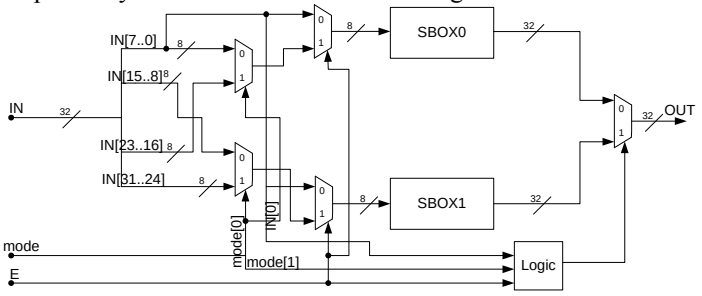

Figure 2. MARS Sboxes in CCproc.

Serpent has 32 rounds and 8 Sboxes (S0 to S7) 4x4 bits each. To gain speed in Sbox access, we designed instructions *serldel,*

serldeh, serlddl and *serlddh*, which use S0 to S3 for encryption, S4 to S7 for encryption, InvS0 to InvS3 for decryption and Invs4 to InvS7 for decryption respectively. For example *serldel* will fetch from RF register ra which is a 32-bit value and consequently contains 8 4-bits values ready to access 8 aliases of the appropriate Sbox, depending on round number.

3. PERFORMANCE EVALUATION

3.1 Algorithm Analysis

To estimate our design's performance, we developed CCproc assembly codes for 4 algorithms, from which 3 used some algorithm specific instructions (Twofish, Blowfish, AES) and one (RC4) that used the existing ISA, without any custom instructions. To obtain timing results we analyzed our assembly codes, measured the stall cycles, and form that we calculated the CPI rate of the simple 5-stage pipelined CCproc. First, we analyzed the branching behavior of these 4 algorithms, counting the overall appearance frequency of UBs, TCBs (taken conditional branches) and UCBs (untaken conditional branches). Table 3 presents the results considering predict not-taken, predict-taken and loop branch handling schemes. It turns out that branching is not a major performance bottleneck, and at worst adds 0.04 to the CPI. The introduction of the loop instruction reduces this overhead to 0.02, but delayed branches essentially eliminate all branch stalls. This is because the branching conditions are mostly *not* data dependent and we can almost always fill the delay slot with useful computation. We also evaluated the stalls due to memory access when we need the value of a memory load (load-use stalls). The CPI cost of load-use hazards is 0.076. In the case of delayed branching, the overall CPI of CCproc's is 1.076. We will use these CPIs to compute the throughput for these 4 algorithms.

Table 3. CCproc CPI costs

B.S.	UBs	TCBs	UCBs	Loop	All-br	Ld-use
Dyn. Freq.	2.1%	2.40%	0.32%	0%	4.8%	7.6%
Predict Not	0.024	0.025	0	0	0.045	0.076
Predict Taken	0	0	0.003	0	0.003	0.076
Dyn. Freq.	2.1%	0.003%	0.2%	2.5%	4.8%	7.6%
Loop	0	0	0.002	0	0.002	0.076
Delayed	0	0	0	0	0	0.076

3.2 Execution Time Results

Table 4 shows the code size for the four algorithms. It lists the the algorithm that was tested, the block size and key size, i.e. the bits number of plaintext to be processed and the number of key bits, the static code size of each algorithm and in parenthesis the key schedule size (KS) and the processing size (E). The 4th column gives the number of instructions needed for the encryption (E) or decryption (D) process, without the key schedule and the last column offers the number of instructions needed for key scheduling.

Table 4. Code size results for 4 algorithms

Algorithm	Plaintext / Key	Static Code Size: Total, (KS, E)	Dynamic Code (E/D)	Dynamic code size (KeySched)
Twofish	128 / 128	1132 (788, 344)	971	1973
Blowfish	64 / 32	1136 (920, 216)	564	137596
Rijndael	128 / 128	620 (140, 480)	1120	179
RC4	128 / 128	564 (292, 272)	293	4153

Code size is an important issue, with smaller values being better, especially for embedded devices. For example, in [19] the smallest size is 8,200 bytes for an Intel's Pentium processor and the biggest size is 23,300 in Motorola's 68040 processor, which means that with the CCproc ISA we have a reduction in code size to 1,132/8,200 = 13.8% and to 1,132/23,300 = 4.8%, or a reduction factor of 7 and 20 respectively. The fact that code size drops so dramatically is an indication of the efficiency of the instruction set that allows us to store the code for all all these 4 algorithms with about 3.5Kbytes of instruction memory. The result is that, despite its design simplicity, CCproc can have the ability to implement many symmetric-key algorithms in a very compact form, becoming a very attractive low-cost solution.

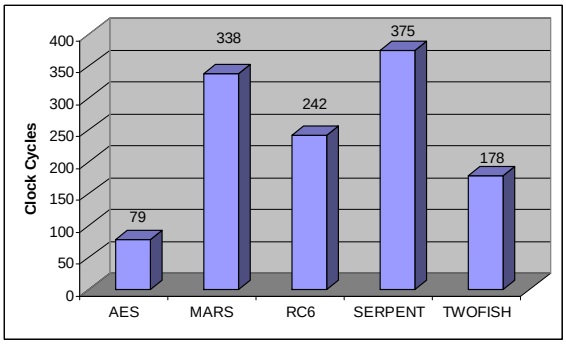

Figure 3. CCproc performance in clock cycles

In terms of execution efficiency, Figure 3 shows the amount of time (cycles) needed per round for the 5 AES finalists. These results, together with the following formula give the throughput per MHz for every CCproc implementation:

$$Throughput = \frac{128 \cdot Frequency}{ClockCyclesPerRound} Mbits/\sec$$

Applying this formula gives 405, 95, 132, 85, and 180 Mbits/second for AES, Mars, RC6, Serpent, and Twofish respectively for out semi-custom implementation.

3.3 Implementation Results

To evaluate the performance of CCproc, we implemented the entire design in VHDL and used Synopsys to synthesize it using the 0.13um UMC High Density Standard Cell Library. Our design used 93K cells, and the total cell area was 5.3mm^2, out of which 0.73mm^2 was the combinational logic, and the rest were devoted to the instruction memory and register file. In terms of performance, the instruction memory standalone maximum operating frequency is 350MHz (2.8nsec cycle time), which of course is a limit to CCproc's performance. The overall processor operating frequency is 250MHz using an "out-of-the-box" approach, without specific optimizations other than selecting the best synthesis strategy and combination of Synopsys commands.

4. RELATED WORK

The field of cryptography algorithms is peculiar in the sense that each algorithm usually has few similarities with others in data processing. The research on how to combine all of them in a design and also support potential new ones, requires studying one by one deeply and carefully. If in all that, one adds the demand of a compact design, this makes it even more difficult.

So far we have seen [21] and [5] as proposals for designing an ISA for symmetric cryptography algorithms. The first is about Cryptomaniac, an architecture which also supports combined

instructions and can process in parallel algorithms if there are no dependencies, by adding extra functional units. The second describes an Alpha instruction set extension to improve symmetric algorithm processing. Another hardware approach for speeding up symmetric-key algorithms processing comes, as stated above, from [9]. The authors take advantage of ICBC (Interleaved Cipher Block Chaining) mode and combine it with usage of Symmetric Multi Processors (SMPs) in order to exploit algorithm parallel processing.

CCproc uses cascaded instructions, as [21] and [5] do. However, we define a very small number of such instructions avoiding the unnecessary generality, and use a very simple datapath and a few custom sboxes. Moreover, CCproc's performance does not rely on additional resources, but almost exclusively on its ISA efficiency.

In the products field, HiFn [26] is a company that designs cryptography acceleration boards, such as their "*Hifn Access HXL*" . The latter supports RSA public-key algorithm and 3DES+SHA-1, AES128+SHA1 and ARC4+MD5 symmetric-key algorithms and hash functions. Via technologies designed Nehemiah core, which is being used in their C3 processor [27]. The latter is an x86 compatible processor and uses the Advanced Cryptography Engine (ACE) which supports AES and can encrypt or decrypt data at a rate of 12.8 Gb/s. For a single encryption or decryption, the effective rate can be even faster, up to 21 Gb/s.

5. CONCLUSIONS – FUTURE WORK

From the study of symmetric-key algorithms and the results so far, we have concluded that the processing performance can be improved in 3 different ways; (a) to design extra instructions specifically for symmetric-key algorithms as we described above, (b) to use simultaneous data processing, and (c) to increase speed i.e. MHz. However, the most widely used mode is CBC due to its good data diffusion properties, but cannot allow parallel plaintext block processing. This result led the cryptography community to propose Interleaved CBC (ICBC) mode as stated in [9], in order to deliver higher performance and enable partial parallel data processing.

As future work, our next step is to create a hardware model for CCproc and perform algorithm simulations and tests and obtain a more realistic image of our design so far. This model may be designed with a hardware description language or by using SimpleScalar processor simulator [4]. A few additions may be added to the design, depending on future results. For example, after further examination of other symmetric-key algorithms, we may design new instructions or improve the existing ones, in order to be also supported by them. Symmetric-key algorithm parallelism is an aspect that we can take advantage of it and add more processing units, improving even more CCproc's overall performance. Last, another design addition may be added to CCproc so as to take advantage of the ICBC mode and boost even more its performance.

REFERENCES

[1] R. Ashruf, G. Gaydadjiev, S. Vasiliadis, "Reconfigurable Implementation for the AES Algorithm", Proc. of ProRISC 2002.

[2] Ross Anderson, Eli Biham, Lars Knudsen, "*Serpent: A Proposal for the Advanced Encryption Standard*", 5th workshop on Fast Software Encryption, 1998.

[3] Albert G. Broscius, Jonathan M. Smith, "*Exploiting Parallelism in Hardware Implementation of the DES*", CRYPTO 1991.

[4] D. C. Burger and T. M. Austin, "*The simplescalar tool set, version 2.0*", Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[5] J. Burke, J. McDonald, T. Austin, "*Architectural Support for Fast Symmetric-Key Cryptography*", ASPLOS 2000.

[6] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O'Connor, M. Peyravian, D. Safford, N. Zunic, "MARS - *a candidate cipher for AES*", IBM Corporation, 1999.

[7] Pawel Chodowiec, Kris Gaj, "*Implementation of the Twofish Cipher Using FPGA Devices*", ECE, George Mason University, July 1999.

[8] Joan Daemen, Vincent Rijmen, "*A specification for Rijndael, the AES Algorithm*", March 2001, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/

[9] Praveen Dongara and T. N. Vijaykumar, "*Accelerating Private-Key Cryptography via Multithreading on Symmetric Multiprocessors*", In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2003.

[10] M. Jung, F. Madlener, M. Ernst and S. A. Huss, "*A Reconfigurable Coprocessor for Finite Field Multiplication in GF(2^8)*", IEEE Workshop on Heterogeneous reconfigurable Systems on Chip, Hamburg, April 2002.

[11] Benjamin Leperchey, Charles Hymans, "*FPGA Implementation of the Rijndael algorithm*", June 16, 2000.

[12] N. Malik, R. Eickemeyer, S. Vassiliadis, "*Interlock Collapsing ALU for Increased Instruction-Level Parallelism,"* Proceedings Micro-25, December 1992.

[13] Maire McLoone, John McCanny, "*Rijndael FPGA Implementations Utilizing Look-up tables*", Journal of VLSI signal processing 34, 261-275, 2003.

[14] M.S. Merkow, CCP and J. Breithaupt, "*The complete guide to internet security*", AMACOM 2000.

[15] Serge Mister,"*Properties of the Building Blocks of Serpent*", Entrust Technologies, May 15, 2000.

[16] National Institute of Standards and Technology, "*Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)*", *Federal Register*, v. 62, n. 117, 12 Sept 1997, pp. 48051 - 48058.

[17] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, "*The RC6 Block Cipher*", August 20, 1998.

[18] Ronald L. Rivest, "*The RC5 Encryption Algorithm"*, RSA Security®.

[19] Bruce Scheier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson, "*Twofish: A 128-bit Block Cipher*", 15 June 1998, Counterpane Systems.

[20] B. Schneier, "*Description of a new variable-length key, 64-bit block cipher (Blowfish)*", Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994.

[21] Lisa Wu, Chris Weaver, and Todd Austin, "*Cryptomaniac: A Fast Flexible Architecture for Secure Communication*", ISCA 2001, June 2001.

[22] "*International Data Encryption Algorithm"*, Technical Description, Mediacrypt®.

[23] Analysis of Systems and Software (ISPASS), March 2003.

Internet links:

[24] www.fact-index.com/r/rc/rc4_cipher.html

[25] www.aci.net/kalliste/des.htm

[26] www.hifn.com

[27] www.via.com.tw/en/products/processors/c3