Instruction Set Extensions for Cryptographic Hash Functions on a Microcontroller Architecture

Jeremy H.-F. Constantin and Andreas P. Burg
Telecommunications Circuits Laboratory, STI-IEL, EPFL
Lausanne, VD, 1015 Switzerland
{jeremy.constantin,andreas.burg}@epfl.ch

Frank K. Gürkaynak
Microelectronics Design Center, D-ITET, ETH
Zürich, ZH, 8092 Switzerland
kgf@ee.ethz.ch

***Abstract*—In this paper, we investigate the benefits of instruction set extensions (ISEs) on a 16-bit microcontroller architecture for software implementations of cryptographic hash functions, using the example of the five SHA-3 final round candidates. We identify the general algorithm bottlenecks, taking into account memory footprints and cycle counts of our optimized reference assembly implementations. We show that our target applications benefit from algorithm-specific ISEs based on finite state machines for address generation, lookup table integration, and extension of computational units through microcoded instructions. The gains in throughput, memory consumption, and the area overhead are assessed, by implementing the modified cores and applications utilizing the developed ISEs. Our results show that with less than 10% additional core area, it is possible to increase the execution speed on average by 172% (ranging from 21% to 703%), while reducing memory requirements on average by more than 40%.**

***Keywords*-Instruction Set Extensions, Embedded Systems, Cryptographic Hash Functions, SHA-3**

I. Introduction

While the performance of microprocessors has continued its exponential growth, small and simple microcontroller units (MCUs) have become increasingly popular for embedded applications. MCUs are regularly integrated together with other components to form Systems-on-Chips (SoCs) that combine the flexibility of programmable components with the cost benefits of increased integration densities and performance due to modern manufacturing technology nodes below 100 nm feature size.

Despite the use of advanced process technologies, MCU implementations in embedded systems are still severely constrained by resources such as total code size and required data memory. Furthermore, achieving a given throughput with the least possible complexity (number of instructions) is critical due to constraints on execution speed and energy consumption. An efficient approach to meeting these constraints is the design of application specific instruction set processors (ASIPs). In many applications, the MCU is integrated in a SoC, where it can be customized to the application with instruction set extensions (ISEs). Such extensions typically enhance the datapath of an architecture, thereby mainly improving operation speed for a target application.

An increasingly ubiquitous application are cryptographic algorithms. In particular, cryptographic hash functions have become of great importance, as reflected by the public competition aiming at the selection of a new standard for cryptographic hashing, started in 2007 by the U.S. National Institute of Standards and Technology (NIST) [1]. For the final, third round (2010) of this competition the field has been reduced to five SHA-3 candidates, with a winner algorithm expected to be announced in 2012. Potential applications of the SHA-3 standard range from multi-gigabit data transmission protocols to radio-frequency identification tags, which typically have to operate with severely constrained resources. As a result, the organizers are not only interested in the cryptographic strength of the candidates, but also in the evaluation of the performance of the algorithms. The public SHA-3 evaluation has attracted many contributions on comparing the performance of candidate algorithms on different platforms. Extensive results have been published for hardware [2–4], and software [5,6] implementations of SHA-3 candidates (including exploration and usage of ISEs [7,8]).

Once an algorithm has been selected as the SHA-3 standard, processors and MCUs with specifically tailored ISEs are expected to appear as IP blocks or stand-alone components, similar to the case of AES in modern CPUs. Furthermore, many companies offer solutions [9–11] that facilitate the design of ASIPs and the customization of commonly used MCUs. Starting from a high-level description, such solutions allow changes to be made to an instruction set architecture, and not only produce the corresponding hardware description for integration into a SoC, but also generate the necessary tool chain (compiler, assembler, linker).

Contributions: In this paper, we consider the use of ASIPs, in particular the optimization of MCUs with ISEs, for improving the execution speed and memory footprint of cryptographic hash functions. We show how one can achieve considerable performance gains, with a type of ISEs that is different from the usual approach of common datapath extensions. Highlighting the algorithm-specific performance bottlenecks, we propose ISEs that facilitate the generation of memory access patterns, lookup table integration, and extension of computational units through microcoded instructions.

For the evaluation of our proposal, we consider the five remaining SHA-3 candidates with respect to their suitability for software implementation utilizing ISEs on present and future MCUs and embedded SoCs. We report the execution

speedup, the memory reduction, and the overhead associated with these extensions.

Outline: The paper is organized as follows: In Section II we give an overview of the cryptographic algorithms chosen as the target applications for this work. Section III describes the reference MCU used throughout this evaluation, while Section IV defines the performance metrics used in this study. The design and evaluation flow used to determine the instruction set extensions for each candidate algorithm, and how they are added to the base MCU is explained in Section V. Section VI describes the proposed types of instruction set extensions developed for this work. The results obtained from this study are presented and compared with results from other publications in Section VII and the sources of errors in this work are briefly discussed in Section VIII. Finally, Section IX provides concluding remarks.

II. Application

The target applications for this work are cryptographic hash functions in general. As our ASIP benchmark we chose the five final round SHA-3 candidate algorithms: BLAKE, Grøstl, JH, Keccak, and Skein. For the competition, NIST has stated that the selection for the final round has been done with the factor of diversity in mind, which makes this particular algorithm subset interesting, as it covers various different approaches for the task of cryptographic message digest calculation (hashing).

All algorithms are based on the common principal of combining message data (in fixed block sizes) with internal state data to produce an output hash value of fixed size (224, 256, 384 or 512 bit). Each message block is processed separately using the same core algorithm, transforming the previous internal state into a new state. The internal state data is commonly partitioned into state words of fixed size (8-64 bit) and the state is modified through multiple passes through a set of core functions for each message block.

All SHA-3 candidate hash functions consist of three main elements: state word permutations, state word transformations and lookups in tables of constants. Permutations generally relate to memory moves, transformations are realized using basic arithmetic and logical operations, such as addition, XOR, AND, NOT and state word rotations. The values of the incorporated constants are algorithm specific and depend on the current round number or internal state data (e.g., S-Box lookup table).

The five SHA-3 algorithms used for benchmarking in this work differ in: internal state size (512-1600 bit) and state organization (e.g., 8x8 matrix of bytes, or 8 64-bit state words), message block size (512-1088 bit), and lookup table utilization (e.g., for initial constants only, or state data dependent lookups). Some algorithms like Keccak apply the mentioned full range of arithmetic and logical operations, while algorithms like Grøstl can be described efficiently only using XOR and memory moves, with the help of suitable lookup tables and an underlying architecture that supports byte-mode memory accesses (cf. Subsection III-A).

III. Embedded Microcontroller Architecture

In this work we use our custom implementation of the Microchip PIC24 16-bit architecture [12], as reference implementation. This microcontroller was selected mainly because a functionally verified description was available through an earlier unrelated project.

A. Instruction Set Architecture Summary

The chosen reference MCU is a 16-bit Harvard architecture with a total of 87 base instructions encoded in a 24-bit instruction word [13]. The majority of the instructions allow for a variety of different addressing schemes and operand modes, resulting in 190 different effective instructions. All operands and destinations can be either addressed in byte or word mode, for most instructions. Word mode represents native 16-bit data addressing, while byte mode only operates on the least significant byte of the corresponding data word. This feature is not necessarily supported by all 16-bit MCU architectures, and can be an important factor for the overall performance of an algorithm implementation (cf. Grøstl). The ALU uses a 16-entry general purpose register array, including a stack pointer register. In addition there are several status and control registers, and a 16-bit repeat loop counter used in conjunction with the `REPEAT` instruction. This command repeats the following instruction as often as specified, allowing for very simple hardware loops, thus reducing the program size and cycle count overhead due to branching.

B. Micro Architecture Summary

The micro architecture of our PIC24 implementation is inspired by and hence very similar to that of the original PIC24. It comprises three pipeline stages and executes almost all instructions in a single cycle. However, a slightly more advanced form of data bypassing is implemented in our design. The commercial implementation issues stalls if a data dependency stems from a register, which is used with a register indirect addressing mode in the subsequent instruction. Our implementation employs full data bypassing which leads to a cycle count reduction of 10-30% for an average application. The memories for program and data are external from the core, each accessible with a latency of one clock cycle. As in the commercial device, the data memory is realized as a dual-port memory, which enables a read- and a write-access in the same clock cycle.

IV. Performance Metrics

Throughout this paper, we report four main performance metrics when presenting our results. In this section we describe these metrics, and explain how they are calculated. Please note that for all algorithms, we only consider the respective versions proposed as replacements for SHA-256.

A. Cycle Count

The cycle count is a measure for the complexity and also for the energy consumption of an implementation on a specific processor. It is also inversely proportional to the throughput

which describes how fast the hash algorithm works for a given clock frequency. Every hash algorithm has a defined *message block length*. For all SHA-3 candidates this is 512 bits, except for Keccak which uses 1088-bit message blocks. In this paper, we report the long message performance (i.e. no finalization round) in the commonly used cycles/byte format.

B. Data Memory

The microcontroller used in this work uses 16-bit wide data memory, realized as a static random access memory (SRAM). In a microcontroller, the total amount of SRAM available for all applications is a scarce resource due to the fact that SRAMs occupy significant circuit area for practical memory sizes. The data memory utilization is always expressed in number of bytes throughout this paper.

C. Program Memory

The PIC24 microcontroller is based on the Harvard architecture with separate data and program memories. This allows each memory to have a different bit-width. The PIC24 microcontroller uses a 24-bit wide program memory.

In practice, the program memory could be implemented as a read-only memory or a one-time programmable memory, both of which have less hardware overhead than an SRAM as used for the data memory. While still a significant burden, program memory is therefore often not as expensive as data memory.

All PIC24 instructions are either one or two instruction words long, and can hence be stored as three or six bytes respectively. The overall comparison tables always list the total number of bytes. We use the word *text* to describe the complete content of program memory. Static initialization data (e.g. a set of hash chain init values) is normally used once per execution of an algorithm. It can therefore in practice be stored in sections linked to the less expensive program memory, and is hence in addition to program code also accounted for as text.

D. Area

A hardware implementation always involves a compromise between operation speed, power and energy used, and the circuit area. In this study, all microcontroller descriptions are synthesized using a standard-cell-based design flow for a 90 nm CMOS technology. Furthermore, all MCU instances reported in this study are extensively analyzed regarding area requirements for the full range of different core clock constraints covering all target constraint corners (cf. Section VII-B). Area overhead of the proposed ISEs with respect to the original implementation or any other absolute area figures given in comparison tables and the continuous text are always for a chosen reference core clock speed of 200 MHz.

Area is expressed in terms of kilo Gate Equivalents (kGEs), where one GE is taken as the area of a 2-input NAND gate with a standard driving strength. The conversion factor is $3.136\,\mu m^2$ per GE. The results are all synthesis results, and do not include post-layout parasitic effects.

The total area of the microcontroller subsystem comprises of the data memory, program memory, and the actual MCU core. In a typical implementation, the two memories would be implemented as large SRAM macros. Even if very modest sizes were to be selected for these memories, the SRAMs would occupy at least two thirds of the total area. Since the memory overhead is large and determining a fair size of memory is not straightforward, we report only the change in the core area in this comparison.

V. Design Flow

The design flow used in our implementation and evaluation process is illustrated in Fig. 1. The flow comprises a hardware implementation path and a software development, optimization and verification path.

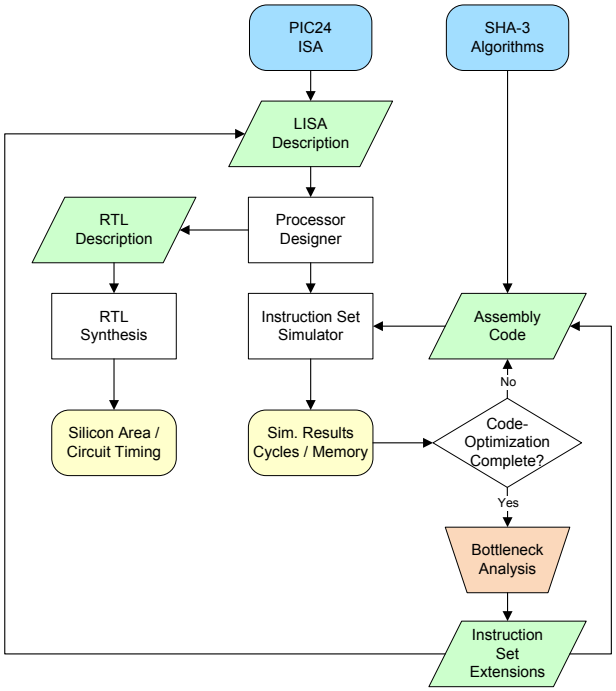

Fig. 1. Illustration of the design flow and tools employed for implementation, benchmarking, and development of instruction set extensions.

On the processor architecture side, the instruction set architecture (ISA) and the microarchitecture of the PIC24 processor are first described in LISA (Language for Instruction Set Architectures), a language tailored to the design of application specific processors [14,15]. This description is then automatically translated into a register transfer-level (RTL) VHDL description that can be synthesized into gates using RTL synthesis tools (Synopsys DC Ultra) to evaluate silicon area and performance. The design is mapped to a standard-performance regular threshold-voltage 90 nm CMOS technology, with nominal supply voltage of 1.0 V. In addition to the hardware description, Processor Designer [9] generates the basic software tool chain (assembler and linker) and a fully cycle-accurate instruction set simulator (ISS).

On the software side, the algorithm specifications for the five SHA-3 candidates provide the starting point for an initial benchmark implementation in the native assembly language of

the PIC24 ISA. Multiple iterations of profiling and software-optimization are carried out, with the help of profiling tools provided by the ISS, to reduce code and memory size and to reduce cycle counts. All implementations are always verified against the provided test patterns to guarantee full compliance with the original algorithm specifications.

At the point, where no further gains are achieved using the standard ISA of the PIC24 microcontroller, the different algorithms are analyzed manually, to identify specific performance and memory bottlenecks. For each candidate, we identify custom instructions that promise an improvement in terms of memory footprint and/or cycle count, while being compatible with the general architecture of the core with only minor modifications. To be more precise, the extended implementations remain fully backward compatible and no changes are made to key components such as the register file, the memory interfaces, the pipeline stages, or the instruction formats. The additional instructions are incorporated into the LISA model and into the assembly descriptions of the SHA-3 kernels. ISEs are fine tuned through multiple iterations of hardware/architecture and code adjustments followed by benchmarking of gains in terms memory utilization and cycle count.

VI. Instruction Set Extensions

Although the reference PIC24 ISA comprises a large number of instructions, our implementations without ISEs mainly focus on a core set of instructions, directly matching the base operations intrinsic to the class of cryptographic hash functions, as described in Section II. During the optimization phase, improvements are mainly achieved by efficient use of all available operand addressing modes provided by the architecture, and applying coding techniques such as unrolling and precomputation, where feasible (execution time versus program and data memory trade-offs).

We investigate three different forms of performance bottlenecks that can not be removed using the reference ISA. First, we analyze conventional basic operations, which however have to be carried out on data word lengths that are wider than the ones supported by the underlying native architecture (e.g. 64-bit data words on a 16-bit MCU). Depending on the operation, these often have to be emulated by a whole set of assembly instructions. A very relevant example in the context of cryptographic hash functions is the rotation of state data words. Furthermore, we examine complex memory access patterns, which produce overhead in form of advanced address arithmetic and potential multiple accesses of the same data. Last, we investigate static data memory requirements in the form of lookup tables, occupying valuable data memory space, and further increasing the number of required memory accesses. Derived from these three types of performance bottlenecks, the following subsections describe the corresponding three classes of proposed ISEs for cryptographic hash functions. Moreover, algorithm-specific examples are given, as well as our assessment of individual applicability regarding the set of five investigated SHA-3 algorithms.

A. Extension of Computational Units through Microcoded Multi-Cycle Instructions

The most common way of extending an ISA is by introducing new computational units to the datapath of the processor. The new instruction hence performs a new computational task, e.g. the calculation of a multiply-accumulate operation. In the case of cryptographic hash functions the main computational bottleneck on an MCU lies in the support for large data word rotations. These rotations in general need to be performed on 64-bit words for all algorithms which use this operation, namely BLAKE, Keccak and Skein.

The problem of a 64-bit data word rotation on a 16-bit architecture can be solved in the form of a microcoded multi-cycle instruction, with minimal additions to the processor datapath. Since the 16-bit architecture supports at most a double register writeback, the 64-bit word rotation instruction needs two cycles to complete. In each cycle, only three specific values are rotated to create two resulting MCU native data words. Thus, the required hardware only amounts to two 16-bit barrel shifters, and not a full 64-bit barrel shifter. The emulation of a generic 64-bit rotate by an arbitrary amount of bits utilizing the PIC24 ISA requires 12 cycles, which results in a speedup of all rotation operations by a factor of 6.

Even more complex arithmetic operations such as a matrix multiplication in the field of F_{256} on a 8x8 matrix of 8-bit state words (as required by Grøstl), can be supported by the help of small extensions to already existing computational units. Adding a few simple shift units and XOR units on small 8-bit input values can provide a considerable speedup factor of 14.4 for this operation, when combined with the right microcoded multi-cycle operand and memory addressing support.

B. Finite State Machines for Data Address Generation

As described in Section II, cryptographic hash functions are heavily based on permutations. These permutations of state words generally follow some fixed permutation tables, which are intrinsic to the algorithm definition. Effectively, this means that state data needs to be loaded and stored in regular patterns, but often not in the sequential fashion in which the state words are mapped in memory. The existence of multiple of these access patterns applied to the same state, often renders general relabeling or physical reordering of state words in the data memory impossible. The assembly implementations therefore have to generate these memory address patterns by the use of additional instructions. Moreover, there is often a trade-off between which data can be loaded efficiently without much overhead at the current position in the algorithm, and which data is actually needed for immediate calculations and can be retained temporarily in the general purpose register file of limited size.

ISEs that enable the efficient generation of memory accesses based on a loop counter (e.g., the PIC24 `REPEAT` counter), can significantly reduce this type of performance bottleneck, especially on resource constrained systems without considerable cache hierarchies. The ISEs essentially map part of the loop-based program flow in the algorithm core onto a

microcoded instruction. Based on a small finite state machine (FSM), the instruction generates for each state (repetition count) the required data memory address offsets, register indices, and operand data. We note that this FSM based method is particularly applicable, if the computational part of the repeated instruction is simple, and performs a clear transformation of the state data without much computational effort.

The proposed replacement of program code by an ISE has two effects in general: execution speedup through memory access optimization, and considerable code size reduction, depending on the repetition count of the new instruction and its comprising computational complexity.

The Grøstl algorithm with its very modular structure of its operations, and their rather low complexity, can especially benefit from this method. All four of the hash function's core operations can be replaced by one single instruction each, utilizing the described ISE type. The BLAKE and JH ISEs also apply this approach for their custom instructions, although to a lesser extent.

C. Lookup Table Integration

Due to the large area occupied by on-chip memory, reduction of data memory consumption is often as important as the speedup of algorithm execution. A prime candidate for significant reduction of data memory is the removal of static lookup tables (LUTs). Almost all of the evaluated algorithms use some form of LUT or constant table of various size, with the exception of Skein, which only requires initial state data (stored as text in the program memory, cf. IV-C).

Removal of constant table data from memory is possible through integration of these LUTs into the processor core. There are two types of LUTs that can be found: tables holding round or data dependent constants, and LUTs providing (often considerable) execution speedup through precomputation. The required indices for these LUTs can come from different sources, such as the current round number or state data.

This class of ISEs has been utilized for all four applicable algorithms, moving every LUT from data memory into the MCU core. Integration of LUTs up to the size of 1344 bytes in the case of JH (for precomputed round constants enabling an efficient bit-sliced implementation of the algorithm), only resulted in negligible core area overhead of around 2 kGE, compared to an SRAM implementation of 14 kGE for an equal data size. In the case of Grøstl, which utilizes an S-Box LUT as it is used in AES, the integration of two parallel 8-bit LUTs in hardware additionally facilitates speedup of the state substitution by a factor of 4.2.

D. Implementation Overview

The implementations of the ISEs map the three proposed concepts onto specific instructions, for each algorithm individually. Table I gives an overview of the three different ISE types and specifies for each candidate which algorithm parts are addressed by which category. Speedup of an ISE is reported relative to the corresponding realization of that particular function using the standard PIC24 ISA.

TABLE I
OVERVIEW OF PROPOSED INSTRUCTIONS AND THEIR INDIVIDUAL PERFORMANCE AND MEMORY GAINS, GROUPED BY ISE-TYPE.

	Extension of Computational Units (VI-A)	
	Function realized by ISE	Speedup
BLAKE	32-bit rotation	6.0
Grøstl	*MixBytes* step	14.4
JH	bit-sliced *S-Box* + *L/MDS*	2.0
Keccak	64-bit rotation (single)	2.5
	64-bit rotation (+mem.permut.)	4.1
Skein	64-bit rotation (generic)	6.0
	FSMs for Address Generation (VI-B)	
	Function realized by ISE	Speedup
BLAKE	constant XOR message	4.0
Grøstl	*AddRoundConstant* step (P/Q)	3.2/1.5
	ShiftBytes step	3.4
	MixBytes step	14.4
JH	*Swap* permutation	5.2
	Lookup Table Integration (VI-C)	
	Function realized by ISE	Mem. [byte]
BLAKE	initialization constants	64
	Sigma permutations	224
Grøstl	*S-Box*	256
	F_{256} Multiplication LUTs	512
JH	bit-sliced round constants	1344
Keccak	round constants	96

For an in-detail description and discussion of all proposed instructions on a software/ISA level please refer to [16].

VII. IMPLEMENTATION RESULTS

Our results are structured into two main parts: covering the aspects regarding software performance utilizing ISEs, and assessing the exact hardware performance of the proposed extended MCU cores.

A. Software Performance

1) Instruction Set Extensions: By developing ISEs for the PIC24 ISA we are able to provide a speedup of about 1.4 for our benchmark algorithms in the general case (for the slowest 4 out of 5 candidates), with the exception of Grøstl for which we achieve a speedup of 8. Table II shows these results in detail and compares the area overhead that the ISEs incur to the reference microcontroller. For an in-depth evaluation of the area and timing requirements of the implemented cores refer to Section VII-B.

It can be observed that for 3 out of 5 candidates, the ISEs did not result in any noticeable core area overhead, while still providing significant improvements, in both cycle-count (reduction on average around 30%) and memory requirements (Table IV). This is mainly due to the factor that some ISEs

TABLE II
IMPROVEMENT OF HASHING SPEED (LONG MESSAGE) AND CORE AREA BY USING ISES FOR ALL SHA-3 CANDIDATES

	Cycles/Byte			Core Area [kGE]	
	PIC24	+ISE	Speedup	+ISE	Overhead[a]
BLAKE	155.2	102.9	**1.51**	22.77	**-0.5**%
Grøstl	462.3	57.6	**8.03**	24.97	**+9.1**%
JH	463.8	383.5	**1.21**	25.20	**+10.1**%
Keccak	188.3	131.7	**1.43**	22.22	**-2.9**%
Skein	157.6	112.6	**1.40**	23.00	**+0.5**%

[a]Reference Core Area: 22.88 kGE at 200 MHz

did not actually add datapath components, but provided small FSMs, that are able to resolve complex but regular addressing schemes to fetch operands for already present datapath components (XOR, AND, ADD). The smallest speed up that is obtained through the ISEs is for JH (about 20%), whereas the largest improvement with a speedup factor of over 8 is obtained for Grøstl. Interestingly, these two implementations both resulted in about 10% area overhead, which is still negligible.

One important parameter when determining how much faster an algorithm can be implemented is the number of memory accesses (read/write) required for the algorithm. Since the memory bandwidth for the given architecture will not change with additional instructions, the program will always be limited by these numbers. Part of the speedup is achieved by eliminating memory accesses, mostly by embedding operational constants into the ISE, and optimization of the load- and store-patterns for state words. In Table III we list the number of read and write memory accesses for all candidate algorithms. It can be seen that only for Grøstl a significant improvement could be made. This also explains the relatively high performance gain for this algorithm.

TABLE III
CHANGE IN THE NUMBER OF MEMORY ACCESSES DURING THE PROCESSING OF ONE MESSAGE BLOCK FOR ALL SHA-3 CANDIDATES.

	Read			Write		
	PIC24	+ISE	Change	PIC24	+ISE	Change
BLAKE	2,370	1,682	**-29%**	1,187	1,187	**0%**
Grøstl	16,566	3,126	**-81%**	13,271	2,391	**-82%**
JH	11,836	9,874	**-17%**	4,141	4,099	**-1%**
Keccak	14,660	14,779	**0%**	7,345	7,416	**+1%**
Skein	5,264	5,264	**0%**	3,289	3,289	**0%**

From a system designers point of view, more often than not the amount of memory used by an algorithm is even more important than its outright execution speed. We have listed the total data and program (text) memory used by all candidate algorithms separately for both the standard and the enhanced instruction set implementation of PIC24 in Table IV. It can be seen that the largest combined improvement is achieved for JH (61.5% total reduction) and Grøstl (71.3% total reduction). Whereas the smallest improvement is achieved for Skein (17.7% total reduction).

Text is generally saved through the compaction of more complex algorithm parts into single instructions, especially by mapping address generation patterns onto FSM based instructions. In the case of frequent use of rotations, their single instruction representation additional reduces the program code size. The data memory footprint is reduced by integrating round constants and other lookup tables into the processor core.

TABLE IV
REDUCTION OF DATA AND INSTRUCTION MEMORY BY USING INSTRUCTION SET EXTENSIONS FOR ALL SHA-3 CANDIDATES.

	Data [byte]			Text [byte]		
	PIC24	+ISE	Reduction	PIC24	+ISE	Reduction
BLAKE	488	200	**-59.0%**	1,028	818	**-20.4%**
Grøstl	982	214	**-78.2%**	2,619	819	**-68.7%**
JH	1,550	206	**-86.7%**	4,649	2,183	**-53.0%**
Keccak	448	352	**-21.4%**	3,480	2,415	**-30.6%**
Skein	242	242	**0.0%**	5,734	4,678	**-18.4%**

2) Reference Implementation: We give a short comparison of the software performance of our reference assembly implementations with other published works on MCU based systems. Since our reference implementations are the basis for the developed ISEs and the reported relative performance gains, we list our implementation results together with other published results in Table V, to show that our standard PIC24 ISA implementations are reasonable in comparison with the current state of the art.

The relative performance of the algorithms is given in relation to BLAKE to allow for easy comparison of the individual performance differences between the various MCU platforms. It can be observed that in general 32-bit MCU architectures clearly benefit in terms of cycle count over a 16-bit architecture, since most algorithms internally use state word sizes that are larger (64-bit) than the native word sizes of the MCUs.

The most important study of SHA-3 algorithm performance on MCU based systems has been done by Christian Wenzel-Benner and Jens Gräf [18]. They maintain a web-page [17], where their results are published. These results have also been included in the eBASH website [5]. Thomas Pornin [6] has developed a library (sphlib) which uses standard C, and therefore could easily be ported to a variety of platforms. The library includes comparisons on several platforms, but does not necessarily reflect the results that are achievable with hand-crafted assembly kernels.

B. Hardware Performance

A general overview of the area requirements for the implemented cores is given in Table II. The digital design flow is known to produce results that vary as much as ±5%, so

TABLE V
COMPARISON OF THROUGHPUT NUMBERS [CYCLES/BYTE] OF PUBLISHED MICROCONTROLLER SHA-3 IMPLEMENTATIONS. NUMBERS IN PARENTHESES SHOW PERFORMANCE NORMALIZED TO BLAKE PERFORMANCE.

	This work		[6]	[6]	[17]	[17]
Architecture	PIC24	PIC24+ISE	ARM-M3	ARM920T	ARMv5TE	ATmega128
Datapath	16-bit	16-bit	32-bit	32-bit	32-bit	8-bit
BLAKE	155 (1.00)	103 (1.00)	89 (1.00)	54 (1.00)	87 (1.00)	1241 (1.00)
Grøstl	462 (2.98)	58 (0.56)	455 (5.11)	313 (5.79)	216 (2.48)	11198 (9.02)
JH	464 (2.99)	384 (3.72)	370 (4.16)	395 (7.31)	361 (4.15)	3829 (3.06)
Keccak	188 (1.21)	132 (1.28)	192 (2.16)	197 (3.65)	-	1115 (0.89)
Skein	158 (1.02)	113 (1.10)	128 (1.44)	129 (2.39)	184 (2.11)	1444 (1.16)

changes smaller than 5% can not reliably be presented. In fact for 2 out of 5 algorithms, the core area for the processor actually decreases slightly when ISEs are added.

It must be noted that in the best case, the core area represents at most one third of the total area of the overall MCU subsystem. The other two thirds are used by data and instruction memory. For example, the generated SRAM macro blocks used for evaluation and complete synthesis of the designs in this work, are of the sizes 2048 instructions (6 Kbyte) and 1024 data words (2 Kbyte), together occupying an area of 50.28 kGE. As such, the net-overhead due to the ISEs is even smaller. Factoring in the possible system memory size reductions due to improved code size and usage of working data, the total area would potentially even decrease.

For assessment of the design corners, emphasizing optimization for high speed or low area, the reference core as well as all five modified versions, each including their algorithm-specific ISEs, are implemented and synthesized for various timing constraints. As shown in Fig. 2, we perform an area-delay trade-off evaluation for all six designs, by synthesizing each architecture multiple times, while sweeping the core clock constraint.

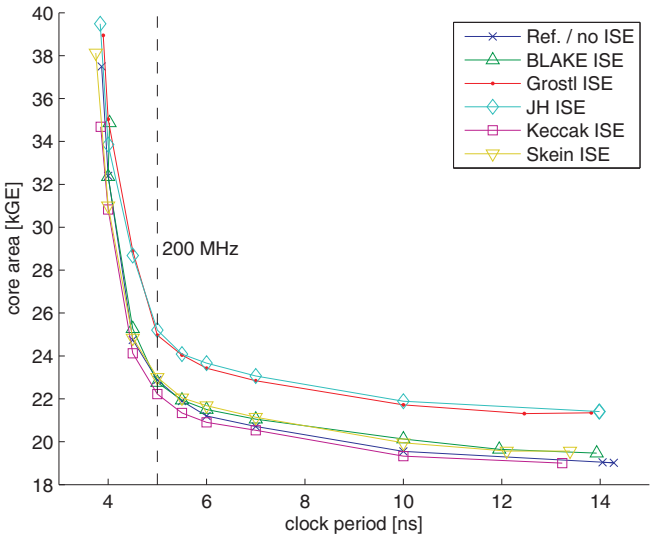

Fig. 2. Maximum core clock period versus core area for the reference design and five cores with individual instruction set extensions each.

Optimization for area shows a size of 19.0 kGE for the reference core as well as the core with Keccak ISEs. The cores with BLAKE ISEs and Skein ISEs each occupy a minimum area of 19.5 kGE, while Grøstl ISEs and JH ISEs each produce extended minimal core sizes of 21.4 kGE, due to integration of relatively large LUTs in the processor datapath. All designs optimized for area can still operate at a core clock frequency of about 70 MHz.

Hard constrained synthesis, i.e. optimization for speed, shows a similar picture of the designs performing on a comparable level regarding maximum clock frequency and required core area. Maximum frequencies are around 260 MHz, with the exceptions of the two cores with BLAKE ISEs (248 MHz) and Keccak ISEs (267 MHz). The occupied area for the high speed designs varies between 34.7 kGE (Keccak) and 39.5 kGE (JH), around the reference core without ISEs (37.5 kGE). These figures can mainly be explained by the mentioned optimization-variations during synthesis, which can especially be observed for the core with Keccak ISEs, which is identical to the reference core plus three additional instructions.

As a result, it can be seen that the area overhead produced by the introduced ISEs is in general negligible, independent of the clock constraint.

To give the synthesizer some room for exploiting various techniques to trade-off speed versus area, the reference clock period for general overhead comparison is chosen at 5 ns, about 1 ns above the achievable minimum, hence allowing for all six designs to meet the same timing constraint. The difference in core area can this way be attributed to the overhead due to the ISEs only.

VIII. SOURCES OF ERROR

Although we have tried to minimize the sources of errors, there are several factors that may have influenced the results, especially regarding the relative performance comparisons of the five SHA-3 candidates, and their suitability for ISEs.

A. Choice of the MCU

All our results are given on a particular MCU. It is fair to say that, this MCU is not the most widely used embedded processor. However we believe that it is not a bad choice as a generalized 16-bit MCU, since it has a simple Harvard architecture, and a relatively large number of single cycle

instructions which makes it easy to implement. We do not think that the evaluation of our presented ISE concepts would differ significantly if another MCU were to be used, however we can not exclude this possibility.

B. Designer Experience

During this work, several critical parts have been performed manually. The design flow used in this project required all candidate algorithms to be manually implemented in assembly language. In the later stages, potential ISEs have been determined by manually analyzing the first implementations. And finally, the optimized implementations utilizing the ISEs have been coded manually as well. Even though we believe that we have tried our best to implement all these steps equally well, it is possible that some optimization possibilities were overlooked in the process.

C. Reporting Overhead

Our experience is that synthesis results have an accuracy of roughly $\pm 5\%$ depending on many factors. All presented numbers are generated using only front-end design data and do not accurately reflect the parasitic effects from placement and routing. However, we have significant experience with back-end design, and do not believe that the relative performances would be much affected during the post-layout phase.

IX. Conclusion

It is well known that instruction set extensions (ISEs) can increase the performance of an application on a given microprocessor architecture, through addition of datapath units. Considering algorithm-specific performance bottlenecks regarding memory consumption and execution time on resource constrained systems, we proposed dedicated ISEs based on lookup table integration and microcoded instructions using finite state machines for operand and address generation, which do not significantly add to the processor datapath.

Looking at the application-specific case of cryptographic hash functions on a 16-bit MCU for SoC, we assessed the potentials of these ISEs for all five SHA-3 final round candidates, representing a variety of different hash algorithms. We show that our proposed ISEs can provide significant speedup, hence enabling throughput increase and reduced energy consumption, in addition to substantial memory footprint reduction.

It was shown that most improvements were made by instructions that provided efficient generation of memory address patterns. Rather than adding datapath units, to calculate complete functions, especially beneficial were instructions that handled complex (but regular) memory accesses as a result of constant permutation templates. Furthermore, moving constant lookup tables from data memory into the processor's datapath turned out to be highly beneficial in reducing memory footprints at negligible core-area overhead. Minor extensions of existing computational units, combined with state driven microcoded instruction execution, to support rotation operations on larger bit-widths (64-bit) within the limits of the native 16-bit architecture, proved to be especially helpful for significant speedup of most of the SHA-3 algorithms.

In three out of five cases, the ISEs had no measurable impact on the core area of the microcontroller, in the two other cases, the overhead was limited to 10%, whereas the execution speed improved by 172% on average over five candidates. In particular, we were able to improve the execution speed of Grøstl by more than a factor of 8, and reduce its memory consumption by more than 70%. This has moved Grøstl from being one of the slowest implementations (about 3 times slower) to the fastest implementation by some margin (1.75 times faster than the next algorithm).

Acknowledgment

The authors would like to thank Prof. H. Meyr (specifically M. Witte and F. Borlenghi) for the inspiring discussions and their support for the work with Processor Designer which was instrumental for getting started on this project.

References

[1] NIST, "Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family," Federal Register, Vol.72, No.212, 2007, http://www.nist.gov/hash-competition.

[2] The ECRYPTII Group, "The SHA-3 zoo," http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[3] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa: Automated Tool for Hardware EvaluatioN," 2011, http://cryptography.gmu.edu/athena/.

[4] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. Gürkaynak, "Developing a hardware evaluation method for SHA-3 candidates," in *Cryptographic Hardware and Embedded Systems, CHES*, vol. 6225, 2010, pp. 248–263.

[5] D. J. Bernstein and T. Lange (editors), "eBACS: ECRYPT Benchmarking of Cryptographic Systems," http://bench.cr.yp.to.

[6] T. Pornin, "sphlib, Update for the SHA-3 third round candidates," http://www.saphir2.com/sphlib.

[7] R. Benadjila, O. Billet, S. Gueron, and M. Robshaw, "The Intel AES instructions set and the SHA-3 candidates," in *Advances in Cryptology, ASIACRYPT*, 2009, vol. 5912, pp. 162–178.

[8] P. Grabher, J. Großschädl, S. Hoerder, K. Järvinen, D. Page, S. Tillich, and M. Wójcik, "An exploration of mechanisms for dynamic cryptographic instruction set extension," in *Cryptographic Hardware and Embedded Systems, CHES*, 2011, vol. 6917, pp. 1–16.

[9] Synopsys, "Automating the design and implementation of custom processors (Processor Designer, LISA 2.0)," http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx.

[10] Tensilica, "Xtensa customizable processors," http://www.tensilica.com/products/xtensa-customizable.htm.

[11] Target Compiler Technologies, "IP designer," http://www.retarget.com/products/ipdesigner.php.

[12] Microchip, "PIC24HJXXXGPX06/X08/X10 Data Sheet," http://ww1.microchip.com/downloads/en/DeviceDoc/70175H.pdf.

[13] ——, "16-bit MCU and DSC programmer's reference manual," http://ww1.microchip.com/downloads/en/DeviceDoc/70157D.pdf.

[14] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," in *Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.20, no.11*. IEEE, 2001, pp. 1338–1354.

[15] O. Schliebusch, R. Leupers, and H. Meyr, *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007.

[16] J. Constantin, A. Burg, and F. K. 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, http://eprint.iacr.org/2012/050.

[17] C. Wenzel-Benner and J. Gräf, "XBX: eXternal Benchmarking eXtension," http://xbx.das-labor.org/trac.

[18] ——, "XBX: eXternal Benchmarking eXtension for the SUPERCOP Crypto Benchmarking Framework," in *Cryptographic Hardware and Embedded Systems - CHES*, vol. 6225, 2010, pp. 294–305.