Investigating the Potential of Custom Instruction Set Extensions for SHA-3 Candidates on a 16-bit Microcontroller Architecture

Jeremy Constantin[1], Andreas Burg[1], and Frank K. Gürkaynak[2]

[1] Telecommunications Circuits Laboratory, EPFL, Switzerland
{jeremy.constantin,andreas.burg}@epfl.ch
[2] Microelectronics Designs Center, ETH Zurich, Switzerland
kgf@ee.ethz.ch

Abstract. In this paper, we investigate the benefit of instruction set extensions for software implementations of all five SHA-3 candidates. To this end, we start from optimized assembly code for a common 16-bit microcontroller instruction set architecture. By themselves, these implementations provide reference for complexity of the algorithms on 16-bit architectures, commonly used in embedded systems. For each algorithm, we then propose suitable instruction set extensions and implement the modified processor core. We assess the gains in throughput, memory consumption, and the area overhead. Our results show that with less than 10% additional area, it is possible to increase the execution speed on average by almost 40%, while reducing memory requirements on average by more than 40%. In particular, the Grøstl algorithm, which was one of the slowest algorithms in previous reference implementations, ends up being the fastest implementation by some margin, once minor (but dedicated) instruction set extensions are taken into account.

Key words: SHA-3 Final Round Candidates, Software Implementation, Assembler, 16-bit Microcontroller, Instruction Set Extensions, ISA Exploration

1 Introduction

In 2007, the U.S. National Institute of Standards and Technology (NIST) started a public competition aiming at the selection of a new standard for cryptographic hashing [13]. The cryptographic community was asked to propose new hash functions and to evaluate the security level of other candidates. In 2008, 51 functions were accepted to the first round, the second round (2009) reduced this number to 14, and for the final, third round (2010) the field has been further reduced to five candidates: BLAKE, Grøstl, JH, Keccak, and Skein. Following the third SHA-3 candidate conference, a winner algorithm is expected to be announced in 2012.

Potential applications of SHA-3 standard range from multi-gigabit data transmission protocols with high performance requirements to radio-frequency identification (RFID) 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 algorithm implemented on different platforms.

Following the success of the public AES selection process, the SHA-3 evaluation has attracted many contributions comparing the performance of candidate algorithms on different platforms. Extensive results have been published for software [2] [14], and hardware [8] [5] [9] implementations of SHA-3 candidates.

In recent years the performance of microprocessors has continued its exponential growth. Modern microprocessors have become increasingly complex structures, consisting of several levels of on-chip cache memory and in most cases multiple parallel cores and execution units. Some recent CPUs even include instructions to execute AES round operations on 128-bit wide registers. At the same time, small and simple microcontroller units (MCU) have become popular for embedded applications. MCUs are regularly integrated together with other components to form Systems-on-Chips (SoC) that combine the flexibility of programmable components with the cost benefits of increased integration densities and the performance due to modern manufacturing technology nodes below 100nm 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. A study by Wenzel-Benner and Gräf has compared the performance of SHA-3 candidates on several MCU platforms [18]. Thomas Pornin [14] has also published a library (sphlib) containing implementations of SHA-3 candidates geared towards MCUs with constrained resources. Unfortunately, many of the available resources provide only little reference on the performance of carefully hand-crafted assembly implementations on 16-bit MCU, which are most frequently used in many embedded applications.

An additional important point about embedded MCUs is that, in many applications, the MCU would be integrated in a custom SoC, where it can be customized with instruction set extensions to the application. The basic idea behind such extensions is to enhance the datapath of an MCU so that an application can be executed with fewer number of instructions increasing operation speed, reducing energy consumption, and saving resources (RAM and ROM). Such new additions could add some overhead (increased circuit size, or reduced maximum operating speed due to additions to the datapath), but in many cases could still offer significant advantages. Most IP vendors already offer a wide variety of options for their embedded MCUs. Once an algorithm has been selected as a standard, processors and MCUs with specifically tailored instruction set extensions 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 [16] that facilitate design of application specific processors and the customization of commonly used MCUs. Starting from a high-level description, such solutions allow changes to be made to a processor architecture, and not only produce the corresponding hardware description for integration into a SoC, but also to generate the tool chain (compiler, assembler, linker) that will support this enhanced processor, allowing it to be used with relative ease.

Contributions In this paper, we compare the five remaining SHA-3 candidates with respect to their suitability for software implementation on present and future MCUs and embedded SoCs. To this end, we start from a small, general purpose 16-bit MCU, based on the PIC24 series of microcontrollers, and evaluate the performance of all five SHA-3 candidates. To match the requirements of resource constrained embedded systems, we consider only hand-crafted assembly implementations of the corresponding kernels, which are expected to become available as library components for the different MCUs once a winner has been selected. Then, for each candidate we examine this reference implementation and identify promising instruction set extensions. We report the performance gain, the memory reduction, and the overhead associated with these extensions. In this way, we aim to uncover if some of the SHA-3 candidates could achieve a better performance when instruction set extensions are used.

Outline The paper is organized as follows: In section 2 we describe the reference MCU used throughout this evaluation. Section 3 defines the performance metrics used in this study. The design and evaluation flow used to determine the instruction set extension for each candidate algorithm, how they were added to the base MCU is explained in section 4. The following section 5 contains details of the implementation for each candidate algorithm. The combined results obtained from this study are presented and compared with results from other publications in section 6 and the sources of errors in this work are briefly described in section 7. Finally section 8 provides concluding remarks. Appendix A contains the tables with the detailed implementation results of the five individual candidates.

2 The Embedded Microcontroller, PIC24

In this work we use our custom implementation of the Microchip PIC24 16-bit architecture [12], as reference implementation. This microcontroller was selected because of its common features for 16-bit MCU architectures, and due to the fact that a functionally verified description was available through an earlier unrelated project.

2.1 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 [11]. 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 named W0-W15, with a stack pointer assigned to register W15. 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.

2.2 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, on the contrary, employs full data bypassing which leads to a cycle count reduction of 10-30% for an average application. 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.

3 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 were calculated.

3.1 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. The input message, which, according to SHA-3 specifications, can be arbitrary long up to $2^{64}-1$ bits, is first *padded* to a multiple of the message block length, and then the hash algorithm processes each message block sequentially. As soon as the last message block has been processed an *output digest* is produced. For SHA-3 the output digest size can be 224, 256, 384, or 512 bits. Some SHA-3 candidate algorithms use an additional *finalization step* when processing the final message block, increasing the run time for the final block slightly.

In this paper, we measure the number of clock cycles used for processing one message block on our microcontroller architecture. However, only four SHA-3 candidates use 512-bit message blocks while Keccak uses 1088-bit message blocks. To normalize for this difference we report complexity in cycles/byte.

3.2 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.

3.3 Program Memory

The PIC24 microcontroller is based on the Harvard architecture that is common for small microcontrollers. In this architecture, data and program memory are separated. 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 (ROM) or a one-time programmable (OTP) 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 therefore be stored as three or six bytes respectively. The overall comparison tables always list the total number of bytes. Some tables use *instructions* when reporting the complexity of individual functions for clarity. The word *Text* is frequently used to distinguish program in the memory from data. We continue this tradition and refer to the program memory in tables as *Text* as well. Static initialization data (e.g. 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 also accounted for as *Text*.

3.4 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 using a 90 nm CMOS technology (see section 4 for details). We have kept the operation speed for all of the implementations the same. All microcontroller instances reported in this study have been optimized for 200 MHz clock speed. At this constant operation speed, the change in circuit area is reported as the overhead of the suggested instruction set extensions with respect to the original implementation without extensions.

Area is expressed in terms of kilo Gate Equivalents (kGEs), in order to get technology independent area numbers. For one GE we have taken the area of a 2-input NAND gate with a standard driving strength (1x). The results are all synthesis results, and do not include post-layout parasitic effects.

The total area of the microcontroller comprises of the data memory, program memory, and the actual microcontroller 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 (i.e., 6 Kbyte for instruction memory, 2 Kbyte for data memory), these memories would occupy at least two thirds of the total area. Since the memory overhead is so large and finding a fair size of the memory is not straightforward, we report only the change in the core area.

4 Design Flow

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

On the hardware side, the instruction set architecture (ISA) and the microarchitecture of the PIC24 processor are first described in LISA, a language tailored to the design of application specific processors [10, 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 to evaluate silicon area and performance. In addition to the hardware description, Processor Designer [16] 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 implementation in the native assembly language of the PIC24 ISA. Multiple iterations of profiling and software-optimization are carried out 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 code of the different algorithms is analyzed manually, with the use of profiling tools provided by the ISS, to identify 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[3]. These instructions are incorporated into the LISA model and in the assembly descriptions of the SHA-3 kernels. Instruction set extensions are again fine tuned through multiple iterations of hardware/architecture and code adjustments followed by benchmarking of gains in terms memory utilization and cycle count.

5 Implementation

In this section, we consider the implementation of the five SHA-3 final round candidates individually in alphabetical order. For each candidate, we first provide a brief introduction of the algorithm. Then we describe our implementation on the reference 16-bit PIC24 architecture and comment on the corresponding implementation issues and bottlenecks. Next, we introduce our instruction set extensions and explain how they are used to improve the respective algorithm implementation.

To settle on a common characteristic regarding internal state sizes and to limit the amount of code to develop and benchmark, only the SHA-3 candidate versions proposed as the replacements for SHA-256 were implemented. These versions were chosen, since they are most suited for use in the context of embedded systems.

Every implementation is fully functional as a standalone application, performing two invocations of the algorithm code (one for single block, one for multi block), followed by an automatic check of the message digest, to verify the correctness of the implementation. All implementations include code written for initialization and preparation of message blocks, however these have not been included for performance evaluation, i.e., only long message performance values are given in this paper. This choice, to focus only on the algorithm kernels was made, since the code for message block preparation (e.g., padding) is similar for all candidates, and only executed once per block, making it the least relevant code regarding any possible optimizations.

As a general remark regarding the implementation of the instruction set extensions for all candidates: If an instruction relies on the information during which round it is executed (e.g., to adjust which constant to choose from a table), this information is generally read from a dedicated fixed working register (W14 has been chosen for this purpose), always containing the current round number during execution of the whole algorithm kernel. The number of the register providing the round information is therefore not explicitly encoded into the instruction word, or syntax of the assembler instruction, for that matter.

The syntax of the presented instructions uses the following nomenclature:

- **`Ws`**: source register
- **`Wd`**: destination register
- **`Wm`**: register holding a memory (base) address (e.g., to point to state data)
- **`#lit`**: literal (number constant)

5.1 BLAKE

BLAKE [1] uses an internal state that is arranged as a 4x4 matrix. For output sizes up to 256 bit, each element of the state is 32 bits wide and the core algorithm is applied for 14 rounds. One round of BLAKE consists of 8 calls to a *G function*, each of which operates on a different permutation of 4 elements of the state. At the end of all 14 round calculations, a short finalization step generates the hash output by combining the current hash value with the previous value and a *salt* value. The *G function* is the only computational kernel, which performs data transformation in the BLAKE hash algorithm. The *G function* uses a combination of 32-bit additions, XOR functions and four different 32-bit right rotate operations by 7, 8, 12 and 16 bits. The function makes use of sixteen 32-bit constants, which are applied in a different order depending on the round. This schedule is known as the *Sigma Permutation*. The input message block data is also selected for each round using this permutation schedule.

[3] 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

Reference Implementation BLAKE is shown to be one of the fastest implementations with the smallest memory footprint in various studies. Table 5 shows the breakdown of the memory resources in our assembler implementation. The core of the implementation is the *G function*, which operates on four 32-bit words and transforms them. To this end, the corresponding words of the internal state are first loaded into four register pairs using the appropriate permutation pattern. After the transformation through the *G function*, they are stored back to their original positions. The *G function* performs a lookup of the indices needed during this round in the *Sigma Permutation* table and uses those indices to address the correct words in the message data block, as well as the round constants data table. The 32-bit rotations by 7, 8 and 12 bits are emulated with 6 instructions each, utilizing the OR, left shift, and right shift instructions of the reference ISA.

Instruction Set Extensions We have developed three dedicated, new instructions for BLAKE.

- **`S3_XCST Wm`**
  During the initialization phase, the first eight 32-bit constants are XOR'ed onto the internal state (Wm). This instruction performs this operation efficiently while iterating over 16 cycles in conjunction with a preceding `REPEAT #15` statement. In each repetition the MCU treats 16-bit of the state and derives the necessary constant and the memory address offset for reading and writing back the modified state from the MCUs internal *rcount register* (loaded by the `REPEAT` statement).
- **`S3_RROT_7/8/12 Ws, Wd`**
  Rotations are among the most time consuming operations of the reference implementation. This family of three instructions performs a 32-bit right rotate by a specific amount of either 7, 8 or 12 bits, as required in the BLAKE algorithm, avoiding the need for a full barrel-shifter in the MCU. The input data is provided in a register pair (Ws) and the result is written to another register pair (Wd), using the MCU's double writeback feature, to perform the complete operation in one cycle. The fourth rotation needed by BLAKE is by 16 bits and can directly be performed efficiently using existing `MOV` instructions.
- **`S3_CXM Wm, Wd`**
  This instruction calculates the term combining constant data and message block data (Wm) by an XOR, used twice per *G function* call (with reversed indices). The instruction directly generates the lookup data from the *Sigma Permutation* table as a memory address offset, as well as the corresponding constant. This avoids costly memory accesses with very little area overhead for implementing the constant tables inside the processor. Since the algorithm operates on 32-bit words, the instruction needs two cycles to complete to be able to fetch the message block data word from memory.

Implementation with Instruction Set Extensions The instruction set extensions developed for BLAKE reduce the program memory by 20% (cf. Table 5), the data memory by 59% and reduce the cycle count by 34%, making an already fast and small implementation even faster and smaller. Memory for *Text* is saved by a compact way to describe the addition of constants without the need for unrolling, the automated *Sigma Permutation*, and the compaction of the rotation operations. *Data* is saved by placing all lookup tables into the processor. Throughput is mainly gained by a speedup of the rotations by a factor of 6, and efficient *Sigma Permutation* access, which eliminate additional memory accesses for the purpose of fetching indicies and constants data. The additional instructions have no noticeable impact on the core area compared to the reference implementation as they do not add new computational units, but modify only register and memory accesses for existing operations.

5.2 Grøstl

Grøstl [6] consists of a series of *f Compression* functions, followed by an *Output Transformation* that is derived from this *f* function. The *f* function is based on two operations, P and Q, which have a similar construction but use different parameters for two of its four sub operations. P and Q are each applied for 10 rounds per *f* function call for output sizes up to 256 bit. The *Output Transformation* consists of 10 rounds of the P operation.

The state in Grøstl consists of an 8x8 array of 8-bit data elements for output sizes up to 256 bits (8x16 array for larger output widths). Both P and Q operations consist of four sub-operations that are very similar to AES. *AddRoundConstant* XORs the current state with a round based constant (different for P and Q), *SubBytes* applies the 8-bit substitution table used for AES to the state, *ShiftBytes* performs a row based permutation of the data words (offsets different for P and Q), and finally *MixBytes* transforms each column of the state by multiplying the state in $\mathbb{F}_{256}$ with a constant matrix.

Reference Implementation Grøstl implementations in small microcontrollers have always been considered to be slow and large. Table 6 provides a breakdown of the memory footprint of our implementations. In terms of throughput our reference implementation requires almost 30,000 cycles per block (462 cycles/byte), making it one of the slowest algorithms.

It is important to note that due to the organization of the state matrix into byte elements that are used in row as well as column order, byte mode support of the MCU architecture is imperative to avoid even costlier implementations in terms of cycles and code size. The byte mode feature is particularly important for addressing single elements in byte increments in the data memory which is commonly organized in more coarse-grained word sizes. The instructions underlying the four sub-operations of P and Q can be reduced to simple XORs and memory moves combined with the use of three lookup tables (LUT). The first LUT is the AES S-Box, which maps every 8-bit value of the state to another 8-bit value. The remaining two LUTs hold precomputed results for the multiplications of any 8-bit value by either 2 or 4 in the finite field of $\mathbb{F}_{256}$ as used in AES.

Instruction Set Extensions Since all four sub-operations of P and Q are very modular and each transforms the complete 64 byte state in memory, without performing too complex arithmetic per state element, it was possible to develop four instructions for Grøstl, each replacing exactly one of the sub-operations. All these instructions are memory-to-memory instructions, i.e., they modify the state matrix in place, sometimes utilizing a set of work registers during their operation.

- **`S3_CST_P/Q Wm`**
  This instruction replaces the *AddRoundConstant* kernel, which XORs the current round number with the entries of a fixed constant matrix onto the entries of the state matrix (Wm). The corresponding operations are performed in a hardware-loop, using the MCU's `REPEAT` instruction and the internal *rcount register* as a state. The P version of the operation has mostly zero-entries in the constant matrix, which allows the use of `REPEAT #7`, to perform the complete transformation in 8 cycles. The Q version on the other hand has to transform every element of the state, requiring the use of `REPEAT #31`.
- **`S3_SBOX Wm`**
  This instruction adds the logic for the AES S-Box lookup table into the processor core (in form of a synthesized LUT) and executes two parallel 8-bit AES SubBytes table lookups per cycle. Therefore the complete state matrix (Wm) can by substituted in 32 cycles, using a `REPEAT #31` instruction, followed by a single `S3_SBOX` instruction.
- **`S3_SHIFT_P/Q Wm`**
  This instruction performs the *ShiftBytes* operation, and exists in both P and Q variants. The instruction is used in conjunction with `REPEAT #35`, performing the complete shift of all state rows (Wm) in 36 cycles. It operates on pairs of state elements from two adjacent rows, since the state matrix is addressed column-wise in memory. Two pairs of elements are combined in each repetition (cycle), normally one pair from memory and one pair from the register file, to form a new pair, which is the main computational extension of this instruction. The complexity of the instruction lies in providing the correct memory addressing schedule from the *rcount register* state, i.e., when to load which pair of elements to perform the entire transformation in the minimum amount of repetitions (cycles). The instruction uses registers W0-W7 as temporary work registers to buffer values that have already been overwritten in memory.
- **`S3_MIX Wm`**
  Most of the performance gain can be attributed to this instruction that performs the *MixBytes* operation. The instruction is used together with `REPEAT #79` and is able to multiply the entire state matrix (Wm) in $\mathbb{F}_{256}$ with the

constant matrix in only 80 cycles. This is achieved by processing the state elements in pairs (subsequent entries of the same column of the state matrix). Each pair is multiplied with all corresponding fields of the constant matrix in two cycles. The 16 extra cycles stem from phases during the instruction where only results can be written back to the memory. The multiplication is performed by a new functional unit that calculates for each input byte the multiplication results for all factors (1, 2, 3, 4, 5 and 7) by using shifts and XORs on 8-bit values only. The instruction uses W0-W3 as accumulator registers for the temporary results of the new column values.

Implementation with Instruction Set Extensions The `S3_MIX` instruction has been instrumental in reducing the cycle count for Grøstl. However, the corresponding instruction does not add complex new components to the datapath. The matrix multiplication is performed by a series of conventional XOR operations. The Instruction merely adjusts the source and destination of such elementary instructions based on the *rcount register* state. In effect the operational flow of the algorithm has been moved from programmed instructions to the instruction decoder and implemented as an FSM, controlled by the repeat counter. This move was possible due to the very modular structure of the operations, and their low complexity.

As the results show (cf. Table 2), it was possible to reduce the cycle-count by more than 87% with these additions, making the ISE modified implementation of Grøstl by far the fastest of the implementations. Furthermore data memory is reduced by 75%, due to cutting of the LUTs originally required for the *MixBytes* multiplication and the move of the S-Box LUT to the core. Instruction memory is also drastically reduced by 69%, since all four sub-operations are replaced by single instructions. The core area overhead for these ISEs is only 10% (2 kGE). Note that this overhead only refers to the MCU core itself, which is one of the smallest elements already in a real system, which also includes memory blocks of considerable area.

5.3 JH

JH [19] uses a 1024-bit internal state, independent of the output size. The message block is XOR'ed to parts of the present state and transformed over 42 rounds. At the end of the rounds the message is once again XOR'ed with the state to produce the next state. The JH function consists of three main steps. For each round a different round constant is used. Depending on the bits of this constant for each four-bit tuple of the state one of two different 4-bit substitution tables (*SBox*) is applied. Note that the four state bits are spread out over the entire state block. As a second step, a linear transformation *L* defined over 8 bits is applied. This is concluded by a permutation function *swap* that shuffles the state.

Reference Implementation While this initial description is suitable for hardware implementations, a so called *bit-sliced* implementation [19] can be used in software to improve efficiency. In *bit-sliced* implementations, instead of using many operations with small bit-widths (i.e., for 4-bit *SBox* functions), large words are formed by combining the same binary digit of multiple inputs. Since the PIC24 is a 16-bit architecture, we have implemented a bit-sliced implementation tailored to a data word size of 16 bit. Operations in JH can be transformed so that 16-bit words can be formed by combining the bits of 16 separate words. To store the precomputed round constants for this bit-sliced implementation, 42x16x16 bits = 1344 bytes are required, which accounts for the majority of the *Data* segment as can be observed in Table 7. This contribution also renders JH the algorithm with the largest memory footprint in terms of *Data* size for the reference implementation (cf. Table 4).

The bit-sliced implementation changes the described *SBox* lookups into an operation using bit-slice instructions only (on four 16-bit values and a constant), namely AND, NOT, and XOR. This is followed by the *L/MDS* transformation, which only uses XORs on 8 different values. The *swap* operation on 8 values each, is performed by either using bit-masks combined with left and right shifts and logic ORs for small shift/interleave values (1, 2 and 4), or simple MOV instructions for larger values (8, 16, 32 and 64).

Instruction Set Extensions The following two instructions were added to the instruction set to improve performance of interleaving state data during the *swap* operation, as well as to remove the large round constants LUT from data memory.

- **S3_CST_ACXM/AMXC #index, Ws, [Wm], Wd**
  This instruction performs a lookup using a LUT built into the processors datapath, holding all round constants, depending on the current round number and a position (0..15) index passed as a literal (#index). This constant is then used in a combined AND-then-XOR operation, using additionally one source register (Ws) and one data word read from memory (Wm). The result is stored in a register (Wd). The instruction has two different modes, both used once in each *SBox* call, effectively exchanging the order of the constant and the data word from memory during execution.
- **S3_SWAP Wm**
  This instruction is used in conjunction with `REPEAT #31` to carry out the *swap* operation in 32 repetitions. The parameter of the swap (the size of the bit-tuples to be swapped, 1, 2, 4, 8, 16, 32 or 64) is thereby derived from the round counter value which is taken from a fixed register.

Implementation with Instruction Set Extensions Our implementation results with ISEs show a reduction of cycles/byte of 17%, mostly due to the improved *swap* operation. However, the main benefit comes from the reduction of costly data memory by 87% since the LUT containing the round constants has been moved into a hardwired LUT in the core, where it only generates an area overhead of about 10% (2 kGE). *Text* is reduced by 53% due to the simplification of the different flavours of the *swap* operation into a single instruction.

5.4 Keccak

Keccak [3] uses a so called *sponge* function where the input message is absorbed (XOR'ed) into a 1600 bit wide state, using a message block size of 1088 bit. Both, state and message block size are therefore uncommon compared to the other algorithms. The state is organized as a 5x5 array. Each entry of this array is called a lane, and consists of a 64-bit word. One message block is processed in 24 rounds, and each round consists of 5 functions called θ, ρ, π, χ, and ι (*Theta*, *Rho*, *Pi*, *Chi* and *Iota*). *Iota* adds a round constant, while *Rho*, *Pi* and *Theta* are linear transformations used for diffusion. The non-linearity required for the hash function is provided by the *Chi* operation, which is applied to each row of the state.

Reference Implementation The *Theta* operation performs a series of XOR combinations on state elements, including a single bit rotation. The *Rho* step performs a 64-bit rotation on a state element, using a different rotation constant for each element of the 5x5 state matrix, depending on position. The *Pi* step permutes the elements of the state matrix and is combined with the *Rho* step in our reference implementation. A state element is loaded from memory, rotated and then directly written to its new destination, according to the permutation table defined by *Pi*. This permutation can be chained, so that only one element has to be buffered for the complete state to be rotated and permuted in place. The next transformation is *Chi*, it applies a combination of logical operations of AND, NOT and XOR. The final step *Iota* XORs the constant depending on the current round onto the state element (0,0).

Instruction Set Extensions The main bottleneck of the algorithm on a 16-bit architecture are the 64-bit rotations, which need 12 cycles per rotation for the generic case in the reference implementation. Hence, this operation is the main focus of the ISEs. Furthermore LUTs were moved into the core where possible to reduce *Text* as shown in Table 8. Eventually, the following three ISEs were chosen:

- **S3_LROT1 Ws, Wd**
  This instruction performs a 64-bit left rotate by a fixed amount of 1 bit in 2 cycles, using a set of four registers as source (Ws) and a set of four other registers as destination (wd). The instruction utilizes the double writeback (to register file) capabilities of the architecture.
- **S3_LROTM_A/B #rot, #coord, Wm**
  This instruction implements a generic 64-bit left rotation by an arbitrary number of bits (#rot). The operation works on a set of four source registers and writes the result to a memory location (Wm) defined by coordinates in the 5x5 state matrix, given as a literal (#coord). Execution takes 4 cycles and the instruction performs the additional

function of saving the data located at the destination memory address to another set of registers, before writing the rotation result to memory. The instruction exists in two variants, differing only in the set of registers used for rotation and for backup storage (W0-W3 and W4-W7). Using these two variants in alternating order allows to effectively store the result of the currently rotated element, while at the same time already pre-loading the next element of the chain, combining the *Rho* + *Pi* operations (rotations + permutations) over all 25 elements of the state matrix.

- **`S3_XCST Wm`**
  This instruction XORs the current round constant onto the internal state element at (0,0) (Wm). Execution time varies between one and three cycles, depending on the constant value, since possible zero bytes are not applied.

Implementation with Instruction Set Extensions The results show a reduction of the number of cycles by 30%, which can mainly be attributed to the improved efficiency of the 64-bit rotation, as well to the fact that it is now combined into the element chaining of the *Pi* state permutation. The Data size is reduced by 21%, by placing the round constant table into the core. The Text size is lowered by 31%, through compact description of the combined *Rho* + *Pi* operation in a single instruction.

All three extensions cause virtually no core area overhead, since they contain no large LUTs and do not add considerably to the datapath. The round constant lookup table can be stored in the core in an even more compressed way compared to the standard implementation, only utilizing 24x11 bits = 33 bytes. For a more detailed discussion of the 64-bit rotation and its resource requirements, please see the corresponding paragraph in the Skein subsection, which also utilizes a very similar instruction, but implements it in a slightly different way.

5.5 Skein

The Skein algorithm [4] constructs a hash function out of a *tweakable* block cipher called *ThreeFish*. The Skein algorithm that we have implemented uses a 512-bit state, with 256-bit output. The main idea behind Skein is to keep the round structure simple, while using a large number of rounds to obtain security. In total there are 72 *ThreeFish* rounds, each of which is made up of a *Mix* operation defined over 128 bits, and a permutation. The *Mix* function has a simple construction, and consists of 64-bit addition, XOR and rotation functions. For every four *ThreeFish* rounds a subkey is XOR'ed onto the state. These subkeys are generated from a *tweak* value and the round counter using a *KeySchedule*.

Reference Implementation The computational core of Skein is the *Mix* function of *ThreeFish*. This *Mix* function consists of simple arithmetic operations on 64-bit words, some of which are well supported by the standard ISA, through the use of addition with carry and standard XOR. The problematic operation is the left rotation over 64-bit words, which has to be emulated using 12 instructions per rotation in the generic case. The cases for rotations by a multiple of 16 can be performed in 4 cycles using MOV instructions.

Instruction Set Extensions Profiling of the reference implementation showed that large gains can be achieved by removing every unnecessary cycle from the *Mix* function core. This lead to the addition of one instruction, performing the rotation in an efficient way, providing a general speedup of 6 for the rotation function.

- **`S3_LROT #rot, Ws, Wd`**
  This instruction performs a generic 64-bit left rotate by an arbitrary amount of bits (given as a literal #rot) in two cycles, using a set of four registers as source (Ws) and a set of four other registers as destination (Wd). The instruction utilizes the double writeback (to the register file) capability of the architecture. Since the instruction uses two cycles to produce the result, in each cycle using three specific input values, the added logic corresponds to two 16-bit barrel shifters, instead of a real 64-bit barrel shifter, and is therefore very small.

Implementation with Instruction Set Extensions Skein did not receive any additional ISE, since the building block of the *Mix* function is already very basic and well supported by the standard ISA. The algorithm also does not utilize any sort of LUT that could be moved into the core. Furthermore the address arithmetic needed for performing the permutations and selection of state words is quite straightforward and does not warrant any special supporting instructions, since possible performance gains would be minimal.

Nevertheless, with a single carefully chosen instruction set extension the results show an improvement in throughput of 29%, with virtually no core area overhead. The Text size is reduced by 18%, but still the largest implementation, because of unrolling of the complete key injection schedule for all 9 cases (mainly caused by the round cycle size of 9, which is not a power of 2). The Data size did not change, but is already small due to the lack of LUTs, as can be seen in Table 9.

6 Results

Embedded systems and particularly performance and/or resource constrained small microcontrollers have been identified as an important application for future SHA-3 implementations. Authors of nearly all algorithms have also published performance results and/or estimations for specific small microcontrollers on their webpages. The most important study has been done by Christian Wenzel-Benner and Jens Gräf [17]. They maintain a webpage [18], where their results are published. These results have also been included in the eBASH website [2]. Thomas Pornin [14] 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.

While large general purpose microprocessors share a similar instruction set architecture, small microcontrollers come in many different variations. They differ in their structure (Harvard, von Neumann), width of their datapath (4-32 bits), and provide differing resources (number of registers, multipliers etc). This makes a general comparison between microcontrollers extremely difficult.

6.1 16-bit MCU reference implementation results

In this work, we have used the PIC24 microcontroller as a representative for the variety of available 16-bit microcontrollers, which complements other published results which so far have focused on 8-bit and 32-bit MCU architectures only. In Table 1 we list our implementation results together with other published results.

Table 1. Comparison of throughput numbers [cycles/byte] of published microcontroller implementations. Numbers in parentheses show performance normalized to BLAKE performance.

	This work		[14]	[14]	[18]	[18]	[7]
Architecture	PIC24	PIC24+ISE	ARM-M3	ARM920T	ARMv5TE	ATmega128	8051
Datapath	16-bit	16-bit	32-bit	32-bit	32-bit	8-bit	8-bit
Specification	third	third	third	third	third	third	second
BLAKE	155 (1.00)	103 (1.00)	89 (1.00)	54 (1.00)	87 (1.00)	1241 (1.00)	643 (1.00)
Grøstl	462 (2.98)	58 (0.56)	455 (5.11)	313 (5.79)	216 (2.48)	11198 (9.02)	1327 (2.06)
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)	1745 (2.71)
Skein	158 (1.02)	113 (1.10)	128 (1.44)	129 (2.39)	184 (2.11)	1444 (1.16)	1944 (3.02)

It can be seen, that BLAKE is consistently fast throughout all published works. The ranking is not so clear for other candidates, however Grøstl and JH are mostly ranked among the slower implementations. It can be seen that our PIC24 reference implementations ranks quite favorably when compared to other implementations. In particular, our

Keccak realization outperforms all published results for microcontrollers, even those for 32-bit architectures. This is partially attributed to the fact that we use hand-crafted assembly, rather than compiled C-code.

6.2 Implementation results with instruction set extensions

By developing instruction set extensions for the PIC24 platform we were able to increase the performance of individual implementations by almost 40%. Table 2 shows these results in detail and compares the area overhead that the instruction set extensions incurred to the reference microcontroller with a core area of 23 kGE. It must be noted that in the best case, the core area represents one third of the total area of the overall MCU subsystem. The other two thirds are used by data and instruction memory. As such, the net overhead due to the instruction set extension will be even much smaller.

It can be seen that for three out of five candidates, the instruction set extensions did not result in any noticeable core area overhead, while still providing significant improvements, in both cycle-count (around 30%) (Table 2) and memory requirements (Table 4). This is due to two factors: First, some instruction set extensions did not actually add datapath components, but provided small finite state machines, that are able to resolve complex but regular addressing schemes to fetch operands for already present datapath components (XOR, AND, ADD). Second, the digital design flow is known to produce results that vary as much as $\pm 5\%$, so changes smaller than 5% can not reliably be presented. In fact for two out of five candidates, the core area for the processor actually decreased slightly when instruction set extensions were added.

The smallest speed up that was obtained through the instruction set extensions was for JH at 17.3%, whereas the largest improvement 87.5% was obtained for Grøstl. Interestingly, these two implementations both resulted in about 10% area overhead, which is still negligible.

Table 2. Improvement of long message hashing speed by using instruction set extensions for all SHA-3 candidates, and overhead of instruction set extensions on the core area of the PIC24 microcontroller.

	Cycles		**Cycles/byte**			**Area Overhead**
	PIC24	+ISE	PIC24	+ISE	**Reduction**	
BLAKE	9,933	6,583	155.2	102.9	**-33.7%**	∼0%
Grøstl	29,585	3,685	462.3	57.6	**-87.5%**	+10%
JH	29,683	24,541	463.8	383.5	**-17.3%**	+10%
Keccak	25,613	17,908	188.3	131.7	**-30.1%**	∼0%
Skein	10,084	7,204	157.6	112.6	**-28.6%**	∼0%

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 speed up is achieved by eliminating memory accesses, mostly by embedding operational constants into the instruction set extensions. In Table 3 we list the number of read and write memory accesses for all candidate algorithms. When comparing the memory accesses to the total number of cycles listed in Table 2, note that the PIC24 architecture allows concurrent read and write operations to the data memory within the same cycle. 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. For Skein, on the other hand, we could not find any instructions that would be able to reduce the number of memory accesses.

As mentioned earlier, in the best case, around two thirds of an embedded processor consist of program and data memory. 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. Generally the Data Memory is more expensive for small microcontrollers, as they have to be realized using costly SRAMs, while program memory can often be implemented

Table 3. 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%**

using more efficient ROMs. We have listed the total data and program (text) memory used by all candidate algorithms listed separately for both the standard and the enhanced instruction set implementation of PIC24 in Table 4. It can be seen that the largest improvement was achieved for JH (61.5% total reduction) and Grøstl (71.3% total reduction). Whereas the smallest improvement was achieved for Skein (17.7% total reduction).

Table 4. Reduction of instruction and data memory by using instruction set instructions 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%**

7 Sources of Error

Although we have tried to minimize the sources of errors, there are several factors that may have influenced the results. In this section we try to outline the possible sources of error in our results, and explain what we have done to address them.

7.1 Choice of the MCU

All our results are given on a particular MCU. This MCU was chosen mainly because a suitable description (for LISA) was available to us at the beginning of the project. 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, it has a simple Harvard architecture, a relatively large number of single cycle instructions which makes it easy to implement. We do not think that the results presented here would differ significantly if another processor were to be used, however we can not exclude this possibility. We are also not aware of a *universally approved* representative processor/architecture for such comparisons, as a results all such studies are bound to have errors in this regard.

7.2 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 implemented in assembly language. This was done manually. In the later stages, potential

instruction set extensions have been determined by manually analyzing the first implementations. And finally, the optimized implementations utilizing the instruction set extensions have been coded in assembly language 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 of the implementations were *better* than the others, and/or some optimization possibilities were *overlooked* in the process.

7.3 Reporting overhead

Enhancing the processor datapath will have two consequences. First of all, in most cases, additional logic will be added to the datapath, increasing the overall area of the processor. In some cases, the additions may also increase the critical path of the processor. Most synthesizers would be able to exploit various techniques to trade-off speed versus area, and could compensate for the increased delay by increasing the area further. However, in some cases the increase is simply too much to be compensated, resulting in a circuit that is not only larger than the original, but also slower.

For this paper, we have first determined the performance limits of our PIC24 architecture for the 90 nm CMOS technology we have available. This was determined to be slightly above 250 MHz clock speed using typical conditions. We have decided to add some margin and synthesized all the implementations so that they achieve 200 MHz clock speed. The difference in core area is attributed to the overhead due to the instruction set extensions, and this overhead figure has been reported in the paper.

Our experience is that synthesis results have an accuracy of roughly $\pm 10\%$ depending on many factors. All numbers used 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.

8 Conclusions

It is well known that instruction set extensions can increase the performance of an application on a given microprocessor. In this work, the question we had asked was, *Would all SHA-3 candidate algorithms benefit equally from instruction set extensions?*. Our results clearly show that some algorithms could potentially benefit much more from instruction set extensions than others. In particular, we were able to improve the execution speed of Grøstl by more than 85%, and reduced its memory footprint 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). In this case the instruction set extensions have increased the core area for the processor by a modest 10%.

We have presented a detailed analysis of all implementations, and for each algorithm have explained what type of new instructions were developed. In three out of 5 cases, the instruction set extensions had no measurable impact on the core area of the microcontroller, in the two other cases, the overhead was limited to 10%, whereas the excution speed improved by nearly 40% on average over five candidates.

It was shown that most improvements were made by instructions that took advantage of data in the register file. 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 processors datapath turned out to be highly beneficial in reducing memory footprint at negligible core-area overhead.

Furthermore, our reference implementations of the five candidate algorithms on a standard PIC24 instruction set architecture provides insight into the performance of the SHA-3 finalists on 16-bit microcontrollers. The comparison shows that BLAKE, Keccak, and Skein achieve a similar number of cycles per byte, which is about one third of that of JH and Grøstl. In terms of program memory footprint (*Text*), the algorithms are ranked as follows (best to worst): BLAKE, Grøstl, JH, Keccak, and Skein with a factor of more than four between the best and the worst. With respect to the data memory, the order is different: Skein, Keccak, BLAKE, Grøstl, and JH, with a factor of almost seven between the best and the worst.

References

1. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Sha-3 proposal blake. Submission to NIST (Round 3), 2010.
2. Daniel J. Bernstein and Tanja Lange (editors). ebacs: Ecrypt benchmarking of cryptographic systems, 2011. http://bench.cr.yp.to.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak reference. Submission to NIST (Round 3), 2011.
4. Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. Submission to NIST (Round 3), 2010.
5. Kris Gaj, Jens-Peter Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y. Brewster. Athena: Automated tool for hardware evaluation, 2011. http://cryptography.gmu.edu/athena/.
6. Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a sha-3 candidate. Submission to NIST (Round 3), 2011.
7. Mourad Gouicem. Comparison of seven sha-3 candidates software implementations on smart cards. Cryptology ePrint Archive, Report 2010/531, 2010. http://eprint.iacr.org/.
8. The ECRYPTII group. The sha-3 zoo, 2011. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.
9. Luca Henzen, Pietro Gendotti, Patrice Guillet, Enrico Pargaetzi, Martin Zoller, and Frank Gürkaynak. Developing a hardware evaluation method for sha-3 candidates. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 248–263, 2010.
10. 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 *Computer-Aided Design of Integrated Circuits and Systems, Transactions on , vol.20, no.11*, pages 1338–1354. IEEE, 2001.
11. Microchip. 16-bit mcu and dsc programmer's reference manual, 2009. http://ww1.microchip.com/downloads/en/DeviceDoc/70157D.pdf.
12. Microchip. Pic24hjxxxgpx06/x08/x10 data sheet, 2009. http://ww1.microchip.com/downloads/en/DeviceDoc/70175H.pdf.
13. 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.
14. Thomas Pornin. sphlib, update for the sha-3 third round candidates, 2011. http://www.saphir2.com/sphlib.
15. O. Schliebusch, R. Leupers, and H. Meyr. *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007.
16. Synopsys. Automating the design and implementation of custom processors (processor designer, lisa 2.0), 2011. http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx.
17. Christian Wenzel-Benner and Jens Gräf. Xbx: external benchmarking extension for the supercop crypto benchmarking framework. In *Cryptographic Hardware and Embedded Systems - CHES*, volume 6225 of *LNCS*, pages 294–305. Springer, 2010.
18. Christian Wenzel-Benner and Jens Gräf. Xbx: external benchmarking extension, 2011. http://xbx.das-labor.org/trac.
19. Hongjun Wu. The hash function jh. Submission to NIST (round 3), 2011.

A Detailed Implementation Results

Table 5. Memory resource breakdown of the standard implementation of the BLAKE algorithm on the PIC24 architecture, and the changes due to Instruction Set Extensions.

	Description	Reference	Δ ISE	PIC24	PIC24+ISE
Data				**488 bytes**	**200 bytes**
	Hash State	32 bytes			
	Salt	16 bytes			
	Counter	8 bytes			
	Message Block	64 bytes			
	Internal State	64 bytes			
	Misc. Variables	4 bytes			
	Stack	12 bytes			
	Constants (Pi)	64 bytes	-64 bytes		
	Sigma Permutations	224 bytes	-224 bytes		
Text				**1028 bytes**	**818 bytes**
	CompressBlock()	274 instr	-39 instr		
	RoundFuncG()	58 instr	-31 instr		
	Initial State	32 bytes			

Table 6. Memory resource breakdown of the standard implementation of the Grøstl algorithm on the PIC24 architecture, and the changes due to Instruction Set Extensions.

	Description	Reference	Δ ISE	PIC24	PIC24+ISE
Data				**982 bytes**	**214 bytes**
	Hash State	64 bytes			
	Message Block	64 bytes			
	Internal State	64 bytes			
	Misc. Variables	6 bytes			
	Stack	16 bytes			
	S-Box	256 bytes	-256 bytes		
	Mult 2 LUT	256 bytes	-256 bytes		
	Mult 4 LUT	256 bytes	-256 bytes		
Text				**2619 bytes**	**819 bytes**
	CompressBlock()	173 instr			
	PermutationPQ()	26 instr			
	AddRoundConstPQ()	70 instr	-70 instr		
	SubBytes()	134 instr	-134 instr		
	ShiftBytesPQ()	246 instr	-246 instr		
	MixBytes()	150 instr	-150 instr		
	OutputTransform()	74 instr			

Table 7. Memory resource breakdown of the standard implementation of the JH algorithm on the PIC24 architecture, and the changes due to Instruction Set Extensions.

	Description	Reference	Δ ISE	PIC24	PIC24+ISE
Data				**1550 bytes**	**206 bytes**
	Hash State	128 bytes			
	Message Block	64 bytes			
	Misc. Variables	4 bytes			
	Stack	10 bytes			
	Round Constants	1344 bytes	-1344 bytes		
Text				**4649 bytes**	**2183 bytes**
	CompressBlock()	144 instr			
	SBox()	432 instr	-37 instr		
	L/MDS()	144 instr			
	Swap	787 instr	-785 instr		
	Initial State	128 bytes			

Table 8. Memory resource breakdown of the standard implementation of the Keccak algorithm on the PIC24 architecture, and the changes due to Instruction Set Extensions.

	Description	Reference	Δ ISE	PIC24	PIC24+ISE
Data				**448 bytes**	**352 bytes**
	Hash State	200 bytes			
	Message Block	136 bytes			
	Misc. Variables	4 bytes			
	Stack	12 bytes			
	Round Constants	96 bytes	-96 bytes		
Text				**3480 bytes**	**2415 bytes**
	CompressBlock()	147 instr			
	Theta()	307 instr	-20 instr		
	RhoPi()	349 instr	-324 instr		
	Chi()	345 instr			
	Iota()	12 instr	-11 instr		

Table 9. Memory resource breakdown of the standard implementation of the Skein algorithm on the PIC24 architecture, and the changes due to Instruction Set Extensions.

	Description	Reference	Δ ISE	PIC24	PIC24+ISE
Data				**242 bytes**	**242 bytes**
	Hash State	64 bytes			
	Key Schedule	72 bytes			
	Tweak	24 bytes			
	Message Block	64 bytes			
	Misc. Variables	2 bytes			
	Stack	16 bytes			
Text				**5734 bytes**	**4678 bytes**
	CompressBlock()	75 instr			
	KeyscheduleGen()	83 instr			
	ThreeFishRounds()	876 instr	-352 instr		
	InjectKey()	844 instr			
	OutputTransform()	12 instr			
	Initial State	64 bytes			