Light-Weight Instruction Set Extensions for Bit-Sliced Cryptography

Philipp Grabher, Johann Großschädl, and Dan Page

University of Bristol, Department of Computer Science
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, U.K.
{grabher,johann,page}@cs.bris.ac.uk

Abstract. Bit-slicing is a non-conventional implementation technique for cryptographic software where an n-bit processor is considered as a collection of n 1-bit execution units operating in SIMD mode. Particularly when implementing symmetric ciphers, the bit-slicing approach has several advantages over more conventional alternatives: it often allows one to reduce memory footprint by eliminating large look-up tables, and it permits more predictable performance characteristics that can foil time based side-channel attacks. Both features are attractive for mobile and embedded processors, but the performance overhead that results from bit-sliced implementation often represents a significant disadvantage. In this paper we describe a set of light-weight Instruction Set Extensions (ISEs) that can improve said performance while retaining all advantages of bit-sliced implementation. Contrary to other crypto-ISE, our design is generic and allows for a high degree of algorithm agility: we demonstrate applicability to several well-known cryptographic primitives including four block ciphers (DES, Serpent, AES, and PRESENT), a hash function (SHA-1), as well as multiplication of ternary polynomials.

1 Introduction

In some sense, the provision of cryptographic schemes to secure information being communicated or stored is a compromise: higher levels of security necessitate higher levels of computational overhead. Given this fact, the study of low-cost implementation techniques that improve the efficiency and/or memory footprint of cryptographic schemes remains an ongoing research topic. In this context one can consider a spectrum of approaches. At one extreme are software-based techniques to manipulate algorithms so they are more efficient or more easily map to the capabilities of the host platform; at the other are hardware-based techniques which re-design or extend the platform to better suit algorithms. Somewhere in this design space is the technique of identifying and implementing Instruction Set Extensions (ISEs) [16,27,35]. The premise is that, after a careful workload characterisation, it is possible to identify a small set of operations that dominate the execution time of a software implementation. By supporting these specific operations using additional or modified hardware and exposing their behaviour to the programmer via the Instruction Set Architecture (ISA), performance can

© International Association for Cryptologic Research 2008

be significantly improved. This is often possible with only minor penalties in terms of datapath disruption and logic overhead; ideally a generic ISE is more attractive than one which is suited for use in only a single algorithm.

The design of custom instructions to support the execution of cryptographic workloads has been actively researched in the recent past. Previous work on cryptography extensions for general-purpose processors covered both public-key [2,16,30] and secret-key algorithms [9,15,8]. An example for the former are the Cryptography Instruction Set (CIS) extensions to the SPARC V8 architecture [17]. The CIS extensions consist of only six custom instructions, but allow one to accelerate the full range of public-key algorithms standardised in IEEE 1363 [20]; these include RSA, DSA, Diffie-Hellman, as well as elliptic curve schemes over prime and binary fields. Therefore, the CIS extensions are referred to as *domain-specific extensions*, in contrast to application-specific extensions like the ones described in [2,30], which support just a single public-key algorithm. The idea of domain-specific extensions is based on the observation that virtually all public-key algorithms of practical importance use either a multiplicative group ($\mathbb{Z}_p^*$ or $\mathbb{Z}_n^*$), a prime field, or a field of characteristic 2 as underlying algebraic structure. Thus, by designing custom instructions that accelerate the arithmetic of large integers and binary polynomials, it is possible to support a wide range of public-key cryptosystems.

Previous work on optimised architectures for secret-key cryptography considered the design Application-Specific Instruction set Processors (ASIPs) and the integration of custom instructions into general-purpose processors. Most of the published instructions are optimised for a single secret-key algorithm such as DES [12] or AES [4,13,35]. Among the few exceptions are the instruction sets of CryptoManiac [36], MOSES [32,33], and PAX [14], which were designed with the objective of more general applicability. CryptoManiac's architecture consists of a conglomeration of different sets of custom instructions, each set crafted for a specific algorithm based on its performance-critical core functions. Unfortunately, the design of custom instructions for a whole domain of algorithms is much harder for secret-key cryptography than for public-key cryptography, mainly due to the large number of different design strategies and underlying basic operations: instructions for accelerating the execution of one secret-key algorithm are in most cases useless for other algorithms.

Look-up tables are a generic and low-cost processor extension to increase the performance of various classes of applications, including secret-key algorithms [38]. These look-up tables can be configured to implement different dataflow subgraphs depending on the application being executed. Using this mechanism reduces the latency of the subgraph's execution and the number of temporary values that need to be stored to the register file. In [29], Patterson demonstrated the merits of this approach taking Serpent as example, whereby he achieved a throughput of 10 Gbit/sec on an FPGA implementation.

In this paper we introduce an ISE which can be used to accelerate a range of cryptographic algorithms that operate on data in a bit-oriented (rather than word-oriented) manner. In particular, we consider bit-sliced algorithms [5]. The exemplar use of bit-slicing is given by Biham, who extracted a 5-fold performance gain from DES [5]. However, beyond pure performance, one can identify another more subtle advantage from the general approach. By, for example, eliminating a (potentially very large) table used to represent S-box content, a typical bit-sliced implementation will have a smaller data memory footprint; despite the fact that the code memory footprint may slightly increase, the overall effect is usually a net gain. Furthermore, elimination of such tables also eliminates the need to execute instructions that access them. Depending on the exact memory hierarchy, this can result in (more) predictable, data-independent execution and thus prevent cache-based side-channel attacks [7,28]. Our proposed ISE capitalises on these significant advantages of bit-sliced implementation while further improving their performance, a factor which is often perceived as a disadvantage.

We organise the rest of this paper as follows. In Section 2 we recap on the concept of bit-slicing and introduce the design of our ISE and the host platform it is embedded into. We then use Section 3 to evaluate the ISE, demonstrating its generic nature by presenting application in six different case studies; in each case we are able to improve performance and reduce memory footprint versus an implementation on the same platform without the use of our ISE. Finally, in Section 5 we conclude and present some areas for further work.

2 ISE Definition

2.1 Bit-Sliced Implementation

Imagine a scalar processor with a w-bit word size, let x_i denote the i-th bit of a machine word x where i is termed the index of the bit. Such a processor operates natively on word-sized operands. For example, with a single operation one might perform addition of w-bit operands x and y to produce $r = x + y$, or component-wise XOR to produce $r_i = x_i \oplus y_i$ for all $0 \leq i < w$. This ability is restricted however when an algorithm is required to perform some operation involving different bits from the same word. For example, one might be required to combine x_i and x_j, where $i \neq j$, using an XOR operation in order to compute the parity of x. In this situation one is required to shift (and potentially mask) the bits so they are aligned at the same index ready for combination through a native, component-wise XOR. The technique of bit-slicing, proposed by Biham for efficient implementation of DES [5], offers a way to reduce the associated overhead. Instead of representing the w-bit value x as one machine word, we represent x using w machine words where word i contains x_i aligned at the same fixed index j. As such, there is no need to align bits ready for use in a component-wise XOR operation. Additionally, since native word-oriented logical operations in the processor operate on all w bits in parallel, one can pack w different values (say $x[k]$ for $0 \leq k < w$) into the w words and proceed using an analogy of a SIMD-style parallelism. Conversion to and from a bit-sliced representation can represent an overhead but this can be amortised if the cost of computation using the bit-sliced values is significant enough.

2.2 CRISP

A quarter century after many design decisions and assumptions were made by the pioneers of RISC, we are still using largely similar processor designs. One expects that such decisions were initially made using a mix of research and common sense based on prevailing technologies of the time. Despite the huge success of these assumptions, the technology landscape has now changed radically: the types of program we execute today are different and many of the constraints which guided initial thinking have disappeared. This is certainly true of cryptographic workloads as evidenced by previous work on application specific processors such as CryptoManiac [36] and Cryptonite [8].

CRISP (short for Cryptographic RISc Processor) represents an attempt to reassess some of these design decisions in the context of cryptography. The aim is to produce a processor design which is general purpose, but unencumbered by the constraints of history. For this paper, it suffices to consider CRISP as a conventional five-stage pipeline which, in contrast with the more conventional 3-address form, allows 6-address instructions. There are 16 general-purpose registers; this enables instructions to be encoded using a fixed 32-bit format. The philosophy is that, although this approach might, for example, dictate a lower clock frequency, central operations are more naturally described. Let the i-th entry in the general-purpose register file be denoted by $GPR[i]$, the datapath width be w and x_j denote the j-th bit of some w-bit word x. A representative example of said philosophy is the instruction for addition which uses three source operands (a, b and c) and two target operands (p and q). A conventional processor would maintain, and specify instruction for manipulating, a carry-flag; since an instruction can produce two results, CRISP treats the carry-flag as a general purpose register. The addition is therefore specified as

$$ADD\; p, q, a, b, c \quad \mapsto \quad \begin{array}{lcl} t & = & (GPR[a] + GPR[b]) + GPR[c] \\ GPR[p] & = & t_{w-1\ldots 0} \\ GPR[q] & = & t_{2w-1\ldots w} \end{array}$$

such that the three source operands are added together and low and high w-bit halves of the result are stored using the two target operands p and q. The clear disadvantage of such an instruction is higher latency; the advantages include removal for special-case management of the carry-flag and higher instruction throughput. We are aiming to improve the instruction throughput with a level of overhead somewhere between single issue and much more expensive multiple issue. Although the 6-address instruction format of CRISP is unconventional, one can imagine mechanisms to specify similar instructions in conventional 3-address architectures. One example is the use of SIMD instructions that pack multiple operands into registers addressed as one unit. Another approach is to serialise the operand transfers from/to the register file, which effectively relaxes the port constraints of instruction set extensions [31].

Within the general CRISP design we include three instructions which target bit-sliced implementation of cryptography. The processor includes two special purpose registers $LUT0$ and $LUT1$, which are used as 4-input, 1-output Look-Up Tables (LUTs). Configuration of the LUTs is performed by two instructions

$$\begin{array}{lcl} CLUT0\ a & \mapsto & LUT0_i = a_i \\ CLUT1\ a & \mapsto & LUT1_i = a_i \end{array}$$

each of which load the given LUT with a 16-bit immediate operand a, essentially configuring the LUTs. Use of the LUTs is performed with a third instruction

$$ULUT\ p, q, a, b, c, d \ \mapsto \ \begin{array}{l} GPR[p]_i = LUT0[8 \cdot a_i + 4 \cdot b_i + 2 \cdot c_i + 1 \cdot d_i] \\ GPR[q]_i = LUT1[8 \cdot a_i + 4 \cdot b_i + 2 \cdot c_i + 1 \cdot d_i] \end{array}$$

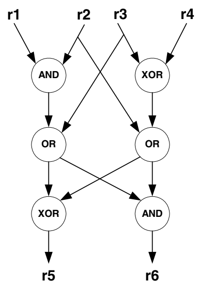
which takes the i-th bit of each source operand and concatenates them to form an index into each LUT; the LUT output forms the i-th bit of the result, two of which are computed in parallel.

To illustrate the benefit of our approach, we use the dataflow subgraph in Figure 1(a) as example, which takes four inputs and produces two outputs via a series of simple logical instructions. On a general-purpose RISC processor, the cost of evaluating this subgraph is exactly six instructions as depicted in Figure 1(b). However, this form of subgraph can be implemented naturally using the LUTs described above; Figure 1(c) shows that the corresponding implementation consists of only two `CLUT` instructions and one `ULUT` instruction.

Since many important block ciphers rely on the efficient computation of bit-level permutations, we include architectural support for this type of operation within our design. Extensive research in this area has been conducted by Lee et al. [23,37,34]. In [23], Lee et al. described how a combination of `GRP` and `SHIFT PAIR` instructions can be used to perform arbitrary bit permutations. The `GRP` instruction is defined as follows

```
GRP Rs, Rc, Rd
```

It moves the bits in the source register `Rs` to the most significant bit positions and to the least significant bit positions according to the control bits in `Rc`. On



(a)

```
AND r5, r1, r2
XOR r6, r3, r4
OR  r5, r5, r3
OR  r6, r6, r2
XOR r5, r5, r6
AND r6, r6, r5
```

(b)

```
CLUT0 14084
CLUT1 51448
ULUT  r5, r6, r1, r2, r3, r4
```

(c)

Fig. 1. An example dataflow subgraph (a) with the corresponding pseudo assembly code for a basic RISC machine (b) and for a LUT-based implementation (c)

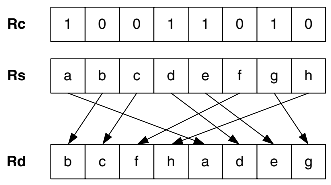


Fig. 2. `GRP` instruction [23]

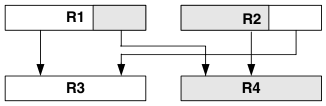


Fig. 3. `Shift Pair` instruction [23]

an n-bit processor, no more than $\log(n)$ `GPR` instructions are required to perform an arbitrary n-bit permutation. Figure 2 illustrates the functionality of the `GRP` instruction in case of 8-bit registers. The `SHIFT PAIR` instruction is instrumental in supporting permutations that cross word boundaries. It concatenates two source registers and separates the contiguous bit regions into two destination registers as depicted in Figure 3.

3 Performance Evaluation

We implemented an early prototype of the CRISP processor using the Processor Designer tool-chain from CoWare. The tool-chain is based on the Language for Instruction Set Architectures (LISA), which allows one to describe a processor architecture at a high level of abstraction; the description allows automatic generation of an instruction set simulator, a complete suite of software development tools, and synthesisable VHDL code. As such, although the results are often less optimal than a hand-written alternative, the tool-chain allows one to quickly explore the ISE design space in order to identify and assess the relative merits of different custom instructions.

Starting with a LISA description of the CRISP 5-stage pipeline, we equipped the processor with Harvard-style data and instruction RAMs, each of 4KB, and synthesised the generated VHDL code using Xilinx ISE 7.3. Our experimental platform was an ADM-XRC-II PCI card which hosts a Xilinx Virtex-II FPGA (XC2V6000-4FF1152) device with $33,000$ slices. The synthesis report indicated that the processor core can operate at a maximum clock frequency of 30 MHz and occupies a total of roughly $9,500$ slices. The integration of our proposed LUTs has no negative impact on the critical path delay and requires about 280 slices. In order to demonstrate correct in-circuit behaviour, we augmented the processor core to include an interface with Xilinx Chipscope. In terms of both performance and area we posit that there is room for improvement: the automatically generated VHDL code is not ideal in a number of cases. For instance, the register file and RAM components are implemented as distributed RAM instead of dedicated block RAM; this leads to a significant overhead in terms of slices occupied by RAM resources and to long routing delays. Moreover, the tool-chain is not able to identify exclusive read operations to the register file from different instructions; it generates a total of 20 read ports although at most four would be sufficient. As a consequence, the critical path of the design lies in this specific part of the implementation and not in the ALU which would allow an operating frequency of nearly 50 MHz. We plan to address these issues at a later stage of the project when the definition of the instruction set architecture has been finalised.

Regardless of the implementation quality, our functional processor model is sufficient to accurately assess the merits of our LUT-based ISE. We developed six case studies which represent different cryptographic primitives with different demands; the results presented below identify each algorithm, the potential for LUT-based acceleration within the algorithm, and compare implementation results (in terms of performance and memory footprint) versus a non-LUT-based alternative. It should be mentioned that bit-sliced ciphers use a non-traditional format to represent data; hence, the format conversion from standard into the bit-sliced domain introduces additional overhead before and after the encryption operation. However, in a closed environment the data can be kept in bit-sliced representation and so the need for a data conversion is omitted. In the following performance evaluation we do not consider the overhead caused by conversion to and from bit-slice representation.

3.1 SHA-1

SHA-1 is a cryptographic hash function which was designed and published by the NIST in 1995. Although SHA-1 is today considered to be cryptographically insecure, it is still employed in a vast range of standard applications and protocols such as SSL, SSH, and IPSec. The algorithm accepts an arbitrary length input message, split into 512-bit blocks, and produces a 160-bit message digest. The state of computation is held in five 32-bit chaining variables a, b, c, d and e which the algorithm processes in four rounds each composed of twenty operations. A different nonlinear function is used in each of the four rounds; for instance the nonlinear function for the third round is given by

$$f_3(a, b, c) : (a \wedge c) \vee ((a \vee b) \wedge c).$$

Using a conventional RISC processor the evaluation of this function takes four instructions; with our LUT-based approach the same function can be realised with one `ULUT` instruction plus one initial `CLUT` instruction to configure the LUT before the round starts. In Table 1, we compare results using our LUT-based approach with the performance of SHA-1 on the same CRISP pipeline without using LUTs; the ISE permits a performance improvement by a factor of 1.11 while code memory footprint is reduced by 21%.

Table 1. Implementation results for SHA-1 compression function

Implementation	Performance (cycles)	Code footprint (bytes)
Standard SHA-1	1602	2620
SHA-1 with LUTs	1441	2060

3.2 Multiplication of Ternary Polynomials

Fast arithmetic in finite fields of characteristic three is important for efficient pairing evaluation using particular parameterisations. In algorithms for pairing evaluation, multiplication in some extension field represents the time-critical operation; the performance of this operation in turn depends on the efficiency of the underlying base field arithmetic.

Using a polynomial basis representation, one can hold an element $a \in \mathbb{F}_{3^n}$ as two n-element bit-vectors a^H and a^L [19]. Using a_i^H and a_i^L to denote the i-th bit of a^H and a^L, respectively, the vectors a^H and a^L are constructed from a such that for all i

$$\begin{aligned} a_i^H &= a_i \text{ div } 2 \\ a_i^L &= a_i \bmod 2. \end{aligned}$$

That is, a^H and a^L are a bit-sliced representation of the coefficients of a where a^H and a^L hold the high and low bits of a given coefficient, respectively. Given such a representation, one can construct component-wise addition using logical operations. For example, a component-wise addition $r_i = a_i + b_i$ of two field elements a and b is specified by

$$\begin{aligned} r_i^H &= (a_i^L \vee b_i^L) \oplus t \\ r_i^L &= (a_i^H \vee b_i^H) \oplus t \end{aligned}$$

where $t = (a_i^L \vee b_i^H) \oplus (a_i^H \vee b_i^L)$.

Using a conventional RISC processor, the cost of each component-wise addition is seven logical operations; with our LUT-based approach the same addition can be collapsed to obtain the high and low bits with two `CLUT` instructions and one `ULUT` instruction. To demonstrate the impact of this, we implemented the comb method for field multiplication in $\mathbb{F}_{3^{97}}$ (the characteristic-two analogue is detailed in [18, Algorithm 2.35]). A summary of the results is shown in Table 2; in comparison to the CRISP processor without LUTs, the LUT-based approach improves performance by a factor of 1.51 while code memory footprint is reduced by 33%.

Table 2. Implementation results for multiplication in $\mathbb{F}_{3^{97}}$

Implementation	Performance (cycles)	Code footprint (bytes)
Standard Multiply	8652	2656
Multiply with LUTs	5750	1784

3.3 Serpent

Serpent was one of five finalists in the AES competition; it is a 32-round substitution-permutation block cipher that operates on 128-bit data blocks. Anderson et al. [1] describe an efficient bit-sliced implementation in which each round is constructed from three layers: a key mixing operation, an S-box operation, and a linear transformation operation. In particular, the S-box layer is realised using a sequence of logical instructions that are applied to four 32-bit input words to produce four output words; each S-box is represented, on average, by about 17 logical instructions.

Implementing each S-box operation on a conventional RISC processor is hampered by the resulting register pressure which, in turn, can enforce costly spills to memory. The advantage of using our LUTs for the S-box layer in Serpent is two-fold: firstly, a series of logical operations can be implemented with only four `CLUT` and two `ULUT` instructions; secondly, we reduce the number of temporary variables such that there is less need to spill values into memory. To further improve the performance of the Serpent encryption operation, specific portions of the linear transformation layer can also be implemented with LUTs.

In Table 3 we compare the LUT-based approach to the original, reference approach of Anderson et al. [1]. The LUT-based approach improves performance by a factor of 2.2 and reduces code memory footprint by 53%.

Table 3. Implementation results for Serpent encryption

Implementation	Performance (cycles)	Code footprint (bytes)
Bit-sliced Serpent	2031	2112
Bit-sliced Serpent with LUTs	922	984

3.4 AES

AES can, by design, be implemented efficiently on 8-bit or 32-bit platforms. In order to perform encryption (resp. decryption), the AES algorithm iteratively applies a round function (resp. inverse round function) to a 4×4 state matrix of elements in $\mathbb{F}_{2^8}$. The round function is composed of four steps: `SubBytes`, a non-linear substitution via an S-box that roughly equates to inversion in $\mathbb{F}_{2^8}$; `AddRoundKey`, the addition of key material via XOR; `ShiftRows`, which simply rotates rows of the state; and `MixColumns`, which multiplies columns of the state by a constant matrix.

An 8-bit implementation typically represents the state matrix as an array of sixteen bytes and implements each step of the round function in a direct fashion [11, Section 4.1]. A 32-bit implementation typically packs the columns of the state matrix into four words and combines the round function steps into a set of table look-ups [11, Section 4.2]. Previous work has developed effective alternatives for bit-sliced implementations of AES [24,25,21]. Könighofer [21] gives a detailed description of a fast bit-sliced AES implementation on a 64-bit AMD Opteron processor. In this work, the state matrix is stored in eight different registers throughout the encryption routine and four blocks are processed in parallel. We implemented Könighofer's method on our CRISP processor; the half-sized datapath width means we process two blocks at a time.

In a bit-sliced AES implementation, `SubBytes` represents the time-critical operation. In contrast to conventional implementations that usually store the S-box as a table in memory, the S-box is expressed by a series of logical operations according to the description of Canright [10]. The basic idea is to decompose the calculation of the multiplicative inverse in $\mathbb{F}_{2^8}$ into the calculation of the inverse in $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^2}$, respectively. Certain parts of these subfield computations can be mapped efficiently to our LUTs, for instance the inverse of $x = (x_0, x_1, x_2, x_3) \in \mathbb{F}_{2^4}$ is given by

$$\begin{aligned} e &= ((x_3 \oplus x_2) \wedge (x_1 \oplus x_0)) \oplus x_3 \oplus x_1 \\ d_1 &= (x_3 \wedge x_1) \oplus e \\ d_0 &= (x_2 \wedge x_0) \oplus e \oplus x_2 \oplus x_0 \end{aligned}$$

On a conventional RISC processor, the cost of computing the inverse in $\mathbb{F}_{2^4}$ is eleven instructions; using a LUT-based approach the computation can be performed with as little as two `CLUT` instructions and one `ULUT` instruction. Similar to the Serpent case, the LUTs are useful in terms of both reducing the number of logical instructions as well as reducing the spills into memory. However, the `ShiftRows` operation requires a closer examination. Each single byte within a register that holds the state needs to be rotated by a different distance; on a conventional RISC processor this can require a number of shift-and-mask type operations. To overcome this problem, one can integrate a custom instruction for efficient bit-level permutation, as proposed by Lee et al. [37], which reduces the cost of `ShiftRows` dramatically. The execution times of these implementations are given in Table 4; comparing our LUT-based implementation to the standard bit-sliced AES implementation of Könighofer [21], we improve performance by a factor of 1.23 and reduce code memory footprint by 36%. Having a dedicated instruction for efficient bit-level permutation further improves performance by a factor of 2.21 and reduces code memory footprint by some 59%. In [3], Bertoni et al. describe a fast non-bit-sliced software implementation of the AES for a 32-bit RISC processor. Comparing this implementation to the fastest bit-sliced version, our ISE permits a performance improvement by a factor of 1.36 on a per-block basis and reduces code memory footprint by 26%.

Table 4. Implementation results on a per-block basis for AES encryption

Implementation	Performance (cycles)	Code footprint (bytes)
Standard AES [3] (i.e. 32-bit)	1662	1160
Bit-sliced AES [21]	2699	2080
Bit-sliced AES with LUTs	2203	1328
Bit-sliced AES with LUTs & perm.	1222	858

3.5 PRESENT

Bogdanov et al. [6] describe PRESENT, a light-weight block cipher that can be efficiently implemented in hardware. PRESENT is a 31-round substitution-permutation network block cipher operating on 64-bit blocks. The S-box layer is realised as a table which maps 4-bit inputs to 4-bit outputs. In a bit-sliced implementation, the 64-bit blocks are stored in sixty four different words such that the i-th bit of each block is held in the i-th word; on a 32-bit datapath this allows us to process thirty two blocks in parallel. The S-box layer is expressed by a sequence of thirty logical operations; in a LUT-based implementation this is realised using four `CLUT` instructions and two `ULUT` instructions. The results are summarised in Table 5; compared to the reference implementation executed on CRISP, our ISE improve performance by a factor of 1.42 and reduce code memory footprint by 18%.

Table 5. Implementation results for PRESENT encryption

Implementation	Performance (cycles)	Code footprint (bytes)
Bit-sliced PRESENT	39986	500
Bit-sliced PRESENT with LUTs	28082	408

3.6 DES

The performance-critical operations in a standard DES software implementation are bit-oriented (e.g. permutation); in some sense this is a result of the hardware based origins of the algorithm. These sorts of operation are costly in software when implemented on a conventional RISC processor. As mentioned previously, Biham [5] described a fast implementation of DES using bit-slicing where the overhead caused by bit-oriented permutations is vastly reduced. Each S-box operation maps a 6-bit input to a 4-bit output and use of bit-slicing means their application is a bottleneck; the S-boxes require at most 132 logical operations and 100 instructions on average.

To reduce this cost, Kwan [22] presented an algorithm to generate S-boxes with an average of 56 logical operations. We examined each S-box using the Mimosys Clarity tool-chain [26] to identify where our ISE could be applied. The tool-chain takes C source code as input and analyses the data-flow graph to find subgraphs which can be implemented using a given ISA; in our case we had it search for 4-input, 2-output subgraphs consisting only of logical operations. The obtained speed-up factors of the bit-sliced DES S-boxes are detailed in Table 6; compared to Kwan's implementation we improve the performance of the S-box layer by a factor of 1.12 using the LUT-mechanism.

Table 6. Analysis of bit-sliced DES S-Boxes

S-Box	S1	S2	S3	S4	S5	S6	S7	S8
Speed-up factor	1.10	1.11	1.09	1.12	1.12	1.13	1.13	1.13

4 Comparison and Discussion

In recent years, custom instructions for secret-key cryptography have been integrated into a wide variety of platforms, ranging from high-performance ASIPs to

Table 7. Comparison of general-purpose processors with crypto extensions

Design	Base arch.	Algorithms and throughput (in cycles per byte)
MOSES [32]	Xtensa (32-bit)	3DES: 42.1 cpb, AES: 87.5 cpb
PAX [15]	RISC (64-bit)	3DES: 79.5, AES: 7.86, Twofish: 39.0, Mars: 85.21
O'Melia [27]	SPARC (32-bit)	3DES: 56.1 cpb, AES: 47.5 cpb, IDEA: 60.6 cpb
CRISP	RISC (32-bit)	SHA-1: 45.0 cpb, AES: 76.4 cpb, Serpent: 57.6 cpb

embedded processors optimised for small area and low power consumption. The plethora of target applications makes a fair comparison of the different designs very difficult, if not impossible. For example, Cryptonite [8] is an ASIP dedicated to cryptographic algorithms[1] and not a general-purpose processor with crypto extensions like CRISP. On the other hand, the custom instructions described in [4,13,35] were designed for integration into general-purpose processors, but their applicability is restricted to a single cryptosystem (AES), while the instructions introduced in this paper allow one to accelerate any cryptographic algorithm that can be implemented via bit-slicing.

Table 7 shows a comparison of CRISP with other general-purpose processors with crypto extensions which followed a similar design strategy, namely support of more than just a single cryptographic algorithm and orientation towards the embedded domain, which requires to consider both performance and hardware cost rather than focussing solely on the former. We omitted CryptoManiac as it is a 4-way VLIW processor optimised for high-bandwidth applications. Even though we restrict our comparison to closely related designs, the figures in Table 7 should be taken with a pinch of salt due to differences in the respective base architectures (e.g. 32-bit vs. 64-bit). MOSES supports only two secret-key algorithms (3DES and AES) while the instruction set of PAX is applicable to a wider range of algorithms of which seven were evaluated in [15] on basis of a 64-bit version of the architecture. The throughput figures of all four designs lie between 40 and 90 cycles per byte for the different algorithms, except of AES on PAX, which is extremely fast. In summary, the results of CRISP compare very well with that of previous work, especially when considering the flexibility and cost-efficiency of its crypto instructions.

5 Conclusions

We have presented a light-weight, generic instruction set extension for a 32-bit RISC processor with a 6-address instruction format (four source registers and two destination registers). Focusing on bit-sliced implementation, the ISE helps to address the disadvantages of this technique; for example, it improves performance and reduces code memory footprint, while maintaining all the advantages including low data memory footprint and predictable execution. Thanks to the

[1] The programmability of an ASIP is limited to applications within the application domain it has been designed for (e.g. cryptography), while a general-purpose processor can execute any kind of application.

generic nature of the proposed extensions, our processor architecture allows for a high degree of algorithm agility. This is a desirable feature when executing algorithm-independent security protocols, such as SSL/TLS or IPSec, where the support of several secret-key algorithms is essential. Moreover, the flexibility of our design can even be exploited by next-generation algorithms rather than being restricted to current-generation algorithms. In terms of hardware cost, the implementation of our ISE represents a modest overhead (just 280 slices of a Virtex-II device). Even though the proposed ISE might not be applicable in a high-performance processor design, it represents an excellent trade-off between implementation quality and cost for embedded and mobile processors.

Acknowledgements

The work described in this paper has been supported by the EPSRC under grant EP/E001556/1 and, in part, by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT. The information in this document reflects only the authors' views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Anderson, R., Biham, E., Knudsen, L.: Serpent: A proposal for the Advanced Encryption Standard. Technical report, `http://www.cl.cam.ac.uk/~rja14/serpent.html`
2. Bartolini, S., Branovic, I., Giorgi, R., Martinelli, E.: A performance evaluation of ARM ISA extension for elliptic curve cryptography over binary finite fields. In: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004), pp. 238–245. IEEE Computer Society Press, Los Alamitos (2004)
3. Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M., Marchesin, S.: Efficient software implementation of AES on 32-bit platforms. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 129–142. Springer, Heidelberg (2003)
4. Bertoni, G.M., Breveglieri, L., Farina, R., Regazzoni, F.: Speeding up AES by extending a 32-bit processor instruction set. In: Proceedings of the 17th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2006), pp. 275–279. IEEE Computer Society Press, Los Alamitos (2006)
5. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 260–272. Springer, Heidelberg (1997)
6. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
7. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Heidelberg (2006)

8. Buchty, R., Heintze, N., Oliva, D.: Cryptonite – A programmable crypto processor architecture for high-bandwidth applications. In: Müller-Schloer, C., Ungerer, T., Bauer, B. (eds.) ARCS 2004. LNCS, vol. 2981, pp. 184–198. Springer, Heidelberg (2004)
9. Burke, J., McDonald, J., Austin, T.: Architectural support for fast symmetric-key cryptography. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), pp. 178–189. ACM Press, New York (2000)
10. Canright, D.: A very compact S-box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
11. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Springer, Heidelberg (2002)
12. Davies, P.L., Robsky, S.R.: Customized processor extension speeds network cryptology. Electronic Design 50(19), 83–88 (2002)
13. Elbirt, A.J.: Fast and efficient implementation of AES via instruction set extensions. In: Proceedings of the 21st International Conference on Advanced Information Networking and Applications (AINA 2007), vol. 1, pp. 481–490. IEEE Computer Society Press, Los Alamitos (2007)
14. Fiskiran, A.M., Lee, R.B.: PAX: A datapath-scalable minimalist cryptographic processor for mobile devices. In: Embedded Cryptographic Hardware: Design and Security, pp. 19–34. Nova Science Publishers (2004)
15. Fiskiran, A.M., Lee, R.B.: On-chip lookup tables for fast symmetric-key encryption. In: Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2005), pp. 356–363. IEEE Computer Society Press, Los Alamitos (2005)
16. Großschädl, J., Savaş, E.: Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
17. Großschädl, J., Tillich, S., Szekely, A., Wurm, M.: Cryptography instruction set extensions to the SPARC V8 architecture (preprint submitted for publication, 2007)
18. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, Heidelberg (2004)
19. Harrison, K., Page, D., Smart, N.P.: Software implementation of finite fields of characteristic three, for use in pairing pased cryptosystems. LMS Journal of Computation and Mathematics 5(1), 181–193 (2002)
20. Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography
21. Könighofer, R.: A fast and cache-timing resistant implementation of the AES. In: Topics in Cryptology — CT-RSA 2008. LNCS, vol. 4964, pp. 187–202. Springer, Heidelberg (2008)
22. Kwan, M.: Reducing the gate count of bitslice DES. Cryptology ePrint Archive, Report 2000/051 (2000), `http://eprint.iacr.org`
23. Lee, R.B., Shi, Z., Yang, X.: Efficient permutation instructions for fast software cryptography. IEEE Mirco. 21(6), 56–69 (2001)
24. Matsui, M.: How far can we go on the x64 processors? In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 341–358. Springer, Heidelberg (2006)
25. Matsui, M., Nakajima, J.: On the power of bitslice implementation on Intel Core2 processor. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 121–134. Springer, Heidelberg (2007)
26. Mimosys. Clarity Product Datasheet (July 2006), `http://www.mimosys.com/pdf/Mimosys_Clarity_Product_Datasheet.pdf`

27. O'Melia, S.R.: Instruction Set Extensions for Enhancing the Performance of Symmetric-Key Cryptography. M.Sc. Thesis. University of Massachusetts, Lowell (2007)
28. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
29. Patterson, C.: A dynamic FPGA implementation of the Serpent block cipher. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 141–155. Springer, Heidelberg (2000)
30. Phillips, B.J., Burgess, N.: Implementing 1,024-bit RSA exponentiation on a 32-bit processor core. In: Proceedings of the 12th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000), pp. 127–137. IEEE Computer Society Press, Los Alamitos (2000)
31. Pozzi, L., Ienne, P.: Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In: Proceedings of the 8th International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2005), pp. 2–10. ACM Press, New York (2005)
32. Ravi, S., Raghunathan, A., Potlapally, N.R., Sankaradass, M.: System design methodologies for a wireless security processing platform. In: Proceedings of the 39th Design Automation Conference (DAC 2002), pp. 777–782. ACM Press, New York (2002)
33. Ravi, S., Raghunathan, A., Potlapally, N.R.: Securing wireless data: System architecture challenges. In: Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002), pp. 195–200. ACM Press, New York (2002)
34. Shi, Z., Lee, R.B.: Bit permutation instructions for accelerating software cryptography. In: Proceedings of the 12th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2000), pp. 138–148. IEEE Computer Society Press, Los Alamitos (2000)
35. Tillich, S., Großschädl, J.: Instruction set extensions for efficient AES implementation on 32-bit processors. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 270–284. Springer, Heidelberg (2006)
36. Wu, L., Weaver, C., Austin, T.M.: CryptoManiac: A fast flexible architecture for secure communication. In: Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA 2001), pp. 110–119. ACM Press, New York (2001)
37. Yang, X., Vachharajani, M., Lee, R.B.: Fast subword permutation instructions based on butterfly networks. In: Media Processors 2000. Proceedings of the SPIE, vol. 3970, pp. 80–86. SPIE (1999)
38. Yehia, S., Clark, N.T., Mahlke, S.A., Flautner, K.: Exploring the design space of LUT-based transparent accelerators. In: Proceedings of the 8th International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2005), pp. 238–249. ACM Press, New York (2005)