RISC-V Cryptographic Extension Proposals

Editors:

Version 0.2.0 (February 19, 2020)
`git:master @ 2de92f9c51006c0953d5d6289b6589df05164417`

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections):
Andy Glew, Barry Spinney, Ben Marshall, Derek Atkins, Ken Dockser, Markku-Juhani O. Saarinen, Nathan Menhorn, Richard Newell

Contents

1 Changelog

17/12/19 - Initial first version shown to Crypto Task group.

10/01/20
- Don't recommend funnel shifts for RV64, 128-bit rotations are not common enough to justify.
- Change LUT4 instruction. Go for very efficient RV64 instruction and split 4-to-2 32-bit instruction. Avoids ternary.

20/01/20
- LUT4 specification updates for 32-bit variant based on TG discussion.
- Initial performance code size impact for SHA256/512 instructions.

24/01/20 Split AES proposals into 3 variants.

19/02/20
- Revisions to AES proposals and added benchmarking results.
- Added SHA256/512 benchmark results.
- Added AES benchmark results.
- Codified scalar extension design policies.

2 Introduction

TODO #2.1: Write the introduction

Question #2.1: How does one detect the presence of the Crypto extension? The `MISA` CSR doesn't have an obvious letter assignment for Crypto yet. Given that the Crypto extension is in three distinct parts (Vector, Scalar, RNG) how should these individually be reported?

3 Vector Extension

TODO #3.1: Vector extension details

4 Scalar Extension

As per the RISC-V Cryptographic Extensions Task Group charter: "*The committee will also make ISA extension proposals for lightweight scalar instructions for 32 and 64 bit machines that improve the performance and reduce the code size required for software execution of common algorithms like AES and SHA and lightweight algorithms like PRESENT and GOST*".

For context, some these instructions have been developed based on academic work at the University of Bristol as part of the XCrypto project [11], and work by Paris Telecom on acceleration of lightweight block ciphers [18].

The scalar cryptography extension is designed with the following policies in mind. Their purpose is to help make design decisions consistent across the extension. Deviation from these policies is allowed if well justified.

Note #4.1: These policies *are up for discussion.*

Policy #1: Where there is a choice between: 1) supporting diverse implementation strategies for an algorithm or 2) supporting a single implementation style which is more performant / less expensive; the scalar crypto extension will pick the more constrained but performant option. This fits a common pattern in other parts of the RISC-V specification, where recommended (but not required) instruction sequences for performing particular tasks are given as an example, such that both hardware and software implementers can optimise for only a single use case.

Policy #2: The extension will be designed to well support *existing* standardised cryptographic constructs. It will not try to support proposed standards, or cryptographic constructs which exist only in academia. Cryptographic standards which are settled upon concurrently with, or after the RISC-V [1] cryptographic extension standardisation will be dealt with by future additions too, or versions of, the RISC-V cryptographic standard extension.

Policy #3: Historically, there has been some discussion [10] on how newly supported operations in general purpose computing might enable new bases for cryptographic algorithms. The standard will not try to anticipate new useful low level operations which *may* be useful as building blocks for future cryptographic constructs.

Policy #4: Regarding side-channel countermeasures: Where relevant, proposed instructions must aim to remove the possibility of any timing side-channels. For side-channels based on power or electro-magnetic (EM) measurements, the extension will not aim to support countermeasures which are implemented above the ISA abstraction layer. Recommendations will be given where relevant on how micro-architectures can implement instructions in a power/EM side-channel resistant way.

4.1 Shared Bitmanip Extension Functionality

Many of the primitive operations used in symmetric key cryptography and cryptographic hash functions are well supported by the RISC-V Bitmanip [13] extension [2]. We propose that the scalar cryptographic extension *reuse* a subset of the instructions from the Bitmanip extension directly. Specifically, this would mean that a core implementing *either* the scalar cryptographic extensions, *or* the Bitmanip extension, *or* both, would be able to depend on the existence of these instructions.

The following subsections give the assembly syntax of instructions proposed for inclusion in the scalar crypto extension, along with a set of use-cases for common algorithms or primitive operations. For information on the semantics of the instructions, we refer directly to the Bitmanip draft specification.

[1]It is anticipated that the NIST Lightweight Cryptography contest, and possibly the NIST Post-Quantum Cryptography contest may be dealt with this way, depending on timescales.

[2] At the time of writing, the Bitmanip extension is still undergoing standardisation. Please refer to the Bitmanip draft specification [12] directly for the latest information, as it may be slightly ahead of what is described here.

4.1.1 Rotations

RISC-V Bitmanip/Crypto ISA

```
RV32, RV64:
    ror    rd, rs1, rs2
    rol    rd, rs1, rs2
    rori   rd, rs1, imm

RV64 only:
    rorw   rd, rs1, rs2
    rolw   rd, rs1, rs2
    roriw  rd, rs1, imm
```

See [12, Section 3.1.1] for exact details of these instructions. Standard bitwise rotation is a primitive operation in many block ciphers and hash functions. It particularly features in the ARX (Add,Rotate,Xor) class of block ciphers [3].

Algorithms making use of 32-bit rotations: SHA256, AES (Shift Rows), ChaCha20

Algorithms making use of 64-bit rotations: SHA512, SHA3

RISC-V Bitmanip/Crypto ISA

```
RV32 only:
    fsl   rd, rs1, rs3, rs2
    fsr   rd, rs1, rs3, rs2
    fsri  rd, rs1, rs3, imm
```

See [12, Section 2.9.3] for exact details of these instructions. The *funnel shift* instructions create a 2∗`XLEN` word by concatenating `rs1` and `rs3`, which is then left/right rotate shifted by the amount in `imm`/`rs2`. These are useful for implementing double-width rotations. There are (currently) no examples of widely used algorithms which use anything larger than a 64-bit rotation, hence the funnel shift instructions are only proposed for inclusion on RV32, since RV64 will support 64-bit rotations naturally.

Algorithms using 64-bit rotations: SHA512, SHA3[4]

4.1.2 Other Permutations: `grev` and `shfl`

RISC-V Bitmanip/Crypto ISA

```
RV32, RV64:
    grev rd, rs1, rs2
    grevi rd, rs1, imm

RV64 only:
    grevw rd, rs1, rs2
    greviw rd, rs1, imm
```

The Generalised Reverse (`grev*`) instructions can be used for "*byte-order swap, bitwise reversal, short-order-swap, word-order-swap (RV64), nibble-order swap, bitwise reversal in a byte*". These operations are useful for various permutation operations needed either by block ciphers and hash-functions directly, or for endianness correction of data. Endianness correction is important because cryptography often occurs in the context of communication, which requires standardised endianness which may be different from the natural machine endianness. This instruction is also useful for getting data into the right form for being posted to a dedicated accelerator.

[3] `https://www.cosic.esat.kuleuven.be/ecrypt/courses/albena11/slides/nicky_mouha_arx-slides.pdf`

[4] SHA3 can avoid awkward double-width rotations using a technique called "Bit Interleaving". Benchmarking will be needed to see if this technique is a worthy mitigation for removing the funnel shifts.

RISC-V Bitmanip/Crypto ISA

```
RV32, RV64:
    shfl    rd, rs1, rs2
    unshfl  rd, rs1, rs2
    shfli   rd, rs1, rs2
    unshfli rd, rs1, rs2

RV64:
    shflw   rd, rs1, rs2
    unshflw rd, rs1, rs2
```

The generalised shuffle instructions are useful for implementing generic bit permutation operations. Algorithms such as DES [5] and PRESENT[5] with irregular / odd permutations are most-likely to benefit from this instruction.

TODO #4.1: More research needed on specific algorithms / use-cases for these instructions. They are included as "hypothetically useful" at the moment.

4.1.3 Carry-less Multiply

RISC-V Bitmanip/Crypto ISA

```
RV32, RV64:
    clmul rd, rs1, rs2
    clmulh rd, rs1, rs2
    clmulr rd, rs1, rs2

RV64 only:
    clmulw rd, rs1, rs2
    clmulhw rd, rs1, rs2
    clmulrw rd, rs1, rs2
```

See [12, Section 2.6] for exact details of this instruction. As is mentioned there, obvious cryptographic use-cases for carry-less multiply are for Galois Counter Mode (GCM) block cipher operations [6]. GCM is recommended by NIST as a block cipher mode of operation [9].

4.1.4 Conditional Move

RISC-V Bitmanip/Crypto ISA

```
RV32, RV64:
    cmov rd, rs2, rs1, rs3
```

See [12, Section 2.9.2] for exact details of this instruction. Conditional move is useful for implementing constant-time cryptographic code and avoiding control flow changes.

4.1.5 Logic With Negate

RISC-V Bitmanip/Crypto ISA

```
RV32, RV64:
    andn rd, rs1, rs2
     orn rd, rs1, rs2
    xorn rd, rs1, rs2
```

[5] One might reasonably argue that given the heritage of DES, it's support shouldn't really be any sort of consideration for a forward looking ISA like RISC-V.

[6] `https://en.wikipedia.org/wiki/Galois/Counter_Mode`

```
packh    a0, a0, a1          slli    a1, a1,  8
packh    a1, a2, a3          slli    a2, a2, 16
pack     a0, a0, a1          slli    a3, a3, 24
.                            or      a2, a2, a3
.                            or      a0, a0, a1
.                            or      a0, a0, a2
```

Figure 1: Comparison of packing four bytes loaded into GPRs $a0..a3$ into a single 32-bit word in $a0$ using the Bitmanip [pack*] instructions v.s. the standard RV32 instructions.

See [12, Section 2.1.3] for exact details of these instructions. These instructions are useful inside hash functions, block ciphers and for implementing software based side-channel countermeasures like masking.

Useful for: SHA3 Chi step, SHA2 Ch step and software based power/EM side-channel countermeasures based on masking.

4.1.6 Packing

```
RISC-V Bitmanip/Crypto ISA
RV32, RV64:
    pack   rd, rs1, rs2
    packu  rd, rs1, rs2
    packh  rd, rs1, rs2

RV64:
    packw  rd, rs1, rs2
    packuw rd, rs1, rs2
```

See [12, Section 2.1.4] for exact details of these instructions. Some lightweight block ciphers (e.g. SPARX [8]) use sub-word data types in their primitives. The Bitmanip pack instructions are useful for performing rotations on 16-bit data elements. They are also useful for re-arranging halfwords within words, and generally getting data into the right shape prior to applying transforms. This is particularly useful for cryptographic algorithms which pass inputs around as byte strings, but can operate on words made out of those byte strings. This occurs for AES when loading blocks and keys (which may not be word aligned) into registers to perform the round functions. See Figure 1 for an example.

Algorithms with sub-word rotations/shifts: SPARX

Algorithms benefiting from packing bytes into words: AES, SHA2, SHA3

4.2 LUT4 Instruction

```
RISC-V Crypto ISA
lut4lo  rd, rs1, rs2    // RV32, RV64
    for i = 0..7 : rd.4[i] = rs2.4[rs1.4[i]&0x7] if (rs1.4[i] <  8) else 0

lut4hi  rd, rs1, rs2    // RV32, RV64
    for i = 0..7 : rd.4[i] = rs2.4[rs1.4[i]&0x7] if (rs1.4[i] >= 8) else 0

lut4    rd, rs1, rs2    // RV32, RV64
    for i = 0..15 : rd.4[i] = rs2.4[rs1.4[i]]
```

The `lut4*` instructions are used to implement 4-bit lookup tables on every nibble in a source word. Many lightweight block ciphers use 4x4 SBoxes: PRINCE[6], PRESENT[5], Rectangle[20], GIFT[2], Twine[17], Skinny, MANTIS[4], Midori [3].

On RV32, the lookup step is split into two stages. The `lut4lo` instruction updates nibbles in the destination with the looked-up value iff the index is less than eight. The `lut4hi` version does the same for index values greater than/equal to eight. The results can then be or'd together. An example implementation of an 8-nibble parallel SBox using these instructions is found in Figure 2

On RV64, the entire set of LUT elements fits in a single source register. The RV64 only `lut4` instruction stores the entire lut in `rs2`, and uses each nibble in `rs1` as an index into it.

Equivalent C code listings for the instructions are found in Figure 6 of Appendix D.1.

Together, the instructions occupy 3 encoding points.

```
sbox_4bit:
    lut4lo   a3, a0, a1       // a0 = indexes, a1 = low  8 LUT nibbles
    lut4hi   a4, a0, a2       // a0 = indexes, a2 = high 8 LUT nibbles
    or       a0, a3, a4       // Or results together.
    ret                       // Function Return
```

Figure 2: Implement 8 parallel 4-to-4 bit SBox operations on RV32 using the `lut4hi` and `lut4lo` instructions. The Inputs to the SBox are stored in `a0` . The high and low 8 elements of the LUT are stored in `a2` and `a1` respectively.

4.3 Multi-precision Arithmetic

Multi-precision arithmetic is commonly used in public key cryptography. RISC-V struggles with long arithmetic for two reasons:

- A lack of carry / overflow detection. This hinders the arithmetic side of implementing multi-precision arithmetic.
- A lack of indexed load and store instructions. Multi-precision arithmetic typically involves stepping through at-least three different arrays at different indices: two operands and a result. This results in a lot of pointer arithmetic at the end of loops on RISC-V.

Question #4.1: These instructions introduce the "double width write-back" idiom to RISC-V. How acceptable is this? Which other instructions could take advantage of this? How well does this overlap with what the P (DSP) extension task group is proposing?

TODO #4.2: Benchmarking flow for these instructions based on long-multiply / modular exponentiation.

4.3.1 Multi-precision Multiply Accumulate Unsigned

```
RISC-V Crypto ISA
RV32, RV64:
    mmulu   rdp, rs1, rs2, rs3  // Variant 1 - double width write-back

    mmulu   rd,  rs1, rs2, rs3  // Variant 2
    mmuluh  rd,  rs1, rs2, rs3  //  - Hi/LO
```

```
// Equivalent RV32IM / RV64IM assembly code listing. rdp = (rd2,rd1)
mul      t1  , rs1, rs2   //         t1   = low(rs1 * rs2)
mulhu    rd2 , rs1, rs2   // (rd2, t1)  =     rs1 * rs2
```

```
add     rd1  , t1  , rs3   //      rd1   = low(rs1 * rs2) + rs3
sltu    t1   , rd1 , t1    //       t1   = rd1 < t1
add     rd2  , rd2 , rs2   //      rd2  += t1
```

The mmulu instruction performs an unsigned multiply of two XLEN sources, forming a 2∗XLEN result. The third XLEN source register is then added to the 2∗XLEN result.

For variant 1, the 2∗XLEN result is then written back to an odd-even register pair **rdp**.

For variant 2, the low or high XLEN bits are written to the destination register for the **mmulu** or **mmuluh** instructions respectively.

4.3.2 Multi-precision Accumulate Unsigned

```
                                          RISC-V Crypto ISA
RV32, RV64:
    maccu   rdp, rs1, rs2, rs3  // Variant 1 : rdp = (rs1 || rs2) + rs3
    maccu   rdp, rs1            // Variant 2 : rdp =  rdp         + rs3
```

```
// Equivalent RV32IM / RV64IM assembly code listing. rdp = (rd2,rd1)
add     rd1, rs2, rs3
sltu    rs1, rd1, rs2
add     rd2, rs2, rs1    // (rd1,rd2) = (rs1,rs2)+rs3
```

Variant 1 of the **maccu** instruction concatenates source registers **rd2** and **rd1** to create a 2∗XLEN word. To this, **rs3** is added, and the full double-XLEN result is written back to an odd/even register pair **rdp**.

Variant 2 does the same, but uses **rdp** as an operand and is destructive.

Note #4.2: This instruction is designed to be used in conjunction with **mmulu** .

```
saes.v1.enc(rs1):
    rd.8[0] =    AESSBox(rs1.8[0])
    rd.8[1] =    AESSBox(rs1.8[1])
    rd.8[2] =    AESSBox(rs1.8[2])
    rd.8[3] =    AESSBox(rs1.8[3])
```

(a) Forward SBox instruction pseudo code.

```
saes.v1.dec(rs1):
    rd.8[0] = InvAESSBox(rs1.8[0])
    rd.8[1] = InvAESSBox(rs1.8[1])
    rd.8[2] = InvAESSBox(rs1.8[2])
    rd.8[3] = InvAESSBox(rs1.8[3])
```

(b) Inverse SBox instruction pseudo code.

Figure 3: AES Instructions variant 1.

.

4.4 Lightweight AES Acceleration

This section details 3.5 proposals for lightweight acceleration of the AES block cipher [1].

Performance and RTL benchmarks for these instruction proposals are found in Section A.1.

4.4.1 Variant 1

RISC-V Crypto ISA

```
RV32, RV64:
    saes.v1.enc rd, rs1
    saes.v1.dec rd, rs1
```

These instructions implement the `SubBytes` [1, Section 5.1.1] and `InvSubBytes` [1, Section 5.3.1] steps of the AES Block Cipher [1]. The low 32-bits of `rs1` are split into bytes. Each byte has the relevant transformation applied, before being written back to the corresponding byte position in `rd`. On an RV64 platform, the high 32-bits of the result are zero extended. Pseudo code is found in figure 3.

This proposal variant requires 2 encoding points with only one source register.

4.4.2 Variant 2

RISC-V Crypto ISA

```
RV32, RV64:
    saes.v2.sub.enc    rd, rs1, rs2
    saes.v2.sub.dec    rd, rs1, rs2
    saes.v2.mix.enc    rd, rs1, rs2
    saes.v2.mix.dec    rd, rs1, rs2
```

These instructions are derived from [11], which in turn adapted them originally from [19]. Each instruction performs either the SubBytes or MixColumns transformation to a single column of the AES state, along with a partial ShiftRows. Example usage can be found in the riscv-crypto repository [7].

Pseudo-code for the sub-bytes and mix-columns instructions are found in figures 4a and 4b respectively.

This proposal variant requires 4 encoding points, where each point requires two source registers and a destination register.

[7] `https://github.com/scarv/riscv-crypto/blob/master/benchmarks/crypto_block/aes/zscrypto_v2/aes_enc.c#L53`

```
saes.v2.sub(rs1, rs2, mode):
    if(mode == enc)
        t0 =    AESSBox(rs1.8[0]), t1 =    AESSBox(rs2.8[1])
        t2 =    AESSBox(rs1.8[2]), t3 =    AESSBox(rs2.8[3])
    else
        t0 = InvAESSBox(rs1.8[0]), t1 = InvAESSBox(rs2.8[1])
        t2 = InvAESSBox(rs1.8[2]), t3 = InvAESSBox(rs2.8[3])
    rd.32 = {t3, t2, t1, t0}
```

(a) AES instruction variant 2: sbox instruction pseudo code.

```
saes.v2.mix(rs1, rs2, mode):
    t0 = rs1.8[0], t1 = rs1.8[1]
    t2 = rs2.8[2], t3 = rs2.8[3]
    if(mode == enc)
        rd.32 =    AESMixColumns(t3,t2,t1,t0)
    else
        rd.32 = InvAESMixColumns(t3,t2,t1,t0)
```

(b) AES instruction variant 2: Mix columns instruction pseudo code.

Figure 4: AES Instructions: Variant 2 Pseudo Code.

4.4.3 Variant 3

```
RISC-V Crypto ISA
RV32, RV64:
    saes.v3.encs      rd, rs1, rs2, bs // Encrypt: SubBytes
    saes.v3.encm      rd, rs1, rs2, bs // Encrypt: MixColumns
    saes.v3.encsm     rd, rs1, rs2, bs // Encrypt: SubBytes & MixColumns
    saes.v3.decs      rd, rs1, rs2, bs // Decrypt: SubBytes
    saes.v3.decm      rd, rs1, rs2, bs // Decrypt: MixColumns
    saes.v3.decsm     rd, rs1, rs2, bs // Decrypt: SubBytes & MixColumns
```

These instructions are a very lightweight proposal, derived from [14]. They are designed to enable a partial T-Table based implementation of AES in hardware, where the SubBytes, ShiftRows and MixColumns transformations are all rolled into a single instruction, with the per-byte results then accumulated. The `bs` immediate operand is a 2-bit *Byte Select*, and indicates which byte of the input word is operated on. In contrast to variants 1 and 2, they perform only a single (Inverse) SBox lookup per instruction. Pseudo-code for each instruction is found in figure 5.

Only 4 of the 6 instructions are strictly needed: one of the following two subsets would suffice:

v3.1: `saes.v3.encs` , `saes.v3.encsm` , `saes.v3.decs` and `saes.v3.decsm` . This set is more performant in terms of instructions executed per round. It also places the SubBytes and MixColumns operations sequentially, and so has a longer timing path.

v3.2: `saes.v3.encs` , `saes.v3.encm` , `saes.v3.decs` and `saes.v3.decm` . This set splits the SubBytes and MixColumns operations into separate instructions. This means more instructions per round, but a shorter critical timing path for the instruction.

Both the v3.1 and V3.2 proposal variants require 4 fixed encoding points with two source registers, one destination register and a single 2-bit immediate. This makes for a total of 16 encoding points.

```
saes.v3.encs(rs1, rs2, bs):                       // SubBytes Only
    t0.8   = rs1.8[bs]
    x.8    = AESSBox(t0)
    u.32   = {0, 0, 0, x}
    rd.32 = ROTL32(u, 8*bs) ^ rs2.32

saes.v3.encm(rs1, rs2, bs):                       // MixColumns only
    x.8    = rs1.8[bs]
    u.32   = {GFMUL(x,3) , x, x, GFMUL(x,2)}
    rd.32 = ROTL32(u, 8*bs) ^ rs2.32

saes.v3.encsm(rs1, rs2, bs):                      // SubBytes & MixColumns
    t0.8   = rs1.8[bs]
    x.8    = AESSBox(t0)
    u.32   = {GFMUL(x,3) , x, x, GFMUL(x,2)}
    rd.32 = ROTL32(u, 8*bs) ^ rs2.32
```

(a) Encrypt instruction pseudo code.

```
saes.v3.decs(rs1, rs2, bs):                       // InvSubBytes Only
    t0.8   = rs1.8[bs]
    x.8    = InvAESSBox(t0)
    u.32   = {0, 0, 0, x}
    rd.32 = ROTL32(u, 8*bs) ^ rs2.32

saes.v3.decm(rs1, rs2, bs):                       // InvMixColumns Only
    x.8    = rs1.8[bs]
    u.32   = {GFMUL(x,11),GFMUL(x,13),GFMUL(9),GFMUL(14)}
    rd.32 = ROTL32(u, 8*bs) ^ rs2.32

saes.v3.decsm(rs1, rs2, bs):                      // InvSubBytes & InvMixColumns
    t0.8   = rs1.8[bs]
    x.8    = InvAESSBox(t0)
    u.32   = {GFMUL(x,11),GFMUL(x,13),GFMUL(9),GFMUL(14)}
    rd.32 = ROTL32(u, 8*bs) ^ rs2.32
```

(b) Decrypt instruction pseudo code.

Figure 5: AES instruction variant 3.

4.5 Lightweight SHA2 Acceleration

```
RISC-V Crypto ISA
RV32, RV64:
    ssha256.s0 rd, rs1 : rd = ror32(rs1, 7) ^ ror32(rs1, 18) ^ srl32(rs1, 3)
    ssha256.s1 rd, rs1 : rd = ror32(rs1,17) ^ ror32(rs1, 19) ^ srl32(rs1,10)
    ssha256.s2 rd, rs1 : rd = ror32(rs1, 2) ^ ror32(rs1, 13) ^ ror32(rs1,22)
    ssha256.s3 rd, rs1 : rd = ror32(rs1, 6) ^ ror32(rs1, 11) ^ ror32(rs1,25)

RV64:
    ssha512.s0 rd, rs1 : rd = ror64(rs1, 1) ^ ror64(rs1,  8) ^ srl64(rs1, 7)
    ssha512.s1 rd, rs1 : rd = ror64(rs1,19) ^ ror64(rs1, 61) ^ srl64(rs1, 6)
    ssha512.s2 rd, rs1 : rd = ror64(rs1,28) ^ ror64(rs1, 34) ^ ror64(rs1,39)
    ssha512.s3 rd, rs1 : rd = ror64(rs1,14) ^ ror64(rs1, 18) ^ ror64(rs1,41)
```

The `ssha256.sX` instructions implement the core of the four sigma and sum functions used in the SHA256 hash function [15, Section 4.1.2]. These operations will be supported for a both RV32 and RV64 targets. For RV32, the entire XLEN source register is operated on. For RV64, the low 32-bits of the XLEN register are read and operated on, with the result zero extended to XLEN bits. Though named for SHA256, the instructions work for both the SHA-224 and SHA-256 parameterisations as described in [15].

The `ssha512.sX` instructions implement the core of the four sigma and sum functions used in the SHA512 hash function [15, Section 4.1.3]. These operations will be supported for RV64 targets only. Though named for the SHA-512 parameterisation, the instructions can be used for all of the SHA-384, SHA-512, SHA-512/224 and SHA-512/256 parameterisations as described in [15].

Performance, static code size and RTL benchmarks for these instructions are found in Section A.2. Together, the instructions occupy 8 encoding points.

Note #4.3: The remaining two core functions which make up the SHA256/512 hash functions are the *Ch* and *Maj* functions:

- Ch(x,y,z) = (x & y) ˆ (˜x & z)
- Maj(x,y,z) = (x & y) ˆ (x & z) ˆ (y & z)

As ternary functions, they are much too expensive in terms of opcode space to consider for inclusion as dedicated instructions for such a specialist use case. They are amenable however to macro-op fusion on cores which implement it.

4.6 Lightweight SHA3 Acceleration

Note #4.4: These instructions may be withdrawn as proposals, since in practice implementations of the SHA3 round function are usually unrolled, which removes the use-case for these instructions.

```
RISC-V Crypto ISA
RV32, RV64:
    ssha3.xy rd, rs1, rs2 : rd = (( rs1    % 5) + 5(           rs2 % 5)) << 3
    ssha3.x1 rd, rs1, rs2 : rd = (((rs1+1) % 5) + 5(           rs2 % 5)) << 3
    ssha3.x2 rd, rs1, rs2 : rd = (((rs1+2) % 5) + 5(           rs2 % 5)) << 3
    ssha3.x4 rd, rs1, rs2 : rd = (((rs1+4) % 5) + 5(           rs2 % 5)) << 3
    ssha3.yx rd, rs1, rs2 : rd = (( rs2    % 5) + 5((2*rs1 + 3*rs2)% 5)) << 3
```

These instructions accelerate code-dense implementations of the SHA3 secure hash function [16]. They work on the low 3 bits of the input `rs1` and `rs2` registers, and compute indices into the state array of the round function. They are designed to replace both canonical implementations of the `index` function and lookup table based implementations. In both cases, these instructions offer substantial improvements in performance, static code size and dynamic instruction bandwidth. A longer discussion of the merits of these instructions can be found in Appendix B.

Note #4.5: SHA3 is not yet widely used, especially with respect to SHA2. This can be expected to change over time, particularly as many of the NIST Post-Quantum Cryptography candidates use SHA3 as an underlying construct.

Note #4.6: If the auto-aligning indexed load and store instructions are included, then the auto-aligning component of these instructions may not be needed.

4.7 Lightweight SM4 Acceleration

Note #4.7: This section is a placeholder. It will contain a proposal for accelerating the SM4 block cipher, based on work in [14].

4.8 Auto-aligning Indexed Load and Store

```
RISC-V Crypto ISA
RV32, RV64:
    lbx     rd,  rs1, rs2 : rd = sext(mem[rs1 + (rs2     )])
    lbux    rd,  rs1, rs2 : rd = zext(mem[rs1 + (rs2     )])
    lhx     rd,  rs1, rs2 : rd = sext(mem[rs1 + (rs2 << 1)])
    lhux    rd,  rs1, rs2 : rd = zext(mem[rs1 + (rs2 << 1)])
    lwx     rd,  rs1, rs2 : rd = sext(mem[rs1 + (rs2 << 2)])
    sbx     rs1, rs2, rs3 : mem[rs1 + (rs3     )] = rs2
    shx     rs1, rs2, rs3 : mem[rs1 + (rs3 << 1)] = rs2
    swx     rs1, rs2, rs3 : mem[rs1 + (rs3 << 2)] = rs2

RV64:
    lwux    rd,  rs1, rs2 : rd = zext(mem[rs1 + (rs2 << 2)])
    ldx     rd,  rs1, rs2 : rd = sext(mem[rs1 + (rs2 << 3)])
    sdx     rs1, rs2, rs3 : mem[rs1 + (rs3 << 3)] = rs2
```

Note #4.8: There is a longer discussion about the inclusion of indexed load and store in the context of RISC-V in Appendix C.

These instructions add indexed load and store functionality to RISC-V. The load instructions take a base register `rs1` and an offset register `rs2` to form an address. The offset is aligned to the data type of the instruction. Zero and sign extending variants are provided. The store instructions do the same to form the effective address, but use `rs3` as the auto-aligned offset. This pattern of register usage keeps the functions of `rs1` and `rs2` the same as the immediate offset load and store instructions.

Indexed load and store are useful for implementing many cryptographic algorithms. For public key cryptography involving modular exponentiation, iterating independently over several arrays without lots of pointer arithmetic is beneficial. For hash functions and symmetric key block ciphers, irregular or non-sequential access to a state array also benefits from being able to directly calculate an address based on an offset in a register.

4.9 Micro-architectural Recommendations

TODO #4.3: Macro-op fusion suggestions, side-channel considerations.

5 Random Bit Generation extension

TODO #5.1: Random bit generator details

References

[1] *Advanced Encryption Standard (AES).* National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197. 2001. URL: `https://www.nist.gov/publications/advanced-encryption-standard-aes`.

[2] Subhadeep Banik et al. "GIFT: a small present". In: *International Conference on Cryptographic Hardware and Embedded Systems.* Springer. 2017, pp. 321–345.

[3] Subhadeep Banik et al. "Midori: A block cipher for low energy". In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2015, pp. 411–436.

[4] Christof Beierle et al. "The SKINNY family of block ciphers and its low-latency variant MANTIS". In: *Annual International Cryptology Conference.* Springer. 2016, pp. 123–153.

[5] Andrey Bogdanov et al. "PRESENT: An ultra-lightweight block cipher". In: *International workshop on cryptographic hardware and embedded systems.* Springer. 2007, pp. 450–466.

[6] Julia Borghoff et al. "PRINCE–a low-latency block cipher for pervasive computing applications". In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2012, pp. 208–225.

[7] Christopher Celio et al. "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V". In: *arXiv preprint arXiv:1607.02318* (2016).

[8] Daniel Dinu et al. "Design strategies for ARX with provable bounds: Sparx and LAX". In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2016, pp. 484–513.

[9] Morris Dworkin. "NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC". In: *US National Institute of Standards and Technology http://csrc. nist. gov/publications/nistpubs/800-38D/SP-800-38D. pdf* (2007).

[10] Ruby B Lee et al. "On permutation operations in cipher design". In: *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.* Vol. 2. IEEE. 2004, pp. 569–577.

[11] B. Marshall, D. Page, and T. Pham. *XCrypto: a cryptographic ISE for RISC-V.* Tech. rep. 1.0.0. 2019. URL: `https://github.com/scarv/xcrypto`.

[12] *RISC-V Bit manipulation extension draft proposal.* URL: `https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-draft.pdf`.

[13] *RISC-V Bit manipulation extension repository.* URL: `https://github.com/riscv/riscv-bitmanip`.

[14] Markku-Juhani O. Saarinen. *Lightweight AES ISA.* `https://github.com/mjosaarinen`. Retrieved 24/01/2020. Jan. 2020.

[15] *Secure Hash Standard (SHS).* National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 180-4. 2015. URL: `https://csrc.nist.gov/publications/detail/fips/180/4/final`.

[16] *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 202. 2015. URL: `https://csrc.nist.gov/publications/detail/fips/202/final`.

[17] Tomoyasu Suzaki et al. "TWINE: A Lightweight Block Cipher for Multiple Platforms". In: *International Conference on Selected Areas in Cryptography.* Springer. 2012, pp. 339–354.

[18] Etienne Tehrani et al. "Classification of Lightweight Block Ciphers for Specific Processor Accelerated Implementations". In: *26th IEEE International Conference on Electronics Circuits and Systems.* Nov. 2019.

[19] Stefan Tillich and Johann Großschädl. “Instruction set extensions for efficient AES implementation on 32-bit processors”. In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2006, pp. 270–284.

[20] Wentao Zhang et al. “RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms”. In: *Science China Information Sciences* 58.12 (2015), pp. 1–15.

A Benchmarking

A.1 AES

The source code for these benchmarks is available as part of the `riscv-crypto` GitHub repository.

AES Proposal	Instrs Enc	Instrs Dec	Instrs Ks Enc	Instrs Ks Dec	Static Code Size (Bytes)
Reference	3260	6704	594	594	6233
TTable	1005	1012	625	1863	14353
V1 Latency	2756	6537	462	462	3575
V1 Size	2874	6681	510	510	-
V2 Latency	325	307	462	581	1867
V2 Size	552	539	462	687	-
V3.1	321	321	238	682	1952
V3.2	480	470	238	826	2264

Table 1: AES instruction execution counts measured using Spike running `rv32imcb_Zscrypto`. Instruction counts are for a single AES 128 block encrypt/decrypt operation, or for a single encrypt/decrypt key schedule. V1 and V2 proposals include results for "latency" optimised and "size" optimised implementations of the underlying instructions. The "latency" rows assume one cycle per instruction. The "size" rows assume four cycles per instruction (corresponding to one implemented SBox), which is modelled by padding each AES instruction with 3 additional `nop` instructions. Static code size is reported as the sum of all `.text` and `.data` sections.

AES Proposal	Longest Topological Path	NAND2 Gates
V1 Latency	12	7461
V1 Size	16	2218
V2 Latency	13	8626
V2 Size	15	2487
V3.1	24	2495
V3.2	18	2455

Table 2: AES proposal RTL implementation benchmarks. Measured using the Yosys simple CMOS flow. The V1 and V2 proposals have "Latency" and "Size" optimised versions. The "Latency" versions use 4 SBox instantiations and compute their results in a single clock cycle. The "Size" versions use 1 SBox instantiation, and compute their results over 4 clock cycles.

A.2 SHA2

The source code for the SHA256[8] and SHA512[9] benchmarks are available as part of the `riscv-crypto` GitHub repository.

Architecture	Static Code Size (Bytes)	Instructions Executed	Performance Gain
rv32gc	14934	78003	1.00x
rv32gcb	10086	56866	1.37x
rv32gcb_Zscrypto	5938	28539	2.73x

Table 3: **SHA256:** Static code size and instructions executed comparison for the `ssha256.sx` instructions on RV32 based architectures. Instruction execution counts are for hashing 1024 bytes of data using SHA256.

Architecture	Static Code Size (Bytes)	Instructions Executed	Performance Gain
rv64gc	20490	73138	1.00x
rv64gcb	14216	53153	1.38x
rv64gcb_Zscrypto	8954	35881	2.04x

Table 4: **SHA512:** Static code size and instructions executed comparison for the `ssha512.sx` instructions on RV64 based architectures. Instruction execution counts are for hashing 1024 bytes of data using SHA512.

Instructions	Longest Topological Path	Area (NAND2 Equivalent)
ssha256.s*	5	787
ssha512.s*	6	1534

Table 5: Area and path length estimates are calculated using a basic Yosys CMOS synthesis flow.

[8]`https://github.com/scarv/riscv-crypto/tree/master/benchmarks/crypto_hash/sha256`
[9]`https://github.com/scarv/riscv-crypto/tree/master/benchmarks/crypto_hash/sha512`

Architecture	Flags	`.text` Bytes	Instructions Executed	Fetch Bandwidth	Data Bandwidth
`rv32im`	-O2	688			
`rv32imc`	-O2	558			
`rv64im`	-O2	504			
`rv64imc`	-O2	354			
`rv32im`	-O3	3328			
`rv32imc`	-O3	2812			
`rv64im`	-O3	1292			
`rv64imc`	-O3	1034			

Table 6: Table of code size and performance comparisons for the SHA3 algorithm, implemented on various RISC-V architecture variants.

B SHA3 Instruction Discussions

This discussion follows on from the instruction specifications in section 4.6.

Note #B.1: These instructions may be withdrawn as proposals, since in practice implementations of the SHA3 round function are usually unrolled, which removes the use-case for these instructions.

The SHA3 secure hash function [16] is based on the KECCAK-P family of permutations. SHA3 is notably slower than SHA2 when implemented in software. It also has a large state size (1600 bits) which is very irregularly accessed, making it difficult to accelerate the core *compute* operations as part of a scalar CPU pipeline. We distinguish between *compute* operations (which modify the round function state) and *address* operations (which calculate indexes into the round function state) when motivating these instructions.

The core operations of the KECCAK-P round function are rotations and XORs, which are already well supported by the RISC-V base and Bitmanip architectures. The round function state is accessed as a $5*5$ array of 64-bit words. When developing lightweight accelerator instructions for SHA3, we consider two broad implementation options:

- Loop-unrolled: Here, all of the loops of the round function are unrolled, meaning that all variations of the index function are computed at compile time, and are emitted as immediate offsets to load and store instructions. In this case, there is little that can be added to a scalar pipeline to accelerate SHA3, other than the bitwise rotation instructions (for RV64) or funnel shift instructions (for RV32). The proposed instructions do not benefit a loop-unrolled implementation.

- Loop-rolled-up: The loops are not unrolled, and the index functions are re-computed on every loop iteration. This means that *either* `rem` instructions are used to compute the modulo 5 operations, or they can be replaced with a lookup table. In both cases, the extra number of instructions executed is substantial. The proposed instructions strongly benefit a loop-rolled-up implementation. This has obvious benefits for embedded applications where code-density is an issue. It may also benefit larger implementations due to better use of instruction caches, especially in the presence of a good branch predictor.

The proposed instructions are *extremely* small to implement, since they only read the low 3 bits of the two source registers, and the result is only ever 5 bits wide prior to double-word alignment and 8 bits afterwards. An example implementation [10] synthesised to a simple CMOS cell library using Yosys comes to approximately 300 cells. Note that this is only the cost of dedicated instruction logic. Decode costs are not included.

[10]`https://github.com/scarv/xcrypto/blob/dev/ben/issue-73/rtl/xc_sha3/xc_sha3.v`

TODO #B.1: Benchmarking flow for SHA3 and the KeccakP1600 round function: Code size, performance, instructions executed, instruction bandwidth, data bandwidth etc.

C Indexed Load and Store Discussion

Note #C.1: The following section is included to stimulate discussion. There are good engineering arguments for and against indexed load and store in the context of RISC-V which this section aims to capture.

RISC-V very deliberately omits indexed load and store instructions from the base architecture [7]. The principle arguments for this are:

- That the extra register read port for the 3-operand store instructions are an unacceptable burden on smaller micro-architectures.
- That macro-op fusion is a sufficient mitigation for the performance penalty. In [7, Sections V, VI], the authors use the example of indexed load to demonstrate this. Their results using the SPECInt benchmarks are very encouraging.

With the proposed inclusion of ternary instructions as part of Bitmanip, the first argument no longer holds as much (or any) weight for CPUs which are implementing many/any of the ternary instructions anyway.

The second argument then requires further discussion. Note that we do not argue against macro-op fusion in general, it is still a useful technique for optimising instruction execution and maintaining ISA cleanliness [11]. In the case of indexed load and store however, it's merits are less certain, both generally, and in the case of RISC-V specifically.

- For narrow memory bus widths (32-bits), one can only fuse adjacent 16-bit opcodes. This can be mitigated with an instruction fetch buffer and the associated costs this brings.
- In [7], the authors focus on fusing opcodes from the compressed ISA. While this makes sense, it severely limits the registers available due to the limited addressing capabilities of the RVC instructions.
- There is no reason 32-bit opcodes cannot be fused, but this implies much wider memory busses, deeper fetch buffers, or both.
- It is an open question how a compiler tuned to generate fusion pairs will interact with a core which doesn't implement fusion. Intuitively, this may mean activating forwarding paths much more often. This may result in more toggling and hence energy consumption. We could find no empirical evidence one way or the other on this.
- The recommended fusable sequence for an auto-aligning indexed load in [7, Section VI.A] is three 16-bit compressed instructions. This is a criterion used by the Bitmanip extension to measure whether instructions are worthy of inclusion: does it replace three instructions, or two instructions and is very commonly used. We would argue that an auto-aligning indexed load and store meets both of these.
- A fused store instruction sequence requires that the calculated address be written back to the GPRs, even if the address is never used again.

[11]For some definition thereof.

- Certain bus standards (AMBA AHB, AXI) which have separate address and data phases or channels, possibly removing the need for indexed stores to access all three operands simultaneously. This implies a possible performance penalty.
- While these instructions are very useful generally, they have particular usefulness in Cryptography. In the case of public key cryptography, one typically needs to do large amounts of multi-precision arithmetic. Basic schemes for this rely on iterating over at-least three different arrays with different indices, for which indexed load and store are very useful. More complex modular exponentiation schemes iterate less regularly over the input/output arrays, making the pointer arithmetic involved even more burdensome. In block ciphers and hash functions which are not loop-unrolled, non-sequential access to a state array is also a common idiom.
- Given the results in [7], it is clear that macro-op fusion is a good scheme for enhancing implementations of RISC-V. It does not offer a comparison of performance based on a hypothetical extension to RISC-V which does include auto-aligning indexed load and store. This work and associated results are essential for a full discussion on the matter.

TODO #C.1: Repeat the work of [7], but include a version of RISC-V with the auto-aligning instructions described in section 4.8.

D Equivalent C-Code Listings

D.1 LUT4

```c
uint32_t lut4_lo (uint32_t rs1, uint32_t rs2) {
    uint32_t result = 0;
    for(int i = 0; i < 32; i += 4) {
        uint8_t idx =  (rs1 >> i) & 0x7;
        uint8_t lo  = ((rs1 >> i) & 0xF) < 8;
        if(lo) { result |= ((rs2 >> (4*idx) & 0xF) << i; }
    }
    return result;
}
uint32_t lut4_hi (uint32_t rs1, uint32_t rs2) {
    uint32_t result = 0;
    for(int i = 0; i < 32; i += 4) {
        uint8_t idx =  (rs1 >> i) & 0x7;
        uint8_t hi  = ((rs1 >> i) & 0xF) > 8;
        if(hi) { result |= ((rs2 >> (4*idx) & 0xF) << i; }
    }
    return result;
}
uint64_t lut4 (uint64_t rs1, uint64_t rs2) {
    uint64_t result = 0;
    for(int i = 0; i < 64; i += 4) {
        result |= ((rs2 >> (4*((rs1 >> i)&0xF)) & 0xF) << i;
    }
    return result;
}
```

Figure 6: Equivalent C code models for the `lut4lo`, `lut4hi` and `lut4` instructions.

E Tentative Encoding Proposals

31 30 29 28 27 26 25		24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7		6 5 4 3 2	1 0	
0110000		rs2	rs1	000	rd		01010	11	lut4lo
0110001		rs2	rs1	000	rd		01010	11	lut4hi
0110010		rs2	rs1	000	rd		01010	11	lut4
rs3	01	rs2	rs1	001	rdp	0	01010	11	mmulu
rs3	00	rs2	rs1	001	rdp	0	01010	11	maccu
0001110		00000	rs1	010	rd		01010	11	saes.v1.enc
0001111		00001	rs1	010	rd		01010	11	saes.v1.dec
0001011		rs2	rs1	010	rd		01010	11	saes.v2.sub.enc
0001001		rs2	rs1	010	rd		01010	11	saes.v2.sub.dec
0000111		rs2	rs1	010	rd		01010	11	saes.v2.mix.enc
0000110		rs2	rs1	010	rd		01010	11	saes.v2.mix.dec
bs	00101	rs2	rs1	010	rd		01010	11	saes.v3.encs
bs	00100	rs2	rs1	010	rd		01010	11	saes.v3.encm
bs	00011	rs2	rs1	010	rd		01010	11	saes.v3.encsm
bs	00010	rs2	rs1	010	rd		01010	11	saes.v3.decs
bs	00001	rs2	rs1	010	rd		01010	11	saes.v3.decm
bs	00000	rs2	rs1	010	rd		01010	11	saes.v3.decsm
0000111		00000	rs1	111	rd		01010	11	ssha256.s0
0000111		00001	rs1	111	rd		01010	11	ssha256.s1
0000111		00010	rs1	111	rd		01010	11	ssha256.s2
0000111		00011	rs1	111	rd		01010	11	ssha256.s3
0000111		00100	rs1	111	rd		01010	11	ssha512.s0
0000111		00101	rs1	111	rd		01010	11	ssha512.s1
0000111		00110	rs1	111	rd		01010	11	ssha512.s2
0000111		00111	rs1	111	rd		01010	11	ssha512.s3
0001000		rs2	rs1	111	rd		00000	11	ssha3.xy
0001001		rs2	rs1	111	rd		00000	11	ssha3.x1
0001010		rs2	rs1	111	rd		00000	11	ssha3.x2
0001011		rs2	rs1	111	rd		00000	11	ssha3.x4
0001100		rs2	rs1	111	rd		00000	11	ssha3.yx
0000000		rs2	rs1	111	rd		01010	11	lbx
0000001		rs2	rs1	111	rd		01010	11	lhx
0000010		rs2	rs1	111	rd		01010	11	lwx
0000011		rs2	rs1	111	rd		01010	11	ldx
0000100		rs2	rs1	111	rd		01010	11	lbux
0000101		rs2	rs1	111	rd		01010	11	lhux
0000110		rs2	rs1	111	rd		01010	11	lwux
rs3	00	rs2	rs1	100	00000		01010	11	sbx