A lightweight ISE for ChaCha on RISC-V

Ben Marshall, Daniel Page and Thinh H. Pham

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
{ben.marshall,daniel.page,th.pham}@bristol.ac.uk

Abstract. ChaCha is a high-throughput stream cipher designed with the aim of ensuring high-security margins while achieving high performance on software platforms. RISC-V, an emerging, free, and open Instruction Set Architecture (ISA) is being developed with many instruction set extensions (ISE). ISEs are a native concept in RISC-V to support a relatively small RISC-V ISA to suit different use-cases including cryptographic acceleration via either standard or custom ISEs. This paper proposes a lightweight ISE to support ChaCha on RISC-V architectures. This approach targets embedded computing systems such as IoT edge devices that don't support a vector engine. The proposed ISE is designed to accelerate the computation of the ChaCha block function and align with the RISC-V design principles. We show that our proposed ISEs help to improve the efficiency of the ChaCha block function. The ISE-assisted implementation of ChaCha encryption speeds up at least 5.4× and 3.4× compared to the OpenSSL baseline and ISA-based optimised implementation, respectively. For encrypting short messages, the ISE-assisted implementation gains a comparative performance compared to the implementations using very high area overhead vector extensions.

Keywords: ChaCha Stream Cipher, Instruction Set Extension, RISC-V

1 Introduction

ChaCha for secure communication Secure communication on the internet typically requires different cryptographic primitives and a common protocol applying these primitives to provide a protected channel between endpoints. The Transport Layer Security (TLS) specifies the leading and standard protocols for secure communication. The TLS protocol defines public key algorithms for establishing symmetric session keys, and different symmetric and MAC algorithms for the subsequent encrypted and authenticated communication. The efficiency of these primitives is essential to achieve good performance for secure communication. ChaCha is a high-throughput stream cipher which is a refinement of the Salsa20 stream cipher. It targets software platforms to aim at improving its security bounds without losing performance. ChaCha stream cipher and Poly1305 authenticator [LCM$^+$16] are specified as one of the cipher suites by the current TLS 1.3 [Res18, Section 9.1]. ChaCha is officially supported by popular cryptographic/TLS projects like OpenSSL, OpenSSH, and MbedTLS. Moreover, there are extensive efforts in the literature to optimise the performance of ChaCha implementation on various platforms, namely optimised software (using ARM Cortex ISA [DSS17] or RISC-V ISA [Sto19]), AVX vectorisation architecture [GG14], and dedicated hardware accelerators [KLA$^+$19, PRH$^+$19].

RISC-V Instruction Set Architecture RISC-V is an open and free ISA with academic origins [AP14] adopting strongly RISC-oriented design principles. The ISA can be implemented, modified, or extended with neither licence nor royalty requirements. As a result of these features and the availability of supporting ecosystems (e.g., compilation tool-chains) from surrounding communities, an increasing number of RISC-V implementations have been available. The RISC-V ISA is designed with 32 registers, denoted GPR[i] for $0 \leq i < 32$: GPR[0] is fixed to 0, whereas GPR[1] to GPR[31] are general-purpose. The width of each GPR[i], and hence the base ISA are defined by *XLEN* of which supported values can be 32, 64, 128 bits. RISC-V has an extremely simple ISA (about 50 general-purpose instructions) that has been designed to be extended. Thanks to the simple base ISA, the implementation of a RISC-V processor even with 64-bit ISA can achieve low area cost that is well suitable for resource-constrained devices. For example, the S2 Series, developed by SiFive[1], is a family of full-featured 64-bit RISC-V embedded processors for area-constrained applications. The base ISA can be supplemented using sets of standard or non-standard extensions to support additional special-purposes. Multiple proposals for standard extensions particularly one of which is the cryptography extension[2] are being developed. The current cryptography extension proposal consists of three main components: Vector, Scalar, and Entropy Source instructions. The Scalar aiming at resource-constrained devices defines a set of algorithm-specific, e.g., AES, SM4, SHA2, instructions. However, ChaCha, a widely-used algorithm, is not explicitly supported by the current proposal. In this context, the paper's main contribution is to propose *the first* design and implementation of dedicated ISEs to accelerate the ChaCha algorithm.

Accelerate cryptographic algorithms via an ISE approach To accelerate a given cryptographic algorithm, techniques can be algorithm-agnostic or algorithm-specific, and based on the use of hardware only, software only, or a hybrid approach. ISE [GB11, BGM09, RI16], as a hybrid approach, has proved its effectiveness. An increasing number of studies recently adopt ISEs for cryptographic application to improve efficiency [RCB+20, MNSW21] as well as address security concerns [GMPP20]. For accelerating performance, the idea is that a set of additional instructions can be, e.g., through benchmarking, identified to leverage special-purpose functionality, vs. general-purpose functionality in the base ISA, and thereby deliver improvement. ISEs are *particularly* effective for resource-constrained devices because they afford a compromise improving footprint and latency vs. a software-only option while also improving area overhead and flexibility vs. a hardware-only option.

Remit and organisation This paper investigates an ISE approach to support for ChaCha software. We favour a lightweight method to accelerate ChaCha performance targeting resource-constrained devices. The paper is organised as follows: Section 2 provides some background and an abstract implementation of ChaCha. Section 3 proposes the design and implementation of ISE variants for ChaCha. In Section 4, we first present the evaluation of ISE variants on ChaCha block function, then realise a complete ISE-assisted ChaCha implementation which is evaluated in comparison to optimised implementations using the RISC-V base ISA and vector extensions. Finally, Section 5 gives some conclusions. In addition, the source code of the proposed ISE can be found at https://github.com/scarv/chacha-ise.

2 ChaCha20 stream cipher

The ChaCha cipher is a 256-bit stream cipher, designed by Bernstein in 2008 [Ber08]. ChaCha based on the Salsa stream cipher has naturally a high-throughput performance on software platforms. Compared to Salsa, ChaCha has better diffusion per round and conjecturally increasing resistance to cryptanalysis.

[1]https://www.sifive.com/cores/s21
[2]https://lists.riscv.org/g/tech-crypto-ext

2.1 Encryption and Decryption

The ChaCha encryption and decryption can have the same process in which the input data stream is encrypted or decrypted by chopping the input into 64-byte blocks and Xor-ing them with the 64-byte output blocks of the ChaCha block function. The block function processes on the ChaCha state matrix of 4×4 word (32-bit) elements to generate the 64-byte output block. The ChaCha state matrix is initiated by a 4-word constant, an 8-word secret key, 4 words of counter/nonce. Note that the original ChaCha uses a 2-word nonce and a 2-word counter to allowing a practically unlimited amount of data to be encrypted while the IETF variant [NL18] increases the nonce size to 3 words, but reduces the counter size down to one word. That limits the amount of encrypted data only up to 256 GB but helps the encryption more safely with a longer (key, nonce) pair. This paper adopts the IETF variant of which the initial state matrix is shown in (1). The ChaCha process for encryption and decryption can be described as in Algorithm 1.

$$S = \begin{pmatrix} 61707865 & 3320646E & 79622D32 & 6B206574 \\ key[0] & key[1] & key[2] & key[3] \\ key[4] & key[5] & key[6] & key[7] \\ counter & nonce[0] & nonce[1] & nonce[2] \end{pmatrix} \tag{1}$$

2.2 Block function

ChaCha cipher family provides three variants which use different number of operations performed on the state (i.e., 8, 12 or 20 rounds). A larger number of rounds increases data diffusion and therefore more secure, but longer processing time. From now on, we focus on the ChaCha20 variant that executes 20 rounds in a block function. The block function processes a state matrix using odd rounds and even rounds alternately. Then, the processed state matrix is added to the input state matrix to result in the output state matrix. The operation of the block function is described in Algorithm 2.

Round function Both of the odd and even rounds have 4 quarter rounds. In the odd (column) round, the quarter rounds operate on four elements of the state matrix's columns, while the quarter rounds of the even (diagonal) round operate on the diagonals of the state matrix. For some implementations, one can view that a diagonal round can be equivalent to a column round, and vice versa, if the state matrix is rearranged (rotating the matrix's columns 1, 2, and 3) before the rounds.

Quarter-Round function A quarter round updates four state words of the state matrix as shown in Algorithm 3. The quarter round based on Add-Rotate-Xor (ARX) operations requires four sets of 32-bit Additions, Xors and rotations operating on the state words. Each word is updated twice and affected by all four input words.

3 Proposed ISEs for Chacha20

The ISE-assisted acceleration for the ChaCha block function is fairly challenging especially for the scalar-register-based ISEs because a) The block function was designed to be very efficient on software without dedicated hardware support. b) The function computes on 32-bit state elements and two consecutive operations involve at least 3 different state elements. The latter poses difficulties for the acceleration using the scalar-register-based ISEs on 32-bit RISC-V architecture without the capable of having more than 2 source operands. Certainly, vector instructions are possible approaches. But in this paper, we focused on the scalar-register-based ISE approaches on RISC-V architecture instead. This provides a lower area overhead compared to vectorisation alternatives.

```
Data: 256-bit $Key$, 96-bit $Nonce$, L-bit $P$
Result: L-bit $C$
function ChaChaProcess($Key, Nonce, P$) begin
    /*Init state matrix*/
    $S[0] \leftarrow$ 0x61707865, $S[1] \leftarrow$ 0x3320646e
    $S[2] \leftarrow$ 0x79622d32, $S[3] \leftarrow$ 0x6b206574
    $S[4..11] \leftarrow Key, S[13..15] \leftarrow Nonce$
    /*Encrypting loop*/
    for $i = 0$ upto $\lceil \frac{L}{512} \rceil - 1$ do
        $S[12] \leftarrow i$
        $K_i \leftarrow$ ChaChaBlock($S$)
        $C_i \leftarrow K_i \oplus P_i$
    end
    return C
end
```

Algorithm 1: ChaCha Stream Cipher Process.

```
Data: input state matrix $S$ of 16 32-bit state elements
Result: output state matrix $K$
function ChaChaBlock($S$) begin
    $X \leftarrow S$
    for $i = 1$ upto 10 do
        /*Odd Round*/
        QuarterRound($X[0]$, $X[4]$, $X[8]$, $X[12]$)
        QuarterRound($X[1]$, $X[5]$, $X[9]$, $X[13]$)
        QuarterRound($X[2]$, $X[6]$, $X[10]$, $X[14]$)
        QuarterRound($X[3]$, $X[7]$, $X[11]$, $X[15]$)
        /*Even round*/
        QuarterRound($X[0]$, $X[5]$, $X[10]$, $X[15]$)
        QuarterRound($X[1]$, $X[6]$, $X[11]$, $X[12]$)
        QuarterRound($X[2]$, $X[7]$, $X[8]$ , $X[13]$)
        QuarterRound($X[3]$, $X[4]$, $X[9]$ , $X[14]$)
    end
    $K \leftarrow X + S$
    return K
end
```

Algorithm 2: ChaCha Block function.

```
Data: $A$, $B$, $C$, $D$ (32-bit state emelents).
Result: $A$, $B$, $C$, $D$ (Updated 32-bit state emelents).
function QuarterRound($A, B, C, D$) begin
    /*Top half*/
    $A \leftarrow A + B,\ D \leftarrow D \oplus A,\ D \leftarrow D \lll 16$
    $C \leftarrow C + D,\ B \leftarrow B \oplus C,\ B \leftarrow B \lll 12$
    /*Bottom half*/
    $A \leftarrow A + B,\ D \leftarrow D \oplus A,\ D \leftarrow D \lll 8$
    $C \leftarrow C + D,\ B \leftarrow B \oplus C,\ B \leftarrow B \lll 7$
    return($A, B, C, D$)
end
```

Algorithm 3: ChaCha Quarter Round.

Our ISE designs obey the wider RISC-V design principles. The ISE designs support simple building-block operations, and their instruction encodings must be at most 2 source and 1 destination registers. By following that the proposed ISEs can be integrated into existing RISC-V implementations with negligible modification. To ensure low area and latency overheads, the proposed ISEs follow further requirements: a) The ISE must store operands and results in the RISC-V general-purpose scalar register file. b) The ISE must not introduce special-purpose architectural states, nor rely on special-purpose micro-architectural state (e.g., registers, caches, or scratch-pad memory). c) The operation of ISE should be executed in one clock cycle in its hardware module and must guaranteeing constant-time execution that can inherently prevent certain attack vectors based on execution latency (see [GYCH18, Section 4]). Overall, the proposed ISEs intentionally target performing ChaCha cipher on low(er)-end, resource-constrained devices such as embedded IoT edge computing that don't support a vector engine nor a dedicated hardware IP module. The focus can be viewed as reasonable because existing work on adding cryptographic support to the standard vector extension [risb] already caters for high(er)-end alternatives.

3.1 ISE for 32-bit architecture

The challenge to accelerate the ChaCha block function using 32-bit ISEs is to fetch sufficient state elements, i.e., operands for the ISE operations. We consider potential options for 32 bit-architectures, namely, accessing register pairs and having additional registers in the ISE. Both options can help to fetch additional state elements for the ChaCha round accelerator. However, the former, apart from violating the design constraints of RISC-V, increases decoder and register files complexity, and induces an increased size of pipeline stage registers to accommodate the additional operands. The latter introduces additional states in the ISE that may cause the dirty state issues in context switching scenarios (e.g., interrupt handling, multitasking) that require additional flushing and/or synchronisation mechanisms leading to significant performance reduction. Because of the aforementioned reasons, these two options are not well suitable for a lightweight solution to accelerate the ChaCha round function.

Another possible alternative is straightforwardly combining rotation and arithmetic/-logical operations. This alternative can obviously reduce the number of instruction count, hence improve the performance, for the ChaCha round function. However, the gains are fairly limited. For example, a ISE combining 32-bit Xor and 32-bit Rotation can reduce the instruction count of the ChaCha quarter round function from 12 to 8. That results in the maximum instruction count reduction achieved by using the ISE to about 66.67% of the instruction count of the implementation without ISEs support.

3.2 ISE Design for 64-bit architecture

The 64-bit architecture allows each instruction to pack more state elements compared to 32-bit architectures. This enables the possibility of efficient accelerations. We arrive at three ISE variants described in the following subsections.

3.2.1 Variant 1 (V_1)

V_1 is based on a performance-oriented approach which aims at executing the ChaCha quarter round with a minimal number of instructions. Due to the 1-destination-register constraint, the ChaCha quarter round can be executed at least two ISEs to compute 4 state elements of the quarter round. Therefore, we propose two ISEs as follows:

- `chacha.v1.ad rd, rs1, rs2`

```
begin
    $a \leftarrow \mathsf{GPR}[\texttt{rs1}]\{63 .. 32\}; d \leftarrow \mathsf{GPR}[\texttt{rs1}]\{31 .. 0\};$
    $b \leftarrow \mathsf{GPR}[\texttt{rs2}]\{63 .. 32\}; c \leftarrow \mathsf{GPR}[\texttt{rs2}]\{31 .. 0\};$
    $a \leftarrow a + b; d \leftarrow (a \oplus d) \lll 16;$
    $c \leftarrow c + d;\ b \leftarrow (c \oplus b) \lll 12;$
    $na \leftarrow a + b;$
    $nd \leftarrow (d \oplus na) \lll 8;$
    $\mathsf{GPR}[\texttt{rd}] \leftarrow (na \ll 32) \vee nd;$
end
```

- `chacha.v1.bc rd, rs1, rs2`

```
begin
    $a \leftarrow \mathsf{GPR}[\texttt{rs1}]\{63 .. 32\}; d \leftarrow \mathsf{GPR}[\texttt{rs1}]\{31 .. 0\};$
    $b \leftarrow \mathsf{GPR}[\texttt{rs2}]\{63 .. 32\}; c \leftarrow \mathsf{GPR}[\texttt{rs2}]\{31 .. 0\};$
    $c \leftarrow c + (a \oplus d \lll 24); b \leftarrow (b \oplus c) \lll 12;$
    $nc \leftarrow c + d;$
    $nb \leftarrow (b \oplus nc) \lll 7;$
    $\mathsf{GPR}[\texttt{rd}] \leftarrow (nb \ll 32) \vee nc;$
end
```

Each instruction takes two 64-bit-register operands `rs1` and `rs2`, each of which packs two 32-bit input elements of the quarter-round function. `rs1` (resp. `rs2`) contains two inputs iA and iD (resp. iB and iC). `chacha.v1.ad` (resp. `chacha.v1.bc`) computes the outputs oA and oD (resp. oB and oC) packed in its destination register `rd`. This ISE variant can be used to implement the quarter-round function as in Algorithm 4.

Data: 64-bit values $X = \{iA \parallel iD\}$ and $Y = \{iB \parallel iC\}$.
Result: 64-bit values $R0$, $R1$ such that $R0 = \{oA \parallel oD\}$ and $R1 = \{oB \parallel oC\}$.

```
function QuarterRound($X, Y$) begin
    chacha.v1.ad R0, X, Y
    chacha.v1.bc R1, R0, Y
    return($R0, R1$)
end
```

Algorithm 4: ChaCha Quarter Round in V_1.

3.2.2 Variant 2 (V_2)

V_2 introduces an area-efficiency-oriented approach which aims at favouring hardware re-usage between ISEs to reduce overall area overhead. We investigate four instructions to compute an entire quarter round as follows:

- `chacha.v2.ad0 rd, rs1, rs2`

```
begin
    $a \leftarrow \mathsf{GPR}[\texttt{rs1}]\{63 .. 32\}; d \leftarrow \mathsf{GPR}[\texttt{rs1}]\{31 .. 0\};$
    $b \leftarrow \mathsf{GPR}[\texttt{rs2}]\{63 .. 32\}; c \leftarrow \mathsf{GPR}[\texttt{rs2}]\{31 .. 0\};$
    $na \leftarrow a + b;$
    $nd \leftarrow (na \oplus d) \lll 16;$
    $\mathsf{GPR}[\texttt{rd}] \leftarrow (na \ll 32) \vee nd;$
end
```

- `chacha.v2.bc0 rd, rs1, rs2`

```
begin
    $a \leftarrow \mathsf{GPR}[\texttt{rs1}]\{63 .. 32\}; d \leftarrow \mathsf{GPR}[\texttt{rs1}]\{31 .. 0\};$
    $b \leftarrow \mathsf{GPR}[\texttt{rs2}]\{63 .. 32\}; c \leftarrow \mathsf{GPR}[\texttt{rs2}]\{31 .. 0\};$
    $nc \leftarrow c + d;$
    $nb \leftarrow (nc \oplus b) \lll 12;$
    $\mathsf{GPR}[\texttt{rd}] \leftarrow (nb \ll 32) \vee nc;$
end
```

- `chacha.v2.ad1 rd, rs1, rs2`

```
begin
    $a \leftarrow \mathsf{GPR}[\texttt{rs1}]\{63 .. 32\}; d \leftarrow \mathsf{GPR}[\texttt{rs1}]\{31 .. 0\};$
    $b \leftarrow \mathsf{GPR}[\texttt{rs2}]\{63 .. 32\}; c \leftarrow \mathsf{GPR}[\texttt{rs2}]\{31 .. 0\};$
    $na \leftarrow a + b;$
    $nd \leftarrow (na \oplus d) \lll 8;$
    $\mathsf{GPR}[\texttt{rd}] \leftarrow (na \ll 32) \vee nd;$
end
```

- `chacha.v2.bc1 rd, rs1, rs2`

```
begin
    $a \leftarrow \mathsf{GPR}[\texttt{rs1}]\{63 .. 32\}; d \leftarrow \mathsf{GPR}[\texttt{rs1}]\{31 .. 0\};$
    $b \leftarrow \mathsf{GPR}[\texttt{rs2}]\{63 .. 32\}; c \leftarrow \mathsf{GPR}[\texttt{rs2}]\{31 .. 0\};$
    $nc \leftarrow c + d;$
    $nb \leftarrow (nc \oplus b) \lll 7;$
    $\mathsf{GPR}[\texttt{rd}] \leftarrow (nb \ll 32) \vee nc;$
end
```

The approach is based on observing that the quarter round function can be split into two almost identical “top”and “bottom” halves. Moreover, each half consists of almost identical operation sequence including Addition, Xor, and Rotation. The only difference between these operation sequences is the left rotation amounts. A similar computational structure of the sequences allows to obtain an effective resource sharing for a low area cost implementation. The first two ISEs (i.e., `chacha.v2.ad0` and `chacha.v2.bc0`) are used to compute the top half of the quarter round while the other two ISEs computer the bottom halve. Again we use the similar packing scheme as in V_1. So, the quarter-round function can be implemented with 4 instructions as in Algorithm 5.

Data: 64-bit values $X = \{iA \ \| \ iD\}$ and $Y = \{iB \ \| \ iC\}$.
Result: 64-bit values $R0$, $R1$ such that $R0 = \{oA \ \| \ oD\}$ and $R1 = \{oB \ \| \ oC\}$.

```
function QuarterRound($X, Y$) begin
    chacha.v2.ad0 X’, X, Y
    chacha.v2.bc0 Y’, X, Y
    chacha.v2.ad1 R0, X’, Y’
    chacha.v2.bc1 R1, X’, Y’
    return($R0, R1$)
end
```

Algorithm 5: ChaCha Quarter Round in V_2.

3.2.3 Variant 3 (V_3)

V_3 follows a parallel oriented approach which is different from the above ISEs approaches. By observing two consecutive ChaCha quarter rounds have a similar computational structure, we investigate a calculation scheme so that two quarter-rounds, called a half-round, can be performed simultaneously. So, a ChaCha round can be performed with two half-rounds instead of four quarter rounds. To perform the half-round effectively, ISEs including one Addition and one Xor-then-Rotation instructions are proposed as below:

- `chacha.v3.add rd, rs1, rs2`

```
begin
    $a_1 \leftarrow \mathsf{GPR}[\mathtt{rs1}]\{63 \mathinner{..} 32\}; a_0 \leftarrow \mathsf{GPR}[\mathtt{rs1}]\{31 \mathinner{..} 0\};$
    $b_1 \leftarrow \mathsf{GPR}[\mathtt{rs2}]\{63 \mathinner{..} 32\}; b_0 \leftarrow \mathsf{GPR}[\mathtt{rs2}]\{31 \mathinner{..} 0\};$
    $r_1 \leftarrow a_1 + b_1; r_0 \leftarrow a_0 + b_0;$
    $\mathsf{GPR}[\mathtt{rd}] \leftarrow (r_1 \ll 32) \vee r_0;$
end
```

- `chacha.v3.xorrol rd, rs1, rs2, imm`

```
begin
    $Dec = \{16, 12, 8, 7\}$
    $a_1 \leftarrow \mathsf{GPR}[\mathtt{rs1}]\{63 \mathinner{..} 32\}; a_0 \leftarrow \mathsf{GPR}[\mathtt{rs1}]\{31 \mathinner{..} 0\};$
    $b_1 \leftarrow \mathsf{GPR}[\mathtt{rs2}]\{63 \mathinner{..} 32\}; b_0 \leftarrow \mathsf{GPR}[\mathtt{rs2}]\{31 \mathinner{..} 0\};$
    $r_1 \leftarrow (a_1 \oplus b_1) \lll Dec[\mathtt{imm}];$
    $r_0 \leftarrow (a_0 \oplus b_0) \lll Dec[\mathtt{imm}];$
    $\mathsf{GPR}[\mathtt{rd}] \leftarrow (r_1 \ll 32) \vee r_0;$
end
```

In this ISE variant, each 64 bit source register packs 2 state elements which are at the same row and at two consecutive columns in a ChaCha state matrix and the corresponding two returned elements are packed in a destination register `rd`. It can be viewed that the ISEs are vector-like instructions with the length of two 32-bit elements. So, the half-round can be implemented (similarly to a quarter-round) with 8 instructions as in Algorithm 6. Here, A, B, C, and D represent 4 rows of the ChaCha state matrix while c and $c+1$ subscript (where $c \in \{0, 2\}$) denote two consecutive columns. Furthermore, in Xor-then-Rotation instruction, the rotate amount `imm` is encoded using only 2 instruction bits. Even though the encoding and decoding the rotating amount are dedicated to ChaCha for instruction encoding efficiency, they can easily be extended with negligible area overhead for other ARX ciphers if required.

3.2.4 Packing assisted ISEs

As the above ISE variants work in packed mode, we present packing instructions to support packing manipulations. The packing instructions can be used at the beginning of ChaCha rounds to pack 16 state elements which represent the state matrix into 8 working registers. Moreover, a ChaCha block function performs 20 rounds alternate between odd and even rounds. Between rounds, we must re-pack the registers so that the odd and even rounds can be executed with the same round operations. These instructions are very similar to the `pack` instructions in the Bitmanip extension [risa]. The packing assisted ISEs are described as follows:

```
Data: 64-bit values $X = \{iA_c \parallel iA_{c+1}\}$, $Y = \{iB_c \parallel iB_{c+1}\}$, $Z = \{iC_c \parallel iC_{c+1}\}$
      and $W = \{iD_c \parallel iD_{c+1}\}$.
Result: 64-bit values $R0 = \{oA_c \parallel oA_{c+1}\}$, $R1 = \{oB_c \parallel oB_{c+1}\}$,
        $R2 = \{oC_c \parallel oC_{c+1}\}$ and $R3 = \{oD_c \parallel oD_{c+1}\}$.
function HalfRound($X, Y, Z, W$) begin
    chacha.v3.add X', X, Y
    chacha.v3.xorrol W', W, X', 0
    chacha.v3.add Z', Z, W'
    chacha.v3.xorrol Y', Y, Z', 1
    chacha.v3.add R0, X', Y'
    chacha.v3.xorrol R3, W', R0, 2
    chacha.v3.add R2, Z', R3
    chacha.v3.xorrol R1, Y', R2, 3
    return($R0, R1, R2, R3$)
end
```

Algorithm 6: ChaCha Half Round in V_3.

- `pack rd, rs1, rs2`

```
begin
    $h \leftarrow$ GPR[rs2]{31 .. 0}; $l \leftarrow$ GPR[rs1]{31 .. 0};
    GPR[rd] $\leftarrow (h \ll 32) \vee l$;
end
```

- `packh rd, rs1, rs2`

```
begin
    $h \leftarrow$ GPR[rs2]{63 .. 32}; $l \leftarrow$ GPR[rs1]{63 .. 32};
    GPR[rd] $\leftarrow (h \ll 32) \vee l$;
end
```

- `packhl rd, rs1, rs2`

```
begin
    $h \leftarrow$ GPR[rs2]{63 .. 32}; $l \leftarrow$ GPR[rs1]{31 .. 0};
    GPR[rd] $\leftarrow (h \ll 32) \vee l$;
end
```

- `packlh rd, rs1, rs2`

```
begin
    $h \leftarrow$ GPR[rs2]{31 .. 0}; $l \leftarrow$ GPR[rs1]{63 .. 32};
    GPR[rd] $\leftarrow (h \ll 32) \vee l$;
end
```

The ChaCha round function using each ISE variant is implemented with different types of packing instructions. Table 1 reports the packing operation overhead of ChaCha round variants in term of the number of instructions executed in functions. It can be seen that the packing scheme of the implementation using V_3 is more efficient compared to the other variants.

Table 1: The numbers of packing instructions are executed in three ISEs assisted ChaCha round functions.

Instructions	V_1	V_2	V_3
`pack`	4	4	0
`packh`	4	4	0
`packhl`	160	160	0
`packlh`	0	0	80

Table 2: Comparison of area overheads between ISE variants when synthesised for a generic CMOS cell library.

Implementations	Size (NAND2 gates)	Depth
V_1	2353	56
V_2	1362	25
V_3	1617	19

3.3 Hardware Implementation

3.3.1 Implementation of ISE variants' submodule

We implement three ChaCha ISE variants in their self-contained module. These modules are implemented purely using combinational logics. It means that all operations of the modules are executed in a clock cycle. Moreover, the modules have a similar simple interface which has two input, one output operands, and input decoded signals. The open-source Yosys [Wol] synthesis tool is used with default settings to obtain post-synthesis circuit area overhead of the implemented modules in terms of NAND2 gate equivalents and circuit depths (in the form of gate delays).

Table 2 reports the comparison of area overheads between ISE variants. It can be seen that the V_2 and V_3 consume lower numbers of NAND2 gates and shorter circuit depths compared to V_1. V_2 uses the smallest number of NAND2 gates which is slightly better than that of V_3 while V_3 results in the shortest circuit depths. Moreover, the comparison of the ISE variants' performance on the ChaCha round function in software is considered and given in Section 4.1.

3.3.2 Integration of ChaCha ISE into a 64-bit RISC-V processor

To evaluate the proposed ISE variants, each of the ISE variants is integrated into a 64-bit RISC-V host core. We opt for a well-known Rocket Chip [AAB+16] as the host core for ChaCha ISEs. The Rocket core is a highly configurable RISC-V core that executes

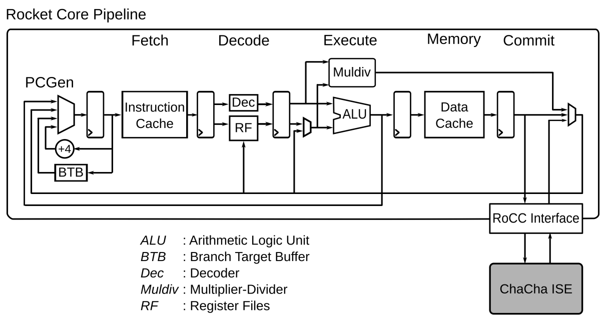

Figure 1: Integrating the ChaCha ISE into the Rocket Chip.

Table 3: Area overheads of ChaCha ISE integration compared to Rocket core and its sub-modules.

	LUTs	FFs	Slices
RV64 Rocket core	9867	3749	2951 (1.00×)
|– ALU	603	0	169 (0.06×)
|– Muldiv	613	214	173 (0.06×)
|– Others	8649	3535	2672 (0.88×)
RV64 Rocket + ISE	10169	3749	3041 (1.03×)

instructions using a 5-stage, in-order pipeline. We take advantage of this to configure for a core supporting RV64IMC instruction set, i.e., the 64-bit base integer ISA [RV:19, Chapter 5] plus standard Multiplication [RV:19, Chapter 7] and Compressed [RV:19, Chapter 16] extensions. It is also configured to support an instruction cache, a data cache, and a branch prediction mechanism. To support ChaCha ISE, the Rocket Chip core has an additional configuration to enable custom instructions, for which we choose the `Custom 0` opcode [RV:19, Chapter 25], to decode the ChaCha ISE. The core accesses the ChaCha ISE submodule via a Rocket Custom Coprocessor (RoCC) interface [AAB+16, Section 4], shown in Figure 1. Since the ISE variants' implementations comply with at most 2 sources and 1 destination operands requirement, no further structural modification is required in micro-architecture.

We implement the evaluated systems on Kintex-7 XC7K160T FPGA device employing Xilinx Vivado 2019.1 version. The default synthesis settings are used, with no effort invested in synthesis or post-implementation optimisation. Table 3 reports the area overheads of the Chacha ISE submodule in the system in comparison to the Rocket core and its submodules, e.g., Multiplier/divisor (Muldiv), ALU. The implementation of V_3 is chosen for the ChaCha ISE submodule to report in the table because it gains the best trade-off between area overhead (see Table 2) and software performance (see Table 4). As can be seen, the ChaCha ISE causes a small increase of 3% in the number of logic Slices of the Rocket core, and consumes no Flip-Flops (FFs). In fact, its overhead is considerably smaller compared to other functional submodules such as ALU, Muldiv. Moreover, the timing report of the implementation shows that the ChaCha ISE does not affect the longest delay paths which reduce the maximum operating frequency of the system.

4 Evaluation

4.1 Single block function performance

We evaluate the accelerated implementations of the ChaCha block function in software compared with a vanilla implementation used in OpenSSL [Ope] as a Baseline. The three accelerated functions denoted as V_1, V_2, and V_3 use the corresponding ChaCha ISE variant sets to accelerate their round operations. Because ChaCha ISEs are encoded as R-type instructions, the accelerated functions easily use assembly RISC-V directives `.insn` to invoke ChaCha ISEs without the requirement of building modified toolchains. For comparison's sake, all accelerated and baseline functions have the same function prototype and looping scheme (i.e., 10 double rounds each of which includes an odd and an even rounds). They are complied using the RISC-V gcc version 9.2.0 with a performance optimisation flag ('-02') and targeting for the `rv64imac` architecture ('-march=rv64imac -mabi=lp64').

The compiled software runs on the Rocket Chip system supporting the relevant set of ISEs via the RoCC interface. The system is implemented on the Kintex-7 XC7K160T FPGA of a SASEBO-GIII platform. The Rocket Chip is configured to run with the clock of 50 MHz.

The evaluation is shown in Table 4 in terms of instruction count, cycle count, and instruction footprint (in bytes) of the ChaCha block function. As can be seen, the accelerated functions have significantly reduced instruction counts to about 20% of the baseline instruction count. Even though the cycle counts are also reduced in the case of the accelerated functions, but the reduction in cycle count metrics is not as good as in instruction count metrics. This could be due to inefficient data forwarding operations supporting ChaCha ISEs via the RoCC interface in the Rocket core micro-architecture. It should be noted that the operation of every ChaCha ISEs is computed in one clock cycle. One can view that as a trade-off between ineffective performance and invasiveness avoiding micro-architectural modifications.

Comparing the ISE variants, V_3 obtains a good trade-off solution which provides the lowest instruction footprint and the second-lowest instruction count (32% and 21%, respectively, compared to the baseline) and consumes a small area overhead.

4.2 Comparison to optimised software implementations

We implement a completed ChaCha encryption/decryption function in which the accelerated block functions implemented in the above subsection is used to generate keystream blocks. The keystream blocks xors with input data-stream blocks to encrypt/decrypt the data streams. We investigate the V_3 set of the proposed ISE to accelerate the encryption function.

The performance of the proposed ISE-assisted implementation is evaluated in comparison to the existing implementations including scalar and vectorisation implementations. For scalar (no vectorisation) implementations, only the standard 64-bit ISA (scalar) instructions of RISC-V are used. We choose the ChaCha implementation of OpenSSL as the Baseline. In addition, we implement an optimised variant denoted V_4 which is written in assembly language to optimise the performance with our best effort. Moreover, we investigate an optimised implementation denoted V_5 to use the Bitmanip extension supporting rotation instructions.

For the vectorisation implementations, we make use of vectorised instructions to accelerate ChaCha encryption/decryption operations. We adopt two approaches, one, denoted `Vector1`, implements a cell-oriented approach used in OpenSSL which processes multiple blocks in parallel. The other, denoted `Vector2`, follows a row-oriented approach presented in [GG14]. This approach packs state elements in ChaCha blocks' rows into the same vectorised registers. Different from the original implementations, the vectorisation implementations are realised using the vector instruction extension set [risb] for RISC-V processors instead of using AVX/AVX2 architecture on `x86_64` processors. In addition, we investigate two versions of vector lengths, namely 128 bits and 256 bits, for the vectorisation implementations.

The implementations are compiled as the same set up in Section 4.1 but targeting to the `rv64imacb` and the `rv64gcv` architectures for the Bitmanip extension and the vector instruction extension, respectively. Currently, the Bitmanip and the vector extensions have not been frozen. We adopt the latest published versions v0.92 and v0.9 for the

Table 4: Comparison of chacha block function performance.

Implementations	Inst. count	Cycle count	Inst. footprint
Baseline	2214 (1.00×)	2991 (1.00×)	852 (1.00×)
V_1	434 (0.20×)	1414 (0.47×)	382 (0.45×)
V_2	594 (0.27×)	2350 (0.79×)	454 (0.53×)
V_3	464 (0.21×)	1420 (0.47×)	274 (0.32×)

Table 5: Comparison of encryption/decryption performance in instruction count for different message sizes between the Baseline, ISA-based optimised implementation, ISE-assisted implementation and different vectorization implementations.

Message size	Baseline	RV64I	RV64IB	ISE	128 bit Vector		256 bit Vector	
	OpenSSL	V_4	V_5	V_3	`Vector1`	`Vector2`	`Vector1`	`Vector2`
64 bytes	2825	1768	1129	523	2001	607	2001	615
128 bytes	5555	3486	2207	989	2001	1182	2001	615
256 bytes	11015	6922	4363	1921	2001	2332	2001	1191
512 bytes	21935	13794	8675	3785	3748	4632	2001	2343
1024 bytes	43775	27538	17299	7716	7242	9232	3748	4647

Bitmanip and the vector extensions, respectively. Since there is not yet an open-source implementation supporting the RISC-V vector instruction extension is available, we use `Spike` [otUoC], an instruction set simulator, to evaluate the implementations.

Table 5 reports the instruction count executing encryption/decryption operations for different message sizes. As expected, all accelerated implementations including the cases of scalar ISE, vectorised ISE gain significant reductions in instruction count compared to the baseline. We observe that the lack of supporting rotation in the current version of the vector instruction extension introduces the disadvantages of the implementations based on this extension. However, for the large messages, the vector-based implementations show their advantage over the scalar ISE based implementation. The `Vector1` implementations provide the lowest instruction count executions when the message size is greater than 512 bytes. The `Vector2` implementations outperform the `Vector1` implementations for shorter messages. Interestingly, the proposed scalar ISE-assisted implementation, V_3, gain the best performance in the case of single block messages. For the message size smaller than 512 bytes, the V_3 implementation have a better performance compared to the `Vector1` implementation. And V_3 outperforms the 128-bit Vector implementation of `Vector2` for all message sizes. But when the vector length increases to 256-bit, `Vector2` shows its advantage over V_3. It is worth noting that the vector instruction extension which is only presented in high(er)-end computational platforms cause a very large overhead in hardware while the proposed ISE approach requires negligible increased hardware cost to gain a good performance for short messages compared to the vectorisation implementations. That makes our ChaCha ISE be suitable for low(er)-end resource-constrained processors.

In comparing the scalar implementation, the V_4 implementation gains a reduced instruction count to 63% of the OpenSSL baseline instruction count. It obtains an encrypting performance of 26.9 instructions/byte (with 1024 byte message) that is almost similar to the result reported in [Sto19] (i.e., 27.9 cycles/byte with most instructions executed in a single cycle). Moreover, the V_5 implementation has further improvement that reduces its instruction count to 40% of the baseline instruction count. Notably, the instruction count of the V_3 implementation is reduced to at least 19% (resp. 29%) of the baseline (resp. V_4) instruction count. In other words, the V_3 implementation achieves a 5.4× (resp. 3.4×) speed-up compared to the baseline (resp. V_4) implementation.

4.3 Comparison to dedicated hardware alternatives

It is possible to implement dedicated hardware as an independent co-processor for ChaCha. Existing works presented in the literature [KLA+19, PRH+19]. The dedicated hardware is obviously much faster compared to the software-only as well as ISE-assisted software alternatives. However, the dedicated hardware IP core consumes a large amount of area overhead that makes it may not suitable for lightweight, resource-constrained systems. For example, the smallest dedicated hardware using logic slices presented in [PRH+19] consumes 852 FPGA logic slides that are about a magnitude larger than the number of slides used by the proposed ISE (i.e., 90 slides). Moreover, the dedicated hardware IP has its fixed functionality dedicating to the ChaCha function, while the proposed ISE could potentially be reused to improve the performance of other cryptographic primitives in the ARX family. For example, the V_3 ISE variant offers effective packing operations of 32-bit Addition and Xor that are intensively used in Speck [BSS+13] and Alzette [BBdS+20].

5 Conclusion

In this paper, we presented the design, implementation, and evaluation of three ISE variants to support ChaCha stream cipher on 64-bit RISC-V architecture. We show that our proposed ISE variants help improve significantly the efficiency of ChaCha block function wrt. both execution latency and memory footprint. Our ISE-assisted implementation of ChaCha encryption speeds up at least 5.4× and 3.4× compared to the OpenSSL baseline and ISA-base optimised implementation, respectively. For encrypting short messages, the ISE-assisted implementation gains a comparative performance compared to the vectorised implementations which demand a large hardware overhead to support vector instruction extension. Moreover, the ISE hardware implementation only causes a negligible increased area overhead, about 3%, on the Rocket Chip system that makes our ChaCha ISE be suitable for resource-constrained processors. While showing potential, the proposed ChaCha ISE needs further investigations in future works to be more generic and efficient for other ARX ciphers.

Acknowledgement

This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme.

References

[AAB+16] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D.A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[AP14] K. Asanović and D.A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, 2014. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[BBdS+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann GroÃ§schÃďdl, LÃĺo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Alzette: a 64-bit ARX-box (feat. CRAX and TRAX). Springer-Verlag, 2020.

[Ber08] D. J. Bernstein. ChaCha, a variant of Salsa20, 2008. http://cr.yp.to/papers.html.

[BGM09] S. Bartolini, R. Giorgi, and E. Martinelli. Instruction set extensions for cryptographic applications. In Ç.K. Koç, editor, *Cryptographic Engineering*, chapter 9, pages 191–233. Springer, 2009.

[BSS+13] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. https://eprint.iacr.org/2013/404.

[DSS17] F. De Santis, A. Schauer, and G. Sigl. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 692–697, 2017.

[GB11] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Trans. on Reconfigurable Technology and Systems*, 4(2):18:1–18:28, 2011.

[GG14] M. Goll and S. Gueron. Vectorization on ChaCha Stream Cipher. In *11th International Conference on Information Technology: New Generations*, pages 612–615, 2014.

[GMPP20] S. Gao, B. Marshall, D. Page, and T. Pham. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):73–98, 2020.

[GYCH18] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[KLA+19] K. Kiningham, P. Levis, M. Anderson, D. Boneh, M. Horowitz, and M. Shih. Falcon - a flexible architecture for accelerating cryptography. In *IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 136–144, 2019.

[LCM+16] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). Internet Engineering Task Force (IETF) Request for Comments (RFC) 7905, June 2016. https://www.rfc-editor.org/rfc/rfc7905.

[MNSW21] B. Marshall, G.R. Newell, D. Page M.-J.O. Saarinen, and C. Wolf. The design of scalar AES instruction set extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(1), 2021.

[NL18] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. Internet Engineering Task Force (IETF) Request for Comments (RFC) 8439, June 2018. https://tools.ietf.org/html/rfc8439.

[Ope] OpenSSL. The Open Source toolkit for SSL/TLS. http://www.openssl.org.

[otUoC] The Regents of the University of California. Spike RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim.

[PRH+19] J. Pfau, M. Reuter, T. Harbaum, K. Hofmann, and J. Becker. A Hardware Perspective on the ChaCha Ciphers: Scalable Chacha8/12/20 Implementations Ranging from 476 Slices to Bitrates of 175 Gbit/s. In *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pages 294–299, 2019.

[RCB+20] B. Rezvani, T. Conroy, L. Beckwith, M. Bozzay, T. Laffoon, D. McFeeters, Y. Shi, M. Vu, and W. Diehl. Efficient simultaneous deployment of multiple lightweight authenticated ciphers. *IACR Crypto. ePrint Archive 2020: 609*, 2020.

[Res18] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Engineering Task Force (IETF) Request for Comments (RFC) 8446, August 2018. https://tools.ietf.org/html/rfc8446.

[RI16] F. Regazzoni and P. Ienne. Instruction set extensions for secure applications. In *Design, Automation, and Test in Europe (DATE)*, pages 1529–1534, 2016.

[risa] RISC-V bit manipulation extension draft proposal. https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-draft.pdf.

[risb] RISC-V "V" Vector Extension draft proposal. https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc.

[RV:19] The RISC-V instruction set manual. Technical Report Volume I: User-Level ISA (Version 20191213), 2019. http://riscv.org/specifications.

[Sto19] K. Stoffelen. Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology – LATINCRYPT 2019*, LNCS 11774, pages 323–340. Springer-Verlag, 2019.

[Wol] Claire Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

A The ISE-assisted Implementation of the block function

```
#define chacha_add(rd, rs1, rs2) asm volatile ( \
  ".insn r CUSTOM_0, 7, 0b11000, %0,%1,%2\n\t" :"=r"(rd):"r"(rs1),"r"(rs2));

#define chacha_xorrol16(rd, rs1, rs2) asm volatile ( \
  ".insn r CUSTOM_0, 7, 0b11001, %0,%1,%2\n\t" :"=r"(rd):"r"(rs1),"r"(rs2));

#define chacha_xorrol12(rd, rs1, rs2) asm volatile ( \
  ".insn r CUSTOM_0, 7, 0b11010, %0,%1,%2\n\t" :"=r"(rd):"r"(rs1),"r"(rs2));

#define chacha_xorrol08(rd, rs1, rs2) asm volatile ( \
  ".insn r CUSTOM_0, 7, 0b11011, %0,%1,%2\n\t" :"=r"(rd):"r"(rs1),"r"(rs2));

#define chacha_xorrol07(rd, rs1, rs2) asm volatile ( \
  ".insn r CUSTOM_0, 7, 0b11100, %0,%1,%2\n\t" :"=r"(rd):"r"(rs1),"r"(rs2));

#define HALF_ROUND(A, B, C, D) {           \
  chacha_add(A,A,B);  chacha_xorrol16(D,D,A); \
  chacha_add(C,C,D);  chacha_xorrol12(B,B,C); \
  chacha_add(A,A,B);  chacha_xorrol08(D,D,A); \
  chacha_add(C,C,D);  chacha_xorrol07(B,B,C); \
}
#define REPACK_ROW(r0,r1,s0,s1) { \
  rv64_packlh(r0,s0,s1); \
  rv64_packlh(r1,s1,s0); \
}
void chacha20_block(uint32_t out[16], uint32_t in[16])
{
  uint64_t * pin  = (uint64_t *)in;
  uint64_t * pout   = (uint64_t *)out;
  uint64_t t2,t3,t6,t7;

  uint64_t a0 = pin[0];       // x[ 1], x[ 0]
  uint64_t a1 = pin[1];       // x[ 3], x[ 2]
  uint64_t a2 = pin[2];       // x[ 5], x[ 4]
  uint64_t a3 = pin[3];       // x[ 7], x[ 6]
  uint64_t a4 = pin[4];       // x[ 9], x[ 8]
  uint64_t a5 = pin[5];       // x[11], x[10]
  uint64_t a6 = pin[6];       // x[13], x[12]
  uint64_t a7 = pin[7];       // x[15], x[14]

  for(int i = 0; i < CHACHA20_ROUNDS; i += 2) {
    HALF_ROUND(a0,a2,a4,a6); // column 1 & 0
    HALF_ROUND(a1,a3,a5,a7); // column 3 & 2
    REPACK_ROW(t2,t3,a2,a3); //  5, 4, 7, 6-> 6, 5, 4, 7
    REPACK_ROW(t6,t7,a7,a6); // 13,12,15,14->12,15,14,13

    HALF_ROUND(a0,t2,a5,t6); // column 1 & 0
    HALF_ROUND(a1,t3,a4,t7); // column 3 & 2
    REPACK_ROW(a2,a3,t3,t2); //  6, 5, 4, 7-> 5, 4, 7, 6
    REPACK_ROW(a6,a7,t6,t7); // 12,15,14,13->13,12,15,14
  }
  chacha_add(pout[0], a0, pin[0]);
  chacha_add(pout[1], a1, pin[1]);
  chacha_add(pout[2], a2, pin[2]);
  chacha_add(pout[3], a3, pin[3]);
  chacha_add(pout[4], a4, pin[4]);
  chacha_add(pout[5], a5, pin[5]);
  chacha_add(pout[6], a6, pin[6]);
  chacha_add(pout[7], a7, pin[7]);
}
```