# Efficient Cryptography on the RISC-V Architecture

Ko Stoffelen

Digital Security Group, Radboud University, Nijmegen, The Netherlands
k.stoffelen@cs.ru.nl

**Abstract.** RISC-V is a promising free and open-source instruction set architecture. Most of the instruction set has been standardized and several hardware implementations are commercially available. In this paper we highlight features of RISC-V that are interesting for optimizing implementations of cryptographic primitives. We provide the first optimized assembly implementations of table-based AES, bitsliced AES, ChaCha, and the Keccak-$f[1600]$ permutation for the RV32I instruction set. With respect to public-key cryptography, we study the performance of arbitrary-precision integer arithmetic without a carry flag. We then estimate the improvement that can be gained by several RISC-V extensions. These performance studies also serve to aid design choices for future RISC-V extensions and implementations.

**Keywords:** RISC-V, AES, ChaCha, Keccak, arbitrary-precision arithmetic, software optimization

## 1 Introduction

The RISC-V project started out in 2010 as a research project at the University of California, Berkeley. The goal was to design an open-source reduced instruction set that was free and practical to use by academics and industry. Today, it comprises a foundation[1] with well over two hundred member organizations, including major industry partners such as Google, Qualcomm, and Samsung. The fact that many large companies are joining this efforts indicates that RISC-V might await a bright future. In particular, no longer having to pay any license fees makes it an attractive alternative and a serious competitor to ARM microcontrollers.

Together, the foundation's members developed a specification for the RISC-V instruction set architecture [RIS17]. RISC-V targets both embedded 32-bit devices and larger 64-bit and even 128-bit devices. While some parts of the specification are still in development, the most important parts have been frozen such that hardware and software could be implemented. Compilers, debuggers, and software libraries with RISC-V support have been around for several years[2].

[1] https://riscv.org/
[2] https://riscv.org/software-status

Commercial boards with fully functional RISC-V SoCs have been available for sale since 2016[3].

There exist several open-source RISC-V CPU designs designed to be easily extensible. This makes the platform an ideal candidate for software-hardware co-design, as was exemplified by a recent implementation of the hash-based signature scheme XMSS [WJW+18]. The underlying hash-function, SHA-256, was implemented in hardware to increase the performance of the full signature scheme. However, it is not always possible to 'simply' add a hardware co-processor of a required cryptographic primitive. In practice, one may have to deal with whatever hardware is available or a developer might lack the capabilities to modify a hardware implementation. More importantly, adding a co-processor to an ASIC will most likely increase the production cost of that chip. In order to make any trade-off decision for software-hardware co-design meaningful, some numbers need to exist to have an idea about the cost of software implementations. To the best of our knowledge, we are the first to provide such numbers for cryptographic primitives.

We explain how AES-128, ChaCha20, and Keccak-$f[1600]$ can be implemented efficiently on RISC-V and we optimize 32-bit RISC-V assembly implementations. We also study the speed of arbitrary-precision addition, schoolbook multiplication, and Karatsuba multiplication for unique and redundant or reduced-radix integer representations. We then draw a parallel to the ARM Cortex-M line of microcontrollers and we show how architectural features such as the availability of native rotation instructions, a carry flag, and the number of available registers impact the performance of these primitives. We continue by estimating what the performance would be if a RISC-V core were to be extended with these features.

In Section 2 we first explain details about the RISC-V instruction set and our benchmarking platform. Sections 3, 4, and 5 cover implementation strategies that are specific to AES, ChaCha, and Keccak, respectively. Arbitrary-precision integer arithmetic is discussed in Section 6. Finally, in Section 7 we compare the relative performance of cryptographic primitives to that on the ARM Cortex-M4 and estimate what the performance would be with RISC-V extensions for several architectural features.

Our software implementations are open-source and placed into the public domain. They are available at `https://github.com/Ko-/riscvcrypto`.

## 2 The RISC-V Architecture

The RISC-V instruction set architecture (ISA) specification is split into a user-level ISA and a privileged ISA. The privileged ISA specifies instructions and registers that are useful when creating, for example, operating systems, but for our purpose we only need to consider the user-level ISA. The user-level ISA is divided in a base ISA and in several standardized extensions that are discussed in Section 2.2. At the time of writing, the base ISAs for 32-bit and 64-bit machines,

[3] `https://www.sifive.com/boards/hifive1`

called RV32I and RV64I respectively, have been frozen at version 2.0. A base ISA for 128-bit machines (RV128I) and a smaller 32-bit variant with fewer registers (RV32E) still have draft status. In this work we focus on the 32-bit RV32I instruction set.

### 2.1 The RV32I Base Instruction Set

RV32I specifies 32 32-bit registers named `x0` to `x31`. However, not all of them can be used freely. The registers have aliases that makes their purpose more clear. For example, `x0` is also known as `zero`: writes to it are ignored and it always reads as the value 0. The others are: `ra` (return address, `x1`), `sp` (stack pointer, `x2`), `gp` (global pointer, `x3`), `tp` (thread pointer, `x4`), `a0-a7` (function arguments and return value), `s0-s11` (saved registers), and `t0-t6` (temporary registers). That means that 27 registers can be used without complications and maybe a few more depending on the environment. Only `sp` and `s0-s11` are callee-saved.

As a true RISC, the number of available instructions is fairly limited. We therefore include a concise but complete overview in this section. All instructions are described in more detail in the official specification [RIS17].

Arithmetic and bitwise instructions have three register operands, or two register operands and a sign-extended 12-bit immediate, denoted by the `I` suffix. The following self-explanatory instructions are available: `ADD`, `ADDI`, `SUB`, `AND`, `ANDI`, `OR`, `ORI`, `XOR`, and `XORI`. There is no `SUBI`, because that is just an `ADDI` with a negative immediate. Similarly, there is no real `NOT` instruction, because it can be implemented with `XORI` and $-1$ as immediate. `NOT` is recognized as a pseudo-instruction by assemblers.

Regarding shifts, the following instructions exist: `SLL`, `SLLI`, `SRL`, `SRLI`, `SRA`, and `SRAI`. These naming convention that is used here is `S`hift (`L`eft or `R`ight) (`L`ogical or `A`rithmetic) (`I`mmediate). Note that the base ISA does not specify a rotation instruction.

To load a value from memory, `LW`, `LH`, `LHU`, `LB`, and `LBU` can be used. The `W` stands for word (32 bits), the `H` for half-word (16 bits), and the `B` for byte (8 bits). With `LH` and `LB`, the value is assumed to be signed and will therefore be sign-extended to a 32-bit register. `LHU` and `LBU` are their unsigned counterparts that perform zero-extension instead of sign-extension. To store a register value to memory, one can use `SW`, `SH`, and `SB`. For all load and store instructions, the base address needs to be in a register. An immediate offset can be specified in the instruction. For example, `LW a1, 4(a0)` loads a word from `a0` + 4 in `a1`. It is not possible to specify the offset in a register or to automatically increment/decrement the address.

The `JAL` and `JALR` instructions specify unconditional jumps. The target address can be specified relative to the program counter (`JAL`) or as an absolute address in a register (`JALR`). On the other hand, `BEQ`, `BNE`, `BLT`, `BLTU`, `BGE`, and `BGEU` denote conditional jumps based on a comparison. Their first two operands are registers of which the values are compared. The `U` suffix denotes that the operands are interpreted as unsigned values for the comparison. The third operand specifies the destination address relative to the program counter.

It is also possible to compare without branching. The `SLT`, `SLTU`, `SLTI`, and `SLTIU` instructions set a destination register to one if the second operand (a register) is less than (signed or unsigned) the third operand (either a register or an immediate). Otherwise, the destination register is set to zero.

The `LUI` (load upper immediate) and `AUIPC` (add upper immediate to program counter) instructions can be used to set values larger than 12 bits in a register.

Finally, for the sake of completeness, there are specialized instructions to deal with synchronization (`FENCE` and `FENCE.I`), to deal with control and status registers (6 `CSR*` variants), to call an operating system (`ECALL`) and to signal debuggers (`EBREAK`). We will not use them, except for reading a cycle counter.

### 2.2 Standardized Extensions

A RISC-V core has to implement a base ISA, and optionally it can implement one or several standardized extensions to the instruction set. Most extensions are denoted by a single letter. The extensions with a frozen specification are $M$ (with instructions for integer multiplication/division), $A$ (atomic instructions), $F$ (single-precision floating point), $D$ (double-precision floating point), $Q$ (quad-precision floating point), and $C$ (compressed instructions).

Other extensions, such as those for bit manipulation, vector instructions, and user-level interrupts still have draft status. To the best of our knowledge the extensions in draft status have not yet been implemented by any commercially available core[4].

### 2.3 Benchmarking Platform

We use a HiFive1 development board as our benchmarking platform, as they are relatively easily available. This contains the FE310-G000 SoC [SiF17] with an E31 core [SiF18]. The core implements the RV32IMAC instruction set, i.e., the RV32I base ISA with the extensions for multiplication/division, atomic instructions, and compressed instructions. Of these, only the M extension is relevant to us.

The RISC-V specification does not specify how long instructions take to execute or what kinds of memory are available. This is left open to the hardware core implementer. Benchmarks across different RISC-V cores therefore need to be compared with caution. To provide more insight, we briefly describe some characteristics of this particular RISC-V core.

The E31 is designed as a 5-stage single-issue in-order pipelined CPU that runs at a clock frequency of at most 384 MHz. There is 64 KiB of DTIM memory that is used as RAM. Outside of the core, there is another 16 MiB of QSPI flash memory. To accelerate instruction fetches from the flash memory, the E31 comes with 16 KiB of 2-way instruction cache.

Most instructions have a result latency of a single cycle. There are a few exceptions. For example, word-loads have a result latency of 2 cycles with a cache hit. With a cache miss, it highly depends on the relative clock frequency of the

[4] https://riscv.org/risc-v-cores

flash controller compared to the core. Half-word-loads and byte-loads have a result latency of 3 cycles in the event of a cache hit. Misaligned DTIM accesses are not allowed and result in a trap signal.

The E31 has an elaborate branch predictor, consisting of a branch target buffer, a branch history buffer, and a return address stack. Correctly predicted branches should suffer no penalty, while wrong guesses receive a penalty of 3 cycles.

The RISC-V specification describes a 64-bit increasing cycle counter that is accessible through two CSR registers. This can be used for accurate benchmarking of code. We aim to unroll the code as much as possible as long as the code still fits in the instruction cache. Tables and constants are stored in the DTIM memory. This way, we manage to get very consistent measurements. Occasionally, a measurement ends up taking much longer than expected. These outliers are ignored.

## 3 AES

32-bit software implementations of AES usually fall into two categories, depending on whether it it safe to use table lookups or not. The fastest encryption implementations *for a single block* use the idea that the various steps of the round function can be combined in large lookup tables, usually called $T$-tables [DR02]. However, this type of implementation is known to be vulnerable to cache-based timing attacks [Ber05a,OST06]. A CPU cache can leak information about which memory address has been accessed during a computation. When this memory address depends on a secret intermediate value as is the case with the $T$-table approach, it can be used to extract secret information.

When *multiple blocks* can be processed in parallel (e.g., in CTR or GCM mode) and the CPU registers are large enough to accommodate multiple blocks, bitsliced implementations can be more efficient [KS09,Kön08]. This type of AES implementation has the additional advantage that lookup tables are easily avoidable, allowing a careful implementer to make it resistant against timing attacks.

Our particular benchmarking platform does not have a data cache. Therefore, it *should* be safe to use a table-based AES implementation on this device. However, this might not be the case on other RISC-V platforms. Table-based implementations might also demand an unreasonable amount of memory on small embedded RISC-V-based devices. This is why we treat both implementation categories.

### 3.1 Table-based Implementations

At Indocrypt 2008, Bernstein and Schwabe explained how to optimize table-based AES implementations for a variety of CPU architectures [BS08]. They describe a baseline of 16 shift instructions, 16 mask instructions, 16 load instructions for table lookups, 4 load instructions for round keys, and 16 xor instructions per AES

round, plus 16 additional mask instructions in the last round and 4 additional round-key loads and 4 xor instructions for the initial AddRoundKey. This baseline excludes the cost of loading the input into registers, writing the output back to memory, and some overhead such as setting the address of the lookup table in a register and storing callee-save registers on the stack when necessary. They then continue by listing various architecture-dependent optimizations.

On RISC-V, very few of these techniques are possible, which is no surprise given that the instruction set is intentionally kept very simple. The `LBU` byte load instruction allows to save 4 mask instructions in the final round. On the other hand, the baseline count assumes that it is possible to load from an address specified by a base value in one register and an offset in another register. While this holds for many architectures, it is not true for RISC-V. Instead, the full address needs to be explicitly computed each time. This means that we require 16 extra `ADD` instructions per round.

With round-key recomputation, only 14 round-key words have to be stored/loaded instead of 44. This saves 30 `SW` instructions in the key expansion, but more importantly, it allows to swap 30 `LW` instructions for 30 `XOR` instructions at the cost of using 4 extra registers of which their values need to be saved on the stack. We expected this to improve performance for encryption on our platform. However, it turned out that this was not the case so we did not employ this technique.

There is more that can be done with the free registers that are available. Some of the round keys could also be cached in registers such that they do not have to be loaded for every block when encrypting multiple blocks. However, to keep the implementation as versatile as possible, we decided not to do this and to encrypt just a single block. This makes it possible to straightforwardly build any mode around it.

**Result.** We implemented and optimized the AES-128 key expansion and encryption algorithms. Both use the same 4 KiB lookup table. Key expansion finishes in 340 cycles and requires no stack memory. Encryption of a single 16-byte block is performed in 912 clock cycles. This uses 24 bytes on the stack to store callee-save registers.

### 3.2 Bitsliced Implementations

With bitsliced AES implementations, the internal parallelism in the SubBytes step usually means that the AES state is represented in such a way that a register is made to contain the $i$th bit of every byte of the state. This means that 8 registers are needed to represent the AES state, but then only 16 bits in the register are used, which is suboptimal. However, when multiple AES blocks can be processed in parallel, they can be stored in the same registers in order to process them simultaneously. Especially when the registers are large, this yields very high throughputs [KS09].

We implement an optimized bitsliced implementation of AES-128 in CTR mode. With 32-bit registers, only 2 blocks can be processed in parallel. The

implementation is inspired by an earlier implementation optimized for the ARM Cortex-M4 architecture [SS16].

For the most expensive operation, SubBytes, we use the smallest known circuit by Boyar and Peralta of 113 gates [BP10]. On the Cortex-M4, this could not be implemented directly because there were not enough registers available. With RV32I, carefully rearranging the instructions permits not having to spill any intermediate value to the stack. We can therefore implement SubBytes in exactly 113 single-cycle bitwise instructions.

ShiftRows with a 'regular' state representation uses rotations over the full rows of the AES state that are stored in registers. The equivalent for the bitsliced state representation requires to do rotations within a byte of a register, which is trickier to implement. The RV32I base ISA does not offer convenient instructions to do this or to extract bits from a register. It therefore has to be implemented by simply masking out a group of bits, shifting them to their correct position and inserting them in a result register. This takes 6 `OR` instructions, 7 `AND(I)` instructions and 6 shift instructions per state register. There are 8 state register, so this has to be done 8 times for one AES round.

On the Cortex-M4, MixColumns could be implemented with just 27 xor instructions, heavily using the fact that one operand could be rotated for free. RISC-V, however, does not have a native rotation instruction in the base ISA at all. Therefore the rotation has to be implemented with two shifts and an `OR` instruction. We study the impact of rotation instructions in more detail in Section 7.2. In total, our MixColumns implementation uses 27 `XOR` instructions and 16 rotations.

The other parts of the implementation are straightforward or are very similar to the Cortex-M implementation [SS16].

**Result.** Key expansion and conversion of all round keys to the bitsliced format takes 1239 clock cycles and 16 stack bytes. For benchmarking encryption, we selected a fixed plaintext size of 4096 bytes. This can be encrypted or decrypted with AES-128-CTR in 509622 cycles, or at 124.4 cycles per byte. 60 stack bytes are used to store callee-save registers and copies of a few other values.

## 4 ChaCha

ChaCha is a family of stream ciphers based on Salsa20 [Ber08]. It is known for its high speed in software and together with a message authentication code called Poly1305 it is used in TLS and OpenSSH [Ber05b,LCM+16,NL18].

ChaCha starts by loading constants, a 256-bit key, a 96-bit nonce, and a 32-bit counter into a 512-bit state. With RV32I, there are enough registers to keep the full state in registers during the whole computation. ChaCha20 is the most commonly used ChaCha variant that performs 20 rounds. Every round contains 4 quarter-rounds and every quarter-round consist of 4 additions, 4 xors, and 4 rotations. Because the RV32I base ISA lacks rotation instructions, every

rotation has to be replaced by 2 shift instructions and an `OR` instruction. In total we require 20 single-cycle instructions to implement the ChaCha quarter-round.

The other parts are straightforward. As long as the input to the stream cipher is longer than 64 bytes, we generate the key-stream and xor it with the input in blocks of 64 bytes. If the input length is not divisible by 64 bytes, there will be some bytes remaining that still need to be encrypted. For those, another 64 bytes of key-stream is generated. These are xored with the input first per word (4 bytes) and finally per byte.

### 4.1 Result

Our implementation of the complete Chacha20 stream cipher requires 32 bytes in the DTIM memory to store constants and another 40 bytes on the stack to store callee-save registers. We benchmark speed with the same fixed input size of 4096 bytes as we used for the bitsliced AES-128-CTR implementation. This can be encrypted or decrypted in 114365 clock cycles, or at 27.9 cycles per byte.

## 5 Keccak

The Keccak-$f$ family of permutations was designed in the course of the SHA-3 competition [BDPA08]. The Keccak-$f$[1600] instance is now at the core of the SHA-3 hash functions and the SHAKE extendable output functions standardized by NIST [NIS15]. It it also used in various other cryptographic functions. An optimized implementation of the Keccak-$f$[1600] permutation therefore benefits all those schemes. In the *Keccak implementation overview* a number of implementation techniques are discussed, including those relevant to 32-bit software implementations [BDP+12].

### 5.1 Efficient Scheduling

The permutation operates on a relatively large state of 1600 bits. Having the RV32I architecture in mind, this state is clearly too large to be able to contain the full state in registers. It is therefore required to swap parts between memory and registers during the computation. Loads from memory and stores to memory are relatively expensive, so for an efficient implementation it is important to keep the number of loads and stores at a manageable level.

The permutation iterates a round function consisting of the steps $\theta$, $\rho$, $\pi$, $\chi$, and $\iota$. The first four steps each process the full state. Computing them one by one would therefore use many loads and stores. The designers described a technique to merge the computation of these steps such that only two passes over the full state are required per round. This is explained in detail in the implementation overview document [BDP+12]. We follow the same approach for our RISC-V implementation.

### 5.2 Bit Interleaving

The state is structured as $5 \times 5$ 64-bit lanes. On a 32-bit architecture, one could simply split the lanes into two halves that are stored in separate registers, but it is more efficient to interleave the bits. The bits with an 'even' index are then stored in one register and those with an 'odd' index in another. The lane-wise translations in $\theta$ and in $\rho$ then become 32-bit rotations. It has been mentioned before that the RV32I base ISA does not contain rotation instructions.

In fact, with both approaches a lane-wise translation costs 6 single-cycle instructions. The difference is that with the interleaved representation, for translation offsets of 1 or -1 only a single register has to be rotated. Those then only cost 3 single-cycle instructions. Because this is the case for 6 out of 29 lane translations per round, bit interleaving still provides a nice improvement.

### 5.3 Lane Complementing

The $\chi$ step computes 5 XOR, 5 AND, and 5 NOT (64-bit) operations on the lanes of every plane of the state. There are 5 such planes and we only have 32-bit instructions, so in total $\chi$ requires 50 `XOR` instructions, 50 `AND` instructions, and 50 `XORI` instructions with $-1$ as immediate per round. The number of `XORI` instructions can be reduced to 10 by representing certain lanes by their complement and by changing some `AND` instructions into `OR` instructions. This comes at the cost of applying a mask at the beginning and at the output of Keccak-$f$. This technique is also described in more detail in the implementation overview document [BDP+12]. This is a useful technique on the RISC-V, because there is no instruction that combines an `AND` with a `NOT` of one of its operands, as is the case on some other architectures.

### 5.4 Result

Our RISC-V implementation is inspired by the fastest Cortex-M3/M4 implementation known to us, which is the `KeccakP-1600-inplace-32bi-armv7m-le` implementation in the eXtended Keccak Code Package[5]. The main differences are that we add lane complementing and that we keep more variables in registers instead of having to store them on the stack.

Memory-wise our implementation requires 192 bytes in the DTIM memory for the round constants and 20 bytes on the stack. To benchmark speed, we measure a single execution of the permutation from the instruction cache. This takes 13774 clock cyles, or 68.9 cycles per byte.

## 6 Arbitrary-Precision Arithmetic

Arbitrary-precision arithmetic on integers, also called big-integer arithmetic, is a core component of public-key cryptographic systems such as RSA and elliptic-curve cryptography. We consider addition and two multiplication algorithms,

[5] https://github.com/XKCP/XKCP

schoolbook and Karatsuba multiplication. The multiplication algorithms make heavy use of the RISC-V M extension. This provides a $32 \times 32$-bit multiplier and the `MUL` and `MULHU` instructions, among some others that we will not use. `MUL` gives the lower 32 bits of the 64-bit multiplication result, `MULHU` the higher 32 bits, interpreting its operands both as unsigned values. On the E31, they each have a result latency of 2 clock cycles.

### 6.1 Carries and Reduced-Radix Representations

An arbitrarily large integer is usually represented as a vector of CPU words. The part of the integer that fits in a single CPU word is called a *limb*. Arithmetic on arbitrary-precision integers then translates to an algorithm that performs arithmetic with the limbs, as those are the only units that a CPU can work with.

The addition of two limbs may result in an overflow. On most CPU architectures, whether an overflow occurred is stored in a carry flag. This can then subsequently be used in an add-with-carry operation.

RISC-V, however, does not specify the existence of a carry flag. Instead, the carry needs to be explicitly computed every time. The `SLTU` instruction (set less than unsigned) is very useful for this. Let $r = a+b$, where $r$, $a$, and $b$ are unsigned 32-bit values. Then the addition produces a carry $c$ whenever $r < a$ (or $r < b$). In assembly, this can be implemented with `ADD r, a, b; SLTU c, r, a`.

This explicit carry handling can be the cause of a significant overhead. One way to avoid this is by guaranteeing that a carry will not occur. This is possible by using a reduced-radix representation, also known as a redundant integer representation. Instead of the full 32 bits, one can use the least significant $k$ bits of every limb, such that the most significant $32 - k$ bits are zero at the start. This *radix-*$2^k$ *representation* requires more limbs to store an integer of the same bit length, but the advantage is that one can do one or even many additions without producing a carry. The carries are accumulated in the most significant $32 - k$ bits of the same limb. Only in the end they may need to be added to the next limb to get back to a unique integer representation.

What is more efficient is highly application-dependent, as that determines how many and which operations are computed on the integers. We aim to keep this generic by studying the performance of both types of addition and multiplication algorithms for an arbitrary number of limbs, without specifying a precise radix.

### 6.2 Addition

Arbitrary-precision addition is a simple operation that consists of a carry chain for full-limb (radix-$2^{32}$) integer representations. The operands are added limb-wise, where every such addition may result in an overflow that has to be carried to the next limb.

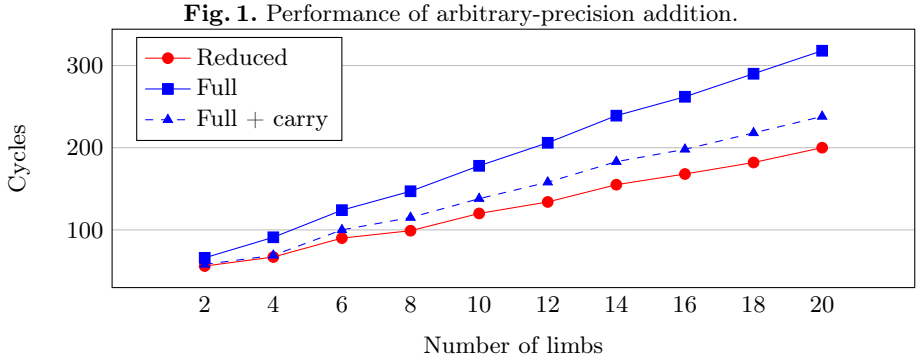
Figure 1 shows how both reduced and full representations compare. It appears that carry handling is a significant part of the computational effort. A reduced-radix representation is approximately 37% faster than a non-redundant

representation. However, one should note that with a reduced-radix representation, more limbs will be required. For example, it is fairer to compare the reduced-radix representation with 12 limbs to the full-radix representation with 10 limbs, when only 27 bits are used in every limb, i.e., in radix $2^{27}$. The cost of carrying at the end to get back to a unique representation also needs to be taken into account.

Still, it appears that reduced-radix representations can be beneficial when multiple additions have to be computed.

Figure 1 also shows the estimated cost of full-limb addition if there were a carry flag and add-with-carry operation. This is discussed in Section 7.4.

**Fig. 1.** Performance of arbitrary-precision addition.




### 6.3 Schoolbook Multiplication

Many algorithms exist to implement arbitrary-precision multiplication. One of the simplest ones is called schoolbook multiplication. With the schoolbook multiplication method multiplying two $n$-limb integers takes $n^2$ single-limb (in our case: $32 \times 32$-bit) multiplications.

A non-reduced representation still has to perform some carry handling, but the cost of this is much less significant with multiplication compared to addition, as can be seen in Figure 2. Schoolbook multiplication with reduced-radix representations is only 8% faster than multiplication with non-reduced representations. And because more limbs will be required, there is actually very little advantage to using a reduced-radix representation.

This can be explained by the fact that the `LW`, `SW`, `MUL`, and `MULHU` instructions take more CPU cycles compared to the simpler bitwise and arithmetic instructions. A reduced-radix representation does not avoid this more significant part of the cost of the inner loop of the algorithm.

### 6.4 Karatsuba Multiplication

The Karatsuba algorithm was the first multiplication algorithm that was discovered that has a lower asymptotic time complexity than $\mathcal{O}(n^2)$ [KO63]. Instead, it can recursively multiply arbitrary-precision integers in $\mathcal{O}(n^{\log_2 3})$. It succeeds in this by effectively trading an $n$-limb multiplication for 3 $\frac{n}{2}$-limb multiplications and several additions.

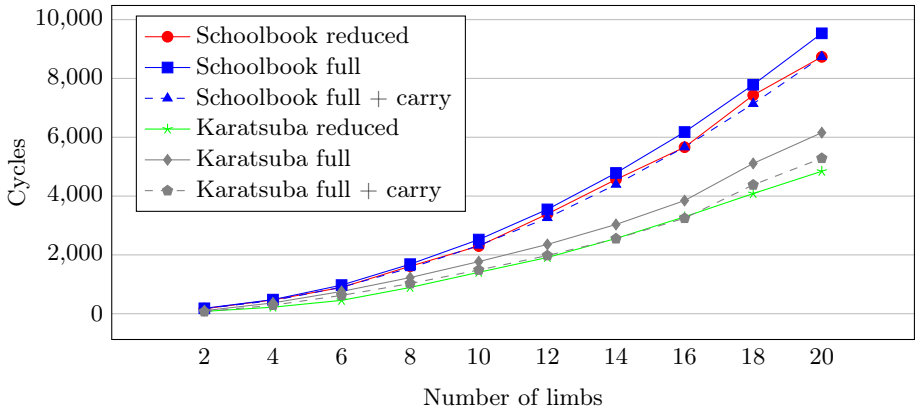
The details of the Karatsuba multiplication algorithm have been extensively covered in other works. It is used in many implementations of cryptographic schemes, most notably for RSA [SV93] and elliptic-curve cryptography [BCL14,DHH+15,FA17], but also for more recent lattice-based [KRS19] and isogeny-based [SLLH18] post-quantum cryptography.

We implement a single level of subtractive Karatsuba that multiplies two equal-length operands with an even number of limbs. This restriction is only there to simplify the performance analysis by being able to omit a few implementation details for dealing with special cases. The case of equal-length operands with an even number of limbs is also in fact the most common scenario in cryptography, which is why it is not even necessarily a relevant restriction.

Figure 2 shows that even for a very small number of limbs, the Karatsuba multiplication algorithm is already faster than schoolbook multiplication. This is not obvious, as the cost of the extra additions and constants in the complexity typically imply a certain threshold where Karatsuba starts to perform better.

The gap between reduced-radix representations and non-reduced or full-limb representations is slightly larger than with schoolbook multiplication, which can be partially explained by the extra additions that need to be computed. Its difference is now approximately 21%. Whether this suffices to make a reduced-radix representation more efficient in practice is hard to conclude from this data. It will depend on the specific application.

**Fig. 2.** Performance of arbitrary-precision multiplication.



# 7 Extending RISC-V and Discussion

## 7.1 Speed Comparison with ARM Cortex-M4

The RISC-V platform that we used has similarities with the ARM Cortex-M family of microcontrollers. Both have 32-bit architectures and are designed for cheap embedded applications. The main difference is that ARM microcontrollers have a richer (proprietary) instruction set. For example, rotations are first-class citizens in the ARMv7-M instruction set and can even be combined with arithmetic instructions in a single CPU cycle. The architecture also provides nicer bit-extraction instructions, a carry flag and a single-cycle add-with-carry. On the other hand, RV32I comes with more registers, which may benefit cryptographic primitives that have a larger state. This can save a lot of overhead of having to spill values to the stack.

At first sight, it is unclear which weighs more heavily. We therefore compare the relative performance of our optimized cryptographic primitives with their counterpart on the Cortex-M4. There already exist AES, ChaCha20, and Keccak-$f[1600]$ assembly implementations optimized for that platform.

**Table 1.** Comparison between the E31 (RV32IMAC) and the Cortex-M4.

| Scheme | Cortex-M4 | | E31/RV32IMAC | |
|---|---|---|---|---|
| | Cycles | Cycles/byte | Cycles | Cycles/byte |
| Table AES-128 key schedule | 254.9 [SS16] | | 340 | |
| Table AES-128 | 644.7 [SS16] | 40.3 | 912 | 57.0 |
| Bitsliced AES-128 key schedule | 1033.8 [SS16] | | 1239 | |
| Bitsliced AES-128-CTR | 414617.6* | 101.2 [SS16] | 509622* | 124.4 |
| ChaCha20 encrypt | 56934.4* | 13.9 [HRS16] | 114365* | 27.9 |
| Keccak-$f[1600]$ permute | 12969† | 64.8 | 13774 | 68.9 |

* When encrypting 4096 bytes.
† We benchmarked `KeccakP1600_Permute_24rounds` from https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/Optimized32biAsmARM/KeccakP-1600-inplace-32bi-armv7m-le-gcc.s on an STM32F407.

Table 1 provides the exact numbers, while Figure 3 visualizes their relative speed. It can be seen that all schemes require more cycles with the RV32I architecture. Of course, this does not directly relate to speed in practice, as we do not take the different CPU clock frequencies into account. It shows that all schemes use instructions that can be computed in a single cycle on the Cortex-M4, but not with RV32I. Relatively, it appears that ChaCha20 has the largest disadvantage because of this. For this scheme, the lack of rotation instructions seems to outweigh the possibility to keep the full state in registers without spilling to the stack, something that is necessary on the Cortex-M4. When the algorithms

are compared to each other, their differences remain very similar with the RV32I instruction set architecture.

### 7.2 The RISC-V B Extension

The RISC-V foundation reserved the $B$ extension for bit manipulation instructions. In 2017 there was an active working group that would develop a specification for the B extension. However, apparently the working group dissolved in November 2017 for bureaucratic reasons[6]. An independent fork was developed outside of the RISC-V foundation, which was merged back and made official again in March 2019.

The latest V0.37 draft specification adds 37 new instructions[7]. While it is unknown which will be used in the end, it is likely that this will include some type of rotation, byte shuffle, and bit-extraction instructions. The current specification also includes an and-with-complement instruction. This would imply that lane complementing would no longer be advantageous for Keccak-$f$[1600].

We estimate the impact that this extension will have, focussing on rotations. For each scheme, we counted all instruction sequences that could be replaced by a rotation instruction. Our table-based AES does not use rotations, while the bitsliced AES implementations uses 144 of them. ChaCha20 uses 320 rotation instructions and Keccak-$f$[1600] 1248.

Assuming that the rotation would be possible in a single cycle, we then calculated how many CPU cycles would be saved by having this instruction. The results can be seen in Table 2 and Figure 3. For Keccak-$f$[1600] and especially for ChaCha20, rotations are a significant part of their computational cost. From Figure 3 it is clear that with rotations, the Keccak-$f$[1600] permutation can be computed in fewer cycles than on the Cortex-M4. This is due to the fact that more registers are available.

**Table 2.** Estimated improvement with a rotation instruction.

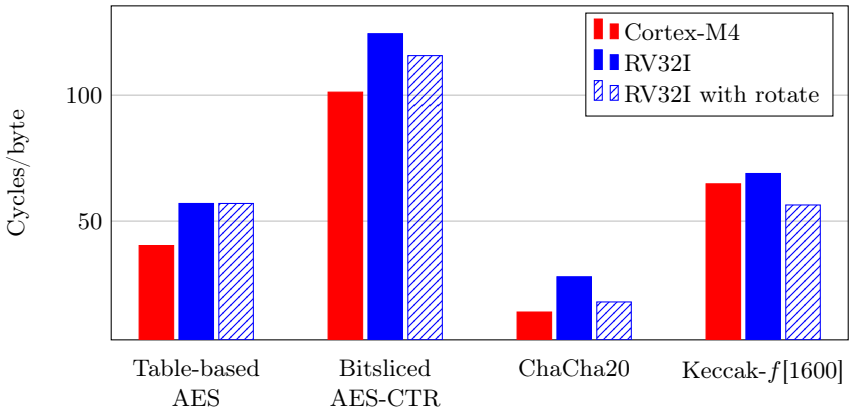
| **Scheme** | Rotations | Improvement | Cycles/byte |
|---|---|---|---|
| Table-based AES | 0 | 0.0% | 57.0 |
| Bitsliced AES | 144 | 7.0% | 115.7 |
| ChaCha20 | 320 | 35.8% | 17.9 |
| Keccak-$f$[1600] | 1248 | 18.1% | 56.4 |

### 7.3 Number of Registers

We already discussed some consequences of the large number of registers that are available on the performance of these implementations. Especially ChaCha and

[6] https://groups.google.com/forum/#!forum/riscv-xbitmanip
[7] https://github.com/riscv/riscv-bitmanip

**Fig. 3.** Speed of cryptographic primitives.




Keccak-$f$[1600], but also the bitsliced AES implementation, benefit from having to spill fewer intermediate values to the stack. It is noteworthy to mention that the RV32E instruction set, which is nearing its completion and which is intended to target embedded devices, will most likely decrease the number of registers from 32 back to 16 [RIS17]. This will set back the performance of aforementioned schemes, but this may be compensated by supporting the B extension with a rotation instruction.

### 7.4 Carry Flag

In Section 6 we studied the performance of arbitrary-precision addition and multiplication with and without reduced-radix integer representations. We now estimate how full-limb representations would perform if an RV32I core was extended with a carry flag and an add-with-carry instruction. We assume that this instruction would have a result latency of a single CPU clock cycle, similar to a regular addition instruction.

For addition, 4 cycles per limb would be saved in our implementation. We then subtracted $4n$ cycles from the full-radix addition results, where $n$ is the number of limbs. The result can be seen in Figure 1. As is to be expected, addition with this instruction is almost as fast as reduced-radix representation, the only difference being the top (most significant) limb that gets set.

With schoolbook multiplication $2n^2$ cycles are subtracted, as we can save 2 cycles in the inner loop with the add-with-carry instruction, which is executed $n^2$ times. For Karatsuba multiplication we computed that the add-with-carry instruction would save $27\frac{n}{2} + 6\left(\frac{n}{2}\right)^2$ cycles. The quadratic term comes from the cycles that are saved with the schoolbook multiplications and the linear part from the cycles that are saved with additions. Figure 2 contains plots for both estimates. With an add-with-carry instruction both schoolbook and Karatsuba multiplication would be approximately as fast as their reduced-radix counterparts.

The reduced-radix implementations use more limbs and still need to carry at the end, so it is appears that an add-with-carry instruction completely compensates for any advantage that a reduced-radix implementation might give.

## 8 Conclusion

We showed how AES, ChaCha, and Keccak-$f$ can be implemented efficiently on the 32-bit variant of the promising open-source RISC-V architecture. We also showed how arbitrary-precision addition and multiplication can be implemented and studied the performance of all these primitives. As the RISC-V is an open design intended to be extensible, we showed for several features, such as a rotation instruction and an add-with-carry instruction, how much improvement exactly could be gained by adding these features. These numbers are essential for making reasonable trade-offs in software-hardware co-design and we hope that they will be found useful by a wide audience.

## References

BCL14. Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 316–334. Springer, Heidelberg, September 2014. 12

BDP+12. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, May 2012. https://keccak.team/files/Keccak-implementation-3.2.pdf. 8, 9

BDPA08. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. NIST SHA-3 Submission, October 2008. https://keccak.team/obsolete/Keccak-main-1.0.pdf. 8

Ber05a. Daniel J. Bernstein. Cache-timing attacks on AES, April 2005. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 5

Ber05b. Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, Heidelberg, February 2005. 7

Ber08. Daniel J. Bernstein. ChaCha, a variant of Salsa20. The State of the Art of Stream Ciphers – SASC, January 2008. https://cr.yp.to/chacha/chacha-20080120.pdf. 7

BP10. Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *LNCS*, pages 178–189. Springer, May 2010. 7

BS08. Daniel J. Bernstein and Peter Schwabe. New AES software speed records. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008: 9th International Conference in Cryptology in India*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, Heidelberg, December 2008. 5

DHH+15. Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2):493–514, Dec 2015. 12

DR02. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. 5

FA17. Hayato Fujii and Diego F. Aranha. Curve25519 for the Cortex-M4 and beyond. In *Progress in Cryptology – LATINCRYPT 2017*, volume To appear, 2017. 12

HRS16. Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. ARMed SPHINCS - computing a 41 KB signature in 16 KB of RAM. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 9614 of *Lecture Notes in Computer Science*, pages 446–470. Springer, Heidelberg, March 2016. 13

KO63. Anatolii Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962. 12

Kön08. Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer, Heidelberg, April 2008. 5

KRS19. Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In *Applied Cryptography and Network Security – ACNS 2019*, volume To appear of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2019. https://eprint.iacr.org/2018/1018.pdf. 12

KS09. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, Heidelberg, September 2009. 5, 6

LCM+16. Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. *RFC 7905: ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. Internet Engineering Task Force, June 2016. https://tools.ietf.org/html/rfc7905. 7

NIS15. NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS 202, August 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 8

NL18. Yoav Nir and Adam Langley. *RFC 8439: ChaCha20 and Poly1305 for IETF Protocols*. Internet Research Task Force, June 2018. https://tools.ietf.org/html/rfc8439. 7

OST06. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, Heidelberg, February 2006. 5

RIS17. RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume 1: User-Level ISA, Document Version 2.2*, May 2017. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. 1, 3, 15

SiF17. SiFive, Inc. *SiFive FE310-G000 Manual, v2p3*, October 2017. https://sifive.cdn.prismic.io/sifive/4d063bf8-3ae6-4db6-9843-ee9076ebadf7_fe310-g000.pdf. 4

SiF18. SiFive, Inc. *SiFive E31 Core Complex Manual, v2p0*, June 2018. https://sifive.cdn.prismic.io/sifive/b06a2d11-19ea-44ec-bf53-3e4c497c7997_sifive-e31-manual-v2p0.pdf. 4

SLLH18. Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):1–20, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7266. 12

SS16. Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, Heidelberg, August 2016. 7, 13

SV93. Mark Shand and Jean Vuillemin. Fast implementations of RSA cryptography. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 252–259, June 1993. 12

WJW+18. Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems – XMSS hardware accelerators for RISC-V. Cryptology ePrint Archive, Report 2018/1225, 2018. https://eprint.iacr.org/2018/1225. 2