

Architecture Support for Bitslicing

Pantea Kiaei, *Student Member, IEEE*, Tom Conroy, and Patrick Schaumont, *Senior Member, IEEE*

Abstract—The bitsliced programming model has shown to boost the throughput of software programs. However, on a standard architecture, it exerts a high pressure on register access, causing memory spills and restraining the full potential of bitslicing. In this work, we present architecture support for bitslicing in a System-on-Chip. Our hardware extensions are of two types; internal to the processor core, in the form of custom instructions, and external to the processor, in the form of direct memory access module with support for data transposition. We present a comprehensive performance evaluation of the proposed enhancements in the context of several RISC-V ISA definitions (RV32I, RV64I, RV32B, RV64B). The proposed 14 new custom instructions use 1.5x fewer registers compared to the equivalent functionality expressed using RISC-V instructions. The integration of those custom instructions in a 5-stage pipelined RISC-V RV32I core requires 4.96% overhead. The proposed bitslice transposition unit with DMA provides a further speedup, changing the quadratic increase in execution time of data transposition to linear. Finally, we demonstrate a comprehensive performance evaluation using a set of benchmarks of lightweight and masked ciphers.

Index Terms—Bitslicing, instruction set extension, direct memory access, system-on-chip, hardware extension, computer architecture.



1 INTRODUCTION

Bitslicing was first introduced as a programming model to boost the throughput of the software implementation of the Data Encryption Standard (DES) cryptographic algorithm [1]. Since then, researchers have explored applications that can benefit from this model of programming in security [2], [3], [4], [5] and dynamic word-length computation [6], [7] among others.

Bitslicing, as a programming model, does not require any changes to the underlying design of the processor. However, bitsliced programs bear significant memory spills due to their extensive amount of live registers [8]. Therefore, hardware support for bitslicing can lead to a significant increase in performance of various bitsliced applications.

Today, many digital circuits consist of a System-on-Chip (SoC). In such systems, hardware support for bitslicing can be in the form of instruction extension in the processor implementation or it can be a hardware module accessible by the processor through a bus. Our goal in this work is to integrate both of these types of hardware support for bitslicing into an SoC. Even though our focus is mostly on security applications, non-security related applications of bitslicing can equally benefit from part of our proposed hardware extensions.

As the open-source RISC-V Instruction Set Architecture (ISA) is gaining more attention both in research as well as in industry, domain-specific Instruction Set Extensions (ISEs) are becoming more and more relevant [9], [10], [11]. In our previous work [12], we proposed Skiva, a 32-bit ISE for the SPARC V8 ISA. Skiva supports protection against

a combination of active and passive physical attacks, i.e., power Side-Channel Analysis (SCA), fault injection, and timing SCA. These protections are in the form of masking [13], redundant computation, and bitslicing.

To account for hardware support for bitslicing at the instruction-set level, in this work, we port the this work, we port the ISE in Skiva to RISC-V and call it Skiva-V. Additionally, we propose the 64-bit version of Skiva-V which supports extra security-related modes. Furthermore, as it is crucial to evaluate the gain achieved by one ISE over the existing ISA, we evaluate Skiva-V in terms of the newly proposed bit-manipulation ISA for RISC-V (RV32B, RV64B)¹. Our focus in this work is on the performance analysis of the proposed instructions. For the security analysis of the custom instructions in Skiva, we refer the reader to our previous work [12]. Furthermore, we note that several authors have proposed a security analysis for similar bitsliced masked software [3], [14], [15].

Finally, we propose a Direct Memory Access (DMA) module, called T-DMA, which is capable of transposing data as part of a memory block transfer. This capability of T-DMA in itself shows how an extra-processor support for bitslicing can be beneficial for any bitsliced implementation. However, we further tune this module to add support for our security-related programming needs, namely on the fly masking and redundancy generation/checking. An extra-processor extension comes with the advantage of integrability with commercial processor cores. We describe how we integrate T-DMA with Skiva-V implementation into a SoC.²

The rest of the paper is as follows: Section 2 gives an overview of the concepts underlying the proposed system. Section 3 describes the definition of the custom instructions in Skiva-V, their ISA-level performance analysis, and implementation footprint. Section 4 demonstrates how to generate bitsliced programs for Skiva-V. Section 5 presents

- P. Kiaei and P. Schaumont were with the Department of Electrical and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA, 01609.
E-mail: {pkiaei,pschaumont}@wpi.edu
- T. Conroy was with the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnique Institute and State University, Blacksburg, VA, 24061.
E-mail: tconroy@vt.edu

This research was supported in part by NSF Award 1931639.

1. <https://github.com/riscv/riscv-bitmanip>
2. We will open-source the design files and the modified GCC compiler before the paper's publication.

our proposed DMA module with support for transposing, masking, and duplicating the data. It further describes its functionality, design, and synthesized implementation footprint. Section 6 describes the integration of Skiva-V processor core and T-DMA into an SoC architecture. Section 7 demonstrates benchmarks to emphasize the impact of hardware support in performance of bitsliced and masked software. Finally, Section 8 concludes the paper.

2 PRELIMINARIES

In this section, we describe a brief introduction to the concepts used in our proposed system.

2.1 Bitslicing

Bitslicing, first introduced by Biham [1], is a technique originally proposed to increase the throughput of a program by running multiple instances of a code in parallel. In bitslicing, all the variables are transposed so that each register contains only one bit of the variable. For example, if a variable is 32 bits wide, in bitsliced program it will reside in 32 different registers and use one bit of each. Each register of width ω then will have the capacity to hold one bit of ω different variables. Consequently, the program needs to be adjusted to work on one bit of its variables at a time. This implies that the adjusted (i.e., bitsliced) program can only contain bit-wise logic operations. Therefore, the bitsliced program will be capable of ω parallel computations.

A fully-bitsliced program needs to be flattened (no branches). In a flattened program, the run-time of the program is known and data-independent. This property of bitslicing benefits the security-sensitive programs as it averts timing side-channel leakage (i.e., correlation between the run-time and the internal data of a program). Furthermore, bitslicing provides a proper base to combine our masking and redundant computation schemes as described in the next section.

2.2 Masking

Power-based SCA [16], [17], as a subset of active physical implementation attacks, has shown vulnerabilities in the implementation of algorithms which are expected to be secure at the algorithm level. In power SCA, the correlation between the power consumption and the internal data is explored to find information about the processed data. A widely-adopted countermeasure against this type of attack is *masking* which tries to break this correlation.

In masking, each signal or variable is divided into *shares* that are independent from the original data. The number of shares depends on the masking scheme. In the d^{th} -order masking scheme, each data bit is divided into $d + 1$ shares. Knowing any strict subset of these shares will not disclose any information about the original data, while knowing all of the shares can reproduce the original data. A simple way to generate these shares is by applying *Boolean masking*. For instance, in Boolean masking for the 1^{st} -order masking scheme, a random bit r is generated (from a uniform distribution) per each original bit b . The shares of the bit b will be computed as the tuple $(b \oplus r, r)$ where \oplus is the exclusive-or operation. Knowledge about one share (either r

or $b \oplus r$) will not give any information about the original data b , however, by knowing both of these shares the original data can be disclosed as the exclusive-or result of the two shares $((b \oplus r) \oplus r = b)$.

Once each data is broken into independently-distributed shares, the algorithm should be modified to work on the shares of the inputs and the intermediate data to generate the shares of the outputs. The operations in the algorithm are categorized into *linear* and *non-linear* operations. An operation is linear if a uniform distribution of its inputs results in a uniform distribution for its outputs. Masking is then applied to each operation according to its linearity. In a linear operation, each share of the output can be implemented as a function of at most one share of each input. This property, however, does not hold for non-linear operations and there exists a vast body of research on how a non-linear operation can be masked [18], [19], [20].

In this work, we break every algorithm into a combination of operations from the set $\{\text{XOR}, \text{XNOR}, \text{AND}, \text{NOT}\}$. Since this set of operations is functionally complete, every operation in the algorithm can be written as a combination of these operations. The AND operation is therefore the only non-linear operation that can appear in the adjusted algorithm. We follow the parallel masked multiplication method proposed by Barthe et al. [18] for our AND operation and a normal masked implementation for our linear operations.

2.3 Redundant Computation

Fault injection [21] is another type of implementation attacks. Redundant computation is a technique to detect whether a fault has been injected in a circuit. In this technique, every computation is done multiple times and the results are compared. A mismatch between the results shows that a fault has happened. For n number of redundant computations, $n - 1$ faults can be detected.

In Skiva-V, our goal is to combine countermeasures against both fault injection and power SCA attacks. As shown in our previous work [12], when the redundant copies of the data are in complementary format, the intensity of power side-channel leakage is decreased. Therefore we support redundancy of two types: direct and complementary. In direct redundancy, the redundant data is a direct (uninverted) copy of the original data, whereas, in the complementary redundancy, half of the redundant copies will be in the inverted format to balance the power consumption of the direct copies.

3 PROCESSOR SUPPORT

We present an instruction set extension (ISE) for both the 32-bit and the 64-bit RISC-V ISAs called Skiva-V. The underlying data representations of Skiva-V are based on the masking order (D) and the spatial redundancy (R_s). In our 32-bit ISE, we follow the same data representations of our previous work, Skiva [12], supporting nine different configurations chosen from the sets $\mathcal{D} = \{1, 2, 4\}$ and $\mathcal{R}_s = \{1, 2, 4\}$. For our 64-bit ISE, we extend the 32-bit representations to add additional masking and redundancy modes. In this new configuration, Skiva-V supports sixteen different configurations from $\mathcal{D} = \{1, 2, 4, 8\}$ and $\mathcal{R}_s = \{1, 2, 4, 8\}$. In all

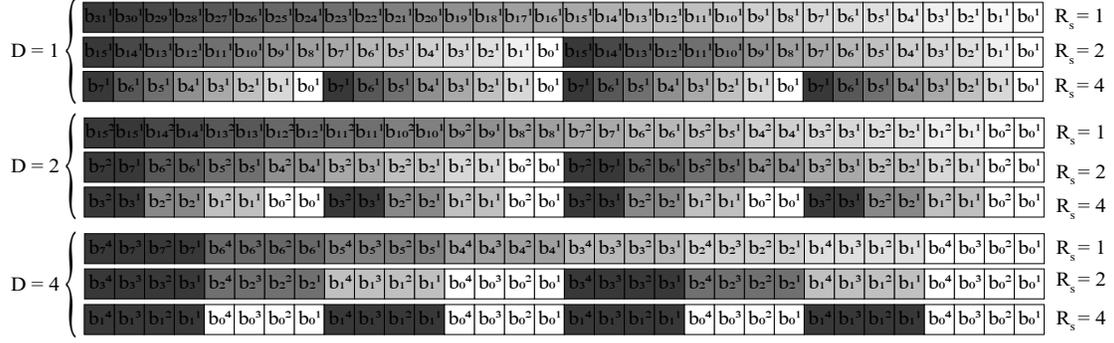


Fig. 1. Bitsliced data representation on 32-bit registers. b_i^j represents j^{th} share of data b_i . Shares of the same variable are shown with the same color.

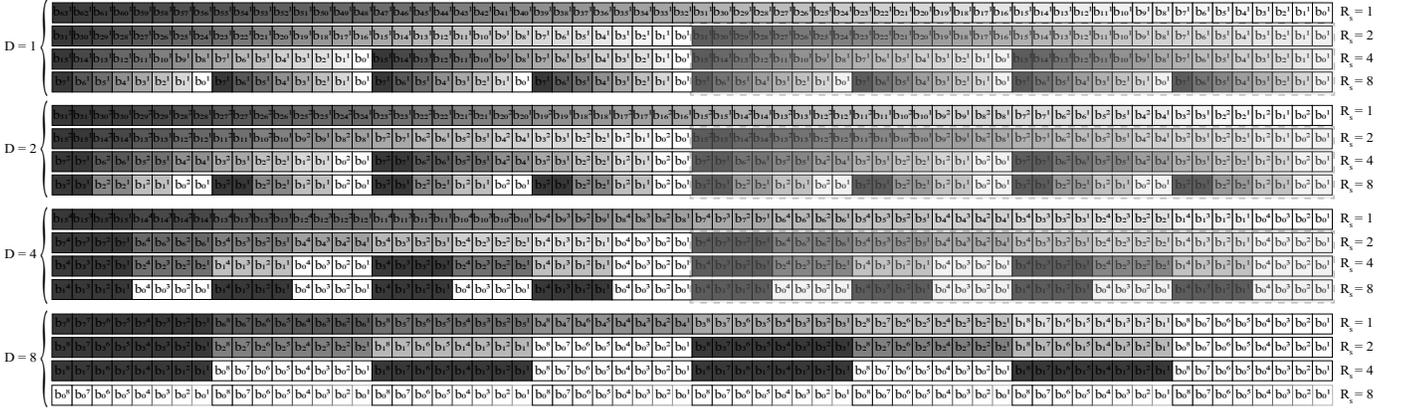


Fig. 2. Bitsliced data representation on 64-bit registers. b_i^j represents j^{th} share of data b_i . Shares of the same variable are shown with the same color. Parts enclosed in dashed lines show the nine possible configurations in the 32-bit architecture proposed in Skiva [12] also shown separately in Figure 1.

of these configurations, the D shares of the same variable reside in the adjacent bits of a register. Next to the shares of one variable, will sit the shares of the next variable for parallel computation. This pattern repeats in the same register R_s times for redundant computation. Thus in each (D, R_s) configuration for the N -bit architecture, Skiva-V supports $p = \frac{N}{D \times R_s}$ parallel computations. Figure 1 and Figure 2 show all the possible configurations in the 32-bit and 64-bit ISEs respectively. In these figures, the i subscripts in b_i data bits show different variables in parallel computation. To support both direct and complementary redundancy, the even-numbered redundant copies can be either inverted or direct.

In the rest of this section, we describe the instructions in Skiva-V, their implementation details and footprint, and how programmers can employ them in their codes.

3.1 Instruction Definitions

Our proposed instruction set extension for RISC-V is divided into three groups: instructions for bitsliced transposition, instructions for masked implementation, and instructions for redundant computation. In the following subsections, we describe each instruction. Table 1 shows the assigned opcodes and formats of the instructions in Skiva-V. The instructions' encodings in Skiva-V follow the RV32I base r-type and i-type instruction formats mentioned in

RISC-V ISA manual [22]. Each of the i-type instructions in Skiva-V has its own immediate encoding that clarifies the masking order (required for subrot instruction) or the redundancy scheme (required for red1/h and ftchk). We describe the immediate assignment of each instruction with their definition in the rest of this section.

Bitsliced transposition

We propose two instructions, i.e. `tr2l rd, rs1, rs2` and `tr2h rd, rs1, rs2`, which, if applied iteratively in the butterfly pattern, can transpose the data from normal representation to its bitsliced format. These instructions take two source registers and reorder their bits in the destination register interchangeably. Instruction `tr2l` reorders the lower half of the source registers while instruction `tr2h` reorders the upper half. To transpose the bitsliced data back to its normal representation, we proposed the inverse of the above instructions, i.e. `invtr2l rd, rs1, rs2` and `invtr2h rd, rs1, rs2`. Figure 3 shows how these instructions work. As an example, Figure 4 shows how four 4-bit registers can be transposed to their bitsliced positions in two iterations of applying these instructions. In general, for N N -bit registers, it takes $\log_2(N)$ iterations of applying the transposition instructions to completely transpose the bits.

TABLE 1
Opcode assignments in Skiva-V

Instruction	Type	funct7 (instr 31-25)	funct3 (instr 14-12)	opcode (instr 6-0)
subrot	i-type	—	0x0	0x0b (custom-0)
redl	i-type	—	0x1	0x0b (custom-0)
redh	i-type	—	0x2	0x0b (custom-0)
ftchk	i-type	—	0x3	0x0b (custom-0)
andc32 (only in 64-bit ISA)	r-type, logic	0x20	0x4	0x0b (custom-0)
andc16	r-type, logic	0x10	0x4	0x0b (custom-0)
andc8	r-type, logic	0x00	0x4	0x0b (custom-0)
xorc32 (only in 64-bit ISA)	r-type, logic	0x21	0x4	0x0b (custom-0)
xorc16	r-type, logic	0x11	0x4	0x0b (custom-0)
xorc8	r-type, logic	0x01	0x4	0x0b (custom-0)
xnorc32 (only in 64-bit ISA)	r-type, logic	0x22	0x4	0x0b (custom-0)
xnorc16	r-type, logic	0x12	0x4	0x0b (custom-0)
xnorc8	r-type, logic	0x02	0x4	0x0b (custom-0)
tr2l	r-type, transposition	0x00	0x5	0x0b (custom-0)
tr2h	r-type, transposition	0x10	0x5	0x0b (custom-0)
invtr2l	r-type, transposition	0x01	0x5	0x0b (custom-0)
invtr2h	r-type, transposition	0x11	0x5	0x0b (custom-0)

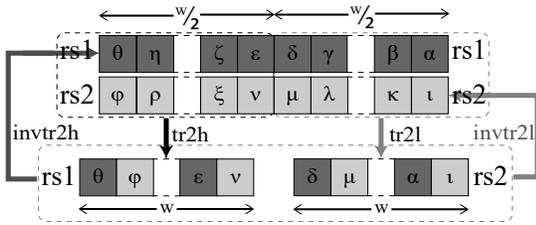


Fig. 3. (inv)tr2h and (inv)tr2l instructions. W represents the length of the registers which can be either 32 or 64 bits. All four instructions take two input registers and store the results in the destination register.

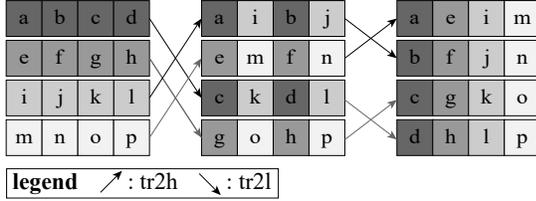


Fig. 4. Applying tr2l and tr2h instructions to four 4-bit registers iteratively in a butterfly pattern to transpose the bits for bitsliced implementation. To transpose the bits back to their initial positions, we can apply invtr2l and invtr2h from right to left.

Masked implementation

In our masked data manipulations, we follow the parallel masked multiplication gadget by Barthe et al. [18]. In this gadget, the shares of a variable are adjacent in a register and during the calculations, we need to rotate the adjacent shares. Rotating parts of a register independently is not part of the RISC-V ISA, however, in our masking schemes will be executed quite often. Hence we add this instruction to Skiva-V. In our 32-bit (resp. 64-bit) representation, Skiva-V supports masked implementation with 2 and 4 (resp. 2, 4, and 8) shares. Therefore we need to be able to rotate 2 and 4 (resp. 2, 4, and 8) consecutive bits in a 32-bit (resp. 64-bit) register. Therefore, we add an instruction (subrot rd, rs1, imm) which takes a source register and an immediate value. If the immediate value is 2/4/8 in 64-bit ISA) respectively 2/4/(8 in 64-bit ISA) consecutive bits will be rotated. Figure 5 shows how this instruction works.

Note. When using the subrot instruction, one must be

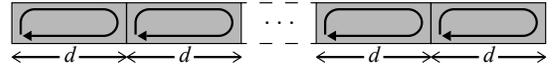


Fig. 5. subrot instruction. This instructions rotates d adjacent bits in a register where d is decided from the immediate input and follows the masking scheme ($d \in \{2, 4\}$ for 32-bit Skiva-V, $d \in \{2, 4, 8\}$ for 64-bit Skiva-V).

TABLE 2

Immediate value assignment for redh and redl instructions. W represents the word length (32 for the 32-bit and 64 for the 64-bit ISA). Source bits column signifies which bits in the source register are being duplicated.

Redundancy (R_s)	Source bits	redh/redl imm.
2 direct	W-1:0	2
2 compl.	W-1:0	3
4 direct	W/2-1:0	4
4 compl.	W/2-1:0	5
4 direct	W-1:W/2	6
4 compl.	W-1:W/2	7
8 direct (only in 64-bit ISA)	15-0	8
8 compl. (only in 64-bit ISA)	15-0	9
8 direct (only in 64-bit ISA)	31-15	10
8 compl. (only in 64-bit ISA)	31-15	11
8 direct (only in 64-bit ISA)	47-32	12
8 compl. (only in 64-bit ISA)	47-32	13
8 direct (only in 64-bit ISA)	63-48	14
8 compl. (only in 64-bit ISA)	63-48	15

careful not to use the same register for both the input and the result (i.e. $rs1 \neq rd$) since this will result in overwriting the shares of the same variable and transiently reducing the intended order of masking scheme. Fortunately, compilers support this type of criteria in their code generation process and we can ensure this property will be held by adding it to the back-end of the compiler (code generator) as a criteria specific to the subrot instruction.

Redundant computation

As mentioned in Section 2, Skiva-V supports both direct and complementary redundant computations. Direct redundancy enables fault detection while complementary redundancy also reduces the intensity of power side-channel leakage. To prepare data for redundant representation, we introduce instructions redl rd, rs1, imm and redh rd, rs1, imm to copy data (both directly and in inverted

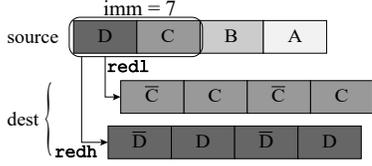


Fig. 6. Example for `redl` and `redh` instructions when the immediate value is 7. In both 32-bit and the 64-bit ISA, `immediate=7` means duplicating the upper half-word (16 bits and 32 bits respectively for the 32-bit and 64-bit architectures) in the complemented format. `redh/redl` copies the upper/lower half of the source's selected bits in its destination register.

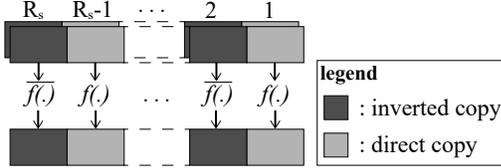


Fig. 7. Complementary logic operations on complementary redundant data.

manner) in the same register. The immediate field in these instructions decides which part of the input register has to be copied and whether it should follow direct redundancy or complementary redundancy. Table 2 shows the immediate value assignment for each redundancy mode.

To demonstrate in more detail how the bits are duplicated in the destination register, Figure 6 demonstrates the result of `redh` and `redl` instructions when their immediate value is 7. According to Table 2, this means the bits in the range $[W-1:W/2]$ ($W=32$ for 32-bit ISA and $W=64$ for 64-bit ISA) should be duplicated in a complementary format.

In cases where our data is in complementary redundancy format, we need a logic operation $\overline{f}(\cdot)$ to calculate $f(\cdot)$ on the direct copies and the inverse ($\overline{f}(\cdot)$) on the complemented copies to result in complemented outputs according to DeMorgan's theorem. Figure 7 shows the structure of complementary logic operations. Therefore, Skiva-V has logic instructions `andcn`, `xorcn`, and `xnorcn` that calculate the logic operation and its inverse on part of the data in their source registers. In the 32-bit (resp. 64-bit) instruction set, n can have the value of 8 and 16 (resp. 8, 16, and 32) to operate in direct/complementary format on n consecutive bits.

Finally, we propose an instruction in Skiva-V to check if the redundant copies of the data agree. The `ftchk rd, rs, 1, imm` instruction will check the redundant copies in the source register based on the immediate value and set the corresponding bit in the destination register to one if the copies of data do not agree (i.e. a fault is detected). To have continuity in the direct and complementary redundancy, the result of `ftchk` operation can be in the complementary format where the comparison result is copied both directly and inversely in the destination register.

Table 3 shows the immediate value encodings for the `ftchk` instruction. For example, if $R_s = 4$ and direct redundancy in the 32-bit ISA, i.e., immediate value is either 4 or 12, the comparison flags are calculated and stored in the destination register (`rd`) based on the source register (`rs`) as

TABLE 3
Immediate value assignment for `ftchk` instruction.

Redundancy (R_s)	ftchk immediate			
	32-bit	32-bit (compl. result)	64-bit	64-bit (compl. result)
2 direct	2	10	2	3
2 compl.	3	11	18	19
4 direct	4	12	4	5
4 compl.	5	13	20	21
8 direct (only in 64-bit ISA)	NA	NA	8	9
8 compl. (only in 64-bit ISA)	NA	NA	24	25

follows for the least significant 8 bits:

$$\begin{aligned}
 rd[i] = & (rs[i] \oplus rs[i + 8]) || \\
 & (rs[i] \oplus rs[i + 16]) || \\
 & (rs[i] \oplus rs[i + 24]); \\
 & \forall i \in [0, 7]
 \end{aligned}$$

The same calculated results will be duplicated directly (when immediate value is 4) or in inverse format (when immediate value is 12) to fill the remaining 16 bits of the destination register (`rd`).

3.2 ISA-level Performance Analysis

We evaluate our instruction set extension on RISC-V for its performance. For each proposed instruction, we write a C code defining the functionality of the instruction. On an implementation of Skiva-V, this C code corresponds to only one instruction. We cross compile the C code with GCC once for RV32 and RV64 instruction sets and once for the RISC-V with bit-manipulation extension (RV32B and RV64B). The GCC with B-extension currently only supports 31 out of 95 proposed instructions in the bit-manipulation draft.

Table 4 and Table 5 show the number of instructions from RISC-V ISA to implement the Skiva-V instructions. Based on our calculations, each of the 32-bit/64-bit Skiva-V operation replaces on average 22.34/29.98 instructions from the RV32/RV64 ISA and 21.84/29.59 instructions from the RV32B/RV64B ISA. All the proposed instructions pass the criteria of replacing a minimum of *three* instructions.

Although the reported numbers for RV32B and RV64B are not significantly different from RV32I and RV64I, the real advantage of the RISC-V's bit-manipulation extension can be much bigger but not yet supported by the GCC code generator. For instance, `rev.p rd, rs, 1` in RV32B/RV64B is functionally equivalent to `subrot rd, rs, 2` in Skiva-V 32/64-bit. However, this was the only instance we found in the bit-manipulation extension that was obviously equivalent to the instructions in Skiva-V.

Furthermore, we calculate the number of registers each instruction-equivalent code snippet uses on RV32I/RV32B and RV64I/RV64B as a measure of register pressure. We calculate the register use of each code snippet to that of its corresponding custom instruction. We make the worst case scenario assumption on the register usage in Skiva-V custom instruction that each r-type instruction (namely `andcn`, `xorcn`, `xnorcn`, `(inv)tr21/h`) uses 3 *distinct* registers and each i-type instruction (namely `subort`, `redl/h`, `ftchk`) uses 2 *distinct* registers. As shown in Table 4 and Table 5, even under our pessimistic assumption, on average, Skiva-V

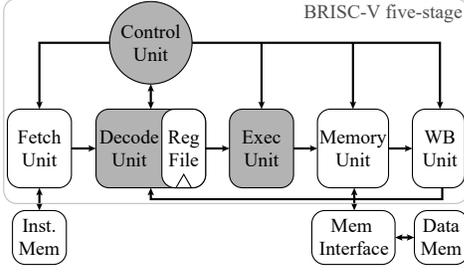


Fig. 8. Skiva-V implementation on BRISC-V five-stage processor. The grey boxes show the modified units for Skiva-V.

custom instructions use $1.47\times/1.65\times$ fewer registers compared to RV32I/RV64I and $1.58\times/1.75\times$ fewer registers compared to RV32B/RV64B ISA.

TABLE 4
ISA-level performance evaluation of Skiva-V 32-bit instructions

Skiva-V 32	RV32I		RV32B	
	# of instr	reg. use	# of instr	reg. use
tr2h rd, rs1, rs2	115	2 \times	115	2 \times
tr2l rd, rs1, rs2	115	2 \times	115	2 \times
invtr2h rd, rs1, rs2	115	2 \times	114	2 \times
invtr2l rd, rs1, rs2	115	2 \times	115	2 \times
subrot rd, rs, 2	9	1.5 \times	9	1.5 \times
subrot rd, rs, 4	9	1.5 \times	9	1.5 \times
redl rd, rs, 2	4	1 \times	3	1 \times
redh rd, rs, 2	4	1.5 \times	4	1.5 \times
redl rd, rs, 3	5	1 \times	4	1.5 \times
redh rd, rs, 3	5	1.5 \times	4	1.5 \times
redl rd, rs, 4	7	1.5 \times	7	1.5 \times
redh rd, rs, 4	8	1.5 \times	8	1.5 \times
redl rd, rs, 5	9	1.5 \times	9	1.5 \times
redh rd, rs, 5	11	1.5 \times	11	1.5 \times
redl rd, rs, 6	8	1.5 \times	8	1.5 \times
redh rd, rs, 6	8	1.5 \times	8	1.5 \times
redl rd, rs, 7	11	1.5 \times	11	1.5 \times
redh rd, rs, 7	10	2 \times	10	2 \times
ftchk rd, rs, 2	6	1 \times	5	1.5 \times
ftchk rd, rs, 3	6	1 \times	5	1.5 \times
ftchk rd, rs, 4	16	1.5 \times	16	1.5 \times
ftchk rd, rs, 5	17	2 \times	16	2.5 \times
ftchk rd, rs, 10	7	1 \times	6	1.5 \times
ftchk rd, rs, 11	8	1.5 \times	8	1.5 \times
ftchk rd, rs, 12	17	1.5 \times	17	1.5 \times
ftchk rd, rs, 13	19	2 \times	16	2.5 \times
andc16 rd, rs1, rs2	7	1 \times	6	1.67 \times
xorc16 rd, rs1, rs2	3	1 \times	3	1 \times
xnorc16 rd, rs1, rs2	4	1 \times	4	1 \times
andc8 rd, rs1, rs2	13	1.33 \times	13	1.33 \times
xorc8 rd, rs1, rs2	13	1.33 \times	11	1.33 \times
xnorc8 rd, rs1, rs2	11	1.33 \times	9	1.33 \times

3.3 Implementation

We integrate the 32-bit Skiva-V instructions into an in-order, five-stage pipeline implementation of the RISC-V RV32I ISA. For this implementation, we use the open-source BRISC-V [23] core. This core consists of five pipeline stages, namely fetch, decode, execute, memory, and write-back. The simplicity of the Skiva-V ISE architecture, enables the easy integration of the instructions which only affect the decode stage, the ALU unit in the execute stage, and the control unit of the processor. The changes applied to the processor are to decode the added instructions, execute them in the ALU, and bypass their outputs to the next immediate instructions in case of dependency (to reduce the number of inserted

TABLE 5
ISA-level performance evaluation of Skiva-V 64-bit instructions

Skiva-V 64	RV64I		RV64B	
	# of instr	reg. use	# of instr	reg. use
tr2h rd, rs1, rs2	244	2.33 \times	243	2.33 \times
tr2l rd, rs1, rs2	243	2.33 \times	244	2.33 \times
invtr2h rd, rs1, rs2	244	2.33 \times	243	2.33 \times
invtr2l rd, rs1, rs2	246	2.67 \times	247	2.33 \times
subrot rd, rs, 8	9	1.5 \times	9	1.5 \times
subrot rd, rs, 4	9	1.5 \times	9	1.5 \times
subrot rd, rs, 2	9	1.5 \times	9	1.5 \times
redh rd, rs, 10	15	1.5 \times	15	1.5 \times
redl rd, rs, 10	16	1.5 \times	16	1.5 \times
redh rd, rs, 11	17	2 \times	17	2 \times
redl rd, rs, 11	19	2 \times	19	2 \times
redh rd, rs, 12	16	1.5 \times	16	1.5 \times
redl rd, rs, 12	16	1.5 \times	16	1.5 \times
redh rd, rs, 13	19	2 \times	19	2 \times
redl rd, rs, 13	19	2 \times	19	2 \times
redh rd, rs, 14	16	1.5 \times	16	1.5 \times
redl rd, rs, 14	16	1.5 \times	16	1.5 \times
redh rd, rs, 15	18	2.5 \times	18	2.5 \times
redl rd, rs, 15	19	2 \times	19	2 \times
redh rd, rs, 2	4	1 \times	4	1 \times
redl rd, rs, 2	4	1 \times	3	1 \times
redh rd, rs, 3	5	1 \times	5	1.5 \times
redl rd, rs, 3	5	1 \times	4	1.5 \times
redh rd, rs, 4	7	1.5 \times	7	1.5 \times
redl rd, rs, 4	8	1.5 \times	7	2 \times
redh rd, rs, 5	13	2 \times	13	2 \times
redl rd, rs, 5	11	1.5 \times	10	2 \times
redh rd, rs, 6	8	1.5 \times	8	1.5 \times
redl rd, rs, 6	9	1.5 \times	8	2 \times
redh rd, rs, 7	10	2 \times	10	2 \times
redl rd, rs, 7	13	2 \times	13	2 \times
redh rd, rs, 8	16	1.5 \times	16	1.5 \times
redl rd, rs, 8	15	1.5 \times	15	1.5 \times
redh rd, rs, 9	19	2 \times	19	2 \times
redl rd, rs, 9	17	2 \times	17	2 \times
ftchk rd, rs, 18	7	1 \times	5	1.5 \times
ftchk rd, rs, 19	8	1.5 \times	7	2 \times
ftchk rd, rs, 20	20	2.5 \times	18	2.5 \times
ftchk rd, rs, 21	21	2.5 \times	19	2.5 \times
ftchk rd, rs, 24	39	2.5 \times	36	3 \times
ftchk rd, rs, 25	39	2.5 \times	36	3 \times
ftchk rd, rs, 2	6	1 \times	5	1.5 \times
ftchk rd, rs, 3	7	1 \times	6	1.5 \times
ftchk rd, rs, 4	18	2 \times	18	2 \times
ftchk rd, rs, 5	19	2 \times	19	2 \times
ftchk rd, rs, 8	36	2 \times	36	2 \times
ftchk rd, rs, 9	36	2 \times	36	2 \times
andc32 rd, rs1, rs2	7	1 \times	6	1.33 \times
xorc32 rd, rs1, rs2	4	1 \times	4	1 \times
xnorc32 rd, rs1, rs2	4	1 \times	3	1.33 \times
andc16 rd, rs1, rs2	9	1.33 \times	9	1.33 \times
xorc16 rd, rs1, rs2	4	1 \times	4	1 \times
xnorc16 rd, rs1, rs2	4	1 \times	4	1 \times
andc8 rd, rs1, rs2	9	1.33 \times	9	1.33 \times
xorc8 rd, rs1, rs2	4	1 \times	4	1 \times
xnorc8 rd, rs1, rs2	4	1 \times	4	1 \times

bubbles in the pipeline). Figure 8 shows the modified units in the five-stage BRISC-V processor core in grey.

Furthermore, to evaluate the area footprint of these instructions, we synthesize the Skiva-V implementation using the complementary metal oxide semiconductor (CMOS) 180nm technology. The addition of Skiva-V instructions comes at the cost of 4.96% area overhead. Table 6 shows the area footprint in more detail.

4 CODING SUPPORT

One of the challenges for bitsliced programming is its code generation. In our previous work [8], we presented an automated technique to generate code for a new programming model, namely Parallel Synchronous Programming (PSP),

TABLE 6

Area overhead of 32-bit Skiva-V integrated into a five-stage pipeline implementation of the RISC-V's RV32I ISA synthesized for the CMOS 180nm technology.

	RV32I	RV32I+Skiva-V	Overhead
area (μm^2)	145,313.46	152,515.91	4.96%

in which the expected run-time of the program is known at the program development stage. Examples of parallel synchronous programs have since been shown in software implementation of light-weight encryption ciphers [24] and variable-precision multiplication used in neural networks [7]. In this section, we demonstrate how bitsliced programs are a subset of the parallel synchronous programs and therefore the automated code generator for PSP (*i.e.* PSPCG) can be used to automate the generation of bitsliced code.

PSP is structurally similar to a finite state machine (FSM). Parallel synchronous programs consist of a *core function* with a status output that shows when the results are ready. This core function will be called iteratively until the status output shows the execution is done.

```
while (!stat_done) {
    core_f(inputs, &outputs, &stat_done);
}
```

We can treat bitslicing as a subset of PSP by unfolding the loop and adding it into the logic of the core function. This results in a flattened function containing only logic operations and therefore is in bitsliced format. Hence we can use the same automatic PSP code generation methodology (PSPCG) for bitsliced codes; We start with the hardware description of the bitsliced function in Verilog. Furthermore, we describe the logic instructions AND, XOR, XNOR, and NOT as logic gates in liberty format. We then use the open source synthesis tool, Yosys [25], to map our Verilog code to a net-list containing only the gates from the given library. From the net-list, we generate the bitsliced C code and replace the gates with their corresponding instructions from the desired ISA in inline assembly format.

Coding for Skiva-V

To generate bitsliced code for Skiva-V, we follow the PSPCG method as mentioned previously. In our custom library for the Yosys synthesis tool, instead of using Skiva-V-specific instructions, we use general instructions such as AND, OR, etc.. In our C code, we define each of these general instructions as a sequence consisting of Skiva-V instructions in the form of inline assembly. We add the proposed instructions to the RISC-V GCC assembler in order for the mnemonics of them to be recognizable by the assembler.

Cost of Transposition with ISE

Even though the transpose instructions (`(inv)tr2h` and `(inv)tr2l`) reduce the overhead of transposing the data significantly (Table 4 and Table 5), the overhead of transposition is still notable. For instance, in the 32-bit ISA, to perform a full transposition over 32 32-bit registers, five rounds of executing `tr2h` and `tr2l` instructions in a butterfly pattern is required. This adds up to 80 `tr2h` and 80 `tr2l`

TABLE 7

Footprint of transposing 32-bit data forward and backward between normal and bitsliced representations.

	tr2h	tr2l	invtr2h	invtr2l	sw	lw	mv	total
transpose	80	80	—	—	40	72	0	272
reverse transpose	—	—	80	80	37	69	4	270

instructions which, depending on the implementation and pipelining, takes around 160 clock cycles to run. If we also consider the memory load and store operations and moving data between registers, this process will be even longer. As shown in Table 7, it takes around 270 instructions to transpose thirty-two 32-bit registers.

The importance of this cost depends on the application. If a program is completely bitsliced, the transposition is only required once at the start and once at the end of the program. However, this overhead becomes troublesome when a program is a combination of bitsliced and non-bitsliced code and the data has to be transposed back and forth multiple times. This motivated our proposal for a Direct Memory Access (DMA) module that handles the transpositions on the fly. In the next section, we describe the proposed DMA.

5 DIRECT MEMORY ACCESS WITH TRANSPOSE SUPPORT

In this section, we describe the Transpose DMA (T-DMA) functionality, design, and the area footprint of the synthesized circuit. T-DMA is capable of performing the same operations as Skiva-V's instructions (`(inv)tr2l`, `(inv)tr2h`, `redl`, `redh`, `ftchk` on the fly on up to 32 consecutive memory locations at once.

5.1 T-DMA Functionality

The proposed T-DMA module is capable of the following:

- Transposing/Reverse transposing an arbitrary number of memory locations (up to thirty-two) starting from a source address and storing the result in given addresses starting from an arbitrary destination address.
- Generating/Removing the masking shares of data in the source address according to the given Skiva-V working mode.
- Generating/Removing the redundancy for the data stored in given source address according to the given Skiva-V working mode.
- Checking for consistency between the redundant copies of the data stored in a given memory address.

5.2 T-DMA Design

Figure 9 shows the design of our proposed T-DMA module. The T-DMA module consists of a *controller* and a *datapath*. The system's processor will program the T-DMA by writing to the controller. Programming the DMA includes telling the controller the D , R_s , direct/complementary redundancy, source memory address, destination memory address, number of memory locations, number of valid bits in each

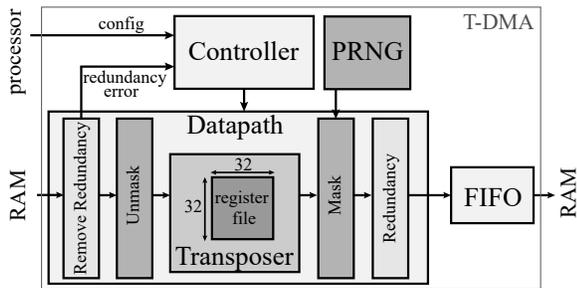


Fig. 9. Design of the T-DMA module.

location, and whether we need to {mask and duplicate} the data or {unmask, and check and remove the redundancy}.

The controller, then, sets the signals for the datapath to perform the transformations. At the core of the T-DMA's datapath design, is the *transposer* with a register file of thirty-two 32-bit registers (128 bytes) tuned for a 32-bit micro-architecture. Once the T-DMA starts the memory transfer, it will load the data residing in a programmable number of locations starting from a source address into the register file. While transferring the data from the system's RAM, the existing redundancy and masking will be removed for backward transposition. In case of a forward transposition, the removal of redundancy and masking are turned off and the masking shares for each bit of the data are generated based on the programmed number of shares ($D \in \{1, 2, 4\}$). We use the Cellular Automata-based PRNG³ to generate the randomness required for masking the data.

Once the masking shares are generated, the data is formatted according to the programmed redundancy scheme ($R_s \in \{1, 2, 4\}$ and direct/complementary copy configuration) and stored in the destination memory locations.

To perform the reverse transposition, the transposer first checks for the correctness of the redundant data. Once the correctness is ensured, it removes the redundancy and unmask the data. Finally, the dis-transposed data will be saved to the destination addresses.

The output of the datapath is stored in a First In, First Out (FIFO) memory. This memory stores the address and data of each output to be sent to the system's RAM. In our implementation, the FIFO is 256 bytes with 32 entries of 64 bits wide (to store the concatenated 32-bit address and 32-bit data). Once the transposition is done, T-DMA starts writing each entry of the FIFO to the system's RAM.

Despite only having a 32×32 register file in its transposer, T-DMA is capable of transposing up to $< 2^{16}$ distinct data each of length $< 2^{12}$ bits by being programmed only once. This feature is enabled by the *stride algorithm*.

Following the stride algorithm, the data is divided into *blocks*, each containing a maximum of 32 distinct data. T-DMA transposes one block at a time; It takes 32-bit parts (starting from the least significant bits) of each data in a block to load the transposer's register file. Subsequently, T-DMA offloads the transposed data to the memory. It then moves to the next 32 significant bits of the data in the block. Once all the bits of the data in the current block are

TABLE 8
Area break-down of the T-DMA modules synthesized for TSMC CMOS 180nm technology.

Module	Absolute area (μm^2)	Percent area
T-DMA	294,398.2676	100%
Contoller	21,969.5614	7.50%
FIFO	159,602.0136	54.20%
Datapath	87,406.2771	29.70%

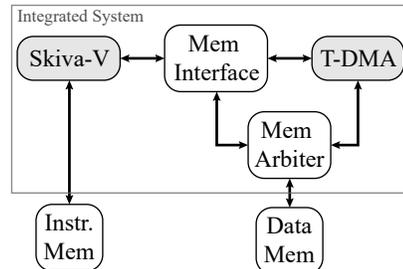


Fig. 10. Integration of Skiva-V and T-DMA.

transposed and stored in the destination addresses, it moves to the next block.

5.3 Implementation

To evaluate the size of T-DMA, we synthesize the circuit for TSMC's 180nm CMOS technology library. Table 8 shows the area break-down of the modules in our implementation of T-DMA. The T-DMA implementation shows a total area of less than 0.3mm^2 . The biggest contributor to this area is the FIFO which is responsible for 54% of the total area. The next biggest contributors are the datapath and the controller modules taking 30% and 7% of the total area respectively. The remaining area (8.6%) is dedicated to the PRNG and the inter-module connections.

6 SYSTEM INTEGRATION

To integrate Skiva-V processor and the T-DMA, we make the T-DMA implementation programmable from the processor by making it address-accessible. We further equip the T-DMA module with a status flag showing when a transaction is in place between the T-DMA and the data memory.

Every memory access from the memory stage of the pipeline goes through the memory interface. Figure 10 shows the connection between the modules in the integrated system. The memory interface detects whether the address is within the range of T-DMA or data memory.

In case of addressing the T-DMA, memory interface starts the transmission with the T-DMA which can include programming the T-DMA (write) or accessing its status bits (read). When the processor is trying to access the data memory, the interface module communicates with the memory arbiter.

Memory arbiter takes care of prioritizing memory accesses from the processor core and T-DMA. When T-DMA is programmed to access the data memory, memory arbiter prioritizes T-DMA's memory access over the memory access requests from the processor core. Therefore, the processor core will insert bubbles into its pipeline while waiting for the result of its memory access.

3. https://github.com/secworks/ca_prng

Implementation

We synthesize the integrated system (Figure 10) for the TSMC’s 180nm CMOS technology library and measure the total area of $518,018.9004\mu\text{m}^2$. Subtracting the synthesized area of the Skiva-V and T-DMA (reported in the previous sections) from the integrated system, consisting of the memory arbiter module and the added logic to the memory interface module, the integration adds around 0.07mm^2 to the overall area.

7 BENCHMARK

In the following, we run all the experiments on our integrated system. We demonstrate the advantage of hardware support for data transposition, the effect of richness of logic instructions performance of bitsliced programs, and the benefit of instruction-support for performance of masked implementations.

7.1 Cost of Transposition

In Section 4, we emphasized the high performance toll of data transposition, even with access to Skiva-V transpose instructions, in terms of the required number of instructions. To characterize the overhead of transposition more thoroughly, we evaluate the cost of transposition in our implemented system in terms of the required number of clock cycles.

We write a program in which K ($2 \leq K \leq 32$) adjacent bits in a 32-bit register need to be transposed to reside in 1 bit of K registers. We run the same program in three different settings: using only standard RV32I instructions, using Skiva-V’s transpose instructions, i.e., `tr2l` and `tr2h`, and using the T-DMA. We compare the first two cases in terms of number of required instructions and all three cases in terms of number of clock cycles.

Using the instructions in Skiva-V provides between $3\times$ to $10\times$ decrease in the number of instructions depending on the value of K . Furthermore, as Figure 11 shows, for each K , the number of clock cycles required to transpose K adjacent bits in a 32-bit register is reduced between $3\times$ to $6\times$ using the Skiva-V instructions. T-DMA and Skiva-V perform closely in this scenario with T-DMA having a better performance for $K \geq 19$.

These results confirm the benefits of the transpose instructions in Skiva-V. However, to demonstrate the benefits of having the T-DMA, we run another experiment in which $K \in \{2, 4, 8, 16, 32\}$ bits in K registers need to be transposed. Figure 12 shows that as K increases, the run-time of this transposition increases linearly ($14k + 19$, $R^2 = 1$) using the T-DMA but quadratically using the instructions in Skiva-V ($1.54k^2 - 4.3k + 64.9$, $R^2 = 1$) and RV32I ($55.1k^2 + 125k - 469$, $R^2 = 1$).

7.2 Effect of Logic Instruction Richness

In this experiment, we implement the bitsliced version of several cryptographic ciphers, namely LED 64 [26], Present [27], Midori [28], and Simon [29], for the RV32I ISA and compare their performance with that of the bitsliced code for Cortex-M4 ISA reported in our previous work [8]. Our goal is to show the effect of the richness of logic operations

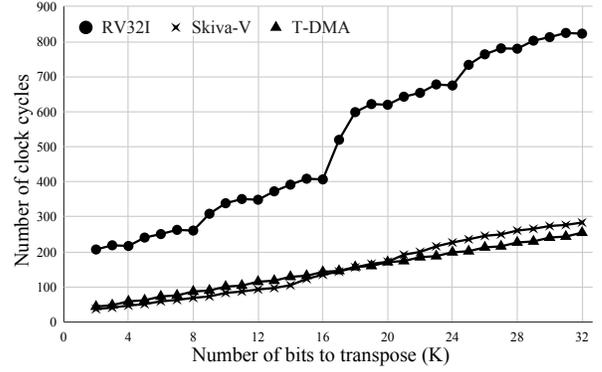


Fig. 11. Number of clock cycles to transpose K adjacent bits of one register.

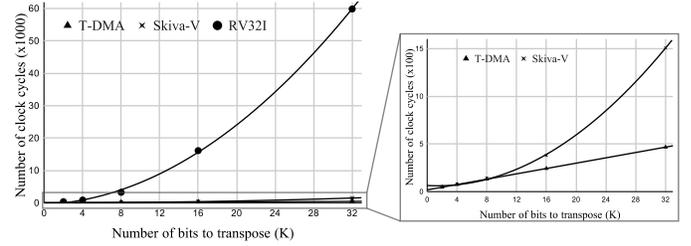


Fig. 12. Number of clock cycles required to transpose K adjacent bits in K registers.

in the performance of bitsliced programs. Table 9 shows that bitsliced programs for Cortex-M4 perform $1.6\times$ to $2\times$ faster than RV32I which is attributed to the richer set of bit-wise logic operations available on Cortex-M4 (i.e., AND, BIC, EOR, MOV, MVN, ORN, ORR).

7.3 Masked Implementations of LWC Ciphers

We take the finalists of the NIST’s Light-Weight Cryptography (LWC) competition that mention masking as their design options; ASCON [30] and GIFT-COFB [31]. We generate the masked implementation of their permutations (shown in Listing 3 and Listing 4) for $D \in \{1, 2, 4\}$ number of shares using the discussed code-generation method (Section 4). In the $D=1$, $D=2$, and $D=4$ settings, we support 32, 16, and 8 parallel executions respectively. We run the generated programs on Skiva-V system and calculate the number of cycles.

To supply the required randomness, we assume that the system has access to a pseudo-random number generator (PRNG) with a high throughput so that accessing a random number is equivalent to reading a register. Listing 1 and Listing 2 show the assembly code for masked 2-input AND instruction with $D=2$ and $D=4$ masked shares which follow the scheme described by Barthe et al. [18] and use the `subrot` instruction available in Skiva-V for rotation of shares sitting adjacently in the registers. Note that in this section, we do not perform any redundant computation, i.e., $R_s = 1$. In case of using complementary redundancy, the corresponding complementary logic instructions (described in Section 3.1) would replace the `and` and `xor` operations in Listing 1 and Listing 2.

TABLE 9
Reciprocal of performance (cycles/byte).

cipher	bitsliced (Cortex-M4) [8]	bitsliced (RV32I)
LED 64	546.676	889.309
Present	399.61	769.156
Midori	236.9	379.332
Simon	744.48	1549.934

Listing 1. First-order masked implementation of AND operation. Inputs are at `a1, a5`, random numbers are at `a0, a4`, the output is written in `a6`.

```

xor t0, a1, a0
subrot s0, a0, 2
xor t2, t0, s0
xor s0, s0, s0
and a7, a5, t2
subrot t5, t2, 2
and t1, t5, a5
xor t5, t5, t5
xor t3, a4, a7
xor t4, t3, t1
subrot t6, a4, 2
xor a6, t6, t4

```

Listing 2. Third-order masked implementation of AND operation. Inputs are at `a2, a4`, random numbers are at `a3, a5`, the output is written in `a1`.

```

xor s2, a3, a2
subrot s4, a2, 4
xor s3, s2, s4
xor s4, s4, s4
and a0, s3, a5
subrot t0, s3, 4
and a6, t0, a5
subrot s0, a5, 4
and a7, s0, s3
subrot t2, t0, 4
and t1, t2, a5
xor t0, t0, t0
xor s0, s0, s0
xor t2, t2, t2
xor t3, a4, a0
xor t4, t3, a6
xor t5, t4, a7
subrot s1, a4, 4
xor t6, t5, s1
xor a1, t6, t1

```

Table 10 reports the number of cycles per byte calculated as $\frac{c}{s \times p}$ where c is the number of clock cycles, s is the size of the state of the cipher in bytes ($\frac{320}{8}$ for ASCON and $\frac{128}{8}$ for GIFT-COFB), and p is the number of parallel runs.

Listing 3. ASCON permutation

```

void ascon_perm (int* state, int* round_const) {
    for (i = 0; i < 12; i++) {
        add_constant(state, round_const[i]);
        substitution(state);
        linear_diffusion(state);
    }
}

```

Listing 4. GIFT permutation

```

void gift_perm (int* state, int* key) {
    for (i = 0; i < 40; i++) {
        sub_cells(state);
        perm_bits(state);
        add_roundkey(state, key);
        key_update(key);
    }
}

```

As a comparison, we compare our results with a similar work, Tornado [32], which reports the same cycles/byte metric for the masked implementations (with the same fast assumption on the PRNG) of the same permutations of the LWC candidates but on Cortex-M4. Table 10 highlights the advantage of having hardware support for bitslicing. First, for an unmasked implementation ($D=1$), Tornado reports higher performance. Note that we assume the data is already in bitsliced (and masked if $D \neq 1$) format hence the transposition is not included in our measurements. Therefore, for unmasked implementations, the Skiva-V instructions are not used and the comparison is between the RISC-V RV32I and Cortex-M4 ISAs and the code generation process. Thus, the higher performance reported by Tornado can be attributed to the more advanced nature of Cortex-M4 ISA compared to the RISC-V ISA and to the code generation tool. Second, for a third-order masked implementation ($D=4$), we observe that Skiva-V can result in $1.5\times$ and $3.2\times$ speedup for ASCON and GIFT-COFB respectively. Since Tornado does not report first-order masking results, we were not able to compare with Skiva-V for the $D=2$ setting.

We further analyze this data in terms of added number of clock cycles per unit increase in the masking order. This criterion depends on the cipher algorithm and the implementation of the algorithm. Since our goal is to compare the implementations, and not the cipher algorithms, we compare this criterion for ASCON and GIFT separately. For this purpose, we make a linear regression of the cycles/byte vs. number of shares (D) as reported in Table 10.

The trend-line of the linear regression for ASCON's performance is $613D - 476$ for Skiva-V and $990D - 889$ for Tornado. This means increasing the order of masking by one, will cause 613 extra clock cycles for Skiva-V and 990 for Tornado ($1.6\times$ increase compared to Skiva-V).

The same experiment for GIFT's performance shows a trend-line of $1002D - 587$ for Skiva-V and $3574D - 3216$ for Tornado. For this cipher, the increase of clock cycles is more significant than ASCON which can be attributed to the multiplicative complexity of its algorithm. Furthermore, for an increase of one in the masking order, Tornado is affected by a $3.6\times$ higher increase in the required clock cycles than Skiva-V.

8 CONCLUSION

Bitsliced programming has been gaining attention in various applications, providing higher throughput to the program. Application-specific instruction set extensions are becoming more relevant since the introduction of RISC-V open-source ISA. We presented hardware support for bitsliced programming which can mitigate the known issues of bitslicing and improve the performance gain of this model of programming. Our hardware support consists of

TABLE 10
 Reciprocal of performance (cycles/byte).
 The PRNG is assumed to have a high enough throughput to not cause any reading delay.
 Tornado results are for ARM Cortex-M4; Skiva-V results are for RISC-V RV32I with extensions.

Cipher	D=1 (no masking)			D=2 (first-order masking)			D=4 (third-order masking)		
	Tornado	Skiva-V	Speed-up	Tornado	Skiva-V	Speed-up	Tornado	Skiva-V	Speed-up
ASCON	101	159.677	0.633	-	717.495	-	3070	1988.903	1.544
GIFT	358	441.941	0.810	-	1378.141	-	11080	3435.656	3.225

custom instructions added to the RISC-V processor and a transposition-equipped DMA module that can be added to an SoC to handle data transformations necessary for bitsliced programs on the fly. We further evaluated the area footprint of our proposed implemented modules and their performance boost through simulated experiments.

REFERENCES

- [1] E. Biham, "A fast new DES implementation in software," in *International Workshop on Fast Software Encryption*. Springer, 1997, pp. 260–272.
- [2] C. Rebeiro, D. Selvakumar, and A. Devi, "Bitslice implementation of aes," in *International Conference on Cryptology and Network Security*. Springer, 2006, pp. 203–212.
- [3] W. de Groot, K. Papagiannopoulos, A. de La Piedra, E. Schneider, and L. Batina, "Bitsliced masking and arm: Friends or foes?" in *International Workshop on Lightweight Cryptography for Security and Privacy*. Springer, 2016, pp. 91–109.
- [4] J. Daemen, M. Peeters, and G. Van Assche, "Bitslice ciphers and power analysis attacks," in *International Workshop on Fast Software Encryption*. Springer, 2000, pp. 134–149.
- [5] S. Matsuda and S. Moriai, "Lightweight cryptography for the cloud: exploit the power of bitslice implementation," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 408–425.
- [6] S. Xu and D. Gregg, "Bitslice vectors: A software approach to customizable data precision on processors with simd extensions," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 442–451.
- [7] R. Singh, T. Conroy, and P. Schaumont, "Variable precision multiplication for software-based neural networks," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.
- [8] P. Kiaei and P. Schaumont, "Synthesis of parallel synchronous software," *IEEE Embedded Systems Letters*, pp. 1–1, 2020.
- [9] A. Zeh, A. Glew, B. Spinney, B. Marshall, D. Page, D. Atkins, K. Dockser, M.-J. O. Saarinen, N. Menhorn, and R. Newell, "Risc-v cryptographic extension proposals."
- [10] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallfloat simd extensions to the risc-v isa," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 654–657.
- [11] P. Kiaei and P. Schaumont, "Domain-oriented masked instruction set architecture for risc-v." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 465, 2020.
- [12] P. Kiaei, D. Mercadier, P.-E. Dagand, K. Heydemann, and P. Schaumont, "Custom instruction support for modular defense against side-channel and fault attacks," in *Constructive Side-Channel Analysis and Secure Design*, G. M. Bertoni and F. Regazzoni, Eds. Cham: Springer International Publishing, 2021, pp. 221–253.
- [13] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media, 2008, vol. 31.
- [14] D. Goudarzi, A. Journault, M. Rivain, and F. Standaert, "Secure multiplication for bitslice higher-order masking: Optimisation and comparison," in *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, ser. Lecture Notes in Computer Science, J. Fan and B. Gierlichs, Eds., vol. 10815. Springer, 2018, pp. 3–22. [Online]. Available: https://doi.org/10.1007/978-3-319-89641-0_1
- [15] S. Dhooghe and S. Nikova, "My gadget just cares for me - how nina can prove security against combined attacks," in *Topics in Cryptology – CT-RSA 2020*, S. Jarecki, Ed. Cham: Springer International Publishing, 2020, pp. 35–55.
- [16] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual international cryptology conference*. Springer, 1999, pp. 388–397.
- [17] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2004, pp. 16–29.
- [18] G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F.-X. Standaert, and P.-Y. Strub, "Parallel implementations of masking schemes and the bounded moment leakage model," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 535–566.
- [19] A. Battistello, J.-S. Coron, E. Prouff, and R. Zeitoun, "Horizontal side-channel attacks and countermeasures on the ISW masking scheme," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 23–39.
- [20] S. Belaïd, F. Benhamouda, A. Passelègue, E. Prouff, A. Thillard, and D. Vergnaud, "Randomness complexity of private circuits for multiplication," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 616–648.
- [21] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1997, pp. 37–51.
- [22] K. Asanovic and A. Waterman, "The risc-v instruction set manual," in *Privileged Architecture, Document Version 20190608-PrivMSU-Ratified*. RISC-V Foundation, 2019, vol. 2.
- [23] S. Bandara, A. Ehret, D. Kava, and M. A. Kinsy, "Brisic-v: An open-source architecture design space exploration toolbox," *arXiv preprint arXiv:1908.09992*, 2019.
- [24] P. Kiaei, A. S. Krishnan, and P. Schaumont, "Parallel synchronous code generation for second round light weight candidates," *Proceedings of the NIST Lightweight Cryptography Workshop*, 2020.
- [25] "Yosys Open Synthesis Suite," <http://www.clifford.at/yosys/>, accessed: 2021-2-18.
- [26] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The led block cipher," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2011, pp. 326–341.
- [27] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2007, pp. 450–466.
- [28] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, "Midori: A block cipher for low energy," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2015, pp. 411–436.
- [29] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The simon and speck lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.
- [30] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schl affer, "Ascon v1.2," *Submission to the CAESAR Competition*, 2016.
- [31] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "Gift: a small present," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 321–345.
- [32] S. Belaïd, P.-E. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff, "Tornado: Automatic generation of probing-secure masked bitsliced implementations," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 311–341.

Pantea Kiaei (Student Member, IEEE) Pantea Kiaei is a Ph.D. student in Electrical and Computer Engineering at Worcester Polytechnic Institute. She received her MS degree in Computer Engineering from Virginia Tech in 2019 and prior to that received her BS degree in Electrical Engineering from Sharif University of Technology, Iran, in 2017. She has reviewed papers for ACM TECS and ACM JETC journals. Her research interests include secure hardware design, computer architecture, and trustworthy secure systems.

Tom Conroy was an MS student in Electrical and Computer Engineering at Virginia Tech. He received his MS degree in Computer Engineering in 2021 and his BS in Computer Engineering in 2019, both from Virginia Tech. He has since joined The Johns Hopkins University Applied Physics Laboratory as an Electronic Systems Engineer. His research interests include secure hardware design, efficient cryptographic implementation on embedded systems, and embedded system security.

Patrick Schaumont (Senior Member, IEEE) is a Professor in Computer Engineering at WPI. He received the Ph.D. degree in Electrical Engineering from UCLA in 2004 and the MS degree in Computer Science from Ghent University in 1990. He was a staff researcher at IMEC, Belgium from 1992 to 2000. He was a faculty member with Virginia Tech from 2005 to 2019. He joined WPI in 2020. He was a visiting researcher at the National Institute of Information and Telecommunications Technology (NICT), Japan in 2014. He was a visiting researcher at Laboratoire d'Informatique de Paris 6 in Paris, France in 2018. He is a Radboud Excellence Initiative Visiting Faculty with Radboud University, Netherlands from 2020. His research interests are in design and design methods of secure, efficient and real-time embedded computing systems.