A Low-Area Yet Performant FPGA Implementation of Shabal

Jérémie Detrey[1], Pierrick Gaudry[1], and Karim Khalfallah[2]

[1] CARAMEL project-team, LORIA, INRIA / CNRS / Nancy Université,
Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
[2] Laboratoire de cryptographie et composants, SGDSN / ANSSI,
51 boulevard de la Tour-Maubourg, 75700 Paris 07 SP, France

Abstract. In this paper, we present an efficient FPGA implementation of the SHA-3 hash function candidate Shabal [7]. Targeted at the recent Xilinx Virtex-5 FPGA family, our design achieves a relatively high throughput of 2 Gbit/s at a cost of only 153 slices, yielding a throughput-*vs.*-area ratio of 13.4 Mbit/s per slice. Our work can also be ported to Xilinx Spartan-3 FPGAs, on which it supports a throughput of 800 Mbit/s for only 499 slices, or equivalently 1.6 Mbit/s per slice.

According to the SHA-3 Zoo website [1], this work is among the smallest reported FPGA implementations of SHA-3 candidates, and ranks first in terms of throughput per area.

Keywords: SHA-3, Shabal, low area, FPGA implementation.

1 Introduction

Following the completion of the first round of the NIST SHA-3 hash algorithm competition in September 2009, fourteen candidates [16] have been selected to participate in the second round [18]. As such, developing and benchmarking software and hardware implementations of these remaining hash functions is key to assess their practicality on various platforms and environments.

Building toward that objective, this paper presents an area-efficient implementation of the SHA-3 candidate Shabal, submitted by Bresson *et al.* [7], on Xilinx Virtex-5 and Spartan-3 FPGAs [20,23]. Even though the core contribution of this work is to demonstrate that Shabal can be brought to area-constrained devices such as smart cards or RFID tags, it also appears from the benchmark results that our design also performs extremely well in terms of throughput per area, ranking first among the other published implementations of SHA-3 candidates.

Roadmap. After a brief description of the Shabal hash function, we explain in Section 2 how this algorithm can be adapted to make full use of the shift register primitives embedded in some FPGA families. A detailed description of our design is given in Section 3, along with implementation results and comparisons in Section 4.

A. Biryukov, G. Gong, and D.R. Stinson (Eds.): SAC 2010, LNCS 6544, pp. 99–113, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Notations. In the following, unless specified otherwise, all words are 32 bits long and are to be interpreted as unsigned integers. Given two such words X and Y along with an integer k, we write the rotation of X by k bits to the left as $X \lll k$; the bitwise exclusive disjunction (XOR) of X and Y as $X \oplus Y$; the bitwise conjunction (AND) of X and Y as $X \wedge Y$; the bitwise negation (NOT) of X as $\overline{X}$; the sum and difference of X and Y modulo 2^{32} as $X \boxplus Y$ and $X \boxminus Y$, respectively; and the product of X by k modulo 2^{32} as $X \boxtimes k$. We also denote by $X \leftarrow Y$ the assignment of the value of Y to the variable X.

Furthermore, given an n-stage-long shift register R, we denote its elements by $R[0]$, $R[1]$, and so on up to $R[n-1]$. Inserting a word X into $R[n-1]$ while shifting the other elements by one position to the left (*i.e.*, $R[i] \leftarrow R[i+1]$ for $0 \leqslant i < n-1$) is denoted by $R \twoheadleftarrow X$, whereas $X \twoheadrightarrow R$ indicates the insertion of X into $R[0]$ while the rest of the register is shifted by one step to the right (*i.e.*, $R[i] \leftarrow R[i-1]$ for $n-1 \geqslant i > 0$).

2 Shabal and Shift Registers

2.1 The Shabal Hash Algorithm

For a complete description of the Shabal hash function, please refer to [7, Ch. 2].

The internal state of Shabal consists of three 32-bit-wide shift registers, A, B, and C, of length 12, 16, and 16, respectively, along with a 64-bit counter W. Shabal splits a message in 512-bit blocks, which are stored into another 16-stage-long and 32-bit-wide shift register called M.

Processing a block M in Shabal involves the following sequence of operations:

1. XOR the counter into the first two words of A: $A[i] \leftarrow A[i] \oplus W[i]$, $i = 0, 1$.
2. Add the message to B: $B[i] \leftarrow B[i] \boxplus M[i]$, for $0 \leqslant i < 16$.
3. Rotate each word of B by 17 bits: $B[i] \leftarrow B[i] \lll 17$, for $0 \leqslant i < 16$.
4. Apply the keyed permutation, as depicted in Fig. 1, for 48 iterations:
   - compute the two 32-bit words P and Q as

$$
\begin{aligned}
V &\leftarrow (A[11] \lll 15) \boxtimes 5, & U &\leftarrow (V \oplus A[0] \oplus C[8]) \boxtimes 3, \\
P &\leftarrow U \oplus M[0] \oplus (\overline{B[6]} \wedge B[9]) \oplus B[13], & Q &\leftarrow \overline{P} \oplus (B[0] \lll 1);
\end{aligned}
$$

   - shift the four registers: $A \twoheadleftarrow P$, $B \twoheadleftarrow Q$, $C[15] \twoheadrightarrow C$, and $M \twoheadleftarrow M[0]$.
5. Add three words of C to each word of A: for $0 \leqslant i < 12$,

$$
A[i] \leftarrow A[i] \boxplus C[(i+3) \bmod 16] \boxplus C[(i+15) \bmod 16] \boxplus C[(i+11) \bmod 16].
$$

6. Subtract the message from C: $C[i] \leftarrow C[i] \boxminus M[i]$, for $0 \leqslant i < 16$.
7. Swap the contents of shift registers B and C: $(B, C) \leftarrow (C, B)$.
8. Increment the counter: $W \leftarrow (W + 1) \bmod 2^{64}$.

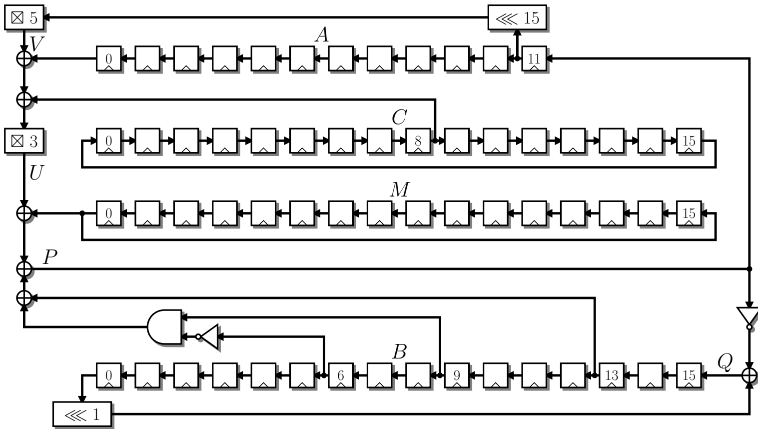

Fig. 1. Main structure of the keyed permutation [7, Fig. 2.4]

2.2 The Xilinx SRL16 Shift Register Primitive

As described in Section 2.1, the internal state of Shabal involves 46 32-bit words of storage and the message block M requires another 16 words. A naive implementation, using one FPGA flip-flop resource for each of these 1 984 bits of storage, would then result in a relatively large circuit.

However, it is important to note here that only a small fraction of the internal state is actually used at any step of the execution of Shabal. For instance, only $A[0]$, $A[11]$, $B[0]$, $B[6]$, $B[9]$, $B[13]$, $C[8]$, and $M[0]$ are required to compute P and Q when applying the keyed permutation (see Fig. 1), the other words simply being stepped through their respective shift registers. We can therefore exploit this fact and take advantage of the dedicated shift register resources offered by some FPGA families in order to minimize the overall area of the circuit.

This is the case in the recent Xilinx FPGAs—such as the high-end Virtex-5 and -6 families, or the low-cost Spartan-3's and -6's—which all support the SRL16 primitive [20,21,23,24]. As depicted in Fig. 2, this primitive implements a 16-stage-long and 1-bit-wide addressable shift register in a single 4-to-1-bit look-up table (LUT), as it is in fact nothing but a 16-bit memory. A dedicated input `DIN` is used to shift the data in, whereas the regular 4-bit input `A` addresses which bit is driven to the LUT output `D` [20, Ch. 7]. It is therefore possible to implement variable-size shift registers with this primitive. For instance, fixing the address `A` to `0000`, `0011`, or `1111` results in 1-, 4-, and 16-stage-long shift registers, respectively. Note that this idea has already been successfully exploited for linear- and non-linear-feedback-shift-register-based stream ciphers such as Grain or Trivium [8].

Furthermore, since the most recent Virtex-5, -6, and Spartan-6 FPGA families are based on 64-bit LUTs—supporting either 6-to-1-bit or 5-to-2-bit modes of operation—it is possible to pack two SRL16 instances in a single LUT, thus implementing a 16-stage-long and 2-bit-wide addressable shift register (see for instance [23, Fig. 5-17]).

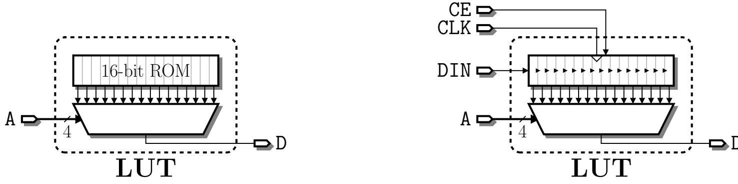

Fig. 2. Xilinx LUT as a look-up table (left) and as an SRL16 shift register (right)

2.3 Adapting Shabal to Use Shift Registers

Since we want to benefit as much as possible from the low area requirements of the SRL16 primitive, we need to implement the full Shabal algorithm using only shift registers. In other words, all the parallel register-wide operations—such as the $B \leftarrow B \boxplus M$ affectation in step 2—should be serialized in a word-by-word fashion.

From the algorithm given in Section 2.1, all steps but steps 4 and 8 should therefore be serialized in such a way. This is a rather simple task for steps 2, 3, 6, and 7, but special care needs to be paid to the other two.

Accumulating *C* into *A*. The first non-trivial serialization to address is that of step 5, which requires the accumulation of three words of the shift register C into each word of A. From the original specification of Shabal [7], we can remark that the indices of the three words of C are actually separated by exactly 12 positions each:

$$\begin{aligned} C[(i+\ \ 3) \bmod 16] &= C[(i+3+0\times 12) \bmod 16], \\ C[(i+15) \bmod 16] &= C[(i+3+1\times 12) \bmod 16], \text{ and} \\ C[(i+11) \bmod 16] &= C[(i+3+2\times 12) \bmod 16]. \end{aligned}$$

We therefore choose to accumulate those words of C in a disctinct shift register of length 12, denoted by D, by executing the iteration $D \twoheadleftarrow D[0] \boxplus C[3]$ and $C \twoheadleftarrow C[0]$ for 36 consecutive cycles (assuming that every word of D was initialized to 0 beforehand).

Once each word $D[i]$ contains $C[(i+3) \bmod 16] \boxplus C[(i+15) \bmod 16] \boxplus C[(i+11) \bmod 16]$, the contents of D are accumulated into A in 12 cycles, both 12-stage-long shift registers A and D stepping simultaneously.

Shifting *C* both left and right. Note that, in the previous situation, the shift register C is rotated by one position to the left at each step (*i.e.*, $C \twoheadleftarrow C[0]$), whereas the keyed permutation (step 4 of the algorithm) requires C to rotate to the right (*i.e.*, $C[15] \twoheadrightarrow C$). In order to solve this issue, we duplicate the shift register C into two separate shift registers C and C', both stepping to the left—so as to match the direction of the other shift registers A, B, D, and M.

- The first shift register, C, is then responsible for delivering the words $C[3]$ in the proper order to accumulate them into D.

- The second one, C', is loaded simultaneously with C—and therefore contains the same data—and is addressed by a 4-bit counter k fed to the `A` port of the SRL16s. So as to constantly point to the word $C[8]$ required by the keyed permutation, k has to be decremented by 2 at each cycle to compensate for C' stepping to the left.

The 64-bit counter W. Looking at step 1 of the Shabal algorithm, it seems natural to consider the 64-bit counter W as a 2-word shift register, synchronized with A, so that we can execute step 1 in two consecutive cycles.

Additionally, this has the interesting side-effect of splitting in half the 64-bit-long carry propagation required for incrementing W in step 8 and which might have been on the critical path. The incrementation of W is then performed word by word, storing the carry output in a separate 1-bit register W_{cy} before reinjecting it as carry-in for the next word of W.

Finally, since implementing 2- or 16-stage-long shift registers requires exactly the same amount of SRL16 primitives—it consists only in changing the `A` input from `0001` to `1111`—we choose to use also a 16-word shift register for W, words $W[2]$ to $W[15]$ set to 0, so as to match the 16-cycle period of registers B, C, C', and M, and thus simplifying the control.

2.4 Scheduling of a Shabal Message Round

From the Shabal algorithm in Section 2.1, it appears that several steps can be merged or performed in parallel. We first present the justification and validity of such merges before giving the resulting scheduling of the algorithm, as implemented in our circuit.

Merging and parallelizing steps in the algorithm. First of all, assuming that the shift register D contains the sums $C[(i+3) \bmod 16] \boxplus C[(i+15) \bmod 16] \boxplus C[(i+11) \bmod 16]$ from the step 5 of the previous round, we can postpone the 12-cycle accumulation of D into A of that previous round to the beginning of the current round, effectively combining it with the XORing of W into A (step 1). During that time, since only the shifted-out value $D[0]$ is required, we can also reset D to 0 by shifting-in 12 successive 0 words.

Additionally, we can merge the 16-cycle steps 2 and 3 by directly rotating $B[i] \boxplus M[i]$ by 17 bits to the left. This can further be combined with the word-by-word loading of the current message block into the shift register M, along with the swapping of B and C (step 7) from the previous round—thus postponed to the current stage—by shifting $B[0]$ into C and C' while simultaneously shifting the newly received message word M_{in} into M and $(C[0] \boxplus M_{\text{in}}) \lll 17$ into B. Finally, as all the registers involved are independent of A and D, this can be performed in parallel with their previously discussed initialization.

We can also accumulate the words of C into D—which takes 36 iterations—while executing the 48 iterations of the keyed permutation in parallel. Furthermore, during the last 16 cycles of those 48, notice that only the shifted-out value $M[0]$ is necessary to the permutation. We can then use the shift register M to

temporarily store the difference $C \boxminus M$ before shifting it again to B in the next round. To that intent, $C[0] \boxminus M[0]$ is then shifted into M during those last 16 cycles. Finally, the incrementation of W can also be performed in parallel during those 16 cycles.

All in all, we end up with a 64-cycle round comprising two main steps:

– a 16-cycle step, during which the shift registers A, B, C, C', D, and M are initialized before performing the keyed permutation; followed by
– a 48-cycle step, actually computing the permutation, while preparing the shift registers D, M, and W for the next round.

Detailed scheduling. As discussed in the previous paragraphs, postponing steps of the Shabal algorithm from one round to the next changed the preconditions on the internal state of Shabal at the beginning of the round, with respect to what was presented as a round in Section 2.1. Therefore, for the scheduling detailed here, we assume (a) that each word $D[i]$ contains $C[(i+3) \bmod 16] \boxplus C[(i+15) \bmod 16] \boxplus C[(i+11) \bmod 16]$, (b) that these sums have not yet been accumulated into A, (c) that the shift register M contains $C \boxminus M$, and (d) that the shift registers B and C have not yet been swapped.

The scheduling of the 64-cycle round then breaks down as follows (where the current cycle is denoted by c, and where all the shifts and assignments are performed synchronously):

$c =$	$0,\ldots,11$	$12,\ldots,15$	$16,\ldots,47$	$48,\ldots,51$	$52,\ldots,63$
$A \twoheadleftarrow$	A_{in}	——	P		
$B \twoheadleftarrow$	B_{in}		Q		
$C \twoheadleftarrow$	$B[0]$		$C[0]$		
$C' \twoheadleftarrow$	$B[0]$		——		
$D \twoheadleftarrow$	0		$D[0] \boxplus C[3]$		——
$M \twoheadleftarrow$	M_{in}		$M[0]$	$C[0] \boxminus M[0]$	
$W \twoheadleftarrow$	$W[0] \boxplus W_{\text{cy}}$				
$W_{\text{cy}} \leftarrow$	Output carry of $W[0] + W_{\text{cy}}$				
$k \leftarrow$	$(k+14) \bmod 16$				

In this scheduling, at each cycle, the two words P and Q are computed as

$$V \leftarrow (A[11] \lll 15) \boxtimes 5, \qquad U \leftarrow (V \oplus A[0] \oplus C'[k]) \boxtimes 3,$$
$$P \leftarrow U \oplus M[0] \oplus (\overline{B[6]} \wedge B[9]) \oplus B[13], \text{ and} \qquad Q \leftarrow \overline{P} \oplus (B[0] \lll 1).$$

Furthermore, A_{in} designates $(A[0] \boxplus D[0]) \oplus W[0]$, B_{in} is $(M[0] \boxplus M_{\text{in}}) \lll 17$, and the input carry W_{cy} is forced to 1 at cycle $c = 48$.

3 FPGA Implementation

3.1 Overall Architecture

The main architecture of our FPGA implementation of Shabal is given in Fig. 3, where the control logic is omitted for clarity's sake. The small rounded boxes

indicate control bits such as the *clock enable* signals for the various shift registers, whereas the light-gray rounded boxes identify how this circuit is mapped onto basic Virtex-5 primitives. Further details about this mapping are given in Section 3.3.

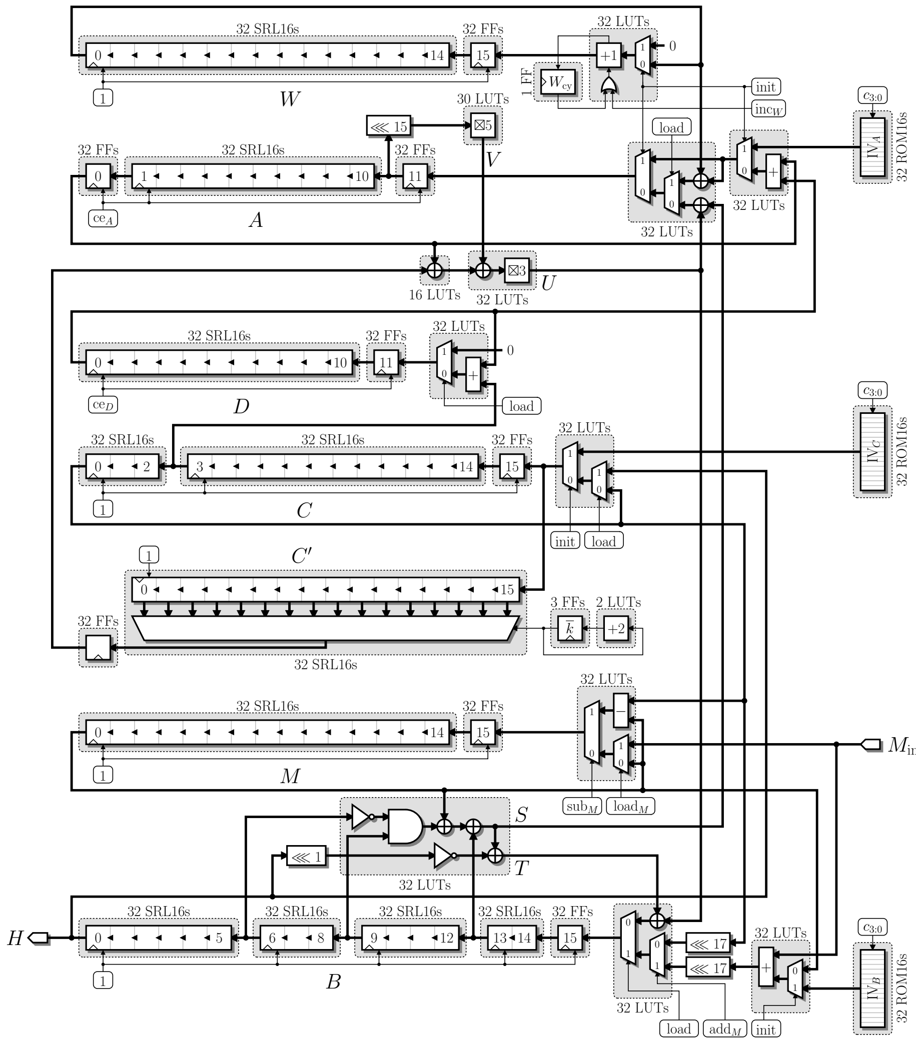

Fig. 3. Main architecture of our Shabal design mapped onto Virtex-5 primitives

Up to now, we have not discussed the initialization of Shabal. According to [7], this can be achieved by setting the registers A, B, and C to specific initialization vectors, denoted here by IV_A, IV_B, and IV_C, respectively. In our architecture, these initialization vectors are stored into small 12- and 16-word

ROMs, addressed by the four least significant bits of c and implemented by means of 16-by-1-bit ROM16 primitives. During the first 16 cycles of the first round of Shabal, the initialization words are then shifted into the corresponding registers. In the meantime, the register W is initialized to 0, except for $W[0]$ which is set to 1.

When addressing an SRL16 primitive such as C', using the address `0000` gives the contents of the first stage of the register (*i.e.*, $C'[15]$) whereas `1111` addresses the last stage (*i.e.*, $C'[0]$). Consequently, as we want to retrieve $C'[k]$ at each clock cycle, we need to address the corresponding shift register with $\overline{k} = 15 - k$ instead of k. We therefore directly store and update $\overline{k}$ by means of a 4-bit up counter. Additionally, since $\overline{k}$ is always incremented by 2, its least significant bit is constant and need not be stored.

So as to shorten the critical path of the circuit, we also use the associativity of the XOR operation to extract some parallelism out of the main feedback loop which computes P and Q. Indeed, while computing V then U as before, we also compute in parallel the two words S and T as

$$S \leftarrow M[0] \oplus (\overline{B[6]} \wedge B[9]) \oplus B[13] \qquad \text{and} \qquad T \leftarrow S \oplus \overline{(B[0] \lll 1)}.$$

We then immediately have P as $U \oplus S$ and Q as $U \oplus T$.

3.2 Control Logic

Although not depicted in Fig. 3, we claim that the logic required to generate the control bits of our architecture entails but a small overhead. This is achieved by restricting the number of control bits to a bare minimum:

– First of all, since the shift registers B, C, C', M, and W are always stepping, according to our scheduling, we fix their *clock enable* signal to `1`. Only A and D have distinct signals—ce_A and ce_D, respectively—as they sometimes have to be stalled for a few cycles due to their shorter length.
– The *load* signal, which identifies the first 16 cycles of each round, is also shared by most shift registers. Only M is controlled by load_M, as no message block M_{in} should be loaded during the three final rounds of Shabal [7].
– An *init* signal also controls whether we are in the first round and therefore should load the initialization values for registers A, B, C, C', and W.
– Finally, the signals sub_M, add_M, and inc_W indicate when to subtract or add the message to C and B, or when to increment W, respectively. Note that these three signals are disabled during the three final rounds of Shabal.

A waveform of those signals during a message round of Shabal is given Fig. 4. Not represented on this waveform, the init signal is high only during the first 16 cycles of the first round, and the load_M signal is identical to the load signal during message rounds. As for the three final rounds, the control bits are the same, except for load_M, sub_M, add_M, and inc_W which are driven low.

The reader should note at this point that we opted for keeping the I/O interface of our circuit as simple as possible, since the choice of which interface

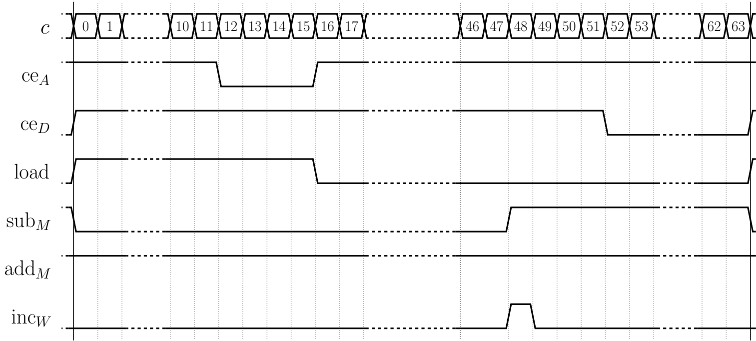

Fig. 4. Waveform of the control signals during a message round

to actually use heavily depends on the usage of the hash function and on the environment in which it will run. Therefore, no message padding mechanism was implemented here, and our circuit assumes that the message blocks are always available and fed to the hash function during the 16 first cycles of each message round. However, disabling the clock signal for the shift registers and the control logic via a circuit-wide clock enable signal would allow the hash function to be interrupted in order to wait for message words, for but a moderate overhead.

3.3 Technology Mapping

As previously mentioned, the light-gray rounded boxes in Fig. 3 refer to the mapping of our Shabal circuit onto basic Virtex-5 primitives.[1] In the figure, next to the boxes, we also indicate the count and type of required primitives, where LUT refers to a Virtex-5 look-up table—in either 6-to-1-bit or 5-to-2-bit mode of operation—FF to a 1-bit flip-flop, SRL16 to an addressable shift register as described in Section 2.2, and ROM16 to a 16-by-1-bit ROM. Note that the fixed-length rotations ($\lll$) involve only wires and do not require any logic resource.

In order to reduce the critical path of the circuit, we choose to implement the first stage of each shift register—except for C'—using flip-flops instead of merging them into the SRL16 primitives. Indeed, this saves the routing delay between the LUT-based multiplexers feeding the registers and the SRL16 primitives, and entails no resource overhead as each Virtex-5 LUT is natively paired with a matching flip-flop on the FPGA.

Furthermore, the clock-to-ouput delay of the SRL16 primitives being quite high [22, Tab. 67], we also use flip-flops to implement the last stage of shift register A which lies on the critical path.

Finally, it is worth noting that, on Virtex-5 FPGAs, two SRL16 or two ROM16 primitives can fit on a single LUT. Since a large part of our implementation is

[1] A similar mapping was made for the Spartan-3 technology, but is not included in this paper for the sake of concision.

Table 1. Resource count of the main architecture as mapped on Virtex-5 (excluding the control logic)

Component	High-level mapping				Actual mapping	
	LUTs	FFs	SRL16s	ROM16s	LUTs	FFs
Register A	64	64	32	32	96	64
Register B	64	32	128	32	144	32
Registers C and C'	34	67	96	32	98	67
Register D	32	32	32	0	48	32
Register M	32	32	32	0	48	32
Register W	32	33	32	0	48	33
Feedback loop	110	0	0	0	110	0
Total	368	260	352	96	592	260

based on these primitives, this reduces ever further the overall area of the design. To illustrate this, we present in Table 1 a complete breakdown of the resource requirements of our circuit, both in terms of high-level primitives—*i.e.*, LUTs, FFs, SRL16s, and ROM16s—and in terms of actually mapped primitives—*i.e.*, LUTs and FFs only.

4 Benchmarks and Comparisons

4.1 Place-and-Route Results

We have implemented the fully autonomous Shabal circuit presented in this paper in VHDL.[2] We have placed-and-routed it using the Xilinx ISE 10.1 toolchain, targeting a Xilinx Virtex-5 LX 30 FPGA with average speed grade (xc5vlx30-2ff324), along with a low-cost Xilinx Spartan-3 200 with highest speed grade (xc3s200-5ft256).

On the Virtex-5, the whole circuit occupies only 153 slices, that is, more precisely, 605 LUTs and 274 flip-flops. Interestingly enough, these figures are very close to the technology mapping estimations from Table 1: the control logic overhead, totaling to 13 LUTs and 14 flip-flops, is quite small, as expected. This design supports a clock period of 3.9 ns—*i.e.*, a frequency of 256 MHz—and since it processes a 512-bit message block in 64 clock cycles, it delivers a total throughput of 2.05 Gbit/s and thus a throughput-*vs.*-area ratio of 13.41 Mbit/s per slice.

On the Spartan-3 target, our Shabal architecture uses 499 slices and can be clocked at 100 MHz. This yields a throughput of 800 Mbit/s, which corresponds to 1.6 Mbit/s per slice.

Note that these results are given for the Shabal-512 flavor, even though changing the digest length does not affect the circuit performance in any way.

[2] This VHDL code is available at `http://hwshabal.gforge.inria.fr/` under the terms of the GNU Lesser General Public License.

4.2 Against Other Shabal Implementations

A comparison between our circuit and previously published implementations is given in Table 2. Since all state-of-the-art papers present benchmarks on Virtex-5 or Spartan-3, we believe this comparison to be quite fair. It is to be noted that, if our Shabal circuit does not deliver the highest throughput, it is by far the smallest implementation in the literature, and also the most efficient in terms of throughput per area.

Table 2. FPGA implementations of Shabal on Virtex-5 and Spartan-3

FPGA	Implementation	Area [slices]	Freq. [MHz]	Cycles / round	TP [Mbps]	TP / area [kbps/slice]
Virtex-5	Baldwin *et al.* [3]*	2 307	222	85	1 330	577
		2 768	139	49	1 450	524
	Kobayashi *et al.* [13]	1 251	214	63	1 739	1 390
	Feron and Francq [9]	1 171	126	25	**2 588**	2 210
		596†	109	49	1 142	1 916
	This work	**153**	256	64	2 051	**13 407**
Spartan-3	Baldwin *et al.* [3]*	1 933	90	85	540	279
		2 223	71	49	740	333
	This work	**499**	100	64	**800**	**1 603**

*Only the core functionality was implemented. †This design requires 40 additional DSP blocks.

Also note that Namin and Hasan published another FPGA implementation of Shabal [17]. However, not only do they present benchmarks on Altera Stratix-III FPGAs—against which it becomes difficult to compare Virtex-5 or Spartan-3 results in a fair way—but it also seems from their paper that they only implement part of the Shabal compression function. We therefore deliberately choose not to compare our work to theirs.

4.3 Against Implementations of the Other SHA-3 Candidates

Since Shabal is but one hash function among the fourteen remaining SHA-3 candidates, we also provide the reader with a compilation of the Virtex-5 and Spartan-3 implementation results for the other hash functions as gathered on the SHA-3 Zoo website [1] in Tables 3 and 4, respectively.

Comparing our Shabal circuit with these other designs, it is clear that, in terms of raw speed, we cannot compete against the high-throughput implementations of ECHO [15] or Grøstl [10] which all exceed the 10 Gbit/s mark on Virtex-5. However, it appears that our implementation ranks among the smallest SHA-3 designs, third only to the low-area implementations of BLAKE and ECHO by Beuchat *et al.* [6,5]. Therefore, if a high throughput is the objective, one can replicate our circuit several times in order to increase the overall throughput by the same factor: for instance, eight instances of our implementation running in parallel would yield a total of 16.4 Gbit/s for a mere 1 224 slices on Virtex-5, thus more than four times smaller than the 5 419 slices of the 15.4-Gbit/s

Table 3. FPGA implementations of SHA-3 candidates on Virtex-5

Hash function	Digest length	Implementation	Area [slices]	Freq. [MHz]	TP [Mbps]	TP / area [kbps/slice]
BLAKE	256	Submission doc. [2]*	1 694	67	3 103	1 832
			1 217	100	2 438	2 003
			390	91	575	1 474
		Kobayashi *et al.* [13]	1 660	115	2 676	1 612
		Beuchat *et al.* [6]	**56**	372	225	4 018
	512	Submission doc. [2]*	4 329	35	2 389	552
			2 389	50	1 766	739
			939	59	533	568
		Beuchat *et al.* [6]	108	358	314	2 907
CubeHash	all	Baldwin *et al.* [3]*	1 178	167	160	136
			1 440	55	110	76
		Kobayashi *et al.* [13]	590	185	2 960	5 017
ECHO	224/256	Lu *et al.* [15]	9 333	87	14 860	1 592
		Kobayashi *et al.* [13]	3 556	104	1 614	454
		Beuchat *et al.* [5]	127	352	72	567
	384/512	Lu *et al.* [15]	9 097	84	7 810	859
Grøstl	224/256	Submission doc. [10]	1 722	201	10 276	5 967
		Baldwin *et al.* [3]*	3 184†	250	6 410	2 013
			4 516†	143	7 310	1 619
			5 878	128	3 280	558
			8 196	102	5 210	636
		Kobayashi *et al.* [13]	4 057	101	5 171	1 275
	384/512	Submission doc. [10]	5 419	211	**15 395**	2 841
		Baldwin *et al.* [3]*	6 368†	144	5 260	826
			10 848	111	4 060	374
			19 161	83	6 090	318
Hamsi	256	Kobayashi *et al.* [13]	718	210	1 680	2 340
Keccak	all	Updated submission [4]	1 412	122	6 900	4 887
			444¶	265	70	158
Luffa	256	Kobayashi *et al.* [13]	1 048	223	6 343	6 052
Shabal	all	Baldwin *et al.* [3]*	2 307	222	1 330	577
			2 768	139	1 450	524
		Kobayashi *et al.* [13]	1 251	214	1 739	1 390
		Feron and Francq [9]	1 171	126	2 588	2 210
			596‡	109	1 142	1 916
		This work	153	256	2 051	**13 407**
Skein	256	Long [14]*	1 001	115	409	409
		Tillich [19]	937	68	1 751	1 869
		Kobayashi *et al.* [13]	854	115	1 482	1 735
	512	Long [14]*	1 877	115	817	435
		Tillich [19]	1 632	69	3 535	2 166

*Only the core functionality was implemented. ‡This design requires 40 additional DSP blocks.
†This design uses several additional RAM blocks. ¶This design uses an additional external memory.

Table 4. FPGA implementations of SHA-3 candidates on Spartan-3

Hash function	Digest length	Implementation	Area [slices]	Freq. [MHz]	TP [Mbps]	TP / area [kbps/slice]
BLAKE	256	Beuchat *et al.* [6]	**124**	190	115	927
	512	Beuchat *et al.* [6]	229	158	138	603
CubeHash	all	Baldwin *et al.* [3]*	2 883	59	50	17
			3 268	38	70	21
Grøstl	224/256	Submission doc. [10]	6 582	87	4 439	674
		Jungk *et al.* [12]	6 136	88	4 520	737
			2 486	63	404	163
		Baldwin *et al.* [3]*	3 183†	91	2 330	732
			4 827†	72	3 660	758
			5 508	60	1 540	280
			8 470	50	2 560	302
		Jungk *et al.* [11]	1 276	60	192	150
			1 672	38	243	145
			4 491†	100	2 560	570
			5 693†	54	2 764	486
	384/512	Submission doc. [10]	20 233	81	**5 901**	292
		Baldwin *et al.* [3]*	6 313†	80	2 910	461
			10 293	50	1 830	178
			17 452	43	3 180	182
		Jungk *et al.* [11]	2 110	63	144	68
			2 463	36	164	66
			8 308†	95	3 474	418
Shabal	all	Baldwin *et al.* [3]*	1 933	90	540	279
			2 223	71	740	333
		This work	499	100	800	**1 603**
Skein	256	Tillich [19]	2 421	26	669	276
	512	Tillich [19]	4 273	27	1 365	319

*Only the core functionality was implemented. †This design uses several additional RAM blocks.

implementation of Grøstl-512 [10]. Of course, this reasoning solely applies when hashing distinct messages in parallel, and not one single large message—in which case only the raw throughput matters.

Consequently, we also detail this throughput per area ratio in the last column of Tables 3 and 4. Reaching 13.4 Mbit/s per slice on Virtex-5, and 1.6 Mbit/s/slice on Spartan-3, our design is the best of the literature according to this metric.

5 Conclusion

We have described an FPGA implementation of the SHA-3 candidate Shabal that provides a decent 2 Gbit/s throughput using as few as 153 slices of a Virtex-5. Obtaining this tiny size was made possible by taking advantage of the specificity of the design of Shabal where only a small percentage of the large internal state of the compression function is active at a given time, thus allowing us to

use the builtin SRL16 shift registers of Xilinx FPGAs. This very good tradeoff between size and speed yields the best throughput per area ratio of all SHA-3 implementations published so far. It demonstrates that Shabal is very well suited for hardware implementations, even in constrained environments.

These results should nevertheless be taken with some caution, as our implementation strongly depends on the underlying FPGA technology and architecture, which in our case allows us to benefit from the cheap shift register primitives. However, Altera FPGAs for instance do not support SRL16-like primitives, but shift-register-capable memory blocks. Porting our circuit to such targets might have an important impact on the overall performance. Similarly, an ASIC implementation of our Shabal circuit might not perform as well against other candidates as our Xilinx implementations do. However, the control simplification and scheduling tricks described in this paper are still applicable independently of the considered targets, and should Shabal be selected for the final round of the SHA-3 contest, we plan to investigate these issues further.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful and encouraging comments.

We would also like to express our gratitude to the whole Shabal team for coming up with such a nice hash function to implement in hardware, and more especially Marion Videau who came to us with this funny challenge in the first place!

References

1. The SHA-3 zoo, `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`
2. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE (October 2008), `http://131002.net/blake/`
3. Baldwin, B., Byrne, A., Mark, H., Hanley, N., McEvoy, R.P., Pan, W., Marnane, W.P.: FPGA implementations of SHA-3 candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. In: 12th Euromicro Conference on Digital Systems Design, Architectures, Methods and Tools (DSD 2009), pp. 783–790. IEEE Computer Society, Patras (August 2009)
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak sponge function family (April 2009), `http://keccak.noekeon.org/`
5. Beuchat, J.L., Okamoto, E., Yamazaki, T.: A compact FPGA implementation of the SHA-3 candidate ECHO. Report 2010/364, Cryptology ePrint Archive (June 2010), `http://eprint.iacr.org/2010/364`
6. Beuchat, J.L., Okamoto, E., Yamazaki, T.: Compact implementations of BLAKE-32 and BLAKE-64 on FPGA. Report 2010/173, Cryptology ePrint Archive (April 2010), `http://eprint.iacr.org/2010/173`
7. Bresson, E., Canteaut, A., Chevallier-Mames, B., Clavier, C., Fuhr, T., Gouget, A., Icart, T., Misarsky, J.F., Naya-Plasencia, M., Paillier, P., Pornin, T., Reinhard, J.R., Thuillet, C., Videau, M.: Shabal, a submission to NIST's cryptographic hash algorithm competition (October 2008), `http://www.shabal.com/?page_id=38`

8. Bulens, P., Kalach, K., Standaert, F.X., Quisquater, J.J.: FPGA implementations of eSTREAM phase-2 focus candidates with hardware profile. Report 2007/024, eSTREAM, ECRYPT Stream Cipher Project (January 2007), http://www.ecrypt.eu.org/stream/papersdir/2007/024.pdf
9. Feron, R., Francq, J.: FPGA implementation of Shabal: Our first results (February 2010), http://www.shabal.com/?page_id=38
10. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl: A SHA-3 candidate (October 2008), http://www.groestl.info/
11. Jungk, B., Reith, S.: On FPGA-based implementations of Grøstl. Report 2010/260, Cryptology ePrint Archive (May 2010), http://eprint.iacr.org/2010/260
12. Jungk, B., Reith, S., Apfelbeck, J.: On optimized FPGA implementations of the SHA-3 candidate Grøstl. Report 2009/206, Cryptology ePrint Archive (May 2009), http://eprint.iacr.org/2009/206
13. Kobayashi, K., Ikegami, J., Matsuo, S., Sakiyama, K., Ohta, K.: Evaluation of hardware performance for the SHA-3 candidates using SASEBO-GII. Report 2010/010, Cryptology ePrint Archive (January 2010), http://eprint.iacr.org/2010/010
14. Long, M.: Implementing Skein hash function on Xilinx Virtex-5 FPGA platform (February 2009), http://www.skein-hash.info/downloads/
15. Lu, L., O'Neill, M., Swartzlander, E.: Hardware evaluation of SHA-3 hash function candidate ECHO (May 2009), http://www.ucc.ie/en/crypto/CodingandCryptographyWorkshop/TheClaudeShannonWorkshoponCodingCryptograpy2009/
16. Naehrig, M., Peters, C., Schwabe, P.: SHA-2 will soon retire: The SHA-3 song. Journal of Craptology 7 (February 2010)
17. Namin, A.H., Hasan, M.A.: Hardware implementation of the compression function for selected SHA-3 candidates. Tech. Rep. 2009-28, Centre for Applied Cryptographic Research, University of Waterloo (July 2009), http://www.cacr.math.uwaterloo.ca/techreports/2009/tech_reports2009.html
18. Regenscheid, A., Perlner, R., Chang, S., Kelsey, J., Nandi, M., Paulu, S.: Status report on the first round of the SHA-3 cryptographic hash algorithm competition. Report NISTIR 7620, National Institute of Standards and Technology (September 2009), http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf
19. Tillich, S.: Hardware implementation of the SHA-3 candidate Skein. Report 2009/159, Cryptology ePrint Archive (April 2009), http://eprint.iacr.org/2009/159
20. Xilinx: Spartan-3 generation FPGA user guide, http://www.xilinx.com/support/documentation/user_guides/ug331.pdf
21. Xilinx: Spartan-6 FPGA Configurable Logic Block user guide, http://www.xilinx.com/support/documentation/user_guides/ug384.pdf
22. Xilinx: Virtex-5 FPGA data sheet: DC and switching characteristics, http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf
23. Xilinx: Virtex-5 FPGA user guide, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf
24. Xilinx: Virtex-6 FPGA Configurable Logic Block user guide, http://www.xilinx.com/support/documentation/user_guides/ug364.pdf