Compact Implementations of BLAKE-32 and BLAKE-64 on FPGA

Jean-Luc Beuchat, Eiji Okamoto, and Teppei Yamazaki

Graduate School of Systems and Information Engineering, University of Tsukuba,
1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573, Japan

Abstract. We propose compact architectures of the SHA-3 candidates BLAKE-32 and BLAKE-64 for several FPGA families. We harness the intrinsic parallelism of the algorithm to interleave the computation of four instances of the G_i function. This approach allows us to design an Arithmetic and Logic Unit with four pipeline stages and to achieve high clock frequencies. With careful scheduling, we completely avoid pipeline bubbles. For the time being, the designs presented in this work are the most compact ones for any of the SHA-3 candidates. We show for instance that a fully autonomous implementation of BLAKE-32 on a Xilinx Virtex-5 device requires 56 slices and two memory blocks.

Keywords: SHA-3, BLAKE, fully autonomous implementation, compact implementation, FPGA.

1 Introduction

In this article we present compact architectures of the SHA-3 candidates BLAKE-32 and BLAKE-64, proposed by Aumasson *et al.* [2], on Field-Programmable Gate Arrays (FPGAs). Such implementations are extremely valuable for constrained environments such as wireless sensor networks or Radio Frequency Identification (RFID) technology, where some security protocols mainly rely on cryptographic hash functions (see for example [17]).

After a short introduction to the BLAKE family of hash functions (Section 2), we explain how to implement the required arithmetic operations on several FPGAs (Section 3). Then, we harness the intrinsic parallelism to interleave several computations, and design two pipelined Arithmetic and Logic Units (ALUs) (Section 4). We have prototyped our architecture on several Altera and Xilinx FPGAs and discuss our results in Section 5. We briefly discuss parallel implementations of BLAKE in Appendix A.

2 Algorithm Specification

The BLAKE family combines three previously studied components, chosen by Aumasson *et al.* for their complementarity [2]: the iteration mode HAIFA, the internal structure of the hash function LAKE, and a modified version of Bernstein's stream cipher ChaCha as compression function. BLAKE is a family of

four hash functions, namely BLAKE-28, BLAKE-32, BLAKE-48, and BLAKE-64 (Table 1). In the following, we focus on BLAKE-32 and refer the reader to [2] for more details about BLAKE-28, BLAKE-48, and BLAKE-64. The main differences lie in the length of words and in some constants involved in the algorithm. Once one has a coprocessor for BLAKE-32, writing a VHDL description of another member of the BLAKE family is therefore straightforward and we will only focus on BLAKE-32 in the following.

Table 1. Properties of the BLAKE family of hash functions (reprinted from [2]). All sizes are given in bits).

Algorithm	Word	Message	Block	Digest	Salt
BLAKE-28	32	$< 2^{64}$	512	224	128
BLAKE-32	32	$< 2^{64}$	512	256	128
BLAKE-48	64	$< 2^{128}$	1024	384	256
BLAKE-64	64	$< 2^{128}$	1024	512	256

BLAKE-32 involves only two arithmetic operations: the addition modulo 2^{32} of two 32-bit unsigned integers (denoted by $\boxplus$) and the bitwise exclusive OR of two 32-bit words (denoted by $\oplus$). The latter is sometimes followed by a rotation of k bits to the right (denoted by $\ggg k$). The compression function of BLAKE-32 produces a new chain value $h' = h'_0, \ldots, h'_7$ from a message block $m = m_0, \ldots, m_{15}$, a chain value $h = h_0, \ldots, h_7$, a salt $s = s_0, \ldots, s_3$, a counter $t = t_0, t_1$, and 16 constants c_i given by:

$$
\begin{aligned}
c_0 &= \texttt{243F6A88}, & c_4 &= \texttt{A4093822}, & c_8 &= \texttt{452821E6}, & c_{12} &= \texttt{C0AC29B7},\\
c_1 &= \texttt{85A308D3}, & c_5 &= \texttt{299F31D0}, & c_9 &= \texttt{38D01377}, & c_{13} &= \texttt{C97C50DD},\\
c_2 &= \texttt{13198A2E}, & c_6 &= \texttt{082EFA98}, & c_{10} &= \texttt{BE5466CF}, & c_{14} &= \texttt{3F84D5B5},\\
c_3 &= \texttt{03707344}, & c_7 &= \texttt{EC4E6C89}, & c_{11} &= \texttt{34E90C6C}, & c_{15} &= \texttt{B5470917}.
\end{aligned}
$$

This process consists of three steps. First, a 16-word internal state $v = v_0, \ldots, v_{15}$ is initialized as follows:

$$
\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}.
$$

Then, a series of ten rounds is performed. Each of these rounds consists of a transformation of the internal state v based on the G_i function described by Algorithm 1, where σ_r denotes a permutation of $\{0, \ldots, 15\}$ parametrized by the round index r (see Table 2). A column step updates the four columns of matrix v as follows:

$G_0(v_0, v_4, v_8, v_{12})$, $G_1(v_1, v_5, v_9, v_{13})$, $G_2(v_2, v_6, v_{10}, v_{14})$, and $G_3(v_3, v_7, v_{11}, v_{15})$.

Algorithm 1 The G_i function of BLAKE-32.

Input: A function index i and four 32-bit integers a, b, c, and d.
Output: $G_i(a, b, c, d)$.

1. $a \leftarrow a \boxplus b$;
2. $a \leftarrow a \boxplus (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$;
3. $d \leftarrow (d \oplus a) \ggg 16$;
4. $c \leftarrow c \boxplus d$;
5. $b \leftarrow (b \oplus c) \ggg 12$;
6. $a \leftarrow a \boxplus b$;
7. $a \leftarrow a \boxplus (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$;
8. $d \leftarrow (d \oplus a) \ggg 8$;
9. $c \leftarrow c \boxplus d$;
10. $b \leftarrow (b \oplus c) \ggg 7$;

Note that each call to G_i updates a distinct column of matrix v. Since we focus on compact implementations of BLAKE-32 in this work, we interleave the computation of G_0, G_1, G_2, and G_3. This approach allows us to design an ALU with four pipeline stages and to achieve high clock frequencies. Then, a diagonal step updates the four diagonals of v:

$G_4(v_0, v_5, v_{10}, v_{15})$, $G_5(v_1, v_6, v_{11}, v_{12})$, $G_6(v_2, v_7, v_8, v_{13})$, and $G_7(v_3, v_4, v_9, v_{14})$.

Here again, each call to G_i modifies a distinct diagonal of the matrix, allowing us to interleave the computation of G_4, G_5, G_6, and G_7.

Table 2. Permutations of $\{0, \ldots, 15\}$ used by the BLAKE functions (reprinted from [2]).

σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

At the end of the tenth round, a new chain value $h' = h'_0, \ldots, h'_7$ is computed from the internal state v and the previous chain value h (finalization step):

$$h'_0 \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8, \quad h'_3 \leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11}, \quad h'_6 \leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14},$$
$$h'_1 \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9, \quad h'_4 \leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12}, \quad h'_7 \leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}.$$
$$h'_2 \leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10}, \quad h'_5 \leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13},$$

In order to guarantee that the length $\ell < 2^{64}$ of a message is a multiple of 512, Aumasson *et al.* suggest the following approach [2]: first, they append a bit 1 followed by a sufficient number of 0 bits such that the length is congruent to 447 modulo 512. Then, they append a bit 1 followed by the 64-bit binary representation of ℓ. The hash can now be computed iteratively (Algorithm 2): the padded message is divided into 16-word blocks $m^{(0)}, \ldots, m^{(N-1)}$ and the chain value $h^{(0)}$ is set to the same initial value as SHA-2 (IV_0 = 6A09E667, IV_1 = `BB67AE85`, IV_2 = `3C6EF372`, IV_3 = `A54FF53A`, IV_4 = `510E527F`, IV_5 = `9B05688C`, IV_6 = `1F83D9AB`, and IV_7 = `5BE0CD19`). The counter $t^{(i)}$ denotes the number of message bits in $m^{(0)}, \ldots, m^{(i)}$ (*i.e.* excluding padding bits). Note that, if the last block contains only padding bits, then $t^{(N-1)}$ is set to zero. In the following, we assume that our coprocessor is provided with padded messages. A hardware wrapper interface for the SHA-3 candidates comprising communication and padding is described in [4].

Algorithm 2 Iterated hash.

Input: A padded message split into N 16-word blocks and a salt s.
Output: A 256-bit digest.

1. $\left(h_0^{(0)}, \ldots, h_7^{(0)}\right) \leftarrow (\mathrm{IV}_0, \ldots, \mathrm{IV}_7)$;
2. **for** $i \leftarrow 0$ **to** $N-1$ **do**
3. $\quad h^{(i+1)} \leftarrow$ **compress** $\left(h^{(i)}, m^{(i)}, s, t^{(i)}\right)$;
4. **end for**
5. **return** $h^{(N)}$;

3 FPGA-Specific Issues and their Implications on the Design of BLAKE

Modern FPGAs are mainly designed for digital signal processing applications involving rather small operands (16 to 64 bits). Several FPGA manufacturers (Altera, Xilinx, etc.) chose to include dedicated carry logic enabling the implementation of fast Carry-Ripple Adders (CRA) for such operand sizes.

Let us study the architecture of a Xilinx Spartan-3 device [16]. Figure 1 describes the simplified architecture of a slice, which is the main logic resource for implementing synchronous and combinatorial circuits (one finds the same kind of slices in several other Xilinx FPGAs: Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-4, etc.). Each slice embeds two 4-input function generators (G-LUT and F-LUT), two storage elements (flip-flops FFY and FFX), carry logic (CYSELG, CYMUXG, CYSELF, and CYMUXF), arithmetic gates (GAND, FAND, XORG, and XORF), and wide-function multiplexers (F5MUX, FXMUX, and FiMUX). Each function generator is implemented by means of a programmable Look-Up Table (LUT).

A Full-Adder (FA) cell computes the sum of a carry-in bit $carry_j$ (coming from a lower bit position) and two bits of same magnitude x_j and y_j. The

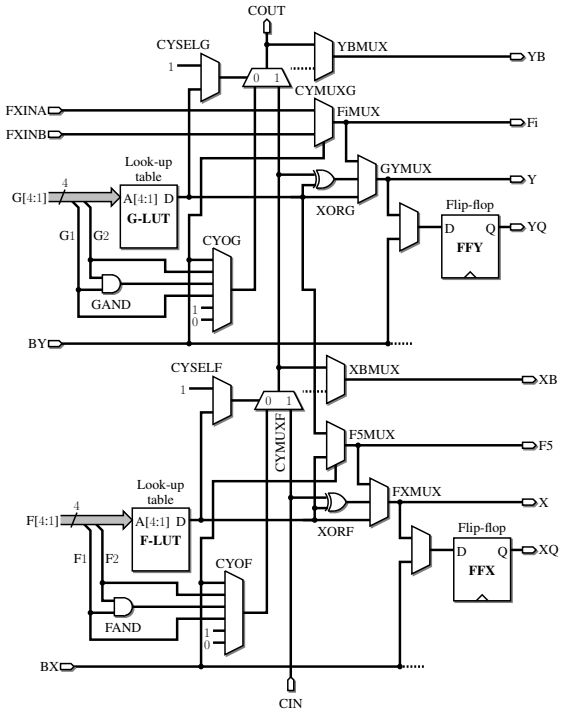


Fig. 1. Simplified diagram of a slice of a Spartan-3 FPGA.

result is encoded by a sum bit sum_j and a carry-out bit $carry_{j+1}$ such that $2carry_{j+1} + sum_j = x_j + y_j + carry_j$. Let $z_j = x_j \oplus y_j$. Then, we have:

$$sum_j = z_j \oplus carry_j, \tag{1}$$

$$carry_{j+1} = \begin{cases} x_j & \text{if } z_j = 0 \ (i.e.\ x_j = y_j), \\ carry_j & \text{otherwise.} \end{cases} \tag{2}$$

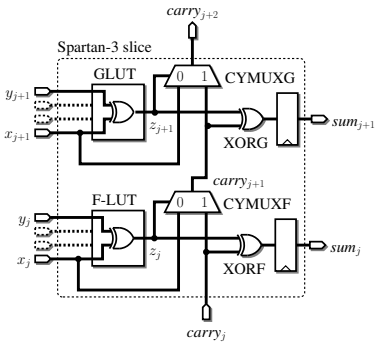
Assume that the F-LUT function generator outputs z_j. Then, the XORF gate computes the sum bit sum_j. The generation of the carry-out bit $carry_{j+1}$ according to Equation (2) involves three multiplexers (CYOF, CYSELF, and CYMUXF). Thanks to the G-LUT function generator, one can implement a second FA cell within the same slice, which thus embeds a 2-bit CRA (Figure 2a). The gates GAND and FAND allows one to build multipliers: one can generate two partial products and compute their sum with a single stage of LUTs.

Since we focus on compact coprocessors for the BLAKE family in this work, we perform a single arithmetic operation at each clock cycle ($\boxplus$ or $\oplus$). A first solution consists in implementing a modular adder and an array of XOR gates, and selecting the operation by means of a multiplexer commanded by a control bit. However, several Xilinx devices (Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-4, and Spartan-3) offer a much more elegant and compact solution (also

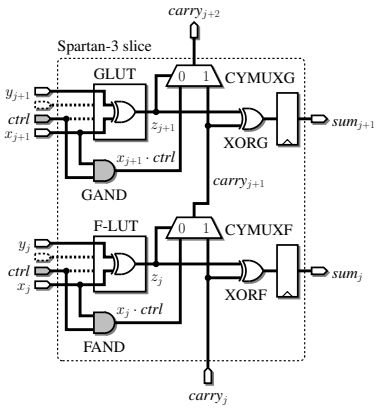
see Appendix B): we can enable or disable carry propagations using a control bit *ctrl* as input to the gates GAND and FAND (Figure 2b). Thus, Equation (2) becomes:

$$carry_{j+1} = \begin{cases} x_j \cdot ctrl & \text{if } z_j = 0 \text{ (\textit{i.e.} } x_j = y_j), \\ carry_j & \text{otherwise.} \end{cases} \tag{3}$$

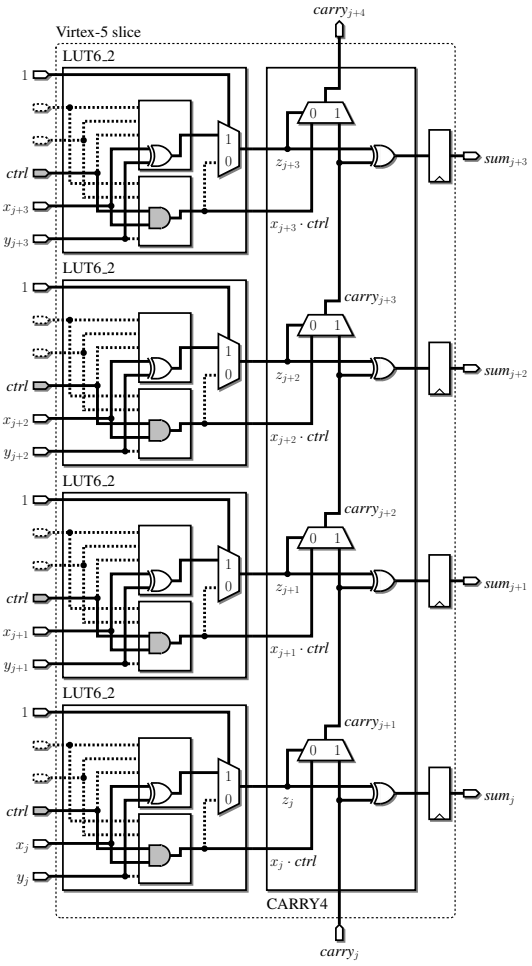
Assuming that $carry_0 = 0$, we easily check that our operator now behaves as a CRA when $ctrl = 1$ and computes the bitwise exclusive OR of its inputs when $ctrl = 0$ (since $ctrl = carry_j = 0$, the output carry $carry_{j+1}$ is also equal to zero and carry propagations are disabled).



(a) Addition of two operands on a Spartan-3 device



(b) Addition or bitwise exclusive OR of two operands on a Spartan-3 device



(c) Addition or bitwise exclusive OR of two operands on a Virtex-5 device

Fig. 2. Addition and bitwise exclusive OR on Xilinx FPGAs.

The latest FPGA families introduced by Xilinx (Virtex-5, Virtex-6, and Spartan-6) are based on 6-input LUTs, each of them having six independent inputs and two independent outputs. It is for instance possible to implement any two 5-input logic functions with shared inputs thanks to this building block (LUT6_2 primitive). Each slice embeds four LUTs and a carry chain that consists of a series of four multiplexers and four XOR gates (CARRY4 primitive). Figure 2c describes how to implement an operator returning either the sum or the bitwise exclusive OR of its two inputs according to a control bit. The carry-out bit $carry_{j+1}$ is determined according to Equation (3), the only difference being that the product $x_j \cdot ctrl$ is now computed within a LUT6_2 primitive. The main drawback of this approach is that design tools are unable to generate such an architecture from a high-level VHDL description of the operator. It is necessary to use specific libraries provided by the FPGA manufacturer and to modify the VHDL code for each device.

Additionally, modern FPGAs feature embedded memory blocks to store relatively large amounts of data. They support several modes (*e.g.* single port, true dual-port, ROM, *etc.*) and port-width configurations. We refer the reader to the technical literature provided by Altera or Xilinx for further details. In this work, we will take advantage of such memory blocks to implement our register file and store the micro-code of our coprocessors.

4 Two Compact Coprocessors for the BLAKE Family

Our compact coprocessor for BLAKE-32 is based on the observation that the four calls to G_i in a column step or a diagonal step can be computed in parallel. In order to achieve a high clock frequency, we suggest to design an ALU with four pipeline stages and to interleave the computation of four G_i functions. A closer look at Algorithm 1 indicates that each instruction involves the result of the previous one. Thus, our ALU includes a feedback mechanism to bypass the register file of the coprocessor.

4.1 Arithmetic and Logic Unit

Figure 3a describes our first ALU designed for FPGAs based on 4-input LUTs. It consists of four stages performing the following tasks:

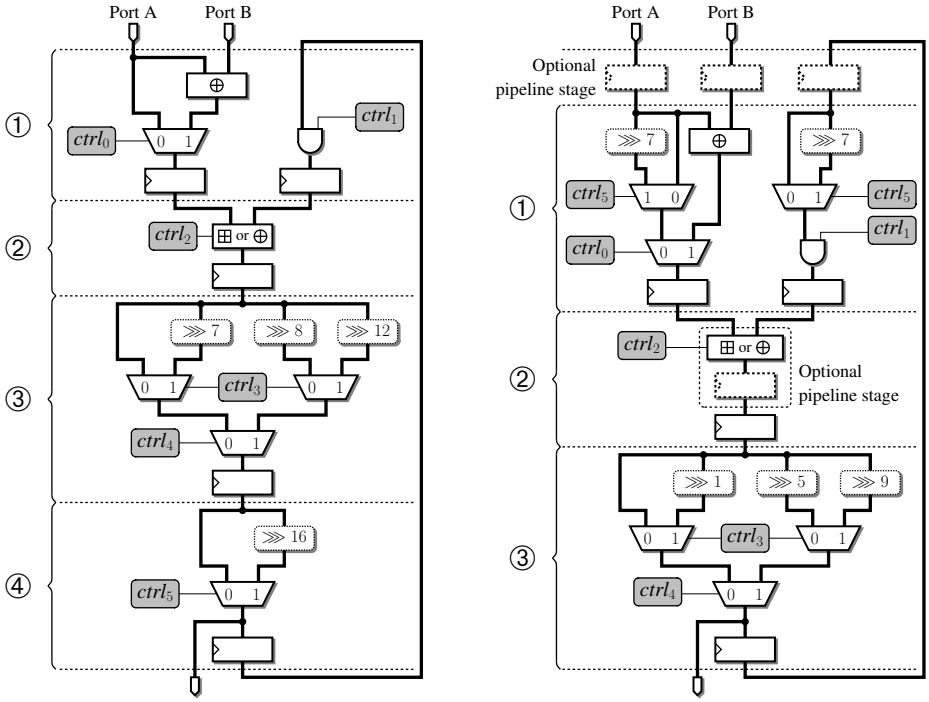
① **Operand selection.** The first operand comes from the register file implemented by means of dual-ported memory. Routing a signal from a memory block to a slice is usually expensive in terms of wire delay and it is recommended to store this signal in a register before performing arithmetic operations. Since a flip-flop is always associated with a 4-input LUT (see for instance Figure 1), we can perform some simple pre-processing without increasing the number of slices of the ALU: a control bit $ctrl_0$ selects either a word read from port A or the bitwise exclusive OR of two words read from ports A and B. This allows us to compute $m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}$ and $m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}$ for free (Algorithm 1, lines 2 and 7).

As explained above, the second input is almost always the result of a previous operation. However, we have to disable this feedback mechanism during the initialization step: the computation of $v_8 \leftarrow s_0 \oplus c_0$ involves for instance two words stored in the register file. An array of AND gates controlled by $ctrl_1$ allows us to force the second operand to zero in such cases. The critical path is limited to a single LUT and a flip-flop.

② **Addition modulo 2^{32} or bitwise exclusive OR.** This stage consists of the arithmetic operator described in the previous section.

③ **Rotation of 0, 7, 8, or 12 bits to the right.** The two multiplexers commanded by $ctrl_3$ are implemented by means of LUTs. On Xilinx FPGAs, the output of this stage is then selected thanks to a F5MUX primitive.

④ **Rotation of 0 or 16 bits to the right.** The final stage allows us to perform the rotation of 16 bits towards less significant bits requested to update d in Algorithm 1 (line 3). Here again, the critical path is limited to a single LUT and a flip-flop.



(a) ALU for FPGAs based on 4-input LUTs

(b) ALU for FPGAs based on 6-input LUTs

Fig. 3. Two arithmetic and logic units for BLAKE-32. (N.B. All control bits $ctrl_j$ belong to $\{0, 1\}$.)

Recall that recent FPGAs embed 6-input function generators. We propose here a simple rewriting of the G_i function that allows us to take advantage of

these new building blocks. It suffices to notice that rotation operations always follow a bitwise exclusive OR and can thus be performed in two steps. We have for instance:

$$(d \oplus a) \ggg 16 = ((d \ggg 7) \oplus (a \ggg 7)) \ggg 9.$$

Algorithm 3 describes an alternative version of the G_i function based on this observation. We obtain a new ALU with three stages (Figure 3b):

① **Operand selection.** We slightly modified the selection of the first operand in order to include the rotation of 7 to the right: the computation of the first operand involves now an array of 5-input LUTs.
② **Addition modulo 2^{32} or bitwise exclusive OR.** This stage consists again of the arithmetic operator described in the previous section.
③ **Rotation of 0, 1, 5, or 9 bits to the right.** This stage simply consists of a 4-input multiplexer implemented by means of an array of 6-input LUTs.

We have two options for the fourth pipeline stage (Figure 3b). The first one consists in storing the inputs in registers in order to reduce the critical path between the register file and the ALU (note that the embedded memory blocks available in several FPGA families include optional output registers). The second one is to introduce pipeline registers to shorten the worst-case carry path of the modulo 2^{32} adder. We strongly recommend to consider both solutions and to select the most appropriate one according to place-and-route results. According to our place-and-route results on Virtex-5 FPGAs, we obtain the best throughput for BLAKE-32 with the first option, whereas the second one seems to be more appropriate for BLAKE-64.

Algorithm 3 The G_i function of BLAKE-32 revisited.

Input: A function index i and four 32-bit integers a, b, c, and d.
Output: $G_i(a, b, c, d)$.

1. $a \leftarrow a \boxplus b$;
2. $a \leftarrow a \boxplus (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$;
3. $d \leftarrow ((d \ggg 7) \oplus (a \ggg 7)) \ggg 9$;
4. $c \leftarrow c \boxplus d$;
5. $b \leftarrow ((b \ggg 7) \oplus (c \ggg 7)) \ggg 5$;
6. $a \leftarrow a \boxplus b$;
7. $a \leftarrow a \boxplus (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$;
8. $d \leftarrow ((d \ggg 7) \oplus (a \ggg 7)) \ggg 1$;
9. $c \leftarrow c \boxplus d$;
10. $b \leftarrow (b \ggg 7) \oplus (c \ggg 7)$;

4.2 Scheduling

We have to be careful in order to avoid pipeline bubbles between a column step and a diagonal step. Figure 4 describes the state of the ALU depicted in

Figure 3a at the end of a column step. It suffices to process the four calls to G_i of the diagonal step in the following order: G_7, G_4, G_5, and G_6. We check for instance that the ALU outputs the new value of v_4 (last instruction of G_0) at time $\tau+4$. If we load v_3 from the register file, we can start the computation of G_7 at time $\tau + 5$. We easily check that this scheduling also avoids pipeline bubbles between a diagonal step and a column step. Since each call to G_i involves ten instructions, we need 80 clock cycles to perform a round of BLAKE-32.

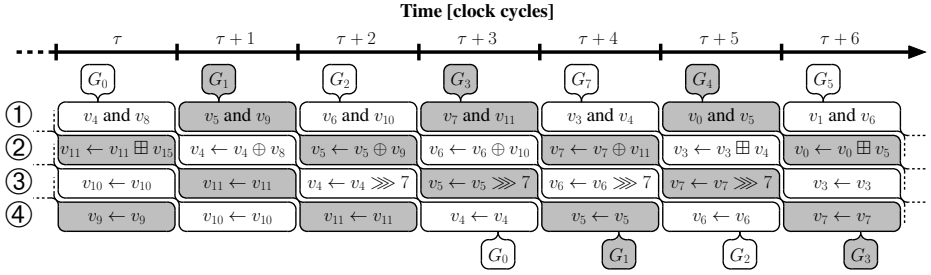


Fig. 4. Avoiding pipeline bubbles between a column step and a diagonal step. The digits ① to ④ refer to the four stages of the ALU depicted in Figure 3a.

The initialization and finalization steps involve 16 and 24 clock cycles, respectively. Furthermore, we need four clock cycles to load v_4, v_5, v_6, and v_7 in the pipeline before the first call to G_0, G_1, G_2, and G_3 (the first operation of G_0 is for instance $v_0 \leftarrow v_0 \boxplus v_4$; recall that we bypass the register file thanks to a feedback mechanism: when we load v_0, we expect the ALU to output v_4). Therefore, we need $16+4+10\cdot 80+24 = 844$ clock cycles to process a 16-word block. In terms of scheduling, the only difference between BLAKE-32 and BLAKE-64 lies in the number of rounds. The latter involves four additional rounds and requires 1164 clock cycles to process a block.

4.3 Register File and Control Unit

The register file stores the 16 constants c_i, a message block $m = m_0, \ldots, m_{15}$, the internal state $v = v_0, \ldots, v_{15}$, the chain value $h = h_0, \ldots, h_7$, the salt $s = s_0, \ldots, s_3$, and the counter $t = t_0, t_1$ (Figure 5). When we process several message blocks (iterated hash), we use the same salt s for each call to the compression function. Therefore, the four words $s_0 \oplus c_0$, $s_1 \oplus c_1$, $s_2 \oplus c_2$, and $s_3 \oplus c_3$ involved in the initialization step are constants that can be computed only once and stored in the register file for subsequent calls. Note that no instruction of BLAKE involves at the same time the salt s and the counter t. Therefore, if we store s and t from addresses 64 to 69, we save a control bit: all variables are accessible from port A (7 address bits), but Port B is restricted to the 64 least significant words of the register file (6 address bits).

The control unit mainly consists of a program counter that addresses an instruction memory implemented by means of a memory block. Our micro-code

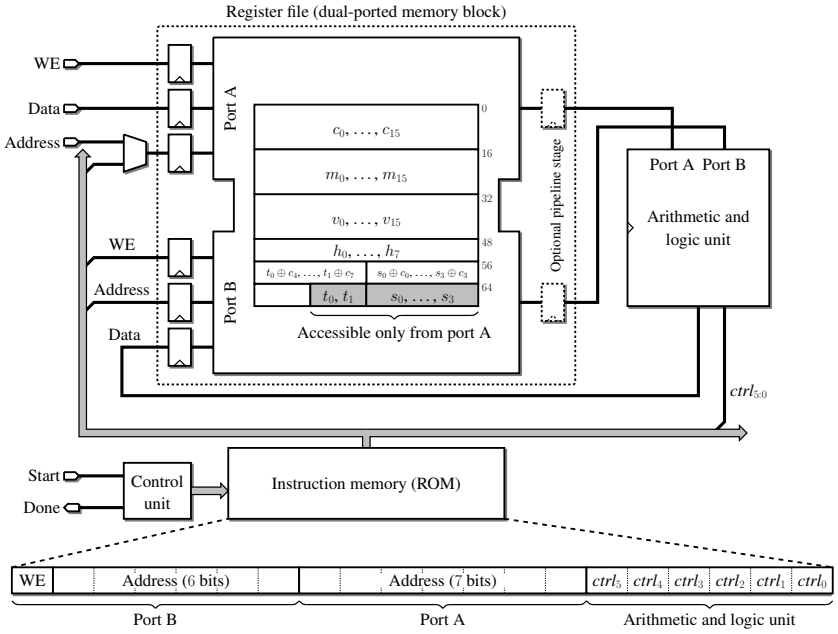


Fig. 5. General architecture of a BLAKE coprocessor.

for BLAKE-32 involves only 14 distinct control words $ctrl_{5:0}$ and it is therefore possible to encode them with 4 bits, thus reducing the size of an instruction to 18 bits at the cost of 6 4-input LUTs. Recall that BLAKE-32 involves 844 instructions[1] and that several FPGAs embed memory blocks whose aspect ratio (*i.e.* width versus depth) is configurable. The 18Kbit blocks available in Spartan-3 or Virtex-4 devices allow one to store 1024 words of 18 bits. Consequently, we can load our micro-code in a single memory block on such FPGAs.

5 Results and Comparisons

We captured our architectures in the VHDL language and prototyped our coprocessors on several Xilinx and Altera FPGAs with average speedgrade (Tables 3 to 6). To our best knowledge, the only compact implementations of BLAKE-32 and BLAKE-64 were proposed by Aumasson *et al.* in their SHA-3 proposal [2]. Their lightweight architecture consists of an initialization unit, a single G_i unit, and a finalization unit. Since they have to read four elements of the internal state v at each clock cycle, they can not implement the register file by means of dual-ported memory and need 16 registers. Our approach leads to a lower

[1] Note that it is possible to reduce the size of the code by storing the table defining the permutation of $\{0 \ldots, 15\}$ parametrized by the round index r (Table 2) and by generating the addresses of $m_{\sigma_r(2i)}$ and $c_{\sigma_r(2i+1)}$ on the fly. However, this approach would require a more complex control unit. As long as the micro-code fits into a single block of memory, there is no need to try to reduce the number of instructions.

throughput, however, our architectures are significantly smaller and improves the area–time trade-off of the compact implementations previously published by Aumasson *et al.* [2]. Note that embedded memory blocks are not a critical resource and we do not report them in the benchmarks.

A few researchers proposed compact implementations of other SHA-3 candidates. We include in our comparisons the results reported in the SHA-3 Zoo [1]. Currently, only five candidates have been evaluated on FPGA, and it is unfortunately difficult to draw conclusions. Among these algorithms, the BLAKE family offers the best area–time trade-off and leads to the smallest coprocessors on reconfigurable devices. We expect the four candidates based on (or at least inspired by) the AES, namely ECHO [5], Fugue [12], Grøstl [10], and SHAvite-3 [8], to require more hardware resources than the BLAKE family. Each of the eight rounds of ECHO involves 32 calls to the AES round function followed by shift operations (BIG.ShiftRows) and arithmetic operations over $\mathbb{F}_{2^8}$ (BIG.MixColumns). To our best knowledge, the smallest hardware implementation of the AES on a Spartan-3 device occupies 163 slices and three memory blocks [14], and computes one round in four clock cycles. The clock frequency on a Spartan xc3s50-4 is equal to 71.5MHz. Note that Good and Benaissa [11] proposed a slightly more compact but slower architecture on Spartan-II. However, since the comparison between the Spartan-II and Spartan-3 families is biased, we focus here on the results reported in [14]. A hardware implementation of ECHO based on such a compact AES coprocessor would already require 1024 clock cycles to compute the AES rounds. Additional slices and clock cycles are then needed to implement the BIG.MixColumns operation. Consequently, our BLAKE-32 coprocessor is smaller in terms of slices and memory blocks, achieves a better clock frequency, and requires fewer clock cycles.

Table 3. Compact implementations of SHA-3 candidates on Xilinx Spartan-3 devices.

	Algorithm	**FPGA**	**Area [slices]**	**Frequency [MHz]**	**Throughput [Mbps]**
This work	BLAKE-32	xc3s50-5	124	190	115
This work	BLAKE-64	xc3s200-5	229	158	138
Jungk *et al.* [13]	Grøstl-224/256	xc3s5000-5	2486	63.2	404
Baldwin *et al.* [3]	Shabal	xc3s5000-5	1933	89.71	540

6 Conclusion

We took advantage of the intrinsic parallelism of the BLAKE family of hash functions to interleave the computation of four instances of the G_i function. Thanks to this approach, we designed an ALU with four pipeline stages and achieved high clock frequencies. A careful scheduling allowed us to totally avoid

Table 4. Compact implementations of SHA-3 candidates on Xilinx Virtex-4 devices.

	Algorithm	**FPGA**	**Area [slices]**	**Frequency [MHz]**	**Throughput [Mbps]**
This work	BLAKE-32	xc4vlx15-11	124	357	216
Aumasson *et al.* [2]	BLAKE-32	xc4vlx100	960	68	430
This work	BLAKE-64	xc4vlx15-11	230	250	219
Aumasson *et al.* [2]	BLAKE-64	xc4vlx100	1856	42	381

Table 5. Compact implementations of SHA-3 candidates on Xilinx Virtex-5 devices.

	Algorithm	**FPGA**	**Area [slices]**	**Frequency [MHz]**	**Throughput [Mbps]**
This work	BLAKE-32	xc5vlx50-2	56	372	225
Aumasson *et al.* [2]	BLAKE-32	xc5vlx110	390	91	575
This work	BLAKE-64	xc5vlx50-2	108	358	314
Aumasson *et al.* [2]	BLAKE-64	xc5vlx110	939	59	533
Bertoni *et al.* [6]	Keccak	xc5vlx50-3	448	265	52
Baldwin *et al.* [3]	Shabal	xc5vlx220-2	2307	222.22	1330
Feron and Francq [9]	Shabal	not specified	596	109	1142

pipeline bubbles and memory collisions. We also addressed FPGA-specific issues: we described how to enable or disable dedicated carry logic in Xilinx devices, thus sharing slices between addition and bitwise exclusive OR of two operands. We showed that a rewriting of the G_i function allows us to fully exploit the 6-input LUTs available in the most recent FPGAs. For the time being, our designs are the most compact ones for any of the SHA-3 candidates.

Acknowledgements

The authors would like to thank Simon Kramer, Jean-Michel Muller, and Francisco Rodríguez-Henríquez for their valuable comments.

References

1. The SHA-3 zoo. `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`.
2. J.-P. Aumasson, L. Henzen, W. Meier, and R.C.-W. Phan. SHA-3 proposal BLAKE (version 1.3). Available online at `http://www.131002.net/blake`, 2009.
3. B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R.P. McEvoy, W. Pan, and W.P. Marnane. FPGA implementations of SHA-3 candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. Cryptology ePrint Archive, Report 2009/342, 2009.
4. B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W.P. Marnane. A hardware wrapper for the SHA-3 hash algorithms. Cryptology ePrint Archive, Report 2010/124, 2010.

Table 6. Compact implementations of SHA-3 candidates on Altera Cyclone III devices.

	Algorithm	**FPGA**	**Area [LEs]**	**Frequency [MHz]**	**Throughput [Mbps]**
This work	BLAKE-32	EP3C5E144A7	285	192	116
This work	BLAKE-64	EP3C5F256I7	542	140	123
Bertoni *et al.* [6]	Keccak	EP3C5F256C6	1559	181	47.8

5. R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 proposal: ECHO. Available online at `http://crypto.rd.francetelecom.com/echo`, 2009.
6. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document (version 2.0). Available online at `http://keccak.noekeon.org`, 2009.
7. J.-L. Beuchat and J.-M. Muller. Automatic generation of modular multipliers for FPGA applications. *IEEE Transactions on Computers*, 57(12):1600–1613, December 2008.
8. E. Biham and O. Dunkelman. The SHAvite-3 hash function (tweaked version). Available online at `http://www.cs.technion.ac.il/~orrd/SHAvite-3`, 2009.
9. R. Feron and J. Francq. FPGA implementation of Shabal: Our first results. Available online at `http://www.shabal.com`, 2010.
10. P. Gauravaram, L.R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S.S. Thomsen. Grøstl – a SHA-3 candidate. Available online at `http://www.groestl.info`, 2008.
11. T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, number 3659 in Lecture Notes in Computer Science, pages 427–440. Springer, 2005.
12. S. Halevi, W.E. Hall, and C.S. Jutla. The hash function "Fugue". Available online at `http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue.index.html`, 2009.
13. B. Jungk, S. Reith, and J. Apfelbeck. On optimized FPGA implementations of the SHA-3 candidate Grøstl. Cryptology ePrint Archive, Report 2009/206, 2009.
14. G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, volume 2, pages 583–587. IEEE Computer Society, 2004.
15. S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely. High-speed hardware implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Cryptology ePrint Archive, Report 2009/510, 2009.
16. Xilinx. Spartan-3 FPGA family, December 2009. Available online at `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`.
17. J. Zhai, C.M. Park, and G.-N. Wang. Hash-based RFID security protocol using randomly key-changed identification procedure. In M. Gavrilova, O. Gervasi, V. Kumar, C.J. Kenneth Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, editors, *Computational Science and Its Applications – ICCSA 2006*, number 3983 in Lecture Notes in Computer Science, pages 296–305. Springer, 2006.

A A Note on Parallel Implementations

In order to achieve a high throughput, Tillich *et al.* [15] recommend to implement four parallel instances of the G_i function, thus computing a round of BLAKE-32 in two clock cycles. They focus on ASIC implementations and use carry-save adders inside the G_i function to shorten the critical path. However, carry-save adders do not take advantage of the dedicated carry logic available in modern FPGAs and other addition algorithms should be considered [7]. Figure 6 describes a parallel implementation of the G_i function. In the following, we explain how to optimize the highlighted part of the circuit by means of Carry-Select Adders (CSAs). Let us consider four 32-bit operands W, X, Y, and Z. We want to minimize the critical path of the operator computing $(((W \boxplus X) \oplus Y) \ggg 16) \boxplus Z$. The main difficulty arises from the rotation operation: if we use CRAs, a first carry propagation occurs when we compute $W \boxplus X$ (Figure 7a). The most significant bit of the sum is then XORed with the most significant bit of Y and becomes an input of weight 2^{15} of the last addition because of the rotation. The critical path includes therefore a second carry propagation (limited to 16 bits).

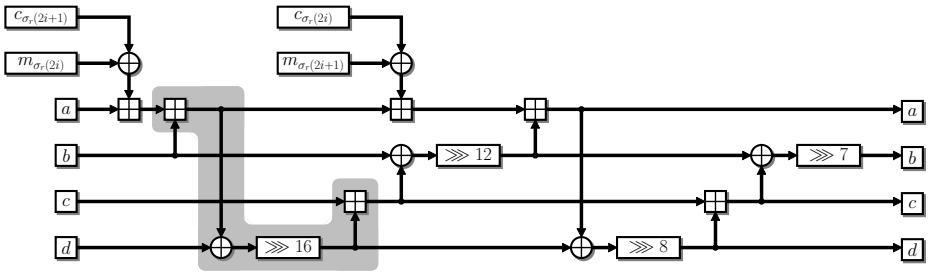


Fig. 6. The G_i function.

A CSA approach allows us to split each operand in two 16-bit chunks. Let $carry_{16}$ denote the carry-out bit of the CRA computing the sum of the lower halves of W and X, namely $W_{15:0}$ and $X_{15:0}$. A first CSA outputs the 16 least significant bits of the result by selecting either $((W_{31:16} \boxplus X_{31:16}) \oplus Y_{31:16}) \boxplus Z_{15:0}$ or $((W_{31:16} \boxplus X_{31:16} \boxplus 1) \oplus Y_{31:16}) \boxplus Z_{15:0}$ according to $carry_{16}$. A second CSA returns the 16 most significant bits of the result, namely $((W_{15:0} \boxplus X_{15:0}) \oplus Y_{15:0}) \boxplus Z_{31:16}$ or $((W_{15:0} \boxplus X_{15:0}) \oplus Y_{15:0}) \boxplus Z_{31:16} \boxplus 1$.

B A Note on Xilinx FPGAs

In Section 3, we described how to enable or disable carry propagations in several Xilinx FPGAs: thanks to a control bit, our operator is able to perform the addition or the bitwise exclusive OR of its inputs. We describe here a slightly more general architecture. On Spartan-3 FPGAs (and on all other Xilinx FPGAs

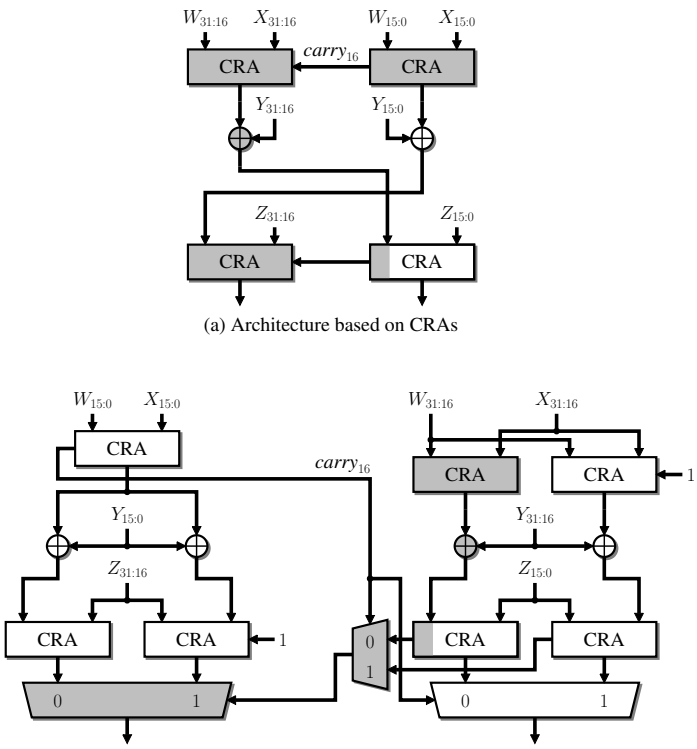


Fig. 7. Computation of $(((W \boxplus X) \oplus Y) \ggg 16) \boxplus Z$. Shaded components belong to the critical path.

based on the same slice architecture), it is possible to compute a sum or any function of up to three Boolean variables (Figure 8).

The LUT6_2 primitive available for instance in Virtex-5 devices offers even more flexibility. One can for instance compute the sum of two partial products or any function of up to four Boolean variables (Figure 8). When the control bit is set to one, we check that:

$$2carry_3 + sum_2 = x_0y_2 + x_1y_1 + carry_2.$$

Otherwise, all carry bits are forced to zero and the circuit computes the function implemented by the LUT4 table.

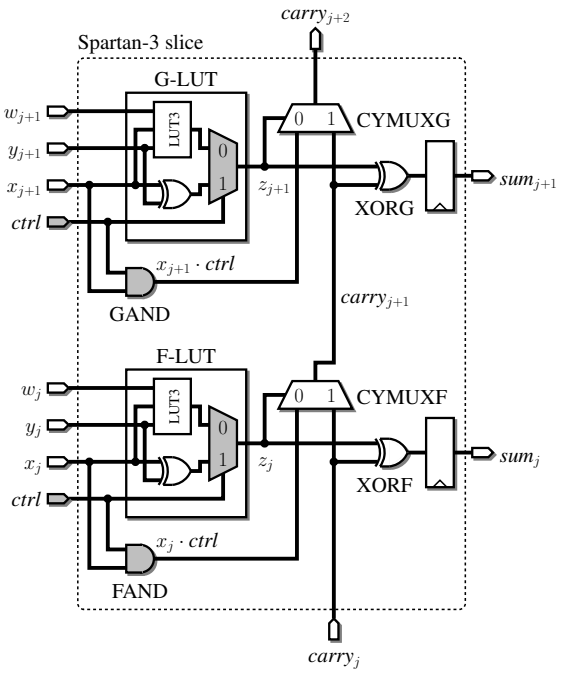


Fig. 8. Addition or function of three Boolean variables.

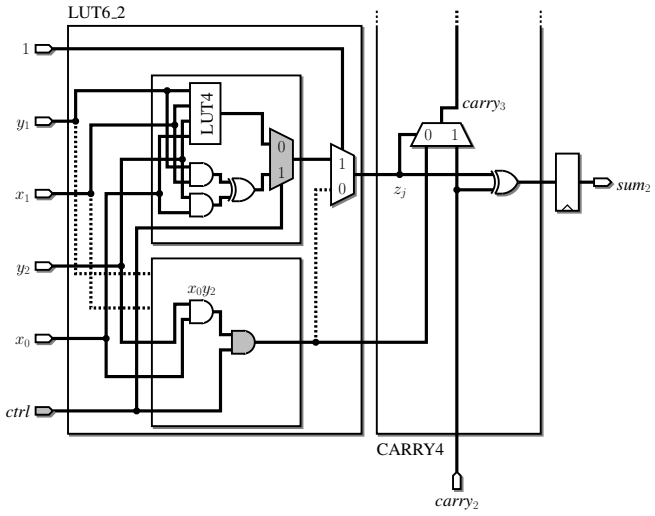


Fig. 9. Sum of two partial products or function of four Boolean variables.