Hardware Framework for the Rabbit Stream Cipher

Deian Stefan*

S*ProCom2// Dept. of Electrical Engineering,
The Cooper Union,
New York NY 10003, USA

Abstract. Rabbit is a software-oriented synchronous stream cipher with very strong security properties and support for 128-bit keys. Rabbit is part of the European Union's eSTREAM portfolio of stream ciphers addressing the need for strong and computationally efficient (i.e., fast) ciphers. Extensive cryptanalysis confirms Rabbit's strength against modern attacks; attacks with complexity lower than an exhaustive key search have not been found. Previous software implementations have demonstrated Rabbit's high throughput, however, the performance in hardware has only been estimated. Three reconfigurable hardware designs of the Rabbit stream cipher – direct, interleaved and generalized folded structure (GFS) – are presented. On the Xilinx Virtex-5 LXT FPGA, a direct, resource-efficient (568 slices) implementation delivers throughputs of up to 9.16 Gbits/s, a 4-slow interleaved design reaches 25.62 Gbits/s using 1163 slices, and a 3-slow 8-GFS implementations delivers throughputs of up to 3.46 Gbits/s using only 233 slices.

Key words: FPGA, Rabbit, eSTREAM, DSP, Stream Cipher

1 Introduction

The widespread use of embedded mobile devices poses the need for fast, hardware-oriented encryption capabilities to provide higher security and protection of private data for end users. Stream ciphers are cryptographic algorithms that transform a stream of plaintext messages of varying bit-length into ciphertext of the same length, usually by generating a *keystream* that is then XORed with the plaintext. In general, stream ciphers have very high throughput, strong security properties, and use few resources, thus making them ideal for mobile applications; well-known examples of stream ciphers include the RC4 cipher used in 802.11 Wireless Encryption Protocol [13], E0 cipher used in Bluetooth protocol [13], and the SNOW 3G cipher used by the 3GPP group in the new mobile cellular standard [26].

* Part of this work was done while the author was visiting EPFL, Switzerland.

The European Union sponsored the four-year eSTREAM project to identify new stream ciphers which address not only strong security properties, but also the need for 1) high-performance software-oriented ciphers and, 2) low-power and low-resource hardware-oriented ciphers. The Rabbit stream cipher is among four software-oriented stream ciphers which were selected for the eSTREAM software portfolio in 2008 [3]. Rabbit performs very well in software (e.g., 5.1 cycles/byte on a 1.7 GHz Pentium 4 and 3.8 cycles/byte on a 533 MHz PowerPC 440GX [6]) and detailed cryptanalysis by the designers and recent studies [2, 20] found no serious weaknesses or attacks more feasible than an exhaustive key search. In [20], Lu *et al.* estimate the complexity of a time-memory-data-tradeoff (TMDT) key-recovery attack to be $2^{97.5}$ with 2^{32} memory usage, 2^{32} precomputations in addition to an exceptionally strong adversary assumption. Moreover, they also present the best distinguishing attack with complexity 2^{158}, which is considerably higher than the exhaustive key search of 2^{128}. The strong security properties of Rabbit makes the cipher a desirable candidate for both software and hardware applications. Until now there were no hardware implementations of Rabbit to evaluate its performance, only estimates of application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) designs; as part of our framework, we present three different architectures suitable for reconfigurable hardware implementations that can be used as standalone hardware or hardware/software co-designs for both cryptographic and cryptanalytic applications.

First we introduce the structure of the Rabbit stream cipher and the mathematical foundations. We then discuss the three hardware architectures of the algorithm: direct, interleaved, and generalized folded structure. The tradeoffs of each are considered along with hardware- and software-based initialization designs. Finally, FPGA implementations and performance benchmarks are presented.

2 Structure of Rabbit

Rabbit is a *symmetric synchronous* stream cipher with a 513-bit internal state derived from the 128-bit key and an optional 64-bit initial vector (IV). From the classical definition of a synchronous stream cipher [22], the internal state during each system iteration is updated according to a *next-state* function dependent on the previous (internal) state, and similarly, the keystream is produced as a function of the internal states, independent of the plaintext or ciphertext. An *output function*, XOR in this case, is

then used to combine the plaintext (ciphertext) message and keystream to produce the output ciphertext (plaintext).

The 128-bit key allows for the safe encryption of 2^{64} plaintext messages [21, 6], while the optional (public) 64-bit IV provides for the safe encryption of up to 2^{64} plaintexts using the same key [8]. Many stream cipher keystream generators are based on the irregular clocking, non-linear combination, or non-linear filtering of the output(s) of linear feedback shift registers (LFSRs) and pseudo-random number generators (PRNGs) [24, 22]. The Rabbit design, although counter-assisted and dependent on the highly non-linear mixing of the internal state, is a novel approach to stream cipher design, adopting random-like properties from chaos theory [7].

The Rabbit 513-bit internal state (at iteration i) is divided into eight 32-bit state variables $x_{j,i},\ 0 \leq j \leq 7$, eight 32-bit counters $c_{j,i},\ 0 \leq j \leq 7$ and a carry bit $\phi_{7,i}$. The design choice of a very large internal state makes TMDT attacks (e.g., key recovery), which rely on "off-line" precomputations to minimize "on-line" computing time, infeasible [5, 6].

2.1 Internal State Update

The internal state update, i.e., a system iteration, is divided into the non-linear next-state update of the state variables $x_{j,i}$'s, and the linear update of the counter variables $c_{j,i}$'s.

Next-state update: At the core of the Rabbit algorithm is the iteration of the state variables, from which the keystream is generated. After the initialization of the internal state (explained in Section 2.2) the next-state function, depending only on the previous state, is used to iterate the system; so, the internal state at iteration $i+1$ depends solely on the non-linear mixing of the internal state at i. Formally, following the notation of [6], the eight 32-bit state variables are updated as follows:

$$x_{j,i+1} = \begin{cases} g_{j,i} + g_{j-1,i} \lll 16 + g_{j-2,i} \lll 16 & \text{for } j \text{ even} \\ g_{j,i} + g_{j-1,i} \lll 8 + g_{j-2,i} & \text{for } j \text{ odd,} \end{cases} \tag{1}$$

where $\lll \alpha$ is a bitwise-rotation by α bits, the additions are mod 2^{32} and all the indices $j-k,\ 0 \leq k \leq 2$ are mod 8 (the number of state and counter variables). The chaos-inspired function g is defined as:

$$g_{j,i} = ((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)) \mod 2^{32}, \tag{2}$$

where $\gg \alpha$ is a bitwise right-shift by α and the inner additions, $(x_{j,i} + c_{j,i+1})$ are mod 2^{32}. The g function is the source of the high non-linearity in the state updates — 256 bits (all the bits of the $x_{j,i}$'s) of the 513-bit internal state are non-linearly transformed; as (1) shows, each state variable is a combination of three outputs from the g function. The g function is the source of the cipher's resistance to algebraic, differential, and linear correlation attacks, which commonly take advantage of ciphers with few non-linear state updates, or the correlation between the difference of inputs and outputs. These attacks seek to determine an output's dependence on the input, find a correlation between the output and internal state or distinguish the keystream from a truly random sequence [12, 1, 7, 6].

Counter update: Similar to the state variable updates, during each iteration the eight 32-bit counter variables are also updated, although linearly, according to:

$$c_{j,i+1} = \begin{cases} c_{0,i} + a_0 + \phi_{7,i} & \text{for } j = 0 \\ c_{j,i} + a_j + \phi_{j-1,i+1} & \text{otherwise} \end{cases} \tag{3}$$

where,

$$a_j = \begin{cases} \texttt{0x4d34d34d} & \text{for } j = 0, 3, 6 \\ \texttt{0xd34d34d3} & \text{for } j = 1, 4, 7 \\ \texttt{0x34d34d34} & \text{otherwise} \end{cases} \tag{4}$$

and the carry $\phi_{j,i+1}$ is:

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } j = 0 \text{ and } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \\ 1 & \text{if } j \neq 0 \text{ and } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

It can be shown that the 256-bit counter state (eight 32-bit counters) has a maximal period length of $2^{256} - 1$ [7], and since the counter variables are used in (2), and thus in the next-state function (1), a lower-bound on the period length of the state variables can also be guaranteed [7, 6].

2.2 Initialization

Key setup: The 128-bit key K is divided into eight 16-bit sub-keys $K = k_7||\cdots||k_0$, where $||$ is the concatenation operation, with the least

significant bit (LSB) bit of k_0 and most significant bit (MSB) of k_7 corresponding to the LSB and MSB of K, respectively. The key is expanded to initialize the counter and state variables according to:

$$x_{j,0} = \begin{cases} k_{j+1}||k_j & \text{for } j \text{ even} \\ k_{j+5}||k_{j+4} & \text{for } j \text{ odd,} \end{cases} \tag{6}$$

and:

$$c_{j,0} = \begin{cases} k_{j+4}||k_{j+5} & \text{for } j \text{ even} \\ k_j||k_{j+1} & \text{for } j \text{ odd,} \end{cases} \tag{7}$$

where the indices $j + k$ are modulo 8. Additionally, the carry $\phi_{7,0}$ is initialized to zero.

Following the key expansion, the system is iterated four times according to the next-state and counter-update functions described in Section 2.1, and finally the counter variables are modified according to:

$$c_{j,4} = c_{j,4} \oplus x_{j+4,4}, \tag{8}$$

where the indices are again mod 8.

The expansion of the key is such that there is a one-to-one correspondence between the key and the 512-bit internal state, while the four system iterations and counter modifications assert both 1) the mixing of all the key bits with every state variable and 2) the combination of the counter with the non-linear state variables [6]. It is important to avoid a many-to-one mapping between the key and internal state as this drastically degrades the strength of the algorithm, for if two keys lead to the same internal state an adversary could potentially generate the same keystream with a different key. Equally essential are the counter modifications, as they prevent key-recovery attacks in which an adversary, with knowledge of the counter's state, can 'clock' the system in reverse and deduce the key. Since the next-state function is resistant to guess-and-verify and correlation attacks [6], and thus resistant to the 'reverse clocking' of the state variables, the modification of the counter variables as in (8) secures against key-recovery attacks.

IV setup: If a 64-bit IV is provided, it is divided into four 16-bit sub-IVs — $IV = iv_3||\cdots||iv_0$ — where the LSB of iv_0 and MSB of iv_3 correspond to the LSB and MSB of IV, respectively. Using the sub-IVs the counters are modified to:

$$c_{j,4} = \begin{cases} c_{j,4} \oplus iv_1||iv_0 & \text{for } j = 0, 4 \\ c_{j,4} \oplus iv_3||iv_1 & \text{for } j = 1, 5 \\ c_{j,4} \oplus iv_3||iv_2 & \text{for } j = 2, 6 \\ c_{j,4} \oplus iv_2||iv_0 & \text{for } j = 3, 7, \end{cases} \tag{9}$$

after which the system is again iterated four times, guaranteeing the non-linear combination of all the IV bits into the state variables [6].

2.3 Keystream Generation

During each iteration i, the state variables $x_{j,i}$ are split into low (L) and high (H) 16-bit sub-states $x_{j,i} = x_{j,i,\mathrm{H}}||x_{j,i,\mathrm{L}}$, from which the 128-bit keystream output, a concatenation of eight 16-bit blocks $s_i = s_{i,7}||\cdots||s_{i,0}$, is extracted according to:

$$\begin{aligned} s_{i,0} &= x_{0,i,\mathrm{L}} \oplus x_{5,i,\mathrm{H}} \quad & s_{i,4} &= x_{4,i,\mathrm{L}} \oplus x_{1,i,\mathrm{H}} \\ s_{i,1} &= x_{0,i,\mathrm{H}} \oplus x_{3,i,\mathrm{L}} \quad & s_{i,5} &= x_{4,i,\mathrm{H}} \oplus x_{7,i,\mathrm{L}} \\ s_{i,2} &= x_{2,i,\mathrm{L}} \oplus x_{7,i,\mathrm{H}} \quad & s_{i,6} &= x_{6,i,\mathrm{L}} \oplus x_{3,i,\mathrm{H}} \\ s_{i,3} &= x_{2,i,\mathrm{H}} \oplus x_{5,i,\mathrm{L}} \quad & s_{i,7} &= x_{6,i,\mathrm{H}} \oplus x_{1,i,\mathrm{L}}. \end{aligned} \tag{10}$$

It is important that adversaries gain no information from the output, that is, they should not be able to distinguish the output of the keystream generator from a truly random sequence [15]. The combination of the outputs of the non-linear g function in the keystream extraction highlights the strength of Rabbit in passing various statistical tests [6], including the NIST Test Suite that seeks to find non-randomness in a sequence [4].

3 Rabbit in Hardware

As previously mentioned, Rabbit is a software-oriented stream cipher and thus was designed to perform well on general purpose architectures, varying from 32-bit Intel processors to 8-bit microcontrollers. Estimates of ASIC and FPGA throughput and area performance are presented in [6], however the implementation details are limited. In the following sections, we consider three architecture designs of the Rabbit algorithm optimized for reconfigurable devices.

3.1 Direct Architecture and General Optimizations

The first architecture we consider is a direct implementation of the algorithm. Observing the relationship between (3) and (5), we note that the counter variables can be updated using a series of chained adders. Each adder takes inputs $c_{j,i}$, a_j and carry-in[1] $\phi_{j-1,i+1}, j > 0$, producing output $c_{j,i+1}$ and carry-out $\phi_{j,i+1}$ each cycle. Figure 1 illustrates the chaining

[1] Note that the carry-in for the first adder is $\phi_{7,i}$.

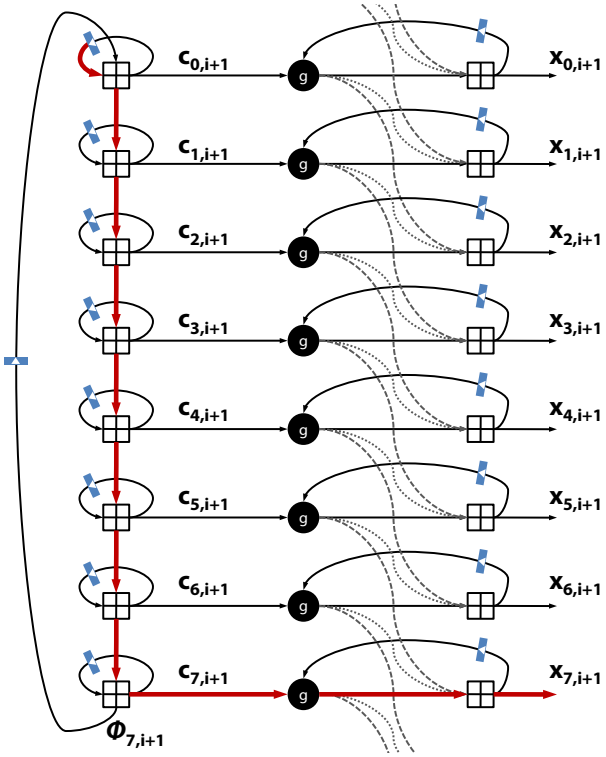


Fig. 1. Direct architecture of the Rabbit algorithm, highlighting the critical path. The $\boxplus$ is a 32-bit adder with carries, while the dotted and dashed lines indicate a variable rotate dependent on whether j is even or odd, see (1). Control logic, a_i inputs, initialization blocks and the keystream extractor are eliminated for clarity.

method within the full architecture design. The updated counters $c_{j,i+1}$ and state variables $x_{j,i}$ are then used as inputs to the g function blocks, the outputs of which, $g_{j,i}$, are combined according to (1) to produce the next state variables $x_{j,i+1}$. Moreover, the next state variables are concurrently combined according to (10) to produce the 128-bit keystream output.

Below, we consider generic hardware optimizations, which are applied to all the designs in the framework, including the direct implementation.

Efficient squaring: In implementing the next-state function, eight parallel realizations of the g function are required. Accordingly, the implementation of g can greatly affect the overall speed performance and area usage. As Boesgaard *et al.* note [6], the most costly part of the g function, the squaring, can be efficiently implemented using three 16-bit multiplies

followed by a 32-bit addition. If we let $u = x_{j,i} + c_{j,i+1}$ and split u into two 16-bit values $u = u_{\mathrm{H}}||u_{\mathrm{L}}$, then the optimization follows directly from the fact that $u^2 = u_{\mathrm{L}}^2 + 2^{32}u_{\mathrm{H}}^2 + 2^{17}u_{\mathrm{L}}u_{\mathrm{H}} \mod 2^{32}$. Thus the full g function, as in (2), can be efficiently implemented using four (2-input) 32-bit adders, three 16-bit multipliers, 3 shifts (which have no cost in hardware, other than routing), and a 32-bit XOR.

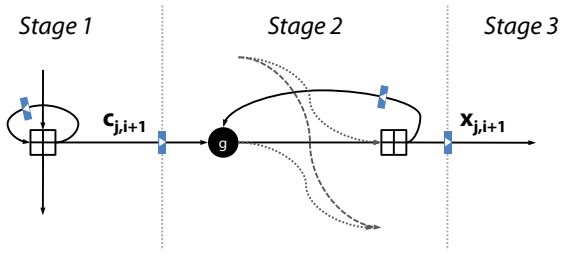


Fig. 2. Three-stage pipeline for the direct architecture of the Rabbit algorithm.

Pipelining: In addition to optimizing g, the speed of the direct design can be further increased by splitting the design into three pipeline stages. Without pipeline registers, the critical path – the path with the highest computational cost between two delay elements – consists of the eight counter adders, a g function (computing $g_{7,i}$), two 32-bit adders (computing $x_{7,i+1}$) and a 16-bit XOR (extracting keystream output); excluding the final XOR, the critical path is highlighted in Figure 1. The critical path can, however, be reduced to either eight 32-bit adders or g and two 32-bit adders[2] by introducing pipeline registers following the counter adders and preceding the keystream output XORs, see Figure 2. To retain correctness, keeping the inputs $c_{j,i+1}$ and $x_{j,i}$ to the g functions synchronized is required and can be accomplished by introducing a latency of one cycle (using clock-enables) for the $x_{j,i}$'s to match the latency introduced by the pipeline register for $c_{j,i+1}$.

***C*-slow retiming:** To further optimize the pipelined design, the critical path, which we experimentally determined to be in the second pipeline stage (the calculation of the the next state variables: g+two 32-bit adders), must be reduced with fine-grained pipelining of the g block, the costliest

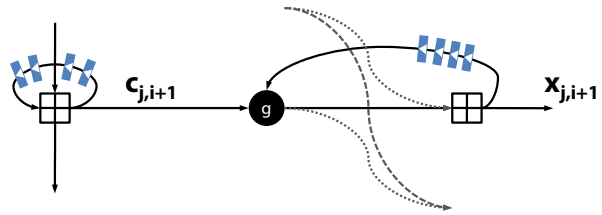
[2] Specifically, the critical path is max(eight 32-bit adders, g+two 32-bit adders).

element in the path. We note that since $g_{j,i+1}$ depends on $x_{j,i+1}$, which is a function of the output of $g_{j,i}$, the direct design cannot take advantage of multiplier pipelining. Instead, we optimize the design with *C-slow retiming*, a DSP system-design technique that allows for the pipelining of structures with feedback loops [23, 27]. C-slow retiming is a modification of a system design in which each register is replaced with C registers (C-slowed) after which the full structure is retimed, whilst retaining algorithmic correctness; we refer the reader to [23] for further details. For $C = 4$, Figure 3(a) illustrates the partial C-slow design before retiming, and Figure 3(b) shows it after retiming, where 3 of the 4 registers were moved into the g block. Retiming stage 2 can thus be seen as fine-grained pipelining of the g function into 3 simpler stages (addition, multiplication, and addition + XOR). Moreover, by pipelining g, the critical path is "reduced" to the eight 32-bit chained counter adders.

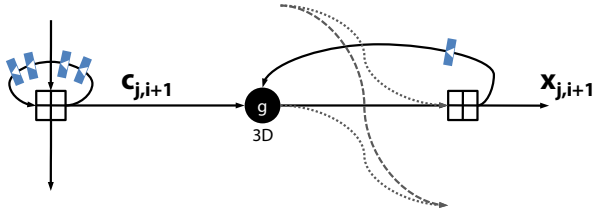
We note that although C-slow retiming can acutely increase the clock rate, the area usage will, in general, increase, as will the number of cycles it takes to complete a single iteration; specifically the number of cycles per iteration will increase to C. Thus to avoid zero-filling the $C - 1$ pipeline registers, it is essential that multiple streams be interleaved, running in parallel, so that during the C cycle system iteration, C independent streams are updated and C different keystream outputs are generated. Multi-stream cipher applications have been studied before (see e.g., [28, 9]), and find use in many applications, including file system encryption, securing virtual private networks, and cryptanalysis.

Initialization: Initialization of the direct architecture requires a key expansion block for (6) and (7), which consist of simple combinations of bit slices used to initialize the state and counter variables; additional control logic (multiplexers) and XORs are needed for the IV setup and modification of the counter as in (8). For multi-stream (C-slow retimed) designs, control logic is necessary to correctly initialize the independent streams.

Alternatively, for hardware/software co-designs, the initialization can be performed in software from which the Rabbit hardware counter, state and carry registers can be loaded; the Rabbit crypto-co-processor and main CPU (e.g., MicroBlaze or PowerPC) can be interfaced using numerous bus protocols that can directly access hardware registers, including the Xilinx Fast Simplex Link (FSL), On-Chip Peripheral Bus (OPB) and the IBM-based Processor Local Bus (PLB). For many security system- and network-on-chip applications, which commonly consist of a CPU and



(a) 4-slow before retiming



(b) 4-slow after retiming

Fig. 3. C-slow retiming for $C = 4$ is accomplished by first replacing each register with C of them, as shown in (a), followed by the retiming, which relocates registers to optimize the design, as shown in (b).

peripherals in addition to the FPGA, initialization in software eliminates the need for additional hardware resources and further simplifies the overall design. Moreover, the saved resources can be dedicated to additional cryptographic cores in multi-stream applications, or to other hardware-assisted applications running concurrently, e.g., MPEG-4 encoder.

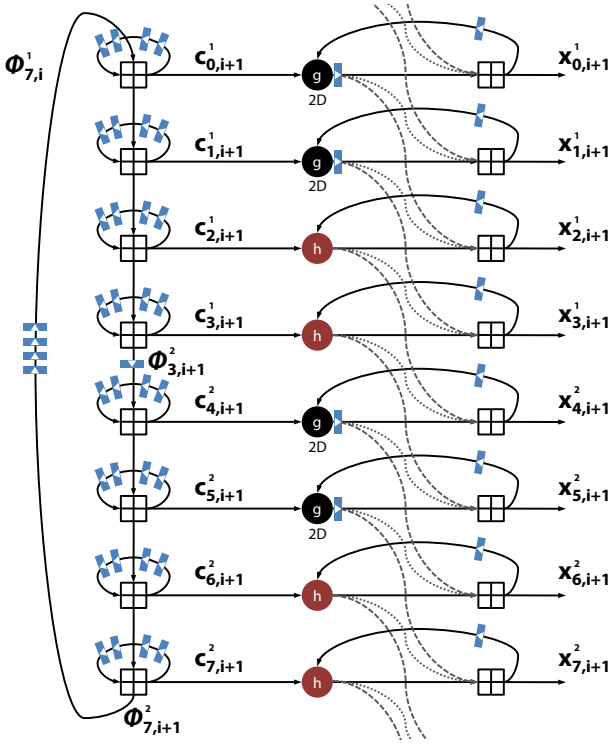
3.2 Interleaved Architecture

Although a C-slow retimed implementation is suitable for hardware, the high data-dependency between the counters (due to the percolating carries $\phi_{j,i+1}$) still poses a limitation on the clock rate. This is because a 256-bit addition[3] must be completed in a single cycle. For a 3-stage pipeline and C-slow retimed design (assuming $C \geq 2$), the cost can be reduced to that of a 128-bit addition using cut-set retiming; in this section we, however, focus on interleaved architecture (IA) design, which is a considerably more balanced structure. See Appendix A for further details on the cut-set retiming approach.

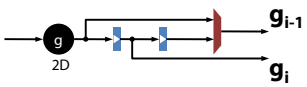
The interleaved design is a generalization of the C-slow retiming approach to fine-grained pipelining of, not only the state variable updates

[3] The eight 32-bit additions with carries is equivalent to a 256-bit addition of $c_{7,j}|| \cdots ||c_{0,j}$ and $a_7|| \cdots ||a_0$ with $\phi_{7,i}$ as a carry-in.

(stage 2 of the pipelined design in Figure 2), but the counter updates as well (stage 1). Given a C-slow design ($C = 2l, l \geq 1$), a C/k-interleaved architecture (in short C/k-IA) interleaves k independent streams in a *single clock cycle* for k cycles (ignoring the initial first cycle used to fill the pipeline), where $k \leq C$ and $k|8$. For example, a 2/2-IA consists of 2 streams which are interleaved such that during the first cycle half of the state variables of each stream are updated and during the second cycle the second half of the variables are updated. As another example, consider the 4/2-IA case; this design is equivalent to interleaving two 2/2-IA streams. We further note that the C-slow retimed design discussed in Section 3.1 is a special case for $k = 1$, i.e., $C/1$-IA.



(a) 4/2-IA design.



(b) h function

Fig. 4. 4/2-Interleaved Architecture design and corresponding h block.

We denote variables of different stream with a superscript, e.g., $c^m_{j,i}$ is the j-th counter variable at iteration i of stream m. For clarity we limit our discussion to the 4/2-IA design shown in Figure 4. From Figure 4 we observe that during the first cycle, half of stream 1's counters $c^1_{j,i+1}, 0 \leq j \leq 3$ and final half of stream 4's counters $c^4_{j,i+1}, 4 \leq j \leq 7$ are updated in the top and bottom of the structure, respectively. Because $\phi_{3,i+1}$ is buffered, in the following cycle we can update $c^1_{j,i+1}, 4 \leq j \leq 7$ in the bottom half, and $c^2_{j,i+2}, 0 \leq j \leq 3$ in the top. Table 1 illustrates the update of the counter variables over time corresponding to Figure 4. With the exception of the first cycle, during every cycle a full-state update is completed.

t	0	1	2	3	4	5	6
Top:	$c^1_{0,i}$	$c^2_{0,i}$	$c^3_{0,i}$	$c^4_{0,i}$	$\boldsymbol{c^1_{0,i+1}}$	$c^2_{0,i+1}$	$c^3_{0,i+1}$
	$c^1_{1,i}$	$c^2_{1,i}$	$c^3_{1,i}$	$c^4_{1,i}$	$\boldsymbol{c^1_{1,i+1}}$	$c^2_{1,i+1}$	$c^3_{1,i+1}$
	$c^1_{2,i}$	$c^2_{1,i}$	$c^3_{2,i}$	$c^4_{2,i}$	$\boldsymbol{c^1_{2,i+1}}$	$c^2_{2,i+1}$	$c^3_{2,i+1}$
	$c^1_{3,i}$	$c^2_{3,i}$	$c^3_{3,i}$	$c^4_{3,i}$	$\boldsymbol{c^1_{3,i+1}}$	$c^2_{3,i+1}$	$c^3_{3,i+1}$
Bottom:	—	$c^1_{4,i}$	$c^2_{4,i}$	$c^3_{4,i}$	$\boldsymbol{c^4_{4,i}}$	$c^1_{4,i+1}$	$c^2_{4,i+1}$
	—	$c^1_{5,i}$	$c^2_{5,i}$	$c^3_{5,i}$	$\boldsymbol{c^4_{5,i}}$	$c^1_{5,i+1}$	$c^2_{5,i+1}$
	—	$c^1_{6,i}$	$c^2_{6,i}$	$c^3_{6,i}$	$\boldsymbol{c^4_{6,i}}$	$c^1_{6,i+1}$	$c^2_{6,i+1}$
	—	$c^1_{7,i}$	$c^2_{7,i}$	$c^3_{7,i}$	$\boldsymbol{c^4_{7,i}}$	$c^1_{7,i+1}$	$c^2_{7,i+1}$

Table 1. Example counter update of 4/2-IA over increasing time t.

Due to the interleaving and need to retain correctness of the algorithm, the retiming of g is slighlty more complex than that of a C-slow design. First, because we start from a 4-slow design, 2 registers can be dedicated to the fine-grained pipelining of g, while the others are used to buffer either 1) the output of g so that the next state variables can be computed according to (1) or 2) the next state variable. As the update is completed over 2 cycles, half of the g blocks need an additional register and a multiplexer (see Figure 4(b)) to select the correct g output; we denote this function by h. For example, in computing $x_{4,j+1}$, the outputs of the first two h blocks (h_2 and h_3) are the previously buffered $x_{2,j+1}$ and $x_{3,j+1}$ (and not the output of the g function).

We further note that for the 4/2-IA, in addition to registers which buffer the next state variables, two keystream extractors are needed in order to produce four 128-bit outputs in four cycles.

3.3 Generalized Folded Structure

Although FPGAs contain digital signal processing (DSP) slices[4] that can be used in implementing an optimized direct or IA design, with the exception of the DSP-enhanced FPGAs (such as the Xilinx Spartan-3A, Virtex-5 SXT and Virtex-4 SX [29]), most FPGAs have a small number of DSP slices which may be necessary for applications other than the encryption module (e.g. Fast Fourier Transform block used for image processing). As such, we seek a more compact implementation of the Rabbit stream cipher.

From (1), (2), (3) and Figure 1, we observe the repeated use of identical circuit blocks in the design (e.g. block g followed by addition), which can be reduced to fewer shared copies at the cost of additional control logic and intermediate state registers. Specifically, the g block, adders and rotation blocks used to update a state variable can be shared to compute all the eight state variables at the cost of 1/8-th the time each computing block is used to update a state variable. Similar to the sharing of resources to update the state variables, the calculation of the eight counter variables at 1/8-th the time per resource can be accomplished by sharing a single adder and carry register.

In DSP terminology, the general design optimization is referred to as a *n-folded* or *n-rolled* design [23], reducing the number of used computational resources (e.g., g blocks) to $1/n$ at the cost of taking n cycles to complete a full iteration. It is constructive to think of folded designs as n threads running on a pipelined system sharing the same computational units, and during every cycle a different thread, cycled in a round–robin fashion, gets a chance to use the computational units (and advance in the pipeline) [16], such that after n cycles all the threads have finished their necessary computations and the iteration is complete.

Although a directly folded design of Rabbit is realizable, it is inefficient because each iteration requires $g_{6,i}$ and $g_{7,i}$ to compute the first two next-state variables, $x_{0,i+1}$ and $x_{1,i+1}$, and as such, an elegant solution buffering only the last two g values is not feasible without the use of an additional g block. Instead, we propose a generalized filter structure that allows access to intermediate values—following the threading analogy: the threads are no longer independent and can share data. Moreover, an n-GFS implementation only requires $1/n$ of the number of computational elements (e.g., adders and g functions) used by a direct implementation.

[4] The design of a DSP slice is FPGA-family-specific, however the most common design is a 18×18 multiplier followed by an adder/accumulator and a small number of registers and multiplexers.

As the counter implementation in an n-GFS architecture is the same as that of a folded design (i.e., in an n-GFS design, the counter system is simply the chaining of $8/n$ adders whose (partial) inputs are n delayed counter variables that need to be updated sequentially), we limit our discussion to the more interesting case of the state updates.

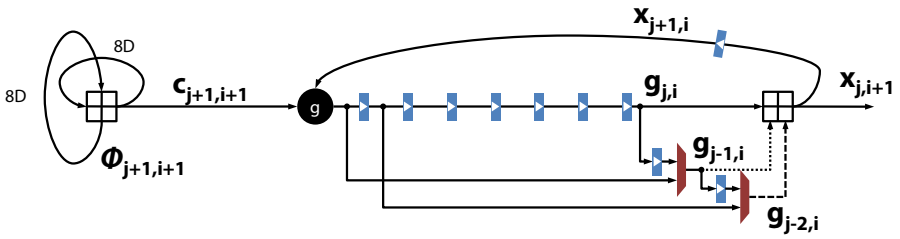


Fig. 5. 8-GFS design. Every 8-th cycle, the multiplexers select the $g_{7,i}$ and $g_{6,i}$ results for the $g_{j-1,i}$ and $g_{j-2,i}$ inputs of the 32-bit adder. The dashed and dotted lines highlight rotations dependent on j.

As shown in Figures 5 the 8-GFS design uses a minimal number of resources, both in terms of the register usage and computational elements (g function and adders). Only two additional registers, which buffer $g_{j-1,i}$ and $g_{j-2,i}$, are needed when computing $x_{j,i+1}$ according to (1). We note that every 8 cycles all intermediate terms, $g_{0,i}$ through $g_{7,i}$, are available and thus any of the next-state variables can be updated, including $x_{0,i+1}$. Similarly, Figure 6 shows the compact 4-GFS design split into a top and bottom pipeline, each computing even and odd next-state variables, respectively. As with the 8-GFS, every $n = 4$ cycles, all the intermediate terms are available and thus $x_{0,i+1}$ and $x_{2,i+1}$ can be computed. A 2-GFS design follows directly from these.

From the figures, we observe that a straight-forward GFS implementation will be limited by the rate at which it can be clocked (due to the fact that the critical path consists of a g block and two 32-bit adders). However, the pipelining and C-slow retiming techniques presented in Section 3.1 are adopted to further speed up the compact n-GFS designs.

Keystream extraction: To extract the keystream output according to (10), a time division demultiplexer (TDD) is needed so that $x_{j,i},\ 0 \leq j \leq 7$ are simultaneously available for the calculation of the s_i's. Since a TDD uses a considerable number of registers, applications of 8-GFS where

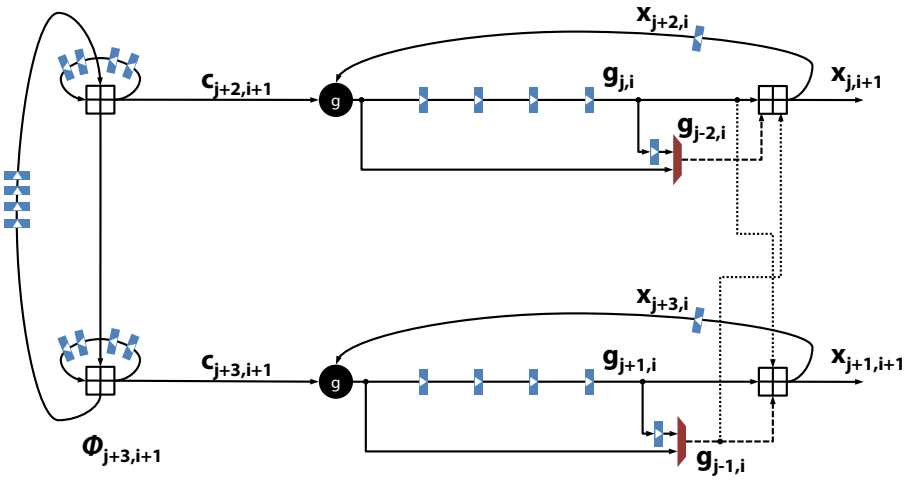


Fig. 6. 4-GFS design with the top pipeline computing every even state variable, and the bottom every odd. Every 4-th cycle, the top and bottom multiplexers select the $g_{6,i}$ and $g_{7,i}$ results, respectively. The dashed and dotted lines highlight rotations dependent on j.

variable output lengths and out-of-order keystreams are acceptable (such as random number generators), the TDD (and following XORs) can be replaced by two 16-bit XORs producing the following output sequence: $s_{i,0}||s_{i,1}$, $s_{i,7}||s_{i,4}$, $s_{i,2}||s_{i,3}$, $s_{i,6}$, $s_{i,5}$. As the 4-GFS does not directly benefit from this optimization, the keystream extractor of 4-GFS consists of a 2-to-8 TDD followed by a series of XORs to generate the output.

Initialization: The generalized filter structure has a very flexible initialization process. For an 8-GFS, the hardware initialization requires additional 1) four registers so that $x_{0,4}$ is available for the modification of $c_{4,4}$ according to (8), 2) two XORs for the mixing of the counters with the state variables and IV, 3) set of control logic. Similar requirements follow for the 4-GFS. We note that although minimal additions are needed for the hardware initialization, software initialization (as discussed in Section 3.1) can be used without the need for any additional resources.

4 Implementation and Discussion

Three direct designs, a 4/2-IA design, and various 4- and 8-GFS designs of the Rabbit cipher were implemented using System Generator and synthesized using Xilinx XST (ISE 11.1). We targeted the Xilinx Virtex-5

LXT (XC5VLX50TFFG1136) FPGA hosted on the Xilinx ML 501 development board, consisting of 7,200 slices, 60 Block RAMs and 48 DSP48 slices. Table 2 summarizes the post-place and route results, where the suffix V is used to identify the implementations with variable output rate (see Section 3.3). We stress the advantage of using C-slow retiming by observing that a direct design can be maximally clocked at 71.58 MHz, while the fine-grained pipelining of the g function increases the clock rate to 141.38 MHz. This nearly doubles the throughput from 9.16 Gbps to 18.10 Gbps, in addition to increasing throughput/area ratio. Although using SLICEM and SLICEL slices (memory- and logic-enhanced slices) for more efficient carry propagation endures a clock rate of 71.58 MHz, we notice the advantage of pipelining the adders in the very high throughput (25.62 Gbps) of the 4/2-IA design; we expect that using C/k-IA designs with $k > 2$ will further allow for an increase in the clock rate, and thus throughput. Furthermore, our results confirm that the estimates made in [6] are reasonably accurate.

Table 2 also shows the performances of the more compact n-GFS designs. The ascent from an 8- to 4-GFS shows a linear increase in the throughput, with only a slight increase in slice count. The single stream 4-GFS and 3-slow 8-GFS are ideal for resource-constrained environments, while delivering reasonably high throughputs (2.74 and 3.43 Gbps, respectively). For cases where variable rate and out-of-order keystream output is acceptable, we recommend the use of the 3-slow 8-GFS, as it outperforms the 4-GFS by more than 26% while using approximately 35% fewer slices, and half the number of DSP slices.

We measured the performance penalty and additional resource of using hardware-initialized designs as compared to hardware/software co-designs to be less than 5% and 10%, respectively. Moreover, since the initialization circuit will not be needed after initialization, we recommend the hardware/software co-design as a very resource efficient design approach.

For completeness, we also compare our results to other stream cipher implementations in Table 2. The table shows previous results of the three eSTREAM hardware-oriented ciphers; a direct comparison is difficult, since [14, 10, 25] are based on the Spartan-3, Virtex-II, and Virtex-II Pro FPGAs and we present results on the Virtex-5 (which is based on the new-generation 6-input LUT architecture). However, we observe that, in general, the throughput/slice ratio of our results is greater than that of Mickey 128 2.0 and comparable with that of Grain. Trivium's throughput/slice is higher than the compared stream ciphers, including

Design	Freq (MHz)	Slices (%)	DSP Slices(%)	Block RAMs(%)	Thruput (Gbps)	Mbps/ Slice
(Rabbit)						
Direct	71.582	568 (7.88%)	24 (50%)	0 (0.00%)	9.16	16.10
Direct, 3-slow	137.155	884 (12.28%)	24 (50%)	0 (0.00%)	17.56	19.86
Direct, 4-slow	141.383	961 (13.35%)	24 (50%)	0 (0.00%)	18.10	18.83
4/2-IA	200.120	1163 (16.15%)	24 (50%)	0 (0.00%)	25.62	22.03
8-GFS	83.724	260 (3.61%)	3 (6%)	0 (0.00%)	1.34	5.15
8-GFS, 2-slow	138.198	368 (5.11%)	3 (6%)	0 (0.00%)	2.21	6.01
8-GFS, 2-slow, V	142.227	239 (3.32%)	3 (6%)	0 (0.00%)	2.28	9.52
8-GFS, 3-slow	214.638	351 (4.88%)	3 (6%)	0 (0.00%)	3.43	9.78
8-GFS, 3-slow, V	216.450	233 (3.24%)	3 (6%)	0 (0.00%)	3.46	14.86
4-GFS	85.697	360 (5.00%)	6 (12%)	0 (0.00%)	2.74	7.62
4-GFS 2-slow	155.982	602 (8.36%)	6 (12%)	0 (0.00%)	4.99	8.29
4-GFS 3-slow	195.198	588 (8.17%)	6 (12%)	0 (0.00%)	6.25	10.62
Estimate [6]	—	—	24	—	17.8	—
(eSTREAM)						
Mickey128 [25]	280.5	392 (2.86%)	0 (0.00%)	0 (0.00%)	0.56	1.43
Grain [14]	155	356 (46.35%)	—	—	2.48	6.97
Grain-128 [10]	181	48 (0.14%)	—	—	0.18	3.77
Trivium [14]	190	388 (10.83%)	—	—	12.16	31.34
(other)						
AES [11]	350	400 (—%)	0 (0.00%)	0 (0.00%)	4.1	10.2
AES [17]	168.3	5177 (37.8%)	—	84 (61.7%)	21.5	4.2
RC4 [18]	64	138 (8.98%)	—	3 (12.5%)	0.22	0.16
LILI-II [19]	—	866 (2.56%)	—	1 (0.69%)	0.24	0.28
SNOW 2.0 [19]	—	1015 (3.00%)	—	3 (2.08%)	5.659	5.57

Table 2. Rabbit Resource Usage and Performance Evaluation.

our 4/2-IA, whose throughput is much higher than all three eSTREAM candidates. We stress that although Rabbit is a software-oriented stream cipher, its performance in hardware is commendable in terms of both throughput and area-usage.

Finally, we compare our results to the Advanced Encryption Standard (AES, Rijndael) and various well-known stream ciphers. In terms of speed, the compact 4-GFS 3-slow Rabbit outperforms all these ciphers, including the Virtex-5 implementation of AES [11], in addition to maintaining the highest throughput/area ratio of 10.62. Similarly, the 4/2-IA outperforms one of the fastest AES implementations [17]; again, a direct comparison is difficult since the AES block cipher of [17] was implemented on older generation Virtex-II Pro FPGAs. In addition to the very high speed performance of Rabbit in hardware, with the exception of RC4,

the compact n-GFS implementations outperform the compared stream ciphers in terms of slices used as well; however we also expect the slice count of the compared ciphers to be lower on a Virtex 5.

5 Conclusion

The first hardware standalone and hardware/software co-designs of the Rabbit stream cipher were presented and optimized using DSP system design techniques. As part of the generalized hardware framework, three different architectures were presented: a direct, interleaved and generalized folded structure. These implementations on the Virtex-5 LXT FPGA outperform previous FPGA implementations of stream ciphers such as MICKEY-128, RC4 and LILI-II, while also maintaining area-efficiencies above 5 Mbps/slice. Future work includes further optimization of Rabbit for ASICs, low-power Spartan-6 FPGAs, and implementation of additional IA and GFS variants.

Acknowledgment The author would like to thank Om Agrawal, David Nummey, and anonymous reviewers for their insightful comments and suggestions. The support of Fred L. Fontaine and `S*ProCom`^{2}, and Arjen K. Lenstra and LACAL is also appreciated.

References

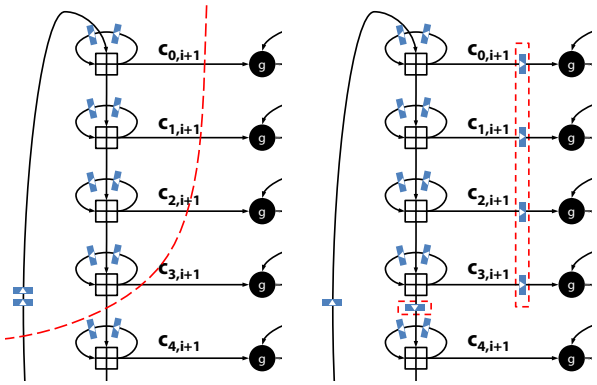
1. Cryptico A/S. Differential properties of the g-function. White paper, `http://www.cryptico.com/Files/filer/wp_differential_properties_gfunction.pdf`, 2003.
2. J.P. Aumasson. On a bias of Rabbit. In *State of the Art of Stream Ciphers Workshop (SASC 2007), eSTREAM, ECRYPT Stream Cipher Project, Report*, 2007.
3. S. Babbage, C. Canniere, A. Canteaut, C. Cid, H. Gilbert, T. Johansson, M. Parker, B. Preneel, V. Rijmen, and M. Robshaw. The eSTREAM Portfolio. *eSTREAM, ECRYPT Stream Cipher Project*, 2008.
4. E.B. Barker, M.S. Nechvatal, E. Barker, S. Leigh, M. Levenson, M. Vangel, G. Discussion, and E. Studies. A Statistical Test Suite For Random And Pseudorandom Number Generators For Cryptographic Applications.
5. A. Biryukov and A. Shamir. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. *Lecture Notes in Computer Science*, pages 1–13, 2000.
6. M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner. The Stream Cipher Rabbit. *ECRYPT Stream Cipher Project Report*, 6, 2005.
7. M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius. Rabbit: A new high-performance stream cipher. *Proc. Fast Software Encryption 2003. Lecture Notes in Computer Science*, pages 307–329, 2003.

8. M. Boesgaard, M. Vesterager, and E. Zenner. The Stream Cipher Rabbit. *New Stream Cipher Designs. Lecture Notes in Computer Science*, 4986:69–83, 2008.
9. J. W. Bos, N. Casati, and D. A. Osvik. Multi-stream hashing on the playstation 3. In *International Workshop on State-of-the-Art in Scientific and Parallel Computing 2008, Minisymposium on Cell/B.E. Technologies*, 2008.
10. P. Bulens, K. Kalach, F.X. Standaert, and J.J. Quisquater. FPGA implementations of eSTREAM phase-2 focus candidates with hardware profile. In *State of the Art of Stream Ciphers Workshop (SASC 2007), eSTREAM, ECRYPT Stream Cipher Project, Report*, 2007.
11. P. Bulens, F.X. Standaert, J.J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. *AFRICACRYPT 2008. Lecture Notes in Computer Science*, 5023:16–26, 2008.
12. N. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology-CRYPTO*, volume 2729, pages 176–194. Springer, 2003.
13. E. Ferro and F. Potorti. Bluetooth and Wi-Fi wireless protocols: a survey and a comparison. *IEEE Wireless Communications*, 12(1):12–26, 2005.
14. K. Gaj, G. Southern, and R. Bachimanchi. Comparison of hardware performance of selected Phase II eSTREAM candidates. *State of the Art of Stream Ciphers Workshop (SASC 2007), eSTREAM, ECRYPT Stream Cipher Project, Report*, 2007.
15. O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press New York, NY, USA, 2000.
16. S. Hauck and A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.
17. A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 308–309, 2004.
18. P. Kitsos, G. Kostopoulos, N. Sklavos, and O. Koufopavlou. Hardware implementation of the RC4 stream cipher. In *Circuits and Systems, 2003. MWSCAS'03. Proceedings of the 46th IEEE International Midwest Symposium on*, volume 3, 2003.
19. P. Leglise, F.X. Standaert, G. Rouvroy, and J.J. Quisquater. Efficient implementation of recent stream ciphers on reconfigurable hardware devices. In *26th Symposium on Information Theory in the Benelux*, pages 261–268, 2005.
20. Y. Lu, H. Wang, and S. Ling. Cryptanalysis of Rabbit. In *Proceedings of the 11th international conference on Information Security*, pages 204–214. Springer, 2008.
21. W. Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.
22. A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of applied cryptography*. CRC press, 1997.
23. K.K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley, 1999.
24. B. Schneier. *Applied Cryptography Second Edition: protocols, algorithms, and source code in C*. John Wiley and Sons, 1996.
25. D. Stefan and C. Mitchell. Parallelized Hardware Implementation of the MICKEY-128 2.0 Stream Cipher. *State of the Art of Stream Ciphers Workshop (SASC 2007), eSTREAM, ECRYPT Stream Cipher Project, Report*, 2007.
26. I.A. UEA2&UIA. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2& UIA2. Document 2: SNOW 3G Specifications. Version: 1.1. *ETSI/SAGE Specification*, 2006.

27. N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 185–194. ACM New York, NY, USA, 2003.
28. C.M. Wee, P.R. Sutton, N.W. Bergmann, and J.A. Williams. Multi stream cipher architecture for reconfigurable system-on-chip. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–4, Aug. 2006.
29. Xilinx. DSP Solutions Using FPGAs. `http://www.xilinx.com/products/design_resources/dsp_central/grouping/fpgas4dsp.htm`, 2009.

A Cut-set Retiming

Given a data flow graph G, cut-set retiming is a technique in which the graph is split into two disconnected subgraphs G_0 and G_1. Further, for every edge from G_0 to G_1, k delays are added and, similarly, for every edge from G_1 to G_0 k delays are removed (note that this assumes the existence of the k delays). We refer to [23] for additional details. Figure A shows part of the chained counter adders of a 4-slow Rabbit cipher with an example of a cut-set (Figure 7(a)) and the respective retiming (Figure 7(b)). This



(a) Determining the cut-set (b) Cut-se retiming

Fig. 7. Example of cut-set retiming to pipeline the chained counter adders.

particular example shows a reduction from a 256-bit addition to two 128-bit additions. Similarly, for $C = 4$ and $C = 8$-slow designs, the 256-bit addition can be further reduced to four 64-bit or eight 32-bit additions, respectively. We note that the IA design of Section 3.2 can be similarly pipelined, however unlike the latter, the cut-set retimed design leads to an unbalanced design with a buildup of many registers between $c_{0,j+1}$ and the g function. As such, we prefer the IA design approach.