VERSITA

TATRA MOUNTAINS
Mathematical Publications
DOI: 10.2478/v10127-011-0039-8
Tatra Mt. Math. Publ. **50** (2011), 87–101

HARDWARE OPTIMIZATIONS OF STREAM CIPHER RABBIT

Jozef Tomecek

ABSTRACT. Stream ciphers form part of cryptographic primitives focused on privacy. Synchronous, symmetric and software-oriented stream cipher Rabbit is member of final portfolio of European Union's eStream project. Although it was designed to perform well in software, employed operations seem to compute efficiently in hardware. 128-bit security, with no known security weaknesses is claimed by Rabbit's designers. Since hardware performance of Rabbit was only estimated in the proposal of algorithm, comparison of direct and optimized FPGA implementations of Rabbit stream cipher is presented, identifying algorithm bottlenecks, discussing optimization techniques applied to algorithm computations, along with key area/time trade-offs.

1. Introduction

The goal of modern cryptography is to deliver security to binary data conveying information. Information is considered secured, when certain cryptographic objectives are fulfilled. As stated in [16], all cryptographic objectives are divided into four main groups—confidentiality (privacy), data integrity, authentication and non-repudiation. Mechanism that defends data utilizing mostly mathematical problems is called cryptographic primitive.

Software oriented stream cipher Rabbit was chosen as a representative of privacy delivering cryptographic objectives. The difference between block cipher and stream cipher relies in encryption transformation [19]. While encryption transformation in block ciphers operates on large blocks of data and remains unchanged, encryption transformation in stream ciphers dynamically evolves with each processed bit.

The goal of the stream cipher is to transform short (hundreds of bits) secret key into long sequence of bits that looks like randomly generated. Most of

2010 Mathematics Subject Classification: 94A60.
Keywords: Rabbit, stream cipher, FPGA, optimization, eStream.
Supported by the Grant VEGA 2/0206/10.

the stream ciphers consist of key-dependent random-like sequence generator and output transformation block. The heart of the cipher is random-like sequence generator characterized by the state of all internal system variables and updated by next-state function. At the beginning of the encryption process internal state variables are initialized with initialization vector (IV). Random-like sequence of bits is extracted from internal state variables regarding output transformation function. Most popular stream cipher architectures are based on linear feedback shift registers (LFSRs) with various output transformations based on irregular clocking of these registers, non-linear combination of values, in particular registers, or non-linear filtering of output values.

Rabbit stream cipher is an iterative cipher with innovative counter-assisted architecture. Moreover, the next-state function in Rabbit is highly non-linear. This algorithm was deeply cryptanalyzed [10], [18], but so far no weaknesses were found in it. It utilizes simple bitwise operations, modular addition and squaring modulo 2^{32}. Performance of software implementation of Rabbit was evaluated by its designers, but the hardware performance was only estimated. Traditional CPUs execute bitwise operations and other specific computations inefficiently, according to their universality and architecture. Also, certain amount of CPU performance is consumed by computation maintenance overhead. So, computations realized on CPUs are generally slower than hardware realizations, no matter how many cores are integrated on the chip and how fast they can run.

On the other hand, FPGAs are slower in terms of frequency, but they are very flexible due to their granularity and very efficient due to concentration of their resources to specific function with smaller maintenance overhead depending on optimization complexity. Designers of hardware algorithms exploit all of these features to overcome CPU computing performance. Implementing an algorithm to perform efficiently in hardware is not an easy job. Compared to software programming in high level language such as C/C++, hardware algorithm designer has to invest approximately ten times more in effort than software designer.

Since hardware performance of Rabbit was only estimated, direct hardware implementation of this stream cipher was developed. Critical parts of the algorithm were optimized using various techniques to accelerate the computation and whole design was simulated, verified and programmed into FPGA.

In the next section, the stream cipher Rabbit is described in detail, identifying computationally most intensive parts. The second section ends with a few words about Rabbit's security. Efficiency improving techniques commonly used in hardware implementations and applied to Rabbit algorithm are outlined in the third section. The last section summarizes performance benchmarks of various optimized implementations of Rabbit and comparison with other hardware implementations of stream ciphers is given.

2. Rabbit stream cipher

Rabbit algorithm consists of the three most important stages. They are *key setup*, *next-state function* and *output transformation*. The algorithm takes 128-bit secret parameter as input and 64-bit initialization vector optionally. All bits of secret key are expanded into internal state variables and counters during key initialization stage. After each iteration of next-state function, 128-bit output sequence is extracted from internal state variables regarding output transformation. Internal state of the cipher consists of 513-bits divided into eight 32-bit state variables and eight 32-bit counters. The one remaining bit stores information about carry from the previous iteration. At the beginning of generating random-like sequence of bits this carry bit is set to 0. Optionally, counter values are initialized with initialization vector after key setup stage.

2.1. Key setup stage

During key setup stage, particular bits of 128-bits long input secret parameter are combined into eight 32-bit internal state variables x_j, $j \in \{0, 1, \ldots, 7\}$ and eight 32-bit counters c_j, $j \in \{0, 1, \ldots, 7\}$. Input sequence $K^{[127..0]}$ of 128-bits defining secret key is divided into eight sub-sequences of 16 consecutive bits $k_0 = K^{[15..0]}$, $k_1 = K^{[31..16]}, \ldots, k_7 = K^{[127..112]}$. Values of internal state variables are given by expanding secret key bits

$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} || k_j & \text{for } j \text{ even,} \\ k_{(j+5 \bmod 8)} || k_{(j+4 \bmod 8)} & \text{for } j \text{ odd,} \end{cases}$$

where $||$ means concatenation of bit strings, k_j represents jth 16-bit sub-sequence from secret key K and identifier $x_{j,0}$ denotes value of jth internal state variable before the first iteration.

Similarly, content of all eight 32-bit counters is initialized by

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} || k_{(j+5 \bmod 8)} & \text{for } j \text{ even,} \\ k_j || k_{(j+1 \bmod 8)} & \text{for } j \text{ odd.} \end{cases}$$

Again, $||$ means concatenation of bit strings, k_j represents jth 16-bit sub-sequence from secret key K and identifier $c_{j,0}$ denotes value of jth counter before the first iteration.

After the key setup stage, the system is iterated four times according to next--state function. Mixing of bits in internal state variables and counters performed during these four iterations dims direct dependencies between secret key bits and values in internal state variables and counters. After four iterations of next-state function, counter values are modified by XOR-ing appropriate state variable value given by:

$$c_{j,4} = c_{j,4} \oplus x_{(j+4 \bmod 8),4},$$

$\oplus$ denotes bitwise addition modulo 2 (or XOR), $c_{j,4}$ denotes value of jth counter after the fourth iteration and $x_{j,4}$ represents value of jth internal state variable after the fourth iteration.

2.2. Next-state function

In the next-state function, all eight 32-bit state variables and all eight 32-bit counters are updated. Preceding update of state variables, new counter values are calculated by:

$$\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} \mod 2^{32},\\
c_{1,i+1} &= c_{1,i} + a_1 + \phi_{0,i+1} \mod 2^{32},\\
c_{2,i+1} &= c_{2,i} + a_2 + \phi_{1,i+1} \mod 2^{32},\\
c_{3,i+1} &= c_{3,i} + a_3 + \phi_{2,i+1} \mod 2^{32},\\
c_{4,i+1} &= c_{4,i} + a_4 + \phi_{3,i+1} \mod 2^{32},\\
c_{5,i+1} &= c_{5,i} + a_5 + \phi_{4,i+1} \mod 2^{32},\\
c_{6,i+1} &= c_{6,i} + a_6 + \phi_{5,i+1} \mod 2^{32},\\
c_{7,i+1} &= c_{7,i} + a_7 + \phi_{6,i+1} \mod 2^{32}.
\end{aligned} \tag{1}$$

In the above equations, $\phi_{j,i+1}$ denotes new $(i+1)$ value of jth counter carry bit, which is computed from actual counter value $(c_{j,i})$, a constant a_j and carry from previous counter value calculation by

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

where constants a_j are defined as follows

$$a_j = \begin{cases} 0x4D34D34D & \text{for } j = 0,3,6,\\ 0xD34D34D3 & \text{for } j = 1,4,7,\\ 0x34D34D34 & \text{for } j = 2,5. \end{cases}$$

Please note that $0x$ denotes number in hexadecimal format. When counter update stage is completed, new state variable values are calculated by:

$$\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16),\\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i},\\
x_{2,i+1} &= g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16),\\
x_{3,i+1} &= g_{3,i} + (g_{2,i} \lll 8) + g_{1,i},\\
x_{4,i+1} &= g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16),\\
x_{5,i+1} &= g_{5,i} + (g_{4,i} \lll 8) + g_{3,i},\\
x_{6,i+1} &= g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16),\\
x_{7,i+1} &= g_{7,i} + (g_{6,i} \lll 8) + g_{5,i},
\end{aligned} \tag{3}$$

where $g_{j,i}$ represents actual value of jth g-function, $g_{j,i} \lll m$ denotes bitwise rotation of binary sequence representing current value of jth g-function by m bits to the left and $x_{j,i+1}$ denotes new value of jth state variable. In state variable update equations all additions are modulo 2^{32}. Each state variable is updated by combination of output from three different g-functions. g-functions deliver high nonlinearity to state variable values and thus whole system. Values of g-functions are computed by rule

$$g_{j,i} = \Big((x_{j,i} + c_{j,i+1})^2 \oplus \big((x_{j,i} + c_{j,i+1})^2 \gg 32\big)\Big) \bmod 2^{32}. \tag{4}$$

Since each g-function value depends on one internal state variable value $x_{j,i}$ and appropriate counter value $c_{j,i+1}$, total number of g-function values is 8, indexed by $j \in \{0, 1, \ldots, 7\}$. Updated value of jth counter is denoted by $c_{j,i+1}$, $x_{j,i}$ denotes current value of jth internal state variable, $(\ldots)^2$ means squaring operation and $(\ldots) \gg m$ denotes shift of binary sequence by m bits to the right. Shifted sequence is filled with m 0s from the left. Again, both additions in all eight g-function computations are modulo 2^{32}.

Output value from jth g-function is produced by combination of current jth internal state variable value $x_{j,i}$ and updated value of jth counter $c_{j,i+1}$. Binary representations of these two values are both 32-bits long. Sum of two 32-bit numbers produces at most 33-bit number, thus only the last 32-bits of the sum are taken. Resulting sum of internal state variable value and counter value is then squared, producing 64-bits long number. Squared sum is shifted by 32 positions to the right afterwards and shifted sequence is filled with sequence of 32 zeros from the left. Shifted version of squared sum is added to the original squared sum, producing 64-bit number. Only the last 32-bits are taken as output value from g-function.

2.3. Initialization vector

By using initialization vector and requesting 2^{32} different IVs, an attacker does not gain an advantage over using the same IV [9]. After key expansion into internal state variables and counters, four iterations of the system, and counter re-mixing with internal state variables in key setup stage, counters in the internal state are modified with initialization vector bits according to:

$$\begin{aligned}
c_{0,4} &= c_{0,4} \oplus IV^{[31..0]}, & c_{1,4} &= c_{1,4} \oplus \big(IV^{[63..48]} || IV^{[31..16]}\big),\\
c_{2,4} &= c_{2,4} \oplus IV^{[63..32]}, & c_{3,4} &= c_{3,4} \oplus \big(IV^{[47..32]} || IV^{[15..0]}\big),\\
c_{4,4} &= c_{4,4} \oplus IV^{[31..0]}, & c_{5,4} &= c_{5,4} \oplus \big(IV^{[63..48]} || IV^{[31..16]}\big),\\
c_{6,4} &= c_{6,4} \oplus IV^{[63..32]}, & c_{7,4} &= c_{7,4} \oplus \big(IV^{[47..32]} || IV^{[15..0]}\big),
\end{aligned}$$

where $\oplus$ denotes XOR, $c_{j,4}$ represents 32-bit value stored in counter j after four iterations of next-state function, $IV^{[u..v]}$ represents bits from u to v of initialization vector and $||$ operation is concatenation of bit strings. When counter values

are updated with initialization vector bits, whole system is iterated four times for security reasons [9]. Overall system function can be visualized as on Figure 1.

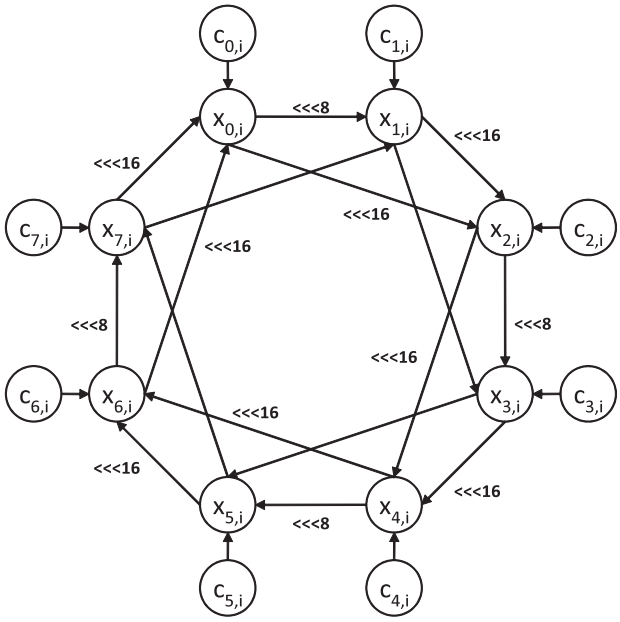

FIGURE 1. Graphical illustration of Rabbit.

2.4. Output transformation

Beginning with the 5th iteration when no IV is used, or 9th iteration when IV is used, output random-like sequence of 128-bits is extracted from internal state variables after the iteration is performed. The extraction rule is defined as:

$$\begin{aligned}
s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]}, & s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]}, \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]}, & s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]}, \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]}, & s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]}, \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]}, & s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]},
\end{aligned} \tag{5}$$

where $\oplus$ denotes XOR, $x_{j,i}^{[u..v]}$ denotes bits from u to v of jth internal state variable in current (ith) iteration and $s_i^{[u..v]}$ represents bits from u to v of output random-like sequence s in actual (ith) iteration. Encryption of the message is simple bitwise XOR of message bits m_i with output sequence bits s_i, so encrypted

message bits are produced by $c_i = m_i \oplus s_i$. Decryption follows by reapplying output sequence bits to the cipher-text bits as $m_i = c_i \oplus s_i$.

2.5. Security of Rabbit

Security of potentially vulnerable blocks of Rabbit stream cipher were deeply analyzed in [3], [5], [6] and [4]. 128-bit security is claimed for Rabbit. It means, that reasonable attack has to be more efficient than 2^{128} trial encryptions [9]. Guess-and-Verify attack to Rabbit g-function has complexity equivalent to 192-bits [9]. Also in Guess-and-Determine attack, attacker must guess more than 128-bits before determining process can start. Known algebraic attacks to stream cipher designs mostly exploit vulnerabilities in linear properties of next-state functions in stream ciphers. Since Rabbit updates its internal state in non-linear fashion it is considered, that algebraic attacks are not applicable to Rabbit [9].

3. Hardware implementation of Rabbit

Rabbit was originally designed as software-oriented algorithm, thus performance was tested on many processors from 32-bit 1.7 GHz Pentium 4 processor to simple 8-bit micro-controllers. It can be seen, that set of operations required by key setup function including initialization process, next-state function and output transformation includes XOR-ing, rotations, shifts, modular addition and modular multiplication. From hardware point of view, XOR-ing, shifting and bit rotation are quite simple operations. On the lowest hardware level, where FPGA devices belong, generally we have no preprogrammed complex functions, even there are no instructions for particular operations. Thus, hardware implementation of any algorithm is more complex than software implementation on any level of abstraction. When programming FPGAs, programmer deals with logic gates, look-up tables, registers and wiring connections between them. On the other side lies flexibility of hardware implementation. The simplest example is adding of two numbers. On 32-bit CPU it is no matter if those two operands are 5-, 12- or 30-bits long. Since 32-bit CPU has only 32-bit wide arithmetic-logic unit, it is harnessed for any of above operations. Hardware designer can implement adder block with as many chained one-bit adders as needed, building adder block with exactly required size. Moreover, with respect to available resources of used logic gate array, more adders can be synthesized and used in parallel.

3.1. Critical path in Rabbit

What is fundamental for hardware algorithm speed is the longest path for electric signal to travel through logic gates from the input pin of logic array

to the output pin. Each logic gate or block of logic gates has some combinational delay. The highest combinational delay on the critical path through logic elements between two synchronous registers corresponds to the worst-case route of electric signal when propagating signal through combinational logic and gives minimal period of one algorithm iteration. Maximum frequency of whole design is then given by inverting this minimal period. Reduction of combinational delays in design leads to significant algorithm acceleration. In case of Rabbit, output pseudo-random sequence is extracted from state variable values (5). Going back to state variable value update during next-state function it is visible, that each state variable value is given by combination of output from three g-functions (3). In the rule for g-functions (4), actual internal state variable value is added to updated counter value, so critical path goes up to new counter value calculation (1), which is performed at the beginning of each iteration. From (3) it follows, that all eight g-function values are needed in each iteration for state variable update. Looking into (1) it is clear, that $c_{7,i+1} = c_{7,i} + a_7 + \phi_{6,i+1} \bmod 2^{32}$ is computed last, because it is waiting for $\phi_{6,i+1}$, which is computed by (2), thus depending on carry from previous counter value calculation and vice versa.

Counting from the first counter, on the critical path of Rabbit is eight 32-bit adders, one 32-bit squaring block and two 32-bit XORs (one in g-function and one in output transformation). Shifts and rotations have no cost in hardware other than routing the wires. Critical path of Rabbit is coarse-dashed on Figure 2.

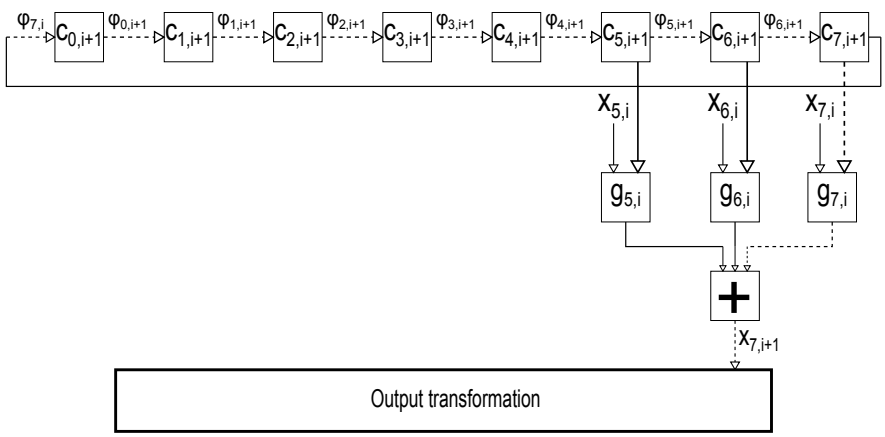

FIGURE 2. Critical path in Rabbit.

3.2. Optimal squaring in g-function

As noted in [9], computationally most complicated operation is squaring of 32-bit number in g-function. Since squaring is in other words multiplying number

by itself, we can save some computation time by replacing 32-bit squaring with equivalent formula of three 16-bit multiplications, two 32-bit additions and two shifts.

Let $w = (x_{j,i} + c_{j,i+1}) \bmod 2^{32}$. We can split 32-bit word w to two 16-bit words $w = w_h || w_l$, then

$$w^2 = w_l^2 + 2^{16} * 2 * w_l * w_h + 2^{32} * w_h^2. \tag{6}$$

Corresponding to basic multiplication of two distinct decadic numbers, before adding up intermediate results, those must be proprietary shifted (w_h^2 is shifted by 32-bits and $w_h * w_l$ is shifted by 16+1-bit, both numbers are shifted to the left). Multiplication with $n-th$ power of two is the same as left shift by n positions in hardware, thus replacing resource demanding 32-bit multiplication with more, but narrower operations seems reasonable. Moreover, in squaring we can save one multiplication, since $w_h * w_l = w_l * w_h$. That's why $w_h * w_l$ is shifted by 16 + one bit to the left. Similarly, when squaring a 16-bit number v, it can be split into two bytes $v = v_h || v_l$ and squared with formula $v^2 = v_l^2 + 2^8 * 2 * v_l * v_h + 2^{16} * v_h$ utilizing only 8-bit multiplication units. Squaring operation performed on 32-bit number partitioned into one-byte words then takes ten multiplications (three multiplications per 16-bit squaring, resting four for inner product multiplication), two 32-bit and four 16-bit additions and six shifts.

3.3. Sequential and parallel tasks

Exploiting great flexibility of FPGA devices, optimal implementation of Rabbit stream cipher algorithm can be designed, compared to sequentially executed software implementation. Adopting methods from parallelizing software algorithms [17], hardware implementation of Rabbit stream cipher algorithm was decomposed to sequentially and parallely executed tasks. Tracing the critical path of algorithm by following data dependency seems most suitable for hardware algorithm decomposition, see Figure 3. Secret key bits are parallely expanded into state variables and counters, counters are parallely updated with initialization vector and reinitialized during key setup and initialization process. Eight parallel implementations of counter update functions with carry prediction were implemented. Carry prediction comes from fact that counter update function is simple two input 32-bit adder with carry input and carry output. If sum of two input 32-bit numbers is greater than or equal to 2^{32}, carry bit is propagated to next counter update function, no matter if there is carry from previous counter. Since particular counters are chained through carry bits, carry prediction can speed-up counter update and thus g-function calculation. Also, eight parallel g-functions are implemented. When output from g-functions is combined into state variable values in next-state function, pseudo-random sequence is extracted in parallel from state variables.

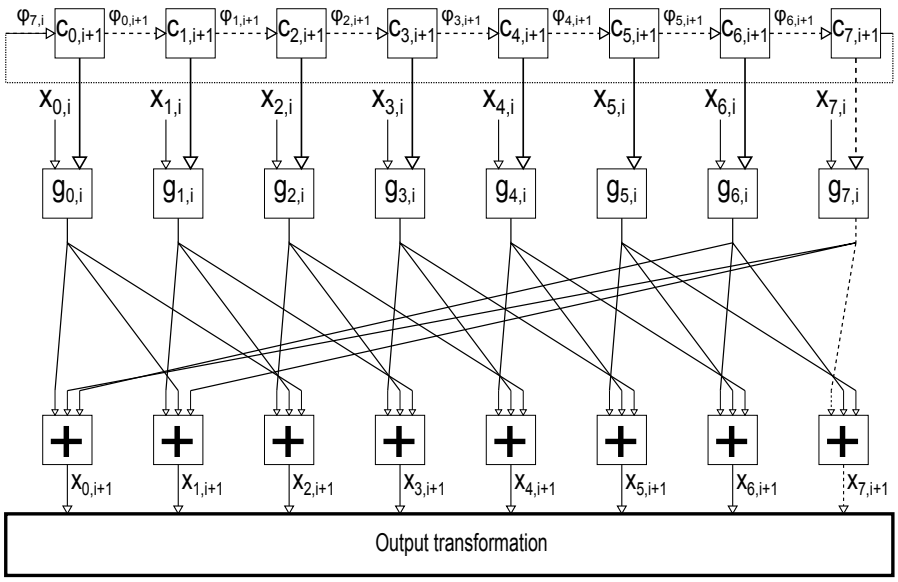

FIGURE 3. Parallel and sequential blocks in Rabbit.

3.4. Efficient resource utilization

The goal of decomposition an algorithm into sequential and parallel blocks is to reach optimal balance between keeping parallel blocks busy with computation, while holding surrounding control logic small and efficient. If the granularity of parallel blocks is very fine, complexity of finite state machines monitoring and controlling them grows, thus slows algorithm execution. So hardware algorithms often face the same challenge as universal microprocessors (CPUs) in efficient resource utilization. Rabbit algorithm can be logically divided into three stages which run almost independently, but have synchronized data input and output. Counter system forms first logic block, because counter values depend only on secret key bits, except counter re-initialization. So after key setup stage is completed, during iterations of next-state function, new counter values are computed independently in counter system block. Output from counter block is registered and readiness of new data is signaled to particular g-function. In the second logic block, g-function values are computed. Within g-function, output value from counter block is combined with current internal state variable value and result is squared, shifted and the sum of original and shifted version of inner product is computed. When g-function values are ready, new state variable values are generated from appropriate g-function output values in the third stage. Whole loop is controlled by finite state machine, which is monitoring data flow and schedules work-flow for particular stages.

3.5. Horizontal folding

When optimizing hardware algorithm, designers are focused on three basic targets: speed, area and power consumption. Algorithm can be accelerated with parallely executed computations, but it needs more combinational and control logic and sometimes more energy. On the other hand, parallel operations can be computed in sequence of many iterations, reducing consumed logic, but increasing time to complete computation. As is visible from (3), state variables are computed by combination of consecutive g-functions. In our resource-efficient implementation of Rabbit stream cipher, there is only one g-function block implemented and shared between eight state variables with registered output, so each g-function is computed once, but used in three state variable calculations. Similarly, counter update block is shared between all eight counters. Sharing logic blocks is called horizontal folding [15] and needs slightly updated control logic to register and synchronize output values from shared blocks. This approach is also scalable, where one counter block and next-state function block can be shared between two, four or all eight data sets.

3.6. Vertical folding

Efficient implementation of multiplication in Rabbit g-function was described in subsection 3.2. The reason for crumbling wide operands in multiplication into more, but narrower 16-bit or 8-bit multiplications is, that some FPGAs have embedded blocks of fixed logic for frequently computed operations. Multiplication unit synthesized in programmable logic is far slower than hardwired multiplication unit of fixed width. Although the speed, size and complexity of FPGA devices is growing, even latest Xilinx Virtex 7 FPGAs have only 25 x 18-bit embedded multipliers [21]. Configurable ($9\times9, 18\times18, 24\times24$, or 36×36) multipliers are more common [1], but native support of wider (24×24-bit and more) multipliers is rare and available mostly in FPGA devices specialized for digital signal processing [2]. Usually, encryption/decryption engine is embedded into complex device and FPGA resources are shared. Likely, embedded FPGA resources are fully utilized by other systems and functions, so cryptographic algorithms tend to have minimal available resources. In Rabbit, scalability of efficient squaring and other functional blocks is important, because it delivers versatility and flexibility to the algorithm implementation. In vertical folding, data-path width and logic block interfaces are narrowed to process shorter operands at the expense of longer processing time. In case of vertically folded [15] 16-bit and 8-bit Rabbit implementations, all routing wires and functional blocks including counter update, next-state function, squaring in g-function and output transformation are implemented with 16-bit and 8-bit widths respectively. Again, additional control logic is required for buffering, partitioning, registering and synchronizing data.

For example in 16-bit version of the next-state function, rotation by 16-bits to the left is performed by reordering output from g-function.

4. Results

Distinct hardware implementations of Rabbit stream cipher algorithm were designed. In the first direct implementation, squaring is implemented using embedded multipliers with no optimization and critical path of algorithm was optimized by parallel implementation of counter update blocks, g-functions, next--state functions and output transformation. In the second implementation (ES), squaring in g-function is replaced with optimal formula 3.2. The third optimization comes with partitioned and pipelined (PP) Rabbit design, where the second implementation is divided into logical blocks, which operate independently with data synchronized between them. The next is area optimized, horizontally folded (HF8) implementation, where eight parallel blocks are merged to one shared pipeline. In vertically folded implementation, partitioned and pipelined design (with parallel blocks) is optimized by datapath and block width reduction to 16 (VF16) and eight (VF8) bits. The last two area optimizations are combined into horizontally and vertically folded architecture with one shared pipeline narrowed to eight bits wide block and datapaths (HF8VF8).

Hardware implementations were designed [20] on Xilinx FPGA Virtex 5 (XC5VLX50T) encompassed with 7,200 slices, 28,800 registers and 48 25×18-bit multipliers in DSP slices. In FPGA design flow, everything starts with hardware description language (HDL) code input. Programmatic code is analyzed, verified for formal correctness and synthesized by software design environment. During synthesis of verified HDL code, so-called post-synthesis netlist of used logic elements and their interconnections is generated. The next step in FPGA design flow is "fitting", when post-synthesis database of logic elements and their interconnections is mapped to available resources of the target device, with respect to designer-defined (or default) constraints. When place and route step is finished, post-fit netlist is analyzed for timing constraints before target FPGA device is configured and programmed. Results summarized in Table 4 were evaluated after the place and route step of the design flow, because frequency estimation of the designed algorithm is based on critical path in post-fit netlist, which leads to worst-case frequency of designed algorithm physically programmed into FPGA device. Fair comparison of hardware implemented algorithms is complicated, because many subjective and objective factors, such as functionality, target hardware platform, design optimization, or stage of the design flow [12] must be taken into account.

TABLE 1. Rabbit hardware implementation comparison.

Rabbit Design	Frequency (MHz)	Throughput (Gbps)	Logic slices (%)	DSP Blocks (%)
Direct	53.658	6.86	654 (9.08)	32 (66)
ES	62.996	8.06	788 (10.94)	24 (50)
PP	76.677	9.81	827 (11.48)	24 (50)
HF8	47.259	6.04	357 (4.95)	4 (8.33)
VF16	44.454	5.69	229 (3.18)	2 (4.17)
VF8	42.349	5.42	211 (2.93)	1 (2.08)
Estimated [9]		17.80		24 (50)

Our best performing implementation is parallelized and pipelined version of Rabbit, running on frequency of 76.677 MHz, processing 9.81 Gb per second, while utilizing 24 multipliers in dedicated DSP blocks. The difference between estimated and obtained throughput probably follows from different pipeline organization of compared designs. The pipeline of our design is divided into three stages (counter updates, g-function computations and state variable updates) and this pipeline realization looks not ideally efficient, when comparing complexity of computations involved in pipeline stages. On the other hand, the more stages are in the pipeline, the more control logic must be implemented to organize data in it. We have no information about FPGA implementation (and pipeline organization) of stream cipher Rabbit used for performance evaluation, other than availability of more than 24 dedicated multipliers with 2.4 ns latency was assumed and two-pipeline design was implemented for performance estimation [9]. Target FPGA device of implementation for performance estimation was also not specified. Future work will be focused on more efficient pipeline organization of the Rabbit hardware algorithm. Horizontally and vertically folded implementations are bit slower, but they utilize less logic blocks and less dedicated multipliers. It is visible, that narrowing datapath and functional block size slightly slows algorithm execution, but reduction of resource utilization is rapid. Versatility of algorithm implementation is important for designs realized in resource-constrained devices. The smallest implementation of stream cipher Rabbit runs on frequency of 42.349 MHz, covers 211 slices and utilizes only one multiplier in DSP block. Throughput of this implementation is 5.42 Gbps. Implementations of other eStream hardware profile candidates on the same or similar FPGA devices are not evaluated. Performance of eStream phase 3 ciphers Trivium [11], Grain [14], Salsa20 [8] and Mickey-128 [7] on Xilinx Spartan 3 FPGA device, optimized for maximum throughput to area ratio was evaluated in [13]. The highest throughput had highly-parallelized Trivium implementation (12.16 Gbps on 388 CLB slices), followed by Grain (2.48 Gbps on 356 CLBs), Salsa20

(1.20 Gbps per 1615 CLBs) and Mickey-128 with maximum throughput of 0.16 Gbps, implemented on 261 CLB slices.

REFERENCES

[1] ALTERA: *Stratix IV device family overview*, 2011, `http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf`.

[2] ALTERA: *Stratix V device family overview*, 2011, `http://www.altera.com/literature/hb/stratix-v/stx5_51001.pdf`.

[3] CRYPTICO A/S: *Algebraic analysis of Rabbit*, 2003, http://www.cryptico.com (White paper)

[4] CRYPTICO A/S: *Analysis of the key setup function in Rabbit*, 2003, http://www.cryptico.com (White paper).

[5] CRYPTICO A/S: *Differential properties of the g-function*, 2003, http://www.cryptico.com (White paper).

[6] CRYPTICO A/S: *Security analysis of the IV-setup for Rabbit*, 2003, http://www.cryptico.com (White paper).

[7] BABBAGE, S.—DODD, M.: *The stream cipher MICKEY-128 (version 1)*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/016, 2005, `http://www.ecrypt.eu.org/stream`.

[8] BERNSTEIN, D.: *Salsa20*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/025, 2005, `http://www.ecrypt.eu.org/stream`.

[9] BOESGAARD, M.—VESTERAGER, M.—CHRISTENSEN, T.—ZENNER, E.: *The stream cipher Rabbit*, eSTREAM, 2006.

[10] BOESGAARD, M.—VESTERAGER, M.—PEDERSEN, T.—CHRISTIANSEN, J.—SCAVENIUS, O.: *Rabbit: A new high-performance stream cipher*, in: Proc. Fast Software Encryption—FSE '03, 10th International Workshop (T. Johansson, ed.), Lund, Sweden, 2003, Lect. Notes in Comput. Sci., Vol. 2887, Springer-Verlag, Berlin, 2003, pp. 307–329.

[11] DE CANNIERE, C.—PRENEEL, B.: *Trivium—A stream cipher construction inspired by block cipher design principles*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005, `http://www.ecrypt.eu.org/stream`.

[12] GAJ, K.—KAPS, J.—AMIRINENI, V.—ROGAWSKI, M.—HOMSIRIKAMOL, E.—BREWSTER, B. Y.: *ATHENa—Automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs*, in: Proc. Field Programmable Logic and Applications—FPL '10, Internat. Conf., Milano, Italy, 2010 (F. Ferrandi et al., eds.), IEEE Comput. Soc., 2010, pp. 414–421.

[13] GAJ, K.—SOUTHERN, G.—BACHIMANCHI, R.: *Comparison of hardware performance of selected phase II eSTREAM candidates*, eSTREAM, ECRYPT Stream Cipher Project, Report 2007/026, 2007, `http://www.ecrypt.eu.org/stream`.

[14] HELL, M.—JOHANSSON, T.—MEIER, W.: *Grain—A stream cipher for constrained environments*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005, `http://www.ecrypt.eu.org/stream`.

[15] HOMSIRIKAMOL, E.—ROGAWSKI, M.—GAJ, K.: *Comparing hardware performance of round 3 SHA-3 candidates using multiple hardware architectures in Xilinx and Altera FPGAs*, in: Proc. ECRYPT II Hash Workshop, Tallinn, Estonia, 2011.

[16] MENEZES, A. J.—OORSCHOT, P. C. VAN—VANSTONE, S. A.: *The Handbook of Applied Cryptography*. CRC Press, Singapore, 1996.

[17] RAUBER, T.—RUNGER, G.: *Parallel Programming*. Springer-Verlag, Berlin, 2007.

[18] SCAVENIUS, O.—BOESGAARD, M.—VESTERAGER, M.—CHRISTIANSEN, J.–
–RIJMEN, V.: *Periodic properties of counter assisted stream cipher*, in: CT-RSA '04, Lecture Notes in Comput. Sci., Vol. 2964, Springer-Verlag, Berlin, 2004, pp. 39–53.

[19] SIMMONS, J. G. (ED.): *Contemporary Cryptology*. IEEE Press, Piscataway, New Jersey, 1992.

[20] XILINX: *Virtex 5 family overview*, `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, 2009.

[21] XILINX: *Xilinx 7 series overview*, `http://www.xilinx.com/support/documentation/data_sheets/ds180_7series_overview.pdf`, 2011.

Received September 22, 2011

Mathematical Institute
Slovak Academy of Sciences
Štefánikova 49
SK–814 73 Bratislava
SLOVAKIA

E-mail: tomecek@mat.savba.sk