Hardware/software codesign for stream ciphers

Patrick Schaumont[1] and Ingrid Verbauwhede[2]

[1] ECE Department, Virginia Polytechnic Institute and State University, USA,
schaum@vt.edu
[2] ESAT/COSIC, Katholieke Universiteit Leuven, Belgium,
ingrid.verbauwhede@esat.kuleuven.be

Abstract. The ESTREAM project has identified two profiles for stream ciphers: a hardware profile and a software profile. The software profile is directly applicable to many computer systems. The hardware profile, on the other hand, does not reflect a complete system design, but instead represents a stand-alone component. In this paper we consider the integration of hardware stream ciphers in software systems for the case of Trivium, Salsa20 and Phelix. We review the different categories of hardware/software interfaces and then present performance- and implementation results for several stream-cipher configurations. Our conclusion is that the wide variety of possible hardware/software interfaces has substantial impact on the resulting performance of the design. We therefore conclude that the hardware profile should consider not only the intrinsic performance and area cost of a stream cipher, but also the required input-output bandwidth for that given encryption performance.

1 Introduction

It is well known that hardware is a generous provider of parallellism. The *hall of fame* for many block cipher implementations in hardware often states throughputs in Gb/s. For example, an AES-128 implementation that runs at 100MHz and requires 11 clock cycles per encryption round has a throughput of 1.16 Gb/s (128 bit /11 * 100MHz). This number reflects the ability of the hardware component to encrypt bits.

In this paper we consider the step beyond the raw processing power of hardware encryption units. How can we harness that encryption performance in software? In this case, we are interested in building flexible, hardware-accelerated solutions.

It is easy to see that communication bandwidth is a critical design factor. For the example of the 1.16Gb/s block cipher stated above, if we assume that each encryption round requires the commmunication of a 128-bit key, 128-bit plaintext and 128-bit cryptext, then we need an input/output bandwidth of about 3.5Gb/s. Dedicated communication hardware (e.g. direct-memory-access chips on fixed-latency buses) may achieve this bandwidth. In many cases however, this bandwidth needs to be provided directly through the software. The bandwidth of 3.5Gb/s (or equivalently 109 million words per second) indeed is outside the capability of most embedded processors.

This shows that the most optimal hardware design (in terms of performance) may not always be the most optimal solution at system level. We will use the term *scenario* to indicate the manner in which a hardware component is integrated in a system. In this paper, we consider some of the possible scenario's for stream ciphers, and apply this to several ESTREAM candidates and several hardware/software architectures.

We consider Trivium [1], Salsa20 [2] and Phelix [3] as candidates for hardware acceleration, and consider the hardware/software interfaces provided with StrongARM and Microblaze processors. The results for StrongARM are estimates obtained using (co-)simulation [4] [5]. The results for Microblaze have been implemented on an FPGA board and measured using a hardware timer. The Trivium codesign was first presented at an ECRYPT summer school in 2006. The Salsa20 and Phelix codesigns are the result of student design projects of a senior course in codesign, organized at Virginia Tech in the Fall of 2006. We can hardly claim optimality for any of the results. Rather, we wish to point out the importance of the *scenario* next to that of the profile of a stream cipher.

2 The scenario of a stream cipher

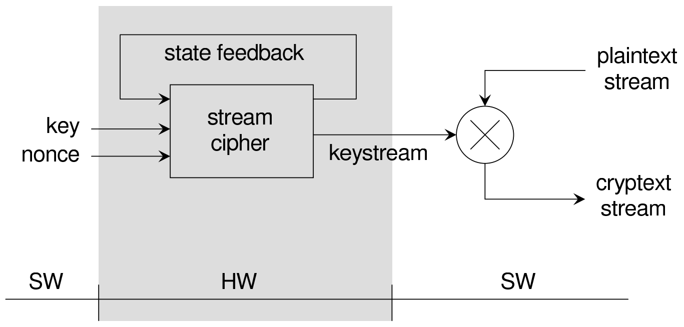


Fig. 1. Example scenario for a stream cipher

In Figure 1, we have modeled a stream cipher as an abstract, iterated block. The cipher is initialized by means of a key and a nonce. After initialization it produces an infinite stream of key values. For each key value, the algorithm performs a series of iterations: the internal state of the cipher is updated based on the previous state values and the key and nonce inputs. For encryption, the keystream is xored with a plaintext stream to produce the cryptext stream.

The scenario of a stream cipher indicates the manner in which the cipher's components or subcomponents are mapped into hardware and software. Figure 1 provides an example of such a mapping. The stream cipher kernel and the iterated state variables are mapped into hardware. Software provides the initial key and the nonce. The output keystream is returned to the software, which also performs encryption. During steady-state encryption, the bandwidth for hardware-software communication is determined by the keystream.

The ESTREAM project defines hardware or software profiles for the implementation of stream ciphers. For a pure hardware or pure software profile, the optimality criteria for hardware and software are well known. In hardware, one wants minimal area (gates) with maximal performance on a given technology. In software, one wants minimal footprint (memory) with maximal performance for a given processor. Thus, by defining the technology for hardware profiles, or the processor for software profiles, one can compare one implementation to the next.

In a scenario that mixes hardware and software, such comparisons are harder. Each scenario in fact can claim properties that are not available in another one, and it is vital to consider the system-level picture of the design. We discuss several of those system-level trade-offs below.

- One may trade-off communication bandwidth for area. It is often desireable to create the keystream in blocks equal to the wordlength of the processor (e.g. 32 bits). This will make optimum use of the available hardware-software bandwidth in the system. On the other hand, if the stream cipher generates less bits per encryption round, multiple stream cipher rounds will be needed for each keystream word. This means that either the hardware clock will have to be increased, or else that multiple parallel iterations need to be created in hardware.
- One may trade-off flexibility for performance. Software can provide fine-granular control on the internal configuration and behavior of a stream cipher. However, performing such control will require additional hardware-software communication which reduces the throughput of the design.
- One may trade-off re-entrancy for performance. A stream cipher can be implemented as a *stateless* hardware design or a *stateful* hardware design. A stateless hardware design will implement the state feedback loop in software. This increases the amount of hardware-software communication significantly, but on the other hand it enables concurrent sharing of stream cipher hardware among different software processes.

For hardware stream cipher profiles, considerations such as the above ones are vital. They imply that classic hardware optimization criteria alone cannot address all of the issues at system level.

In the following, we will briefly review different types of hardware-software interfaces in SoC context, and give several examples in terms of FPGA technology. Next, we will discuss three cases of hardware/software codesign for stream ciphers: one for Trivium, one for Salsa20, and one for Phelix.

3 Hardware/software interfaces

Figure 2 illustrates the three commonly available hardware/software interfaces. Each of these interfaces is distinguished by the proximity of the custom hardware (coprocessor) to the processor pipeline.

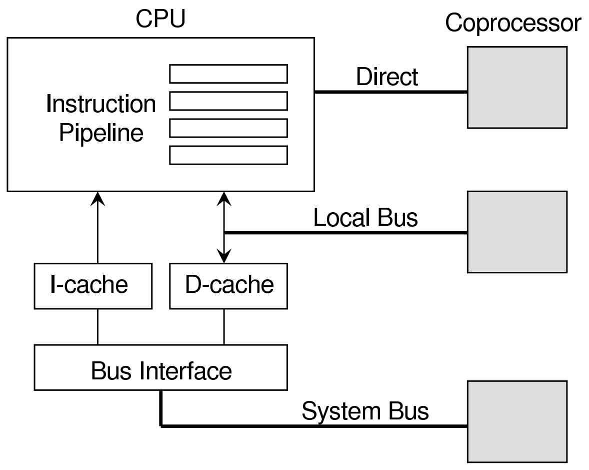
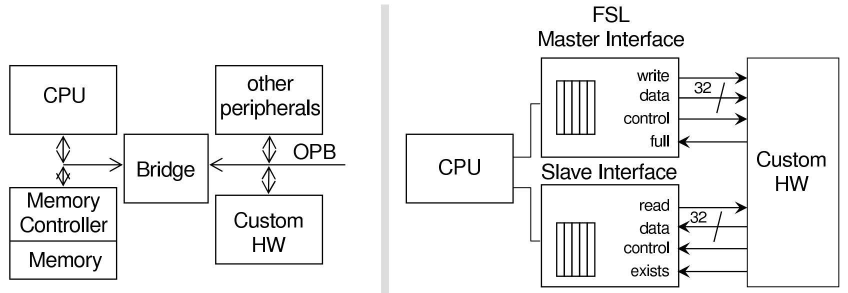


Fig. 2. Hardware/software interfaces on a typical embedded core

- A **Direct-Connection** provides direct access to the processor pipeline. This results in a low-latency, high-bandwidth bus. A CPU will have specific instructions to support this interface, since there is no address space at this level in the processor. These dedicated instructions may introduce (data-) dependencies on the processor pipeline and cause pipeline stalls. The degree of coupling between the processor pipeline and the coprocessor hardware must be evaluated carefully.
- A **Local Bus** provides a dedicated, low-latency and fixed-latency link to the processor. Local busses are addressable. Processors caches are attached to the local bus. Some processors allow dedicated shared memories (scratchpads) to be attached to the local bus, which in turn connect to coprocessor hardware.
- A **Bus Interface** provides a general-purpose link to connect processor hardware. Modern bus interfaces support sophisticated transactions to adapt for the heterogeneity of the components on the system bus. On the flip-side, general-purposes system busses do not have a low latency nor a fixed latency.

Direct-connection busses and local-busses are processor specific. In contrast, bus-interfaces are generic. We will briefly discuss the example of a direct-connection bus (Fast Simplex Link), followed by the example of a general-purpose bus (On-chip Peripheral Bus). Figure 3 illustrates both interfaces, along with their system-level cosimulation model in GEZEL [5].

The OPB interface is a traditional memory-mapped interface for peripheral components. The OPB bus is a shared, variable latency bus which is part of IBM's CoreConnect specification. It is also used to interconnect soft-core and hard-core processors in a Xilinx FPGA. The hardware side of an OPB interface consists of a decoder for a memory-read or memory-write cycle on a selected address in the memory range mapped to the OPB. The decoded memory cy-



Software (write-side only)

```
int main() {
  volatile int *v = (int *) 0x41000000;
  ..
  *v = 0x5;
  ..
}
```

```
int main() {
  ..
  putfsl(5 /* value */, 0 /* device */);
  ..
}
```

Hardware (read-side only)

```
ipblock opb1(out data : ns(32)) {
   iptype "armsystemsource";
   ipparm "core=arm1";
   ipparm "address=0x80000000";
}
```

```
ipblock fsl1(out data   : ns(32);
             out exists : ns(1);
             in  read   : ns(1)) {
   iptype "armfslslave";
   ipparm "core=arm1";
   ipparm "write=0x80000000";
}
```

Fig. 3. Hardware/Software Interface: OPB (left) and FSL (right)

cle is translated to a read-from or a write-into a register in the coprocessor. A memory-mapped interface is an easy and popular interface technique, in particular because it works with standard C on any core that has a system bus. The drawback of this interface is the low-speed connection between hardware and software. Even on an embedded core, a simple round-trip communication between software and hardware can run into several tens of CPU clock cycles.

The FSL interface is a point-to-point communications link from hardware to software. The hardware side of this interface includes a full-handshake protocol and can transfer a token of 32 data-bits and 1 control-bit per handshake. An internal queue can buffer up to 8 tokens and provides additional uncoupling between hardware and software. The CPU has dedicated instructions to write into and read from this interface, and these instructions can stall the CPU pipeline if this would be required by the handshake protocol. The FSL is defined specifically for the MicroBlaze processor (a soft-core FPGA processor by Xilinx).

4 Example Scenario's

We will now discuss three example scenario's for stream ciphers, starting with a Trivium codesign, followed by Salsa20 and Phelix.

4.1 Trivium with a custom-instruction-set processor

Our first scenario is to attach a hardware Trivium stream cipher as a custom-instruction onto a processor. Custom-instruction-set extensions are supported with application-specific instruction-set processors (ASIP). A custom instruction has a limited set of inputs and outputs, in order to limit the number of read- and write-ports on a processor's register file. For the Trivium design, we considered the use of 2-input, 2-output custom instructions (*OP2x2*) and 3-input, single-output instructions (*OP3x1*).

The Trivium kernel is, as suggested by the authors [1], very easy to parallelize. The design in Figure 4 uses a 64-times unrolled kernel. This unrolled kernel reads 288 state bits from the state register, produces 64 keystream bits, and generates 288 next-state bits. The 64 keystream bits are captured by an OP2x2 custom instruction.

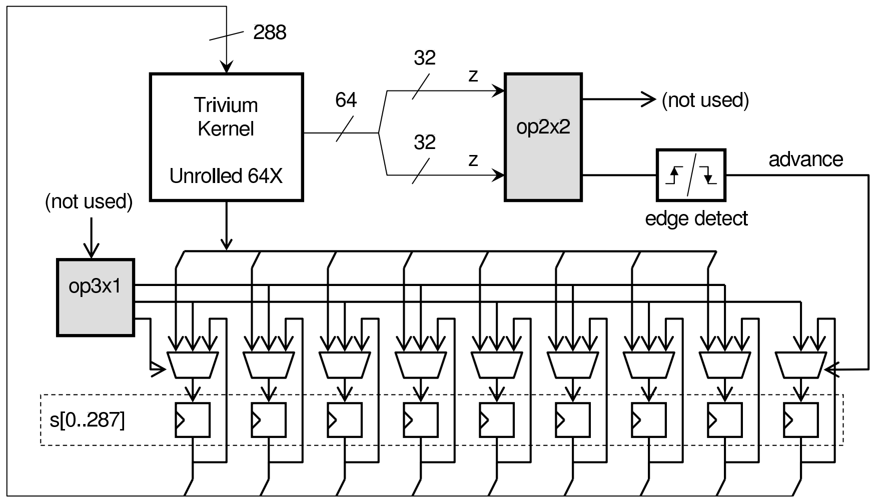


Fig. 4. Hardware Trivium kernel attached to a custom instruction-set interface

In order to ensure proper synchronization between the Trivium hardware and the software that accepts the keystream, the state update of the Trivium hardware is under control of software. A single output bit of the OP2x2 instruction is edge-detected. An upgoing or downgoing transition will enable a single Trivium state update, which produces the 64 next keystream bits. The Trivium design also uses an OP3x1 instruction to initialize the stream cipher. The OP3x1 instruction passes three fields to the Trivium hardware. Two are used to initialize Trivium state (64 bits per OP3x1 execution), and one field is used to index the proper register within the Trivium state variable.

We now examine the steady-state behavior of this design. In steady-state, the C code accepts a keystream coming from the Trivium hardware, and controls the update of the stream cipher's state variable.

```
unsigned z1, z2, stream[4*N];
for (i=0; i<N; i++) {
  OP2x2_1(z1, z2, 1, 0);  // read 64 keybits, toggle control high
  stream[4*i]   = z1;     // store low 32 keybits
  stream[4*i+1] = z2;     // store high 32 keybits
  OP2x2_1(z1, z2, 0, 0);  // read 64 keybits, toggle control low
  stream[4*i+2] = z1;     // store low 32 keybits
  stream[4*i+3] = z2;     // store high 32 keybits
}
```

The OP2x2 instruction repeatedly updates the Trivium state variable, while capturing the output from the Trivium kernel. The keystream is stored in data memory. Table 1 shows the detailed analysis of one iteration of the above loop during execution from cycle 11923 up to cycle 11934. The `umullnv` instructions are unused opcodes of the StrongARM which are reused by our cosimulator to implement OP2x2 instructions. One loop iteration completes in 11 clock cycles, and apart from the branch instruction at the end of the loop there are no pipeline control hazards. Individual instructions have a latency of 5 clock cycles, corresponding to the 5 pipeline stages of the StrongARM. Instructions with 6 clock cycles include a data hazard (one additional cycle of delay).

Table 1. Loop analysis for Trivium software driver

Cycle	PC	Latency	Instruction
11923	8274	5	`umullnv r2, ip, r0, r3`
11924	8278	6	`str r2, [r1, #-2048]`
11925	827c	6	`str ip, [r1, #-2044]`
11927	8280	5	`umullnv r7, r6, r3, r3`
11928	8284	5	`subs r5, r5, #1`
11929	8288	5	`str r7, [r1, #-2040]`
11930	828c	5	`str r6, [r1, #-2036]`
11931	8290	5	`add r1, r1, #16`
11932	8294	2	`bpl 0x8274`
11933	8298	1	(stall cycle)
11934	8274	5	(next iteration)

During these 11 cycles per iteration, we can collect 128 bits of keystream in the processor's data memory. This corresponds to 11.63 bits per clock cycle. Compared to the orginal 64-times unrolled Trivium kernel (which generates 64 keystream bits per clock cycle), this is 5.5 times slower. This overhead is a direct consequence of accepting the keystream with a sequential processor. Note also that the compiler is run with full optimization, and that the assembly code has good quality.

This example illustrates that even an optimized hardware/software interface can cause a significant slowdown of a stream cipher.

4.2 Salsa20 Co-design

Our next example illustrates the effect of stateless stream cipher design. The Salsa20 design [2] generates a keystream using a 64-byte hash function. Figure 5 illustrates how the hash is driven from a Nonce and a key. The Nonce and key are combined and expanded into a 64-byte Salsa20 hash input. The hash input is transformed in 10 iterations of row round and column round functions, which consist of 32-bit additions, xor and rotations. After 10 iterations, the original Salsa20 hash input is added to the hashed output to create 64 bytes of the key stream. The next 64 bytes of the key are found by incrementing the Nonce and repeating the process.

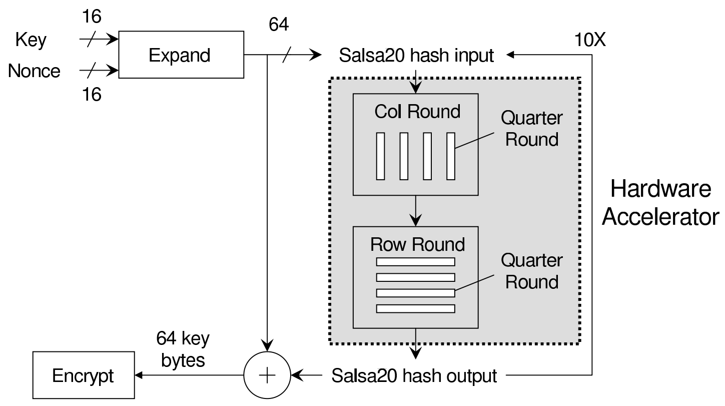


Fig. 5. A Salsa20 hardware accelerator

The Salsa20 design was studied in an undergraduate course at Virginia Tech called 'Introduction to Codesign'. Three groups of two students implemented the Salsa20 design as a coprocessor. Through profiling of the reference C code, all three student groups converged to a design that implements the shaded area in Figure 5 as a hardware accelerator. By profiling C code, it is indeed easy to show that the bulk of the computations during Salsa20 encryption will be performed inside of the Salsa20 hash function. *However, profiling of C code does not reveal the cost of communicating state variables.* As a result, all three student groups, assuming that doing the computations efficiently in hardware is sufficient to build a powerful hardware accelerator, implemented a stateless stream-cipher design. For each iteration of the hash, 64 bytes must be moved to and from the coprocessor.

The implementation results for these teams are illustrated in Table 2, and indicate that the speedup obtained for these stateless stream ciphers is limited. In hindsight, this bottleneck can be easily removed by including the hash state, and possibly also the nonce and key expansion, into the coprocessor.

Table 2. Coprocessor Design Results

Team	**Algorithm**	**C code**	**GEZEL code**	**Coprocessor area**	**Interface**	**Speedup**	**Overall speedup**
		lines	*lines*	*Spartan3e slices*		*HW vs SW*	*HW+SW vs SW*
A	**Salsa20**	220	533	1381	FSL	-	2.54
B	**Salsa20**	178	641	1107	OPB	-	1.34
C	**Salsa20**	437	281	554	OPB	24	2
D	**Phelix**	607	357	568	OPB	81	1.2
E	**Phelix**	408	554	4301	OPB	-	*219*

Table 2 lists linecounts for the initial C algorithm and the resulting coprocessor in GEZEL. Next, the table mentions the coprocessor area in terms of slices of the underlying Spartan3E FPGA technology. As a measure of comparison, the coprocessors are attached to a 32-bit Microblaze of approximately 1000 slices. Thus, a coprocessor area of 870 slices implies that the design size nearly doubles. The last two columns of Table 2 document the resulting performance of the design after introducing coprocessor hardware. Column 7 considers only the hardware acceleration and ignores the hardware-software communication overhead. Column 8 includes this communication overhead. While the design speedup is often dramatic considering only hardware acceleration, the resulting speedup after integration can be much lower. This is a typical effect of aggressive hardware/software partitioning with insufficient consideration of the resulting communications bottleneck. In all of the cases, the main processor was in charge of feeding data into the coprocessor and retrieving it.

One particular speedup figure (219 for the case of Phelix) jumps out of Table 2. However, that number turned out to be wishful thinking, as will be discussed next.

4.3 Phelix Co-design

Our final example illustrates the effect of overly optimistic parallellization of C code when creating hardware. Like Salsa20, Phelix generates a keystream using a series of additions, xor, rotations. The keystream is created by repeated transformations on 5 state words, as shown in Figure 6. Each transformation includes two half-block operations consisting of 10 rounds (rotation/addition). Each half-block uses one of 8 key-words created through a key-expand function. The key stream is created by xoring two half-block outputs. Control and initialization in Phelix are slightly more complex compared to Salsa20.

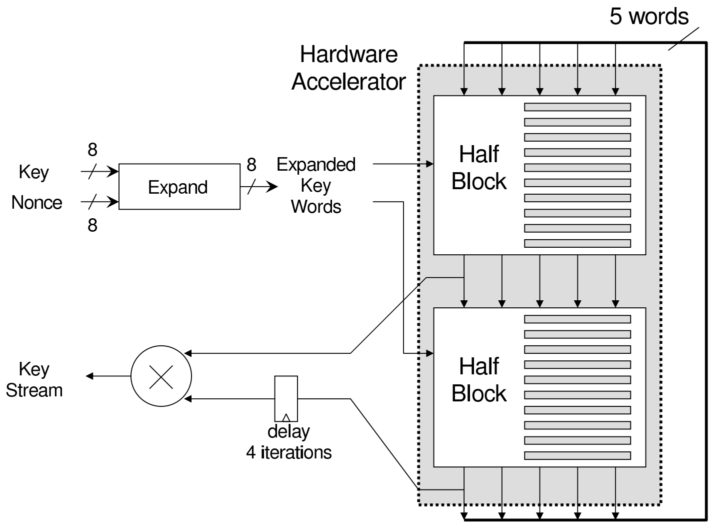


Fig. 6. A Phelix hardware accelerator

Also Phelix was studied in the codesign course at Virginia Tech. Two student teams created a coprocessor for Phelix, but both took a very different approach. A first team worked in an incremental fashion, starting with an accelerator for each half-block. The second team took a much more aggressive approach: they unrolled Phelix to implement up to 8 full blocks in parallel in a single clock cycle. Both teams converged to an implementation in which a full-block iteration is implemented with a hardware accelerator, as shown in Figure 6. This mapping allows the accelerator to be implemented as a combinatorial function, and leaves as much as possible of the control complexity in software. The downside of this mapping is that this introduces a communication bottleneck between software and hardware (which the second team was hoping to avoid by aggressive parallellization).

The team with a more aggressive approach to parallellization is listed as team E in Table 2. They concluded at first that a significant speedup could be expected since the cycle count of an 8-fold unrolled Phelix version is greatly reduced. After technology mapping onto a Spartan-3E step-5 however, the resulting design turned out to be too large and too slow. As illustrated in Table 3, subsequent design iterations created a smaller and faster coprocessor. This trade-off between design parallellism and achievable clock frequency is well known to a hardware designer. On the other hand, the trade-off is unfamiliar to a software designer who focuses on reducing the cycle count only.

Table 3. Unrolling Phelix in hardware (in Spartan 3E step 5)

Version	**Area** *(slices)*	**Clock** *(MHz)*
8x Full Block	4301	7.7
Full Block	1190	48.6
Half Block	459	76

5 Conclusions

The ESTREAM project has identified *profiles* to describe the technological targets for stream cipher development. In this paper, we conclude that the currently defined profiles are coarse definitions at best. We think that especially the hardware profile is unable to predict the performance of a stream cipher in the context of an actual application.

Our alternative was to extend the hardware profile into a *scenario*, which clarifies how the interfaces of a hardware stream cipher are attached to a system. The examples of Trivium, Salsa20, and Phelix demonstrate the sensitivity of system performance to the scenario.

6 Acknowledgements

The results presented in this paper for Salsa20 and Phelix were obtained by the students of the ECE4984 'Introduction to Codesign' class at Virginia Tech. Hence we acknowledge the diligence and commitment of Justin Chong, Mike Bertosh, Thomas Connors, Amy Anderson, George Eichinger III, Tim Doyle, Mike Gora, Bill Klinefelter, Jesse Manning, Steve Ressler, John Morris, Syed Haider, Senwen Kan, Anupam Srivastava, Sumit Ahuja, Mainak Banga, Ian Thornton and Joe Amato.

References

1. C. De Canniere, B. Preneel, "Trivium Specifications," available from ESTREAM (http://www.ecrypt.eu.org/stream/triviump2.html).
2. D. J. Bernstein, "Salsa20 specification," available from ESTREAM (http://www.ecrypt.eu.org/stream/salsa20p2.html).
3. D. Whiting, B. Schneier, S. Lucks, F. Muller, "Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive," available from ESTREAM (http://www.ecrypt.eu.org/stream/phelixp2.html).
4. W. Qin, S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation, Proceedings of 2003 Design Automation and Test in Europe Conference (DATE 03), Mar, 2003, pp.556-561.
5. The GEZEL homepage, online at http://rijndael.ece.vt.edu/gezel2.