Cryptography with Asynchronous Logic Automata

Peter Schmidt-Nielsen, Kailiang Chen, Jonathan Bachrach, Scott Greenwald, Forrest Green, and Neil Gershenfeld

MIT Center for Bits and Atoms, Cambridge, MA

Abstract. We introduce the use of asynchronous logic automata (ALA) for cryptography. ALA aligns the descriptions of hardware and software for portability, programmability, and scalability. An implementation of the A5/1 stream cipher is provided as a design example in a concise hardware description language, Snap, and we discuss a power- and timing-balanced cell design.

Keywords: asynchronous, cellular, cryptography, stream cipher, power balance.

1 Introduction

Cellular architectures have long been attractive for cryptography [1,2,3,4,5]. Cellular automata, by discretizing time, space, and state, with cell transitions defined by indexing a rule table with a bit string representing states in a local neighborhood, offer bit-level parallelism with simple local dynamics.

These have, however, had little impact on technological practice. Field Programmable Gate Arrays are now routinely used to implement high-performance cryptosystems [6]. CAs, in comparison, have lacked both hardware platforms and design workflows to implement cryptographic algorithms.

Use of FPGAs does conventionally assume synchronously clocked logic. Self-timed cryptographic circuits have been developed [7,8]; these can have benefits for speed, power consumption, and robustness against side-channel attacks, but have typically been developed for special-purpose applications rather than as a general-purpose architecture.

FPGAs also rely on a fitter to map a design onto a gate array, which can require significant extra effort in logic synthesis, and has led to the introduction of increasingly large functional modules on the die. Because chip edges differ from their interiors, there is not a straightforward route to divide a single design across multiple gate arrays.

We present an alternative approach to implementing cryptosystems that lies at the intersection of cellular logic, gate arrays, and asynchronous circuits. It is based on Asynchronous Logic Automata (ALA), a model of computation that seeks to align the descriptions of hardware and software. In the following sections we introduce ALA, illustrate its use with an implementation of the A5/1 stream cipher used in GSM, and discuss the design and balancing of circuits.

2 ALA

Software can represent physical quantities, but is not typically itself written with physical units. While this common abstraction from hardware is intended to ease programming, it presents challenging optimizations and many changes in representation in going from the description of a program to its execution, it introduces increasingly severe bottlenecks in physically emulating the virtual interconnect and memory model, and requires additional management of execution threads and interprocessor communication.

Asynchronous Logic Automata (ALA) is instead based a description of computation that is maintained from software to hardware. Programs can be hierarchical and modular, but the underlying representation is maintained throughout, much as the geometry of a map does not change in zooming from city to state to country.

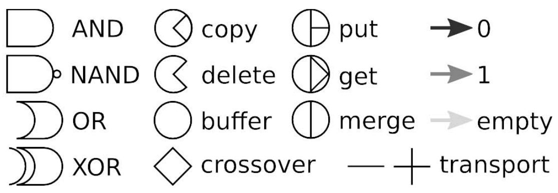

Fig. 1. ALA cells

ALA is based on cells passing tokens on a lattice; it is best understood not as a new model but as the intersection of the scaling end-points of many familiar ones [9]. The dimensionality of the lattice reflects the underlying hardware dimension, here taken to be 2D. Cells are locally connected by directed links that can either be empty or contain a 0 or 1 logical token. When a cell has valid tokens on its input and no tokens on its output, it pulls the former and pushes the latter. The cell types are shown in Figure 1; there are cells for performing logic, for creating and destroying tokens, for switching and merging them, and for performing blocking and non-blocking transport.

Figure 2 shows an AND cell firing, and Figure 3 shows the steps in single-bit addition. The implementation of pipelining is implicit in the asynchronous data dependencies. In ALA, the distance that information travels is proportional to the time that it takes, the number of operations that can be performed, and the amount that can be stored; these are all coupled as they are through the underlying physics.

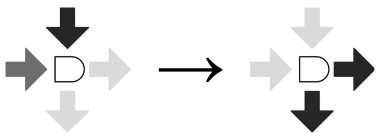

Fig. 2. Example of an AND ALA cell update. Dark arrows denote 0 tokens, light arrows 1 tokens, and grey arrows empty links.

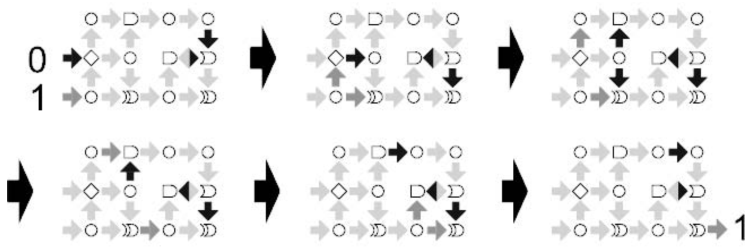

Fig. 3. Example ALA computation: one-bit addition

3 Design

Because ALA programs are spatial, their development shares elements of software, circuit, and mechanical design. One approach that has been used is a visual dataflow programming environment [10]. This has the feature that there is a one-to-one mapping from the high-level description to its implementation in ALA cells; there is no need for a scheduler or execution environment.

An alternative approach is a textual hardware description language, Snap [10]. Snap was written as a module for the Scala programming language. It is based on hierarchically assembling blocks of ALA modules, and linking them with smart glue connections. This Snap code:

```
hc(">->", ">->/1", ">->")
```

produces the simple circuit shown in Figure 4. The function `hc` horizontally concatenates a list of modules, lining up their corresponding inputs and outputs. ALA cells are referenced by strings formatted with three components: the input directions, a specifier of the type of gate to place, and the output directions. Thus `">->"` specifies a wire cell (`"-"`) with an input coming in from the west (`">"`) and outputting to the east (also `">"`). Optionally, gate strings may be followed by a token to be preloaded (e.g., `"/1"`).

Here is a more complicated example, which defines a parametric LFSR specified by a length and set of tap bits:

```
def LFSR(length: Int, taps: Seq[Int]) = hc(
       vc("^->", "<-^"),
       hrep(length, i =>
              if (taps contains i) vc(">->/1", "<vX<")
              else vc(">->/1", "<-<")),
       vc(">-v", "v-<>"))
```

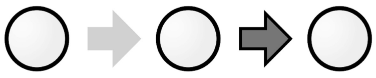

Fig. 4. `Snap example`

The function `vc` is the vertical version of `hc` which vertically stacks its arguments. The function `hrep` takes a number of repetitions n, and a function f that takes an integer and returns a module, and `hc`s the results of applying f to each integer from 0 to $n-1$. `X` corresponds to an XOR gate when used in a string. Using these functions, an LFSR is created by horizontally connecting three modules with `hc`:

1. On the left, `vc("^->", "<-^")`, which corresponds to the U-turn at the left of figure 5.
2. In the middle of the sandwich, `hrep`, which is horizontally concatenating a series of vertical slices, either `vc(">->/1", "<-<")` where there is not a tap bit, or `vc(">->/1", "<vX<")` where there is.[1]
3. On the right, `vc(">-v", "v-<>")`, which corresponds to the U-turn at the right of figure 5. Note that this also fans out bits to the east – forming the output of the whole LFSR.

Given this function, we can create a length 4 LFSR with taps at bit positions 3 and 4 with the code `LFSR(4, List(3, 4))`, shown in Figure 5.

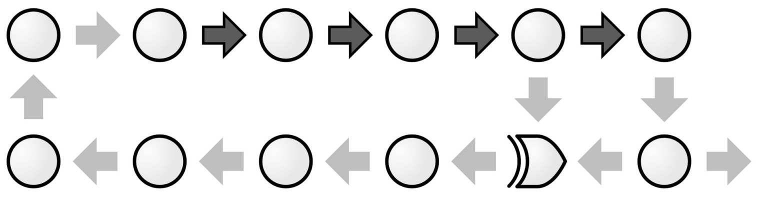

Fig. 5. `LFSR(4, List(3, 4))`

With a parametric LFSR, a shrinking generator can be defined:

```
val shrinking_generator = hcg(
      vc( LFSR(5, List(3, 5)),
          LFSR(6, List(5, 6)) ),
      glue((0, 0), (1, 1)),
      vc(">-v", "v>D>"))
```

Here we are using the smart-glue function `hcg` to connect a stack of two LFSRs with a single delete gate that uses the bits from one LFSR to selectively delete bits from the other LFSR. Glue is specified by a list of ordered pairs of outputs and inputs to connect. In this case we simply want to connect output 0 (the output of the lower LFSR) to input 0 (the data channel of the delete gate), and output 1 (the output of the upper LFSR) to input 1 (the control channel of

[1] Note that we specified two tap bits in the argument to the function `LFSR` in figure 5, but only got one XOR gate. This is because `hrep(4, f)` calls `f` from 0 to 3, and thus the final tap bit 4 specified is actually both ignored and assumed, and thus the code `LFSR(4, List(3))` would produce the same result.

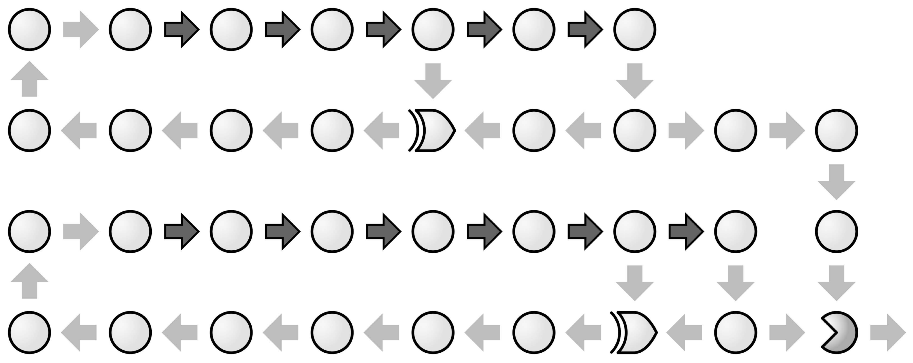

Fig. 6. Shrinking generator

the delete gate), and thus we write `glue((0, 0), (1, 1))`. As a shortcut, `hcg` allows one to omit a glue specification to imply that corresponding inputs and outputs should be connected up. The resultant circuit can be seen in figure 6.

Using just these primitive components, here is the complete code for specifying the A5/1 cipher used in GSM cellphone encryption (Figure 7:

```
// An LFSR that also outputs the bits at ‘siphon_point’.
def siphoned_LFSR(length: Int, siphon_point: Int, taps: Seq[Int]) = hcg(
       vc("^->", "<-^"),
       hrep(length, i =>
              vc( if (i == siphon_point) hc("^->") else noop,
              if (taps contains i) vc(">->/r", "<vX<")
              else vc(">->/r", "<-<"))
              ),
       vc(">-v", "v-<>"))

val A51_LFSRs = List(
       siphoned_LFSR(19, 8, List(13, 16, 17, 18)),
       siphoned_LFSR(22, 10, List(20, 21)),
       siphoned_LFSR(23, 10, List(7, 20, 21, 22)) )

// Takes three inputs, and outputs the majority bit.
val majority_voter = hcg(
       vc(">->", ">->", ">->"),
       glue((0, 0), (0, 2), (1, 1), (1, 4), (2, 3), (2, 5)),
       vc(">-v", ">v&>", ">-v", ">v&>", ">-v", ">v&>"),
       vc(">-v", ">v|>"),
       vc(">-v", ">v|>"))

// Takes three inputs, and for each input outputs
// if the input agrees with the majority.
val agreement = hcg(
       vc(">->", ">->", ">->"),
       glue((0, 0), (1, 1), (2, 2), (0, 3), (1, 4), (2, 5)),
        majority_voter,
        glue((0, 0), (0, 2), (0, 4), (1, 1), (2, 3), (3, 5)),
        vc(">-v", ">vX>",">-v", ">vX>",">-v", ">vX>"))

// Takes the output of an LFSR, and a control line,
// and clocks the LFSR only on a 0-bit from the control line.
val LFSR_duplicator = hcg(
        vc("<-v","v->", ">->", ">->"),
        glue((0, 0), (2, 1), (1, 2), (2, 3), (3, 4)),
        vc("<-<", ">-v/0", "v>C>", ">-v/0", "v>C>"))

// A5/1 cipher, by gluing together all the sub-components.
// The three LFSRs are fed into the agreement module,
// which is in turn fed back into the LFSR_duplicators,
// to only clock those LFSRs that agree with the majority.
// The three LFSRs are XORed together to form the output bit.
val A51 = hcg(
        vrep(3, i => hcg(A51_LFSRs(i), LFSR_duplicator) ),
        glue((1, 0), (4, 1), (7, 2)),
        agreement,
        glue((0, 0), (1, 2), (2, 4), (4, 1), (6, 3), (8, 5)),
        vc("^-<", ">-^", "^-<", ">-^", "^-<", ">-^"),
        vc(">-v", ">vX>"),
        vc(">-v", ">vX>"))
```

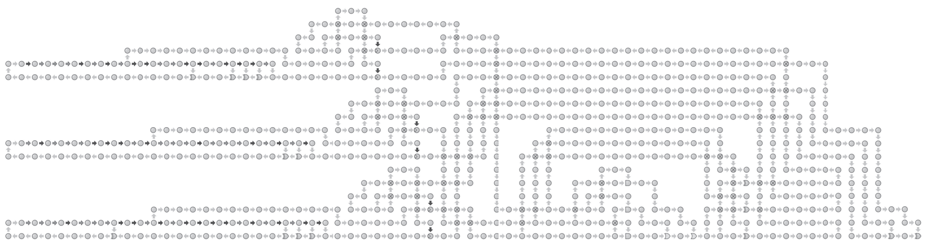

Fig. 7. A5/1 cipher

3.1 Hardware

Snap provides a concise definition of an ALA circuit that is portable across technologies: any process technology that can provide the local cell updates will operate correctly globally, because all of the design rules are contained within the cells. ALA has been ported to parallel multicore and microcontroller array emulators, and designed in CMOS [11]. With a CMOS library of the ALA cells, any design (such as the A5/1 example) can immediately be taped into into a chip, with timing and performance projected from simulation token counts. Here we show that it is straightforward to balance the ALA cells, so that their power and timing are independent of data.

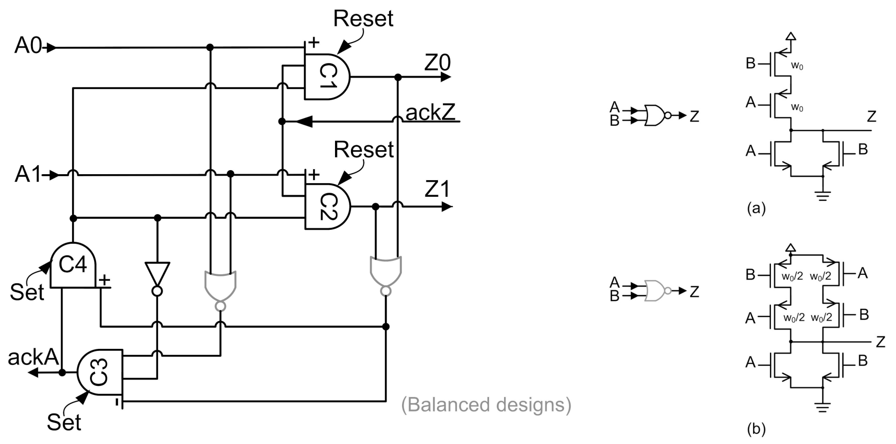

Fig. 8. Schematic for a performance balanced buffer cell

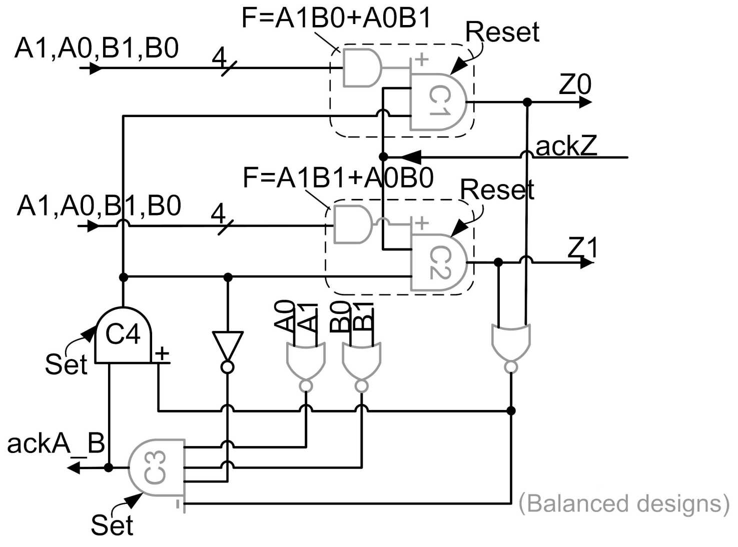

Fig. 9. Schematic for a performance balanced XOR cell

Figure 8 shows the schematic for an ALA buffer cell, built out of asymmetric C-elements [11] and Boolean NOR gates. The data dependency originates from the data-dependent behavior of the Boolean NOR gates. In a traditional NOR gate design shown in Figure 8(a), the transition behavior of the rising edge of state Z is dependent on the relative sequence of the falling edge of A and B. This is because the two PMOS transistors in series form the pull-up network; whether the top PMOS or the bottom PMOS transistor conducts first has a slight effect on the rising edge behavior due to the parasitic capacitance at the node between the two transistors. This asymmetry can be broken by splitting the

PMOS chain into two halves and swapping the sequence of the PMOS transistors in one half, as shown in Figure 8(b). The balanced NOR gate is now symmetric; when it replaces the conventional NOR gates the buffer ALA cell becomes power-balanced and has a data-independent latency.

Other ALA cells can likewise be balanced. Figure 9 shows a balanced XOR cell, in which the light blocks are re-designed. All asymmetric NMOS and/or PMOS chains connected to data lines are replaced with two half-sized transistor chains in parallel with different input sequences.

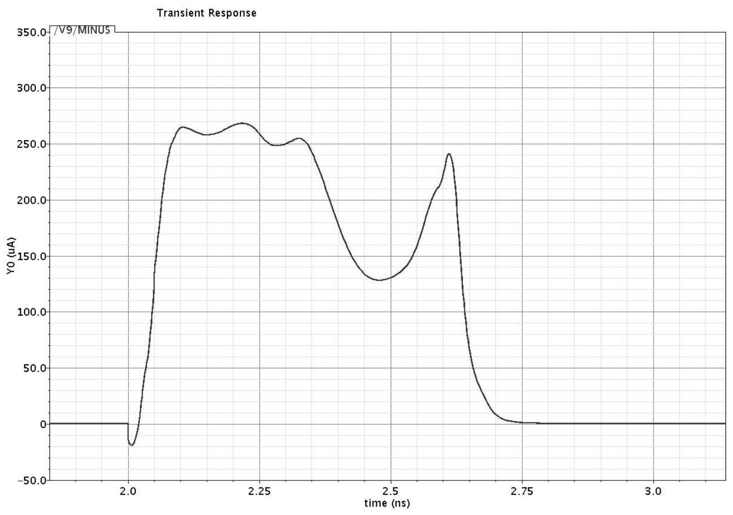

Fig. 10. Current consumption trace for an XOR cell firing

Figure 10 shows a trace of the current consumption over the course of a single XOR cell firing. The first token arrives before the two nanosecond mark. When the second token arrives at the two nanosecond mark, it triggers a firing which completes in less than one nanosecond. Because of the balanced design style, the current consumption waveforms are the same for different data input combinations. And because of ALA's hierarchical modularity, balancing the cells will eliminate system logic power and timing data dependency (although token copying and deletion could be observed if it is controlled by secret data).

4 Conclusions

We have introduced the use of Asynchronous Logic Automata in cryptography, with an example of the A5/1 stream cipher implemented in the Snap language. This provides a concise description with bit-level parallelism and implicit asynchronous data dependencies that is portable across technologies, and parametrically scalable over the homogeneous hardware substrate. Exposing rather than hiding hardware in this way can ease design, by being able to transparently span levels of description rather than requiring differing representations.

Because communication, computation, and storage are locally linked in ALA as they are in physics, there is an opportunity to not just implement existing cryptosystems but also develop entirely new ones that are based on the fundamental properties of information propagation. Future work will report on these spatio-temporal systems.

References

1. Delsarte, P., Quisquater, J.J.: Permutation cascades with normalized cells. Information and Control 23, 344–356 (1973)
2. Wolfram, S.: Cryptography with cellular automata. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 429–432. Springer, Heidelberg (1986)
3. Nandi, S., Kar, B.K., Chaudhuri, P.P.: Theory and applications of cellular automata in cryptography. IEEE Transactions on Computers 43, 1346–1357 (1994)
4. Seredynski, F., Bouvry, P., Zomaya, A.Y.: Cellular automata computations and secret key cryptography. Parallel Computing 30, 753–766 (2004)
5. Das, D., Ray, A.: A parallel encryption algorithm for block ciphers based on reversible programmable cellular automata. Journal of Computer Science and Engineering 1, 82–90 (2010)
6. Chelton, W.N., Benaissa, M.: Fast elliptic curve cryptography on FPGA. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16, 198–205 (2008)
7. Moore, S., Anderson, R., Cunningham, P., Mullins, R., Taylor, G.: Improving smart card security using self-timed circuits. In: Proceedings of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC 2002), p. 211 (2002)
8. Feldhofer, M., Trathnigg, T., Schnitzer, B.: A self-timed arithmetic unit for elliptic curve cryptography. In: Proceedings of the Euromicro Symposium on Digital System Design (DSD 2002), p. 347 (2002)
9. Gershenfeld, N., Dalrymple, D., Chen, K., Knaian, A., Green, F., Demaine, E.D., Greenwald, S., Schmidt-Nielsen, P.: Reconfigurable asynchronous logic automata (RALA). In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 1–6. ACM, New York (2010)
10. Bachrach, J., Greenwald, S., Schmidt-Nielsen, P., Gershenfeld, N.: Spatial programing of asynchronous logic automata (2011) (manuscript)
11. Chen, K., Green, F., Gershenfeld, N.: Asynchronous logic automata ASIC design (2011) (manuscript)