Cellular Automata on the Micron Automata Processor

Ke Wang
Department of Computer Science
University of Virginia
Charlottesville, VA
kewang@virginia.edu

Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, VA
skadron@cs.virginia.edu

University of Virginia Dept. of Computer Science Technical Report TR# CS-2015-03

April 2015

Abstract

A cellular automaton (CA) is a well-studied and widely used time-evolving discrete model. CAs are studied in many fields of science, such as computability theory, mathematics, physics, complexity science, theoretical biology and microstructure modeling. Some CA models have been proven to be Turing Complete, such as the elementary cellular automaton (ECA) of Rule-110 and Conways Game of Life. Micron's Automata Processor (AP) is a hardware implementation of non-deterministic finite automata (NFAs). The core feature of the AP, the state transition element (STE), was designed for highly parallel recognition of complex patterns in big data. The additional features of the AP, the on-chip Boolean and counter elements, theoretically expand the APs computation ability beyond those of traditional NFAs. However, the computational power of the AP remains undiscovered. To answer this question, we implement CAs on the AP, and use the successful implementations to demonstrate the AP's Turing Completeness. When mapping CAs on to the AP, one will face four major implementation difficulties: self-evolution, memory, communication, and computation of evolution rules. To handle these implementation challenges, this paper illustrates several novel primitives, including splitting the input symbol set into data symbols and instruction symbols, storing the live/dead states of a CA cell with activation/deactivation status of an STE, gathering the information of neighbor cells by a star structure or a ring structure, and translating evolution rules of CAs into Boolean logic or the combination of Boolean logic and counters. By using these primitives, we successfully implement and run the examples of ECA Rule 110 and Game of Life on the AP simulator. These results indicate that the AP chip implements a Turing-complete computational model. We also show several optimization strategies to improve performance and reduce the resource usage.

Keywords— Automata Processor, Cellular Automaton, Turing Complete

1 Introduction

Since the concept of a cellular automaton (CA) was originally discovered in the 1940s by Stanislaw Ulam and John von Neumann while they were contemporaries at Los Alamos National Laboratory, a variety of different kinds of CAs were widely studied. The computation models of CAs have been used in many application domains. Among the previous researches on the CAs, one of the most significant discoveries was that some CAs were proven to be Turing-complete machines which have the universal computation ability.

Besides the elaborate algorithms designed for simulating the evolution processes of CAs, hardware acceleration solutions were proposed to speedup the computation of CAs. Micron recently proposed a hardware implementation of the finite state machine - the Automata Processor (AP). With huge amount of on-chip state units, the AP has shown massively parallel computation ability, particularly for high-throughput pattern mining in big data. The AP have achieved incredibly high performance in the applications domains of fuzzy and exact pattern searching [1, 2]. The additional features of the AP, the on-chip Boolean and counter elements, theoretically expand the AP's computation ability beyond those of traditional NFAs. However, none of the previous works has discovered the computational power of the AP.

In this work, we successfully implement the Elementary Cellular Automata and Game of Life on the AP. We make the following contributions:

1. We illustrate several novel primitives for CA designs on the AP including splitting the input symbol set into data symbols and instruction symbols, storing the live/dead states of a CA cell with activation/deactivation status of an STE, gathering information of neighbor cells by a star or a ring structure, and translating evolution rules of CAs into Boolean logic or the combination of Boolean logic and counters (Section 4)

2. By using these primitives, we successfully implement and run the examples of ECA Rule 110 and Game of Life on the AP (Section 5 and 6)

3. We therefore prove that the AP chip implements a Turing-complete computational model (Section 5.4 and 6.6)

4. We also show several optimization strategies to improve performance and reduce the resource usage (Section 5.2 and 6.4)

Section 2 and 3 give brief introductions to the Cellular Automata and the Automata Processor respectively.

2 Cellular Automaton

The concept cellular automaton (CA) was originally proposed by Stanislaw Ulam and John von Neumann. As a time-evolving discrete model, CA is studied in many fields of science, such as computability theory, mathematics, physics, complexity science, theoretical biology and microstructure modeling [3].

A cellular automaton consists of a regular grid of cells, each in one of a finite number of states. The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state (time t = 0) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously, though exceptions are known, such as the stochastic cellular automaton and asynchronous cellular automaton.

2.1 Elementary cellular automaton

An elementary cellular automaton (ECA) is a one-dimensional cellular automaton where there are two possible states (labeled 0 and 1) and the rule to determine the state of a cell in the next generation depends only on the current state of the cell and its two immediate neighbors. As such it is one of the simplest possible models of computation. Nevertheless, there is an elementary cellular automaton (rule 110, defined below) which is capable of universal computation.

2.2 Game of Life [4]

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway. The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves or, for advanced players, by creating patterns with particular properties. The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seedbirths and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

2.3 Implementation Notes

For any CA, there are four fundamental elements need to be implemented:

- Self-evolution: the whole system evolves with time without inputs
- Memory: each CA cell needs to remember its status temporarily
- Communication: each CA cell needs to know the status of its neighbors
- Computation: each CA cell needs to calculate its next status according to its status and status of its neighbors

We will discuss the general methodologies to implement the above elements on the AP in Section 4 and specific designs of automata for ECA Rule-110 and Game of Life in Section 5 and 6.

3 Introduction to Micron Automata Processor

Micron's Automata Processor (AP) is a massively parallel non-von Neumann accelerator designed for high-throughput pattern mining.

3.1 Function elements

The AP chip has three types of functional elements - the state transition element (STE), the counters, and the Boolean elements [5]. The state transition element is the central feature of the AP chip and is the element with the highest population density. Counters and Boolean elements are critical features to extend computational capabilities beyond NFAs.

STEs have two exclusive states - activated and deactivated. An STE is activated if the incoming signal is high. If there are many incoming connections, an implicit rule of "or" will be applied to all these incoming connections. All activated STEs in one automaton will accept the same symbol of input stream at a given cycle. Each STE can be configured to match a set of any 8-bit symbols. When an activated STE matches a symbol of input stream, the output connections (if any) will produce a high signal. An STE can be configured as a self-activation STE by connecting an output wire back to its incoming port.

The Boolean element can be configured as "or", "and", "nor", "nand", "inv", "POS" (product-of-sum), "SOP" (sum-of-product), "NPOS" (not POS), "NSOP" (not SOP). The last four are Booleans element with multiple (6) input terminals, say $I_0, I_1, I_2, I_3, I_4, I_5$ and one output. An SOP is the sum (OR) of product (AND) terms. A POS is the product (AND) of sum (OR) terms. An NSOP is an SOP with its activation value inverted. An NPOS is POS with its activation value inverted. For example, the "POS", $output = (I_0\ or\ I_1)\ and\ (I_2\ or\ I_3)\ and\ (I_4\ or\ I_5)$; the "SOP", $output = (I_0\ and\ I_1)\ or\ (I_2\ and\ I_3)\ or\ (I_4\ and\ I_5)$.

A counter has two input ports - counting port (a small "C" on the counter in the following automaton illustrations) and reset port (a small "R" on the counter in the following automaton illustrations). When the counting port receives a high signal at a cycle, the counter will count one incrementally (only incrementally). A counter will be reset if it receives high signal from its reset port. A counter needs to be configured with an integer threshold. When the threshold is reached, a counter will assert its output. Depending on the operation mode configured, one counter can have three different output behaviors when the threshold is reached: latch mode, i.e., holds high output forever; pulse mode, i.e., produces high signal at that cycle only; and roll mode, i.e., produces a high signal at that cycle and reset. One counter can count up to 2^{12}.

3.2 Speed, capacity and power

Micron's current generation AP - D480 chip is built on 45nm technology running at an input symbol (8-bit) rate of 133 MHz. The D480 chip has two half-cores and each half-core has 96 blocks. Each block has 256 STEs, 4 counters and 12 Boolean elements. In total, one D480 chip has 49,152 processing state elements, 2,304 programmable Boolean elements, and 768 counter elements [5]. Each AP board can have up to 48 AP chips that can perform pattern matching in parallel [6]. Each AP chip has a worst case power consumption of 4W [5]. The power consumption of a 48-core AP board is expected to be similar to a high-end GPU card.

3.3 Input and output

The AP takes input streams of 8-bit symbols. Each AP chip is capable of processing up to multiple separate data streams concurrently, although we do not use this feature for this work. The data processing and data transfer are implicitly overlapped by using the input double-buffering of the AP chip. Any STE can be configured to accept the first symbol in the stream (called start-of-data mode, a small "1" in the left-upper corner of STE in the following automaton illustrations), to accept every symbol in the input stream (called all-input mode, a small "∞" in the left-upper corner of STE in the following automaton illustrations) or to accept a symbol only upon activation. The all-input mode will consume one extra STE.

Any type of element on the AP chip can be configured as a reporting element; one reporting element generates a one-bit signal when the element matches an input symbol. One AP chip has up to 6144 reporting elements. If any reporting element reports at a cycle, the chip will generate an output vector which contains signals of "1" corresponding to the elements that report at that cycle and "0"s for reporting elements that do not report. If too many output vectors are generated, the output buffer can fill up and stall the chip. Thus, minimizing output vectors is an important consideration for performance improvement.

3.4 Programming and reconfiguration

Automata Network Markup Language (ANML) is an XML language for describing the composition of automata networks. ANML is the basic way to program automata on the AP chip. Besides ANML, Micron provides a graphical user interface tool called the AP Workbench for quick automaton designing and debugging. A "macro" is a container of automata for encapsulating a given functionality, similar to a function or subroutine in common programming languages. A macro can be defined with parameters of symbol sets of STEs and counter thresholds which can be instantiated with actual arguments. A macro also can have input ports and output ports to connect the wires outside the macro. Micron's AP SDK also provides C and Python interfaces to build automata, create input streams, parse output and manage computational tasks on the AP board.

4 Methodology

In this section, we propose several primitives to implement 1) self-evolution 2) temporary and local storage of cell status 3) communication between cells 4) computation for next status. In conjunction to self-evolution, we will discuss a strategy of splitting the global set of input symbol to subsets of data input and instruction input. This strategy makes it possible to re-initialize the CA without an automaton recompilation.

4.1 Self-evolution

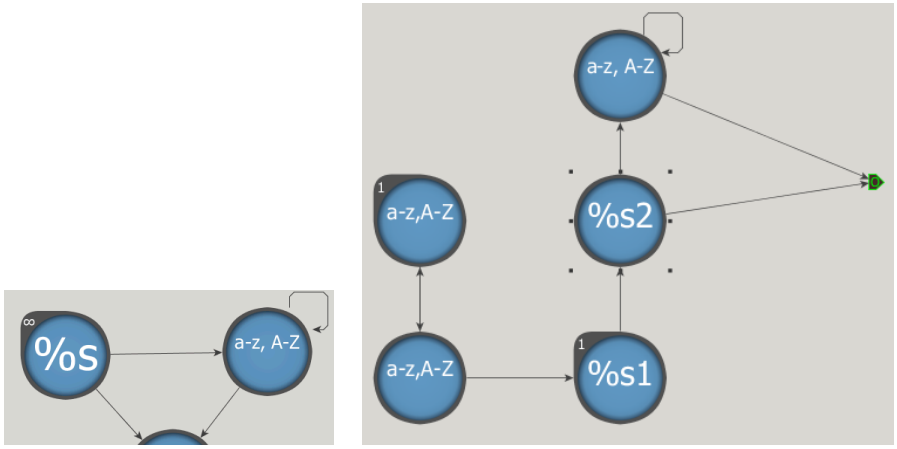
For a self-evolving system like a CA, it only consumes "cycles" instead of the specific symbols. Theoretically, any input symbol should work for a CA. However, a better strategy is to define a set of symbols as instructions to control the information storage, communication, computation. In this work, we define a set of instruction symbols to "drive" the self-evolution of the CAs.

It is possible to set the initial status of CA cells by configuring the activation statuses of STEs. But this strategy will need re-compilation of the whole automaton onto the AP chip to setup the initial condition for another run. To avoid such device re-compilations, we design automaton structures, namely labeling structures, to initialize the CA cells. Accordingly, we define a set of instruction symbols and a set of label symbols. Each CA cell is named with a unique label, which can be one symbol or a pair of symbols depending on the design. When a label is seen in the input, the corresponding CA cell is forced to configured as "alive" status via labeling structures. In the present work, we use digit symbols, e.g. 0-9 as instruction symbols and alphabet symbols, e.g. a-z and A-Z as label symbols. Other choices of instruction symbol set and label symbol set also work, as long as there is no overlap between them.

Figure 1 shows two automaton structures for one-symbol and two-symbol labels.

4.2 Storage of cell status

A typical non-deterministic finite automata (NFA) has no straightforward way to store information. We propose a primitive strategy to store one bit of information by using the activated/deactivated status of an STE. We propose two different storage schemes: static storage and dynamic storage.



(a) An automaton for one-symbol label

(b) An automaton for two-symbol label

Figure 1: Automaton structures for the initialization of a CA cell.

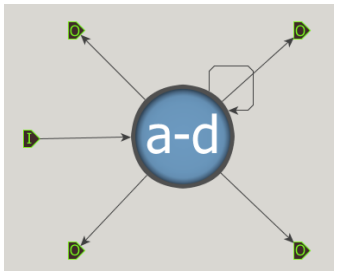


Figure 2: An example of an automaton structure for static storage.

4.2.1 Static storage

The static storage scheme utilizes an STE with a self-activating connection. In this scheme, the central STE is configured to match a symbol set S. After the central STE is activated, it keeps itself activated and outputs high signals as long as the input symbols are in set S. The storage will be cleared when the input symbol is out of the range of S. Figure 2 shows an example of static storage where the $S = \{a, b, c, d\}$. In this example, after the central STE is activated, the activation information will be held and produce high signals to the four output terminals when the input symbol is one of $\{a, b, c, d\}$.

4.2.2 Dynamic storage

The static storage scheme creates a path of STE where the information of activation will pass on. In this scheme, the STEs in the path are figured to match a symbol set S. After the first STE is activated, the information of activation will pass on to next STE as long as the input symbols are in S and output terminals associated with the STEs will produce high signals in order. Figure 3 shows an example of dynamic store where the $S = \{a, b, c, d\}$. In this example, after the first STE is activated the activation information will pass on to the next STE and the output terminals provide high signals one-by-one when the input symbols belong to set a, b, c, d. When compared with static storage scheme another difference is that the storage lifetime of the dynamic storage is also controlled by the length of STE path. The example of Figure 3 can only hold the information for 4 cycles.

4.3 Communication

The communication among CA cells depends on the connections on the AP chip. In some cases, e.g. the Game of Life, a CA cell needs to deal with the incoming information of neighbors one-by-one due to the limited computation bandwidth. Therefore an one-by-one information gathering strategy is introduced. The corresponding implementations of a star structure and a ring structures are proposed as well. We will dive into the details of this strategy

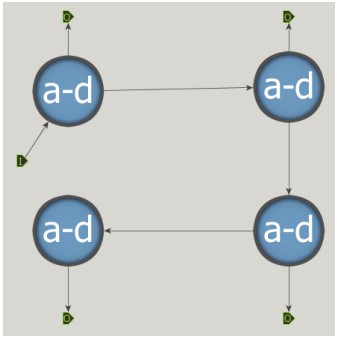


Figure 3: An example of automaton structure for dynamic storage.

in Section6.

4.4 Computation

The computation for evolution is the core feature of a CA. On the AP chip, the computation is implemented by Boolean elements, counter elements and the combination of these two kinds of elements.

In summary, a set of STEs, Boolean and counter elements to implement the functionality of memory and computation are grouped together to form a cell of a CA. The CA cells are wired together for communication by the connections of the AP chip according to the topology of the CA.

5 Elementary Cellular Automata

5.1 Design of ECA

The cells in an elementary cellular automaton (ECA) form an 1D linear array where the cells evolve according to the its status and the status of its two neighbors.

5.1.1 Macro of ECA

Figure 4 shows the macro of automaton designed to represent a cell of ECA rule 110. One macro of rule 110 has two output ports, "alive" and "dead", to represent the *alive* and *dead* status of this cell. Since a cell should be either "alive" or "dead", there must be one and only one of these two output ports carry "high" signal in the computation stage. One rule 110 macro has four input ports, "left_alive", "left_dead", "right_alive" and "right_dead", to connect and read the *alive/ dead* status of left and right neighbors, respectively.

5.1.2 Data and instrcutions

The data input is defined as alphabet symbols while the instruction input is defined as 0,1. A two-symbol sequence [1, 0] in the input stream drive the ECA automaton to finish one life-cycles.

The initialization component of this macro is shown in the pink box. We use one-symbol labeling scheme but it is also possible to use two-symbol scheme.

5.1.3 Storage

A dynamic storage scheme is adopted with a two-step STE path. Different from the basic example shown in Section 4.2.2, we have two branches here with an invertor associated with one branch. This design guarantees there should be one and only one STE of "0" is activated. Note that if two Boolean elements connected directly the

Table 1: An example of input stream for ECA Rule 110

operation	initialization				evolve for 3 life-cycles						re-initialization		evolve for 2 life-cycles			
input symbol	a	c	m	z	1	0	1	0	1	0	k	y	1	0	1	0

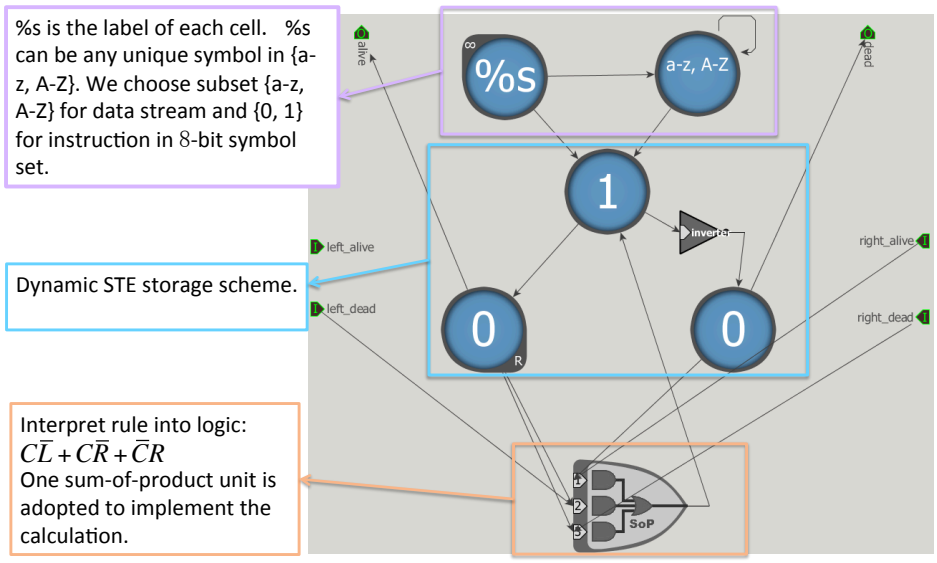


Figure 4: The macro of ECA rule 110

process rate of the AP chip will be cut to half due to the propagation delay. For this reason, we have to add the STEs of "0" to isolate two Boolean elements.

5.1.4 Computation

A Rule 110 automaton evolves according to the set of rules in Table 2. If we use Boolean variable L, C, R to represent the "alive" status of left neighbor, central cell and right neighbor. The rules in Table 2 can be expressed as $C\overline{L}+C\overline{R}+\overline{C}R$. One Boolean element configured as SOP is adopted to carry out this calculation.

5.1.5 Communication

For a given ECA cell, its "left_alive" and "left_dead" input ports should connect to the "alive" and "dead" output ports of it left neighbor to gather the information from its left neighbor; its "right_alive" and "right_dead" input ports should connect to the "alive" and "dead" output ports of its right neighbor. For left-most and right-most cells, there are two types of boundary conditions: 1) periodical boundary condition, assuming the left neighbor of left-most cell is the right-most cell and the right neighbor of right-most cell is the left-most cell. 2) constant boundary condition, giving high signal to one of the "left_alive" and "left_dead" of the left-most cell to assume an "always alive" or "always dead" left neighbor of left-most cell; the same to the right-most cell.

Putting everything together, Figure 5 shows a small scale three-cell ECA with a constant boundary.

5.2 An alternative design of ECA

The above automaton for ECA needs two cycles for one life-cycle because the two Boolean elements are connected in a cycle and two STEs are needed to separate them. We propose an alternative design of ECA which use two Boolean elements in parallel. The idea is to design a "shadow" cell which evolve with the negative rules of 110. The negative rules of 110 (Table 3) can be translated to the Boolean formula $(C+\overline{R})(L+\overline{C})(\overline{C}+R)$. A Boolean element is configured as POS to implement such computation. The macro of this design is shown in Figure6. The initialization component of the shadow cell is designed to get negative initialization. That is if symbol "s" is seen, the "0" STE is deactivated in the corresponding shadow cell. This automaton is driven by instruction symbol "0" with one cycle for one life-cycle.

Table 2: Rules of ECA Rule 110

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	1	0	1	1	1	0

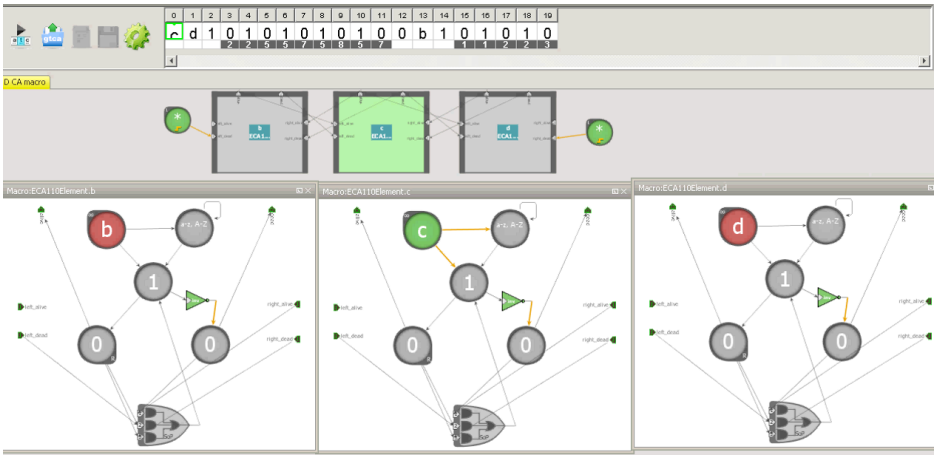


Figure 5: An example of ECA rule 110

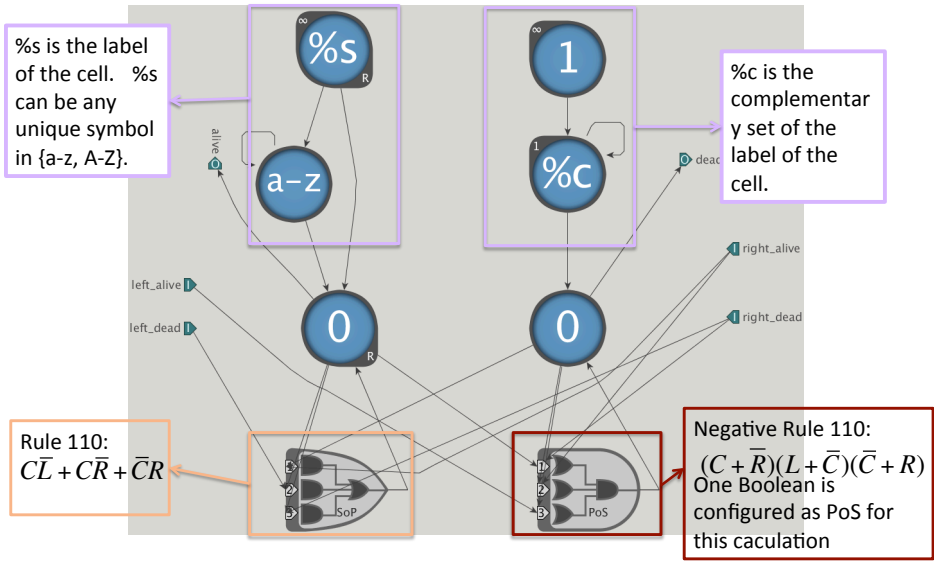


Figure 6: An alternative design of macro for ECA rule 110

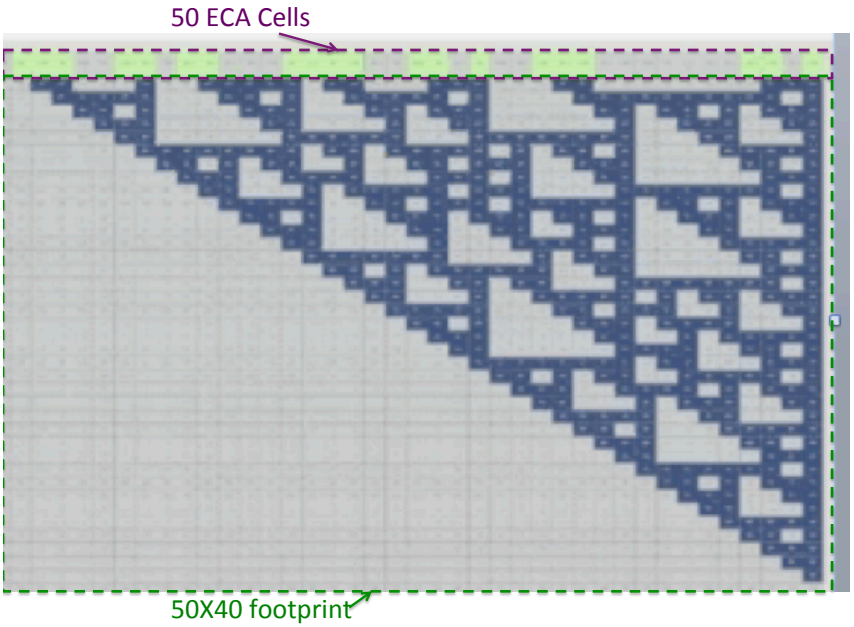


Figure 7: An example of ECA rule 110 (50 cells and 37 life-cycles)

5.3 Results

The Figure 7 shows the result of proposed automaton for ECA rule 110 with 50 ECA cells. The ECA cells are in the array on the top of the Figure 7. A automaton structure, called *footprint*, is designed to receive the dead/life status of ECA cells from input port, hold the status for a life-cycle and pass the status to next *footprint* below. In Figure 7, the array of ECA cells are passing their status to a 2D 50X40 array of *footprint*. Each row of the *footprint* represent a generation of the ECA array; the top row shows the newest generation while the bottom row shows the oldest generation. As shown in the figure, the ECA array starts with one alive cell and generats a pattern of Mollusks Shell, which is a well-known behavior of ECA array.

5.4 Turing Completeness

The Rule 110 has been proven to be Turing complete (universal) by showing the feasibility to use the rule to emulate another computational model, the cyclic tag system, which is known to be universal [3, 7]. Wolfram [3] first showed a number of spaceships, self-perpetuating localized patterns, that could be constructed on an infinitely repeating pattern in a Rule 110 universe. He then proposed a method to combine these structures to interact in a way that could be exploited for computation [7].

6 Game of Life

6.1 Data and instructions

The alphabet symbols are defined as the data input while digits are defined the instruction input. A multiple-symbol sequence of digits in the input stream drive a Game of Life automaton to finish one life-cycles. The instruction

Table 3: Negative Rules of ECA Rule 110

current pattern	111	110	101	100	011	010	001	000
new state for center cell	1	0	0	1	0	0	0	1

Table 4: Game of Life Rules

No. of alive neighbor	< 2	2	3	> 3
alive	dead	alive	alive	dead
dead	dead	dead	alive	dead

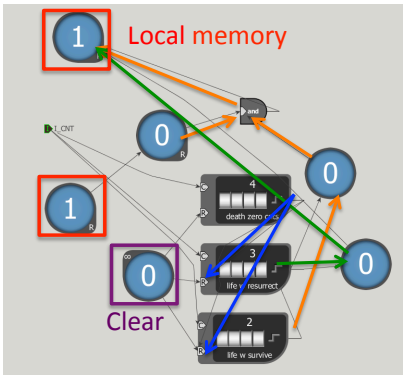


Figure 8: The computation component of Game of Life Cell

sequence can be further divided into counting instructions (or counting cycles) by which the counters gather the states of the neighbors and computation instructions (or computation cycles) which compute the next state of the central cell. The instruction sequence depends on the specific design of storage and computation components (we will talk about them below).

6.2 Computation of evolution

Game of Life (GL) is a two-dimensional cellular automaton. One GL cell needs to know the *alive/dead* status of its 8 neighbors. We translate the rules of GL shown in Section 2.2 in to a table (Table 4). If we use Boolean elements only, Table 4 can be translated to a logic expression of 88 SOP terms. Note the Boolean elements are scarce resource on the AP chip, and the Boolean elements can only connect to the STE within the same block. These constraints make Boolean-only design unfeasible on the AP chip.

We adopted a computation scheme which utilize a combination of counter and Boolean elements to reduce the complexity in design. In general, a counter can only count one neighbor per cycle. So we split the counting process into several cycles. The general strategy is that at each counting cycle the storage component provides the information of "alive/dead" to a neighbor in a direction while the computation component gathers the information of a neighbor in opposite direction.

Figure 8 shows the implementation of rules in Table 4 by three counters and one Boolean element of "AND". The orange and green paths shown in Figure 8 correspond the orange and green columns in Table 4 which lead to the "alive" state of the central cell. The blue paths correspond the blue column in Table 4 where the counter called "death zero cnt" (the top counter in the figure) fires and force to reset other two counters to suppress the potential fires of the lower two counters in the current life circle. The cases indicated in the black (first) column in Table 4 will not lead to a threshold-reached situation of any counter and therefore causes the "dead" state of the central cell. The STE in purple box helps to reset all three counters in the end of each life cycle. The top and below STEs in red boxes denote the enter point and exit point of the memory component respectively. The specific designs of memory component are discussed below.

6.3 Star structure - a static storage scheme

Figure 9 shows an automaton structure of static storage scheme, called the *star structure*. The input symbols in the range of 0-7 hold the state of the central STE in this structure and activate the 8 peripheral STEs. The sequence of storage instructions "012345678" for this design will match the 8 peripheral STEs one by one. Note the first "0" is not removable because it transfer the information from the central memory STE to one of the 8 peripheral STEs.

Putting storage, computation and labeling components together, Figure 10 shows a design of the automaton macro for the Game of Life cell using static memory scheme. Each macro has one input port of the computation

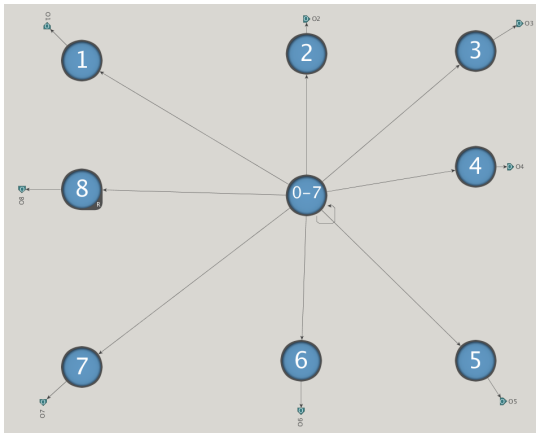


Figure 9: The storage component in star structure for a Game of Life Cell

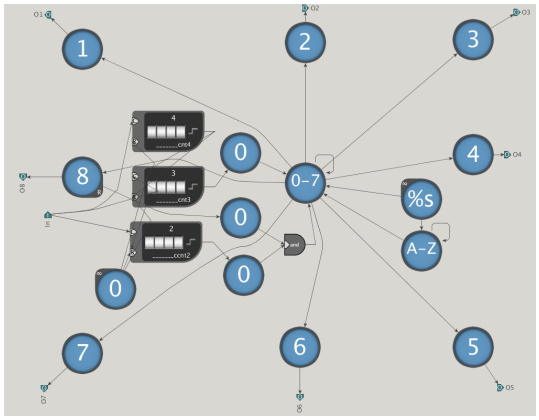


Figure 10: The automaton macro for a Game of Life cell with static memory

component connecting the output ports of the 8 neighbors and eight output ports of memory component connecting in the input ports of 8 neighbors. With one more computation instruction "0", the instruction sequence for a life-cycle is "0123456780".

6.4 Ring structure - a dynamic storage scheme

The *start structure* has two issues: 1) it wastes one cycle in transfer information from the central STE to the peripheral STEs of storage component 2) it creates an connection hot spot of the central STE increasing the difficulty of placement and routing and preventing the design to be extended to other topology, e.g. 3D cellular automata.

To solve these issues, we provide an alternative design using the dynamic memory scheme (shown in Figure 11). This design creates a information delivery path of 8 STEs without a central STE. In each counting cycle, one of the STE in the path is matched and provides the status of the central cell to the neighbor in the corresponding direction. In the same cycle, the computation component reads the information of the neighbor in the opposite direction. The sequence of storage instructions "11111111" for this design will enable the 8 peripheral STEs one by one.

Figure 10 shows a design of the automaton macro for the Game of Life cell using dynamic memory scheme. Each macro has one input port of the computation component connecting the output ports of the 8 neighbors and eight output ports of memory component connecting in the input ports of 8 neighbors. With one more computation instruction "0", the instruction sequence for a life-cycle is "111111110".

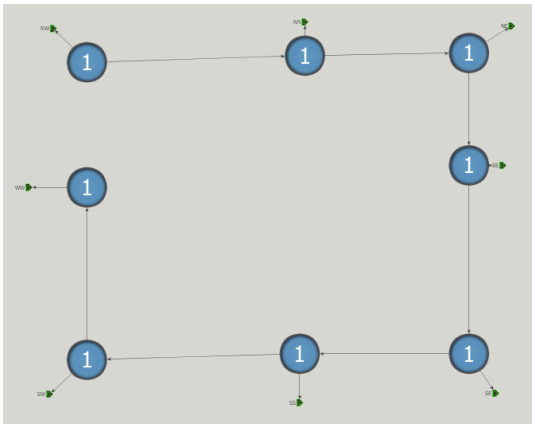


Figure 11: The storage component in ring structure for a Game of Life Cell

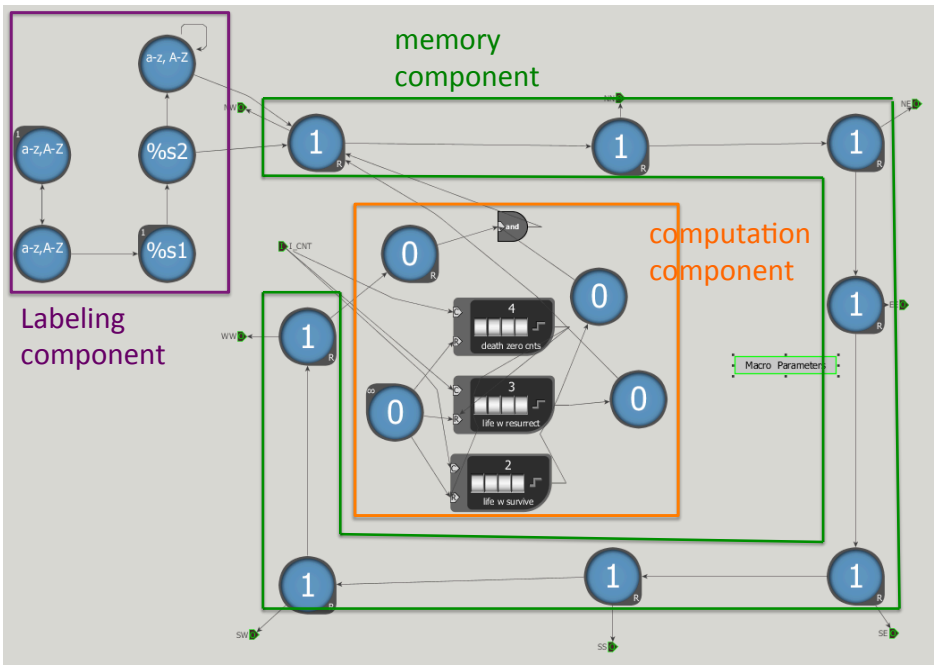


Figure 12: The automaton macro for a Game of Life Cell with dynamic memory

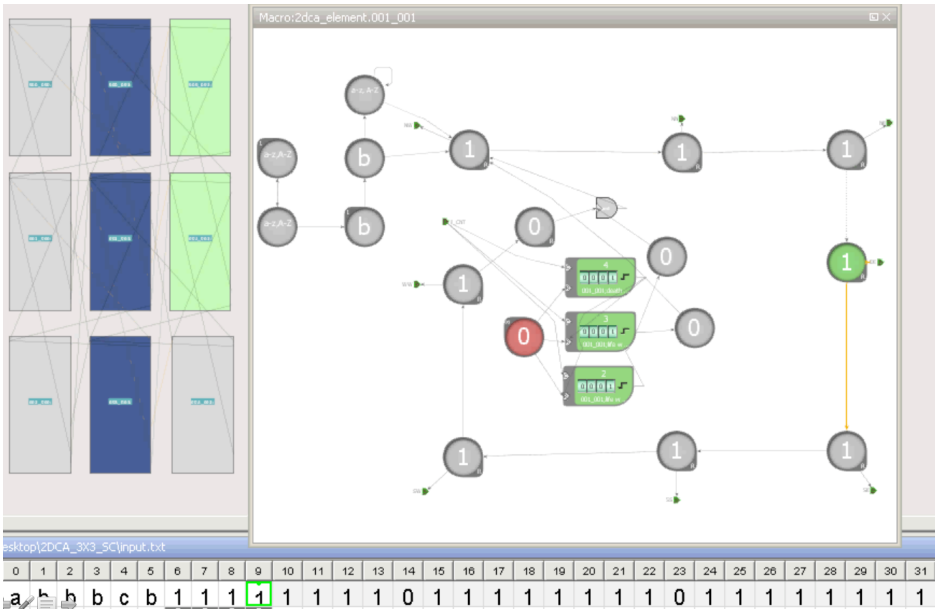


Figure 13: An automaton implementing Blinker with 3X3 grid of Game of Life cells

6.5 Results

Figure 13 and 14 show the automata for two typical evolution patterns of the Game of Life - blinker and Gosper Glider Gun. The cycle-by-cycle simulations of these two automata by the Micron's AP Workbench (an automata simulator) show correct function of these automata.

6.6 Turing complete

Some evolution patterns of the Game of Life automata can be used to construct logic gates (such as AND, OR and NOT) and counters [4]. It is also possible to build a pattern that acts like a finite state machine connected to two counters. The resulting Game of Life automaton has the same computational power as a universal Turing machine [4]. Therefore the Game of Life is theoretically as powerful as Turing complete machine.

7 Conclusions

To map CAs on to the AP, we propose solutions to handle four major implementation issues including self-evolution, memory, communication, and computation of evolution rules. These solutions illustrate several novel primitives including splitting the input symbol set into data symbols and instruction symbols, storing the live/dead states of a CA cell with activation/deactivation status of an STE, gathering information of neighbor cells by a star or a ring structure, and translating evolution rules of CAs into Boolean logic or the combination of Boolean logic and counters. We successfully implement and verified the implementations of ECA Rule 110 and Game of Life on the AP. We therefore illustrate the ability of universal computing of the AP chip. We also propose several optimization strategies to improve performance and reduce the resource usage in the automaton designs of CAs.

Acknowledgment

This work was supported by a grant from Micron Technology.

References

[1] I. Roy and S. Aluru, "Finding motifs in biological sequences using the micron automata processor," in *Proc. of IPDPS'14*, 2014.

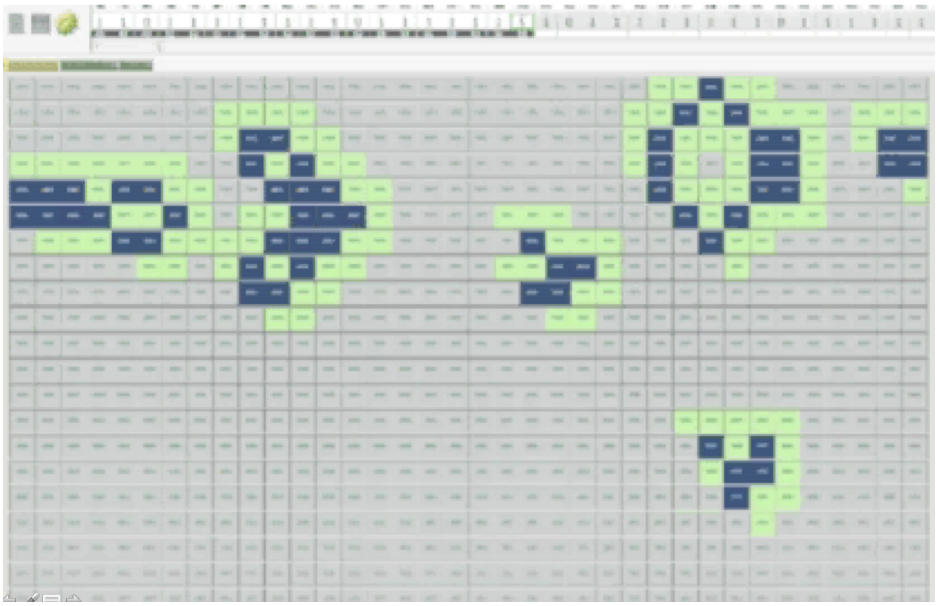


Figure 14: An automaton implementing Gosper Glider Gun with 22X36 grid of Game of Life cells

[2] K. Wang *et al.*, "Association rule mining with the micron automata processor," in *Proc. of IPDPS'15*, to appear.

[3] S. Wolfram, *A new kind of science*. Wolfram media Champaign, 2002, vol. 5.

[4] Wikipedia, "Plagiarism — Wikipedia, the free encyclopedia," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

[5] P. Dlugosch *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE TPDS*, vol. 25, no. 12, 2014.

[6] H. Noyes, "Micron's automata processor architecture: Reconfigurable and massively parallel automata processing," in *Proc. of Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, June 2014, keynote presentation.

[7] Wikipedia, "Plagiarism — Wikipedia, the free encyclopedia," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Rule_110