Definition of a Configurable Architecture for Implementation of Global Cellular Automaton

Christian Wiegand, Christian Siemers, and Harald Richter

Technical University of Clausthal, Institute for Computer Science, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
{wiegand|siemers|richter}@informatik.tu-clausthal.de

Abstract. The realisation of Global Cellular Automaton (GCA) using a comparatively high number of communicating finite state machines (FSM) leads to high communication effort. Inside configurable architectures, fixed numbers of FSM and fixed bus widths result in a granularity that makes mapping of larger GCA to these architectures even more difficult. This approach presents a configurable architecture to support mapping of GCA into a single Boolean network to omit increasing communication effort and to receive scalability as well as high efficiency.

1 Introduction

Cellular Automaton (CA) are defined as a finite set of cells with additional characteristics. The finite set is structured as n-dimensional array with well-defined coordinates of each cell and with a neighbourhood relation. Each cell is capable to read and utilise the state of its neighbouring cells. As the cells implement a (synchronised) finite state machine, all cells will change their states with each clock cycle, and all computations are performed in parallel. Data and/or states from non-neighbouring cells are transported stepwise from cell to cell when needed. Useful applications to be implemented within CA consist of problems with high degrees of data locality.

Mapping a CA to real hardware – whether configurable or fixed – shows linear growth of communication lines with the number of cells. These links are fixed and of short length, resulting in limited communication efforts to implement a CA. If complexity of the functionality is defined by the RAM capacity to realise this function inside memory – this is normally the case inside configurable, look-up table based structures – the upper bound of the complexity will grow exponentially with the number of inputs and linear with the outputs. The number of input variables is the sum of all bits to code the states of all neighbouring cells as well as the own state. Cell complexity is therefore dominated by the number of communication lines.

The concept of Global Cellular Automaton (GCA) [1] overcomes the limitations of CA by providing connections of a cell not only to neighbouring but to any cell in the array. The topology of a GCA is therefore no longer fixed, GCA enable application-specific communication topologies, even with runtime reconfiguration. The number of communication lines per cell might be fixed by an upper limit.

C. Müller-Schloer et al. (Eds.): ARCS 2004, LNCS 2981, pp. 140–155, 2004.
© Springer-Verlag Berlin Heidelberg 2004

As the number of possible links between k cells will grow with k^2, the number of realised communication lines per cell will also grow with order 2. The complexity of a single cell and the Boolean function inside will depend on the number of communication inputs, as discussed in the case of cellular automaton. If any GCA is mapped to a reconfigurable architecture like an FPGA, each cell must be capable of realising any Boolean function with maximum complexity.

If the cells are mapped to a reconfigurable array of ALUs, each with local memory, each cell may integrate any complex functionality. The communication effort grows with the square number of cells, and the granularity of the circuit is defined by the number of cells and the bit width of the communication links between them. This architecture is well suited to realise a GCA, when the number of cells and the bit width fits well, because even complex computations might be performed inside one ALU. The disadvantage of this approach is that the cycle time is deterministic but not bounded, because any algorithm could be realised within one ALU but might use several cycles to perform. Even worse, mapping of GCA with non-fitting characteristics will be difficult if not impossible.

To map the GCA to another type of reconfigurable cell array, each with programmable Boolean functionality, results in cells capable of computing data from each other cell including the own state. This means that all binary coded states from all cells might form the input vector of this function, while the output vector must be capable of coding the state of the cell. Consequently the complexity of the single cell will grow exponentially with the input vector size, while communication will grow in polynomial order.

The approach in this paper presents an architecture capable of realising a GCA into a single Boolean network, where the output vector at the time t_n form part of the input vector for the next state t_{n+1}. This omits the resulting complexity by the communication lines, which is important for any reconfigurable architecture. Even more important, this architecture makes no assumption about the granularity, only the resulting complexity of the GCA is limited.

The remainder of the work contains the definition of the architecture in chapter 2. The introduced structure is capable of containing Boolean functions with large number of input and output lines. Chapter 3 discusses the mapping of GCA to this architecture and presents an example for realising an algorithm in a GCA and mapping this on the introduced architecture. Chapter 4 finally gives an outlook to future work.

2 A Reconfigurable Boolean Network for Large Functions

To design a reconfigurable Boolean network, one of two basic architectures are normally used and are discussed below:

The function to be implemented may be defined completely by storing all values inside a RAM memory. The input vector forms the address bit vector and addresses a well-defined memory cell for any combination. The contents of this memory cell defines the function at this point, and the data bus lines form the output vector. This is known as the look-up table (LUT) architecture.

The most important advantages of this architecture are its simplicity, the simple reconfigurability, the high density of memory circuits and the fixed (and fast) timing.

The with the input vector size exponentially growing number of memory cells is of course disadvantageous, limiting the practical use of LUT structures to small functions.

The second possibility to implement any Boolean function inside a reconfigurable architecture is to use a configurable network consisting of 2 or *k* stages. This mostly minimises the use of gates to implement the functionality, and theory has developed representations (e.g. Reed-Muller Logic), algorithms for minimising logic and partitioning it for several stages.

The advantage of this approach is that minimised number of gates are used to implement the function. Especially fixed implementations are well supported, but for reconfigurable architectures, the effort again grows exponentially with the input vector sizes (but at different rates, compared to the LUT-based architecture).

2.1 Introducing the New Architecture

To combine the advantages of the first architecture – high degree of integration, simplicity of the circuit – with the reduced number of gates of the second approach, the following approach is considered inside this paper. The basic idea consists of the balanced combination of storing functionality inside RAM-based memory and introducing 3 stages inside the architecture to reduce memory size and complexity.

2.1.1 Three Stage Approach

First Stage

The input vector of the Boolean function is represented by the input lines to the complete network. The first stage consists of several memory arrays (or ICs) in parallel, addressed by the input lines. The input vector is partitioned and the parts are each mapped to one corresponding memory array. The so-called minterms of the application, derived from logic minimisation e.g. using Quine-McCluskey or Espresso [2], are stored inside these memory arrays of the first stage. Each part of the array stores a well-defined representation of the (partial) input vector with the actual values (true, false, don’t care) and defines a representing code for this minterm, the so-called minterm-code.

Each memory array of the first stage is addressed by a subset of the input lines and compares this address to each of its 3-valued partial minterms. If a match is found, the minterm-code is sent via the data bus lines to the second stage. If an address doesn't match any partial minterm of one memory array, no data is returned. After processing a complete input vector, the first stage returns a bit pattern that represents all minterms of the Boolean function, which correspond to the input vector.

Second Stage

The minterm-code addresses the memory of the second stage. The memory cells hold the corresponding bit pattern of the minterm-codes and the output vectors. If any input vector of this stage matches one of the stored codes, the stored output information is read out via the data lines and given to stage three for further computation.

The addressing scheme uses again three-valued information, but this time the output consists only of two-valued information. The address information is compared to all stored information in parallel, and a matching hit results in presenting all stored data on the data bus of this second stage memory array. If no matching hit occurs, the corresponding memory array returns ‘0’.

Third Stage
The third stage combines all output values from stage 2 via the OR-function.

2.1.2 Detailed Implementation

Figure 1 shows the complete implementation of a 3-stage network, using an example with 12 input lines, 10 minterms and 8 output lines.

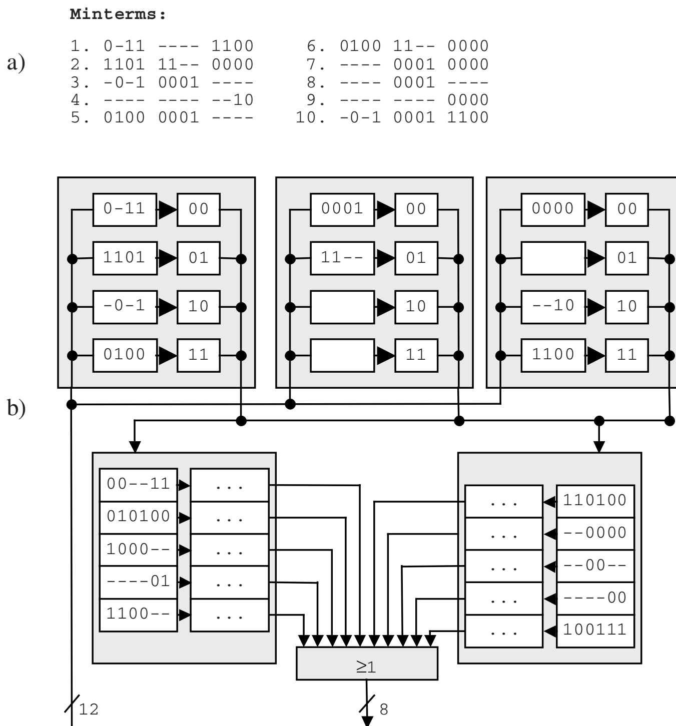

Fig. 1. 3-stage reconfigurable Boolean network: a) Representing minterm table b) Circuit structure

The complete input vector is partitioned into 3 parts each containing 4 3-valued data. The minterm table in fig. 1 a) shows similarities to the open-PLA representation [3]. The stored value for the corresponding combination is the minterm code, and the resulting minterm code vector by simple concatenation of the responds is the complete code for the actual input vector.

There might be several matching minterm codes for one input. The input vector contains binary information, but 3-valued information is stored, and the ‘-’-value for don’t care matches to ‘0’ and ‘1’ by definition. Figure 1 shows as an example that the minterms 3, 4 and 8 matches e.g. to “0011 0001 0010”, and for the minterms 4, 5 and 8 matching input vectors are possible too. This means that for correct computation, the system must be capable of computing more than one minterm code vector.

The resulting minterm code vectors are used by the second stage of the circuit, where the corresponding output vectors are read. The responds of the second stage have to be coupled using the OR-gate of stage three. This is the final result of the operation.

One choice for storing the minterms could be an architecture similar to fully associative cache memory, as shown in figure 2. The minterms are stored as TAG information, the data field will hold the corresponding minterm code. A positive comparison, called compare hit, is marked in the Hit-field.

No.	TAG (3-valued address)	Data (3-valued code)	Hit
No.	TAG (3-valued address)	Data (3-valued code)	Hit
.....			
No.	TAG (3-valued address)	Data (3-valued code)	Hit

Fig. 2. Structure of fully associative memory cells, used as stage-1-memory

A difficulty arises, if the minterms contain unbounded variables, coded as ‘don’t care’ (DC). The comparison must be performed for all stored minterms, and all compare hits must be marked. In summary, there might be several hits per partial minterm memory (equivalent to a row in figure 1a), and all partial hits must be compared again to extract all total hits.

It will be advantageous to use normal structured RAM arrays as minterm memory. Each of these RAMs will be addressed by a partial input vector, and for all DC-parts of the minterm, the minterm code is stored on both addresses. If the data bus of the RAM array exceeds the bit width necessary for storing the code, more bits to code the context or other information might be stored in addition to show invalid conditions etc.

Any normal RAM architecture will not be capable to communicate a data miss, therefore a specialised minterm code must be used to provide stage 2 with this information that the minterm is not stored inside. This is necessary to receive completeness inside minterm coding.

Figure 3 shows partial minterms, first mapped to a Tag-RAM (b) with Don't-Care-comparison, then mapped to a conventional RAM (c). Please note that there is no need to store partial minterms consisting only of ‘Don't-Cares’ (DC, '-') in the Tag-RAM, because these partial minterm are matching any bit pattern of the input vector. The partial minterms no. 3 and 10 as well as no. 5 and 6 lead to the same bit pattern

and need only to be stored once. Therefore code 2 in figure 3 b) indicates that the first part of the input vector matches minterm 3 **and** 10, and the code 3 indicates the minterms 5 **and** 6.

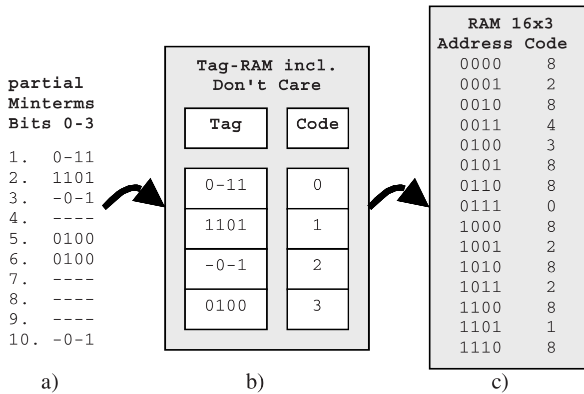

Fig. 3. Mapping partial minterms to RAM a) partial minterms b) Mapping to Tag-RAM c) Mapping to conventional RAM

When the tags are mapped to a conventional RAM, every address of this memory, which binary representation matches the bit pattern of a 3-valued tag, stores the appropriate minterm code. The address “0011” of the RAM matches the tags of the minterms 1 and 3, so the new code 4 is stored here to indicate the occurrence of these partial minterms. All codes of 5–8 represent no matching minterm and may be used for context information, e.g. illegal input vector.

The RAM structure in stage 2 has to use DC coding equivalent to stage 1. This incorporates that comparison has to be provided including this case. Again, using conventional RAM means that the DC codes are decoded to all addresses storing the corresponding data inside stage 2. The RAM of this stage is addressed by the minterm combinations of stage 1. If a Tag-RAM is used in stage 2, every 3-valued tag represents a combination of minterms, which are present at the input lines and detected by stage 1. The OR-combined output-vectors of all minterms, which are represented by one combination, must be stored as the appropriate value.

Again several addresses might match the same minterm combination. To provide the parallel capacity, the memory of stage 2 is pipelined (figure 4): The resulting minterm code of stage 1 addresses the first pipeline stage of the stage-2-RAM, called combination-RAM. This RAM contains an index for every valid minterm code vector, and this index addresses the second pipeline stage RAM storing the output variables for the minterm combination. This RAM is called output-vector-RAM, and the scheme enables the mapping of different minterm combination, resulting from DC digits, to the same output value.

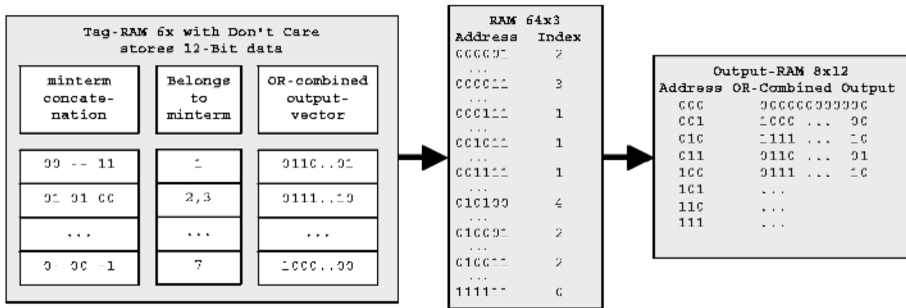

Fig. 4. Pipelining of combination-RAM and output-vector-RAM in stage 2

As any single RAM stores only a single index, the output-vector-RAM must hold all output vector values, combined by a logical OR. This implies that the RAM must hold all possible values of the function. As this results in exponential growth of RAM size, the combination- and output-vector-RAM are segmented in parallel parts, and the results are combined using the OR-operation of stage 3.

If different minterm combinations create the same output-vector, these combinations are mapped to the same output value too. Note that in the shown example the minterm combination "111111" belongs to no minterm-combination, the appropriate index leads to the output vector "000000000000". The indices 5–7 are not used here.

To use as much capacity as possible of the segmented RAM, a configurable crossbar-switch is used to connect stage 1 and stage 2. This switch maps the minterm code vector to the address inputs of the combination-RAM. Non-used data lines might be used for additional information as context, visualisation of invalid codes etc., as already mentioned.

The third stage contains the OR-operation, as discussed before. To use the inverted version of the sum-of-products structure, a configurable exclusive-OR-operation (XOR) is included in this stage. The last operation uses the contents of a stored bit vector to invert the corresponding output bits. This results in the sample architecture in figure 5.

The sample architecture is shown for 12 input- and 12 output-lines. To store a complete Boolean function with 12 input- and output-variables in RAM, 12 bits of data must be stored for all 4096 possible input-combinations. Therefore an amount of 6144 bytes is necessary for complete storage in a LUT-architecture.

The amount of memory to configure the sample architecture sums up to:

3x minterm-RAM 16x4	24	Bytes
3x combination-RAM 64x4	96	Bytes
3x output-vector-RAM 16x12	72	Bytes
Crossbar-Switch configuration	18	Bytes
Inverting register 12x1	1,5	Bytes
Sum	**211,5**	**Bytes**

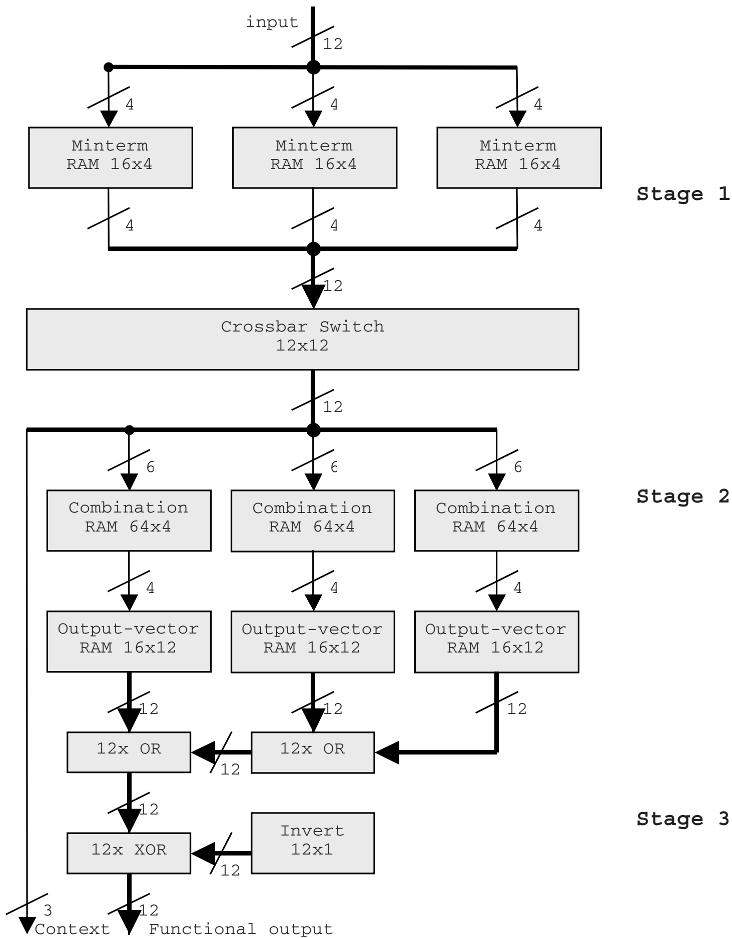

Fig. 5. Sample architecture for a $(0,1)^{12} \rightarrow (0,1)^{12}$ function

Of course this architecture is not capable of containing all function. The impact of the minimising, partitioning and mapping algorithms on the results and the usability will be great and is subject to future work. At this point, the architecture should be introduced as one possible architecture to implement GCA.

3 Mapping of a GCA on This Architecture

3.1 Sample Implementation of the Architecture

It is assumed for the discussion inside this chapter that all binary coded states of a GCA at t_{n-1} are the input to the next cycle t_n when the GCA computes the next state. In this case, the GCA might be mapped to the introduced architecture. To store the

actual state, an additional set of binary valued registers must be added to the architecture. These registers decouple the actual state from the new computation. In most cases, edge-sensitive registers will be used.

The unused data lines, originating from stage 1, may be used as context signals. If the architecture provides more than one layer of RAM in the crossbar-switch, the stage-2-RAM and the inverting register, a context switch can be performed within 1 cycle. This leads the way to multi-context implementations of a GCA. Figure 6 shows an implementation for realising GCA with 64 state bits.

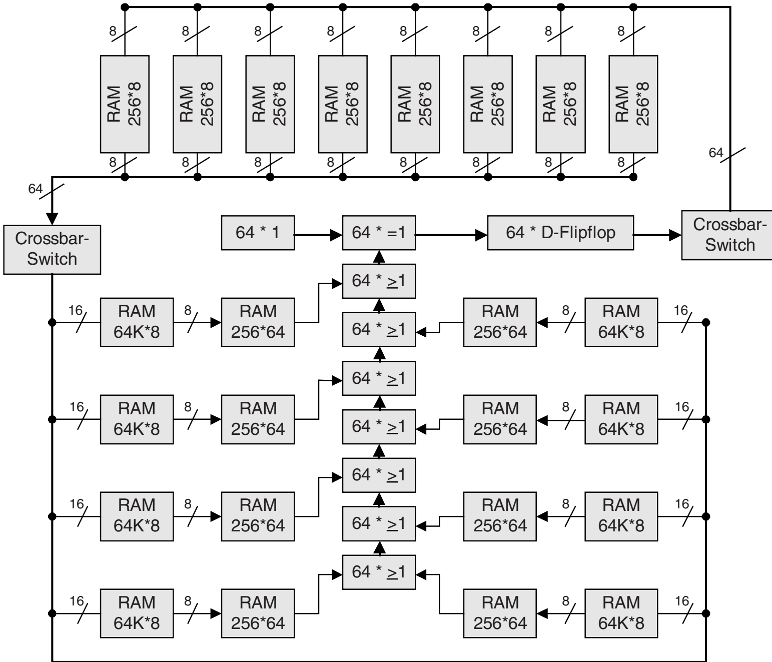

Fig. 6. Sample architecture for implementing GCA

The computation will use one cycle per step in normal cases. As the capacity of the circuit is limited by the combination-RAM, the application might be to large for implementation in the architecture. If all methods of minimising and mapping fail, computation can be split up in partial steps. This results in more then one cycle per step to receive the states of the GCA.

The memory demand per memory layer for this implementation is assumed by the following table:

8x minterm-RAM 256x8	2	KBytes
8x combination-RAM 64Kx8	512	KBytes
8x output-vector-RAM 256x64	16	KBytes
2xCrossbar-Switch configuration	1	KBytes
Inverting register 64x1	8	Bytes
Sum	**531**	**KBytes**

3.2 One Example: Mapping a 4-Bit-CPU on the Implementation

This chapter introduces a simple 4-Bit-CPU, implemented as GCA and mapped on the new hardware architecture. The CPU has a simplified instruction set and consists of only a few internal registers.

Internal Registers

Address, 8-Bit	The memory-address-register. The content of this register is directly mapped to the address-lines of the processor and vice versa
Data, 4-Bit	The memory-data-register. The content of this register is mapped to the data-lines of the processor and vice versa
Accu, 4 Bit	Internal register, where all calculations occur
Code, 4-Bit	Instruction register. This register holds the opcode of the current instruction during execution-time
PC, 4-Bit	Program counter. The content of this register represents the memory-address of the current instruction

Instruction set:

bne Address	Branch not equal. Jump, if content of Accu is not 0
beq Address	Branch equal. Jump if content of Accu is 0
lda Address	Load Accu with the data from Address
sta Address	Store the content of Accu at the given Address
and Data	Calculate Accu and Data and store the result in Accu
or Data	Calculate Accu or Data and store the result in Accu
add Address	Calculate Accu + data from Address and store the result in Accu
sub Address	Calculate Accu - data from Address and store the result in Accu

An unconditional jump can be achieved by a bne- followed by a beq-instruction. There is neither stack processing or subroutines nor a carry flag. The instruction set consists of 8 instructions and can be coded in a 3-bit instruction code. Two operand formats are used: 8-bit addresses and 4-bit data.

3.2.1 Mapping the CPU to a GCA

The GCA consist of 7 named calls. The cells 'Address', 'Data', 'Accu' and 'Code' correspond directly to the registers. The additional cells 'RW', 'Reg0' and 'Reg1' are needed to realise the program flow of the CPU. The cell 'Address' has 256 states, the cell 'RW' two, the cell 'Code' eight, and all other cells have 16 different states. The states of 'Address', 'RW' and 'Data' correspond to the I/O-lines of the CPU, and if these lines change, the states of the corresponding cells change too. Thus the states of all cells of the GCA are coded in 28 bits.

The instructions will be processed in several phases. Each phase needs another configuration of the GCA. This includes different functionality of the single cells as well as different communication links between the cells. These reconfigurations are directed by the context lines, which select different levels of memory to change the behaviour of the GCA. The following phases are used:

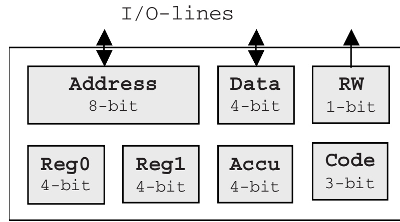

Fig. 7. GCA to realise the 4-bit CPU

Instructions	OpCodes	Phases
bne, beq	000, 001	Fetch OpCode Fetch Addr#1 Set Address regarding to Accu
lda	010	Fetch OpCode Fetch Addr#1 Save&Set Address Set Accu and restore Address
sta	011	Fetch OpCode Fetch Addr#1 Save&Set Address and store Accu Restore Address
and, or	100, 101	Fetch OpCode Calculate Accu
add, sub	110, 111	Fetch OpCode Fetch Addr#1 Save&Set Address Calculate Accu and restore Address

Every instruction starts with the 'Fetch OpCode'-Phase. After completing the instruction, the next phase is 'Fetch OpCode' again. The sequence of phases, determined by the OpCode, is coded in the cyclic change of the context lines determined by the content of the memory arrays of the first stage of the architecture and the configuration of the crossbar switch between stage one and two. All context-states are referred with the name of the corresponding phase in the further text. The last phase of every sequence must leave the context lines in the state to select the configuration of the 'Fetch OpCode'-phase again. To allow the same configurations to be used by different instructions and in different sequences of configurations, the OpCode is stored in the cell 'Code'. This cell is not part of the registers of the CPU.

Some of the phases used to process the instructions are described in detail below:

Fetch OpCode (Fig. 8a)
During this phase the state of the cell 'Address' represents the current Program Counter (PC), and the content of the cell 'Data' represents the current OpCode read from RAM. The cell 'RW' is in state 'read', the state of cell 'Accu' represents the current content of the register 'Accu'. The Program Counter is increased by one, the OpCode is stored in the cell 'Code'. The context lines change to context 'Calculate Accu' or to context 'Fetch Addr#1', according to the OpCode.

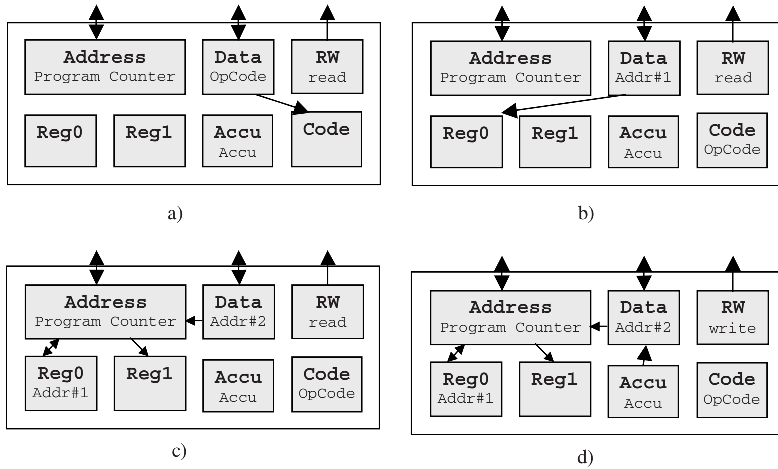

Fig. 8. Instruction execution phases a) Fetch OpCode b) Fetch Addr#1 c) Save&Set Address d) Save&Set Address and Store Accu

Fetch Addr#1 (Fig. 8b)
The cell data, linked to the state of the data-lines of the CPU, contains the first part of an address used by the current instruction. This partial address is stored in the state of cell 'Reg0'. The Program Counter is increased by one. According to the state of cell 'Code', the next context is 'Save&Set Address' or 'Save&Set Address and Store Accu'.

Save&Set Address (Fig. 8c)
This phase follows directly after the phase 'Fetch Addr#1'. The cell 'Address' represents the current PC, the cell 'Reg0' the first part of the address to be read, the cell 'Data' represents the second part of this 8-bit address. In this phase, the state of the cell 'Address' will be copied to Reg0 and Reg1, while the states of the cells 'Reg0' and 'Data' give the new state of 'Address'. According to the state of cell 'Code' the next context is 'Calculate Accu and restore Address' for each of the instructions 'lda', 'add' and 'sub'.

Save&Set Address and Store Accu (Fig. 8d)
As in the phase 'Save&Set Address', this phase saves the current PC in the cells 'Reg0' and 'Reg1' and sets the new Address. Unlike the last phase, the state of the cell 'Accu' is copied to the cell 'Data' and the cell 'RW' changes its state from 'read' to 'write'. In this way, the content of the CPU-register 'Accu' is stored to the given address. The next phase is 'Restore Address'.

3.2.2 Mapping the GCA on the Circuit

The states of all cells of the GCA request 28 bits for coding. This architecture is mapped on a circuit consisting of 8 16x4-RAMs minterm-memory in the first stage and 4 8x8-RAMs combination-memory as well as 256x32-RAMs output-memory in the second stage.

The cells of the GCA are mapped to the following bit positions in the state vector:

```
Accu        bit 4..7
Code        bit 8..10
RW          bit  11
Reg0        bit 12..15
Reg1        bit 16..19
Data        bit 20..23
Address     bit 24..31
```

The phases of the instruction processing consist of 6 different Boolean functions, which have to be mapped on the architecture in different combinations. These are:

– Increment of an 8-bit value, used to increment the PC
– Copy one bit, used to copy registers. This function is used in parallel up to 20 times to copy the PC, the Accu and the new Address
– 4x Calculation of a 4-bit-value from 2 4-bit-values, for the operations 'add', 'sub', 'and', 'or'.

All these Boolean functions may be executed in parallel, as long as they set different output bits. The phases 'Fetch OpCode' and 'Save&Set Address and Store Accu' are explained in detail now.

Fetch OpCode, Context '0000'

This phase combines the increment function with copying 4 bits. The context lines are set to 'Calculate Accu' or 'Fetch Addr#1', according to the current OpCode.

Figure 9 shows the mapping of the cell states to the minterm-memories MIN0 to MIN7. Only the RAMs MIN0, MIN1, MIN2, MIN7, COMB0, OUT0, COMB1 and OUT1 are used. MIN0 and MIN1 are linked to COMB0, which is completely filled, while bit 0..2 of MIN2 are linked to COMB1. The data-lines of MIN7 are linked directly to the context-lines of the circuit.

The data-path from COMB0 leads to the output-memory OUT0, where the next address bits are generated and where the read-write-state is set to 'read'. The data-value of the bits 0..2 from MIN2 representing the OpCode is stored via OUT1 into the GCA as the state of the cell 'Code'.

The context lines are set to '0001', coding the context 'Fetch Addr#1', for any instruction except 'and' and 'or'. If the OpCode-Value is 'and', context is set to '0010' for 'Calculate And' and to '0011' for 'Calculate Or'.

After execution of 'Fetch OpCode', the cell 'Address' keeps the new PC, the cell 'Code' stores the OpCode, the cell 'RW' in in the state 'read' and the context is either 'Fetch Addr#1', 'Calculate And' or 'Calculate Or'.

Save Address and Store Accu, Context '0110'

This phase is only used when the instruction 'sta' is processed. The cell 'Address' stores the current PC, the cell 'Data' stores the second part of the storage address (Addr#2), the cell 'Accu' stores the current content of the accumulator register and the cell 'RW' is in state 'read'.

This phase uses the RAMs MEM0-MEM5, COMB0, COMB1, OUT0 and OUT1 completely and COMB2 and OUT2 partially. The cell 'RW' is set to state 'write', the cell 'Data' stores the content of the accumulator, and the cell 'Address' stores the address, where the content of 'Data' must be written. The context lines are not set, the next context-value is '0000', the 'Fetch OpCode'-Phase.

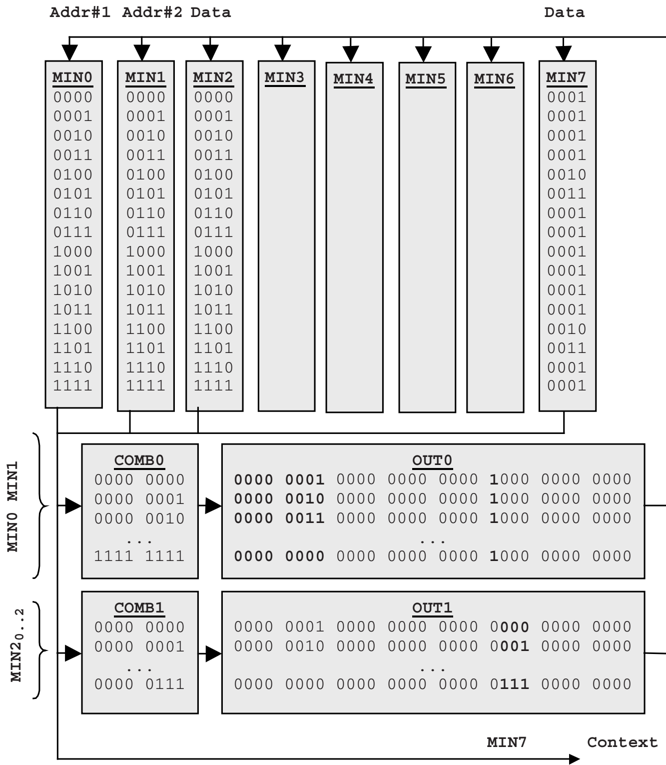

Fig. 9. Fetch OpCode

4 Conclusion and Outlook

Starting with the problem to map a global cellular automaton on a physical circuit, where the cells could possibly but not likely have the full complexity and are linked to every other cell, this paper has introduced a new concept of realising Boolean functions with many input- and output variables. A GCA mapped to one Boolean function avoids the costs of the communication between the separated cells, which are mapped to the complexity of the single resulting Boolean function. This approach shifts the problem of communication to the theory of minimising Boolean functions and to the design of algorithms. If a Boolean function would be still too complex to fit on the architecture, the processing can be divided up in several independent steps with less complex functions.

The example of the CPU, mapped to the new architecture approach, visualises the possibilities of this way to realise global cellular automata. At the same time it shows the limitations: the majority of the Boolean functions of the CPU like copying or incrementing the address are completely defined functions where every possible input combination and every possible output combination of variables may occur. For this

reason no simple way to map these functions on the architecture and save memory and complexity at the same time could be found.

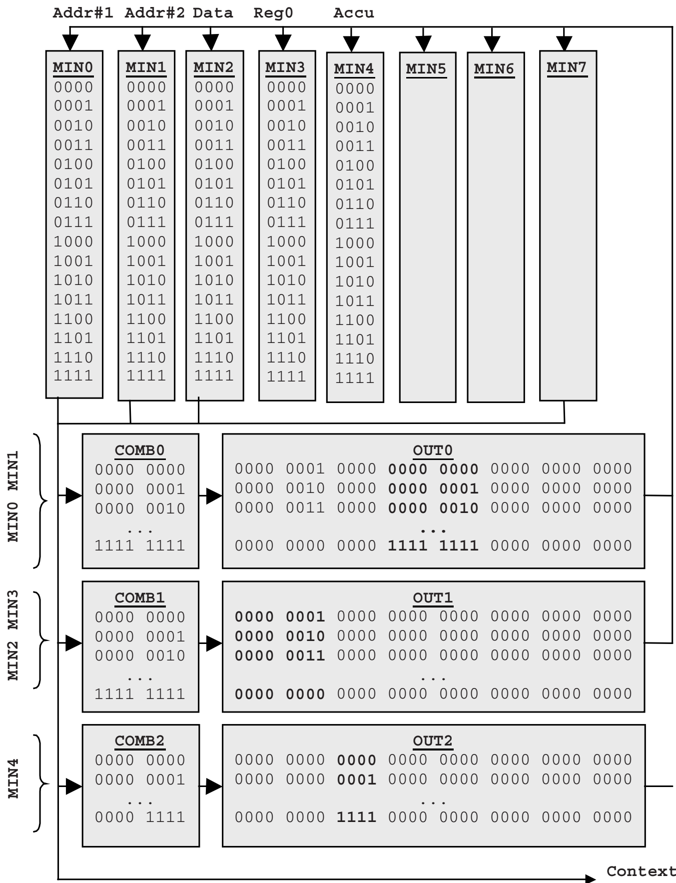

Fig. 10. 'Save Address and Store Accu'

The results of certain considerations show that a lot of improvement is possible by special algorithms and software. This will be the topic of further research. Another topic should be the exploration and improving of the architecture itself. Because only part of the first and second stage RAM was used, it could be advantageously to provide a circuit for an automaton with wider status-register and more status bits, compared to the input lines the first stage can handle. Another way to improve the circuit could be the introduction of a special context-RAM that handles context sequences to be processed.

In summary it can be concluded that this kind of approach gives new possibilities and chances worth for future considerations.

References

[1] Rolf Hoffmann, Klaus-Peter Völkmann, Wolfgang Heenes, ”Globaler Zellularautomat (GCA): Ein neues massivparalleles Berechnungsmodell”. Mitteilungen – Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen, ISSN 0177-0454 Nr. 18, S. 21–28 (2001) (in German language).
[2] R.K Brayton et.al., ”Logic Minimization Algorithms for VLSI Synthesis”. Kluwer Academic Publishers, 1984.
[3] Mike Trapp, “PLD-design methods migrate existing designs to high-capacity devices”. EDN Access, Febr. 1994.
[4] Wolfgang Heenes, Rolf Hoffmann, Klaus-Peter Völkmann: ”Architekturen für den globalen Zellularautomaten“. 19th PARS Workshop, March 2003, Basel (in German language). http://www.ra.informatik.tu-darmstadt.de/publikationen/pars03.pdf