ToPoliNano: NanoMagnet Logic Circuits Design and Simulation

Marco Vacca, Stefano Frache, Mariagrazia Graziano$^{(\boxtimes)}$, Fabrizio Riente, Giovanna Turvani, Massimo Ruo Roch, and Maurizio Zamboni

Dipartimento di elettronica e telecomunicazioni, Politecnico di Torino, Turin, Italy
mariagrazia.graziano@polito.it

Abstract. Among the emerging technologies Field-Coupled devices like Quantum dot Cellular Automata are one of the most interesting. Of all the practical implementations of this principle NanoMagnet Logic shows many important features, such like a very low power consumption and the feasibility with up-to-date technology. However its working principle, based on the interaction among neighbor cells, is quite different from CMOS circuits. Dedicated design and simulation tools for this technology are necessary to further study this technology, but at the moment there are no such tools available in the scientific scenario.

In this chapter we present ToPoliNano, a software developed as a design and simulation tool for NanoMagnet Logic, that can be easily adapted to many other emerging technologies, particularly to any kind of Field-Coupled devices. ToPoliNano allows to design circuits following a top-down approach similar to the ones used in CMOS and to simulate them using a switch model specifically targeted for high complexity circuits. This tool greatly enhances the ability to analyze and optimize the design of Field-Coupled circuits.

1 Introduction on Simulation of Complex NML Circuits

Among the emerging technologies NanoMagnet Logic (NML) is one of the most intriguing. In this technology single domain nanomagnets with only two stable states are used to represent the logic values '0' and '1' [1,2], as shown in Fig. 1. They represent a particular application of the Quantum dot Cellular Automata [3] idea, and more generally of the Field-Coupled principle, where the computation is performed by the interaction of neighbor cells [4–7]. Molecular QCA is the other main implementation of the Quantum dot Cellular Automata principle [8,9], which relies on complex molecules to represent the digital values [10]. The specific advantages of NanoMagnet Logic are represented by low power consumption [11], the possibility of combining memory and logic in the same devices, high radiation resistance and, not less important, the possibility to fabricate circuits with up-to-date technology [12,13].

In this technology logic circuits can be fabricated placing cells on a plane [14]. Signal propagation and logic computation are obtained through magnetic coupling among neighbor cells [15,16], because magnets align themselves in order

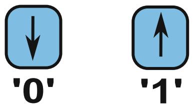

Fig. 1. Single domain nanomagnets are used as basic cells. At the equilibrium only two stable states are possible.

to reach the minimum energy state. The alignment is antiferromagnetic (every element is in the opposite state of its neighbors) if magnets are aligned horizontally, while the alignment is ferromagnetic (every element is in the same state of its neighbors) if magnets are aligned vertically [17]. However, the magnetic field generated by a magnet is not sufficient to cause a state alteration in its neighbors. To switch magnets from one state to the other it is necessary to use a mechanism called clock [18]. The behavior is depicted in Fig. 2. Magnets are forced in an unstable (RESET) state through an external mean, like a magnetic field [19, 20]. When the magnetic field is removed magnets realign with a domino-like effect following the input element. With this mechanism signals propagate correctly through the circuit. As well as a magnetic field, other systems can be used to clock circuits, like STT-current coupling [21] or an electric field [22].

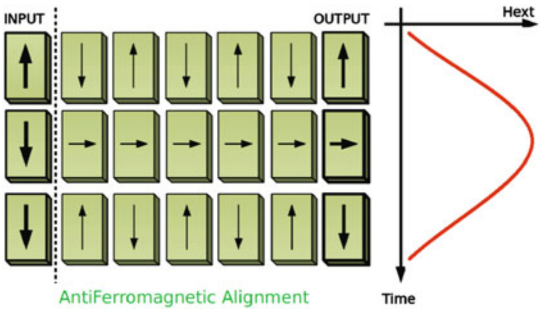

Fig. 2. Clocking mechanism for NML logic. Magnets are forced in an intermediate unstable state through an external mean, like a magnetic field that in a particular portion of time reaches a maximum appropriate value.

The RESET state is unstable. If too many magnets are cascaded some of them along the chain will switch in the wrong state due to external influences, like thermal noise [23, 24]. To have a safe signal propagation no more than 5 magnets should be cascaded [23]. As a consequence a multiphase clock system is required. Circuits are divided in small areas, called clock zones. Each zone is made by a limited number of magnets. Every clock zone requires the application of a different clock signal, like shown in Fig. 3(A) where three clock signals with a phase difference of 120° are used. As depicted in Fig. 3(B), when magnets of a clock zone are in the SWITCH state (the magnetic field is slowly removed) they see from one side magnets that are in the HOLD state (no magnetic field applied). Magnets in the HOLD state can assume a value of logic '0' and '1' so they are seen as an input by magnets that are switching. At the same magnets near the other side of the switching zone are in the RESET state (magnetic field is applied) and therefore they have no influence on the switching ones. Figure 3(B) shows the circuit time evolution when a multiphase clock system is used.

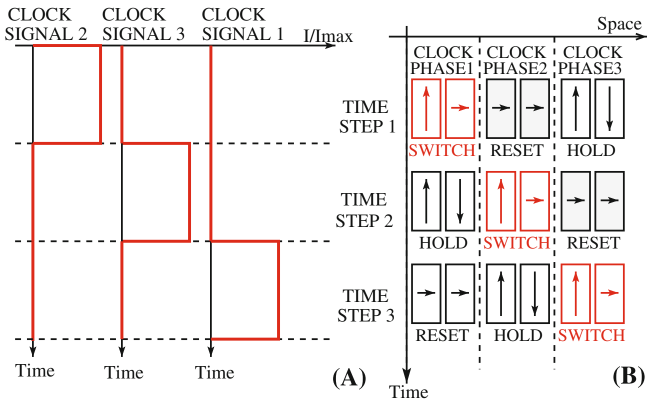

Fig. 3. 3-phase clock. Circuits are divided in areas, called clock zones, made by a limited number of magnets. (A) Three clock signals with a phase difference of 120° are selectively applied to clock zones. (B) When magnets of a clock zone are in the SWITCH state, magnets on the left are in the HOLD state and act like an input, while magnets on the right are in the RESET state and have no influence on the switching magnets.

To design circuits, clock zones must be arranged following a proper layout. Moreover the layout must take into account the constraints related to the technological fabrication of the clock generation network. For example the magnetic field is normally generated by a current flowing through a wire placed under the magnets plane [25] (Fig. 4(A)). With this clock mechanism the clock zones layout is made by parallel stripes (Fig. 4(B)). Each stripe corresponds to one of the clock wires used to generate the various clock signals [13]. While this layout was developed for the magnetic field clock, and other clock systems can have different layouts, it has the advantage to synchronize signals propagation. Thanks to the multiphase clock the circuit is intrinsically pipelined, that means every group of 3 consecutive clock zones has a delay of 1 clock cycle. As a consequence, if the length of input wires of a logic gate is not the same, signals will have different propagation delay and propagation errors will occur. This problem is called "layout=timing" [26,27]. With the clock zones layout shown in Fig. 4(B) the length of every input wire of any gate inside the circuit is always equalized, solving therefore the "layout=timing" problem. For this reason this clock zones layout is chosen as a reference regardless to the clock mechanism used.

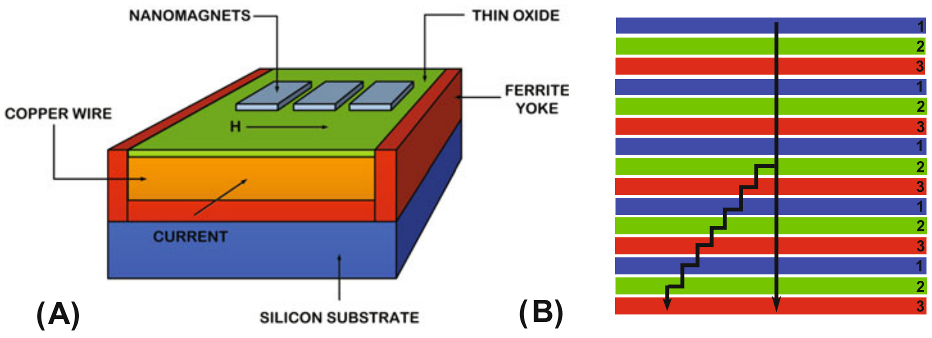

Fig. 4. (A) A magnetic field can be generated by a current flowing through a wire placed under the magnets plane. (B) Clock zones layout is made by parallel strips that follows the wires used to generate the magnetic fields.

To simulate QCA circuits a dedicated simulator, called QCADesigner, was developed [28]. Unfortunately QCADesigner does not support magnetic circuits. To simulate NML circuits two paths can be followed. First of all low level magnetic simulators, like OOMMF [29], NMAG [30] or [31], can be used. Low level simulators allow to obtain the most accurate simulation, but they are very slow and only small circuits can be simulated due to the high memory usage.

As a second option circuits can be studied using a RTL model [32]. The idea is to describe using the VHDL language a CMOS circuit that behaves exactly like its NML counterpart. For example, starting from the NML circuit of Fig. 5(A), its RTL model can be built as shown in Fig. 5(B) and then described with VHDL language. Registers are used to model the propagation delay. This is possible due to the intrinsic pipelined behavior of the technology. Ideal logic gates without delay are used to model the logic function. The logic gate set available in this technology is based on majority voters [16], AND/OR gates [33] and crosswires [13], that are particular blocks that allows to cross two wires on the same plane. At every register one of the three clock signals shown in Fig. 5(C) is then applied. In this way complex circuits can be easily described using VHDL and simulated using the powerful CAD tools available in CMOS technology, like Modelsim [34]. We applied this model successfully designing complex NML circuits in [26,27,35].

2 ToPoliNano

Both simulation mechanisms available have their flaws. The situation is summarized in Fig. 6.

To have the most accurate results physical simulators are required, but they can be used only on very small circuits due to their computational requirements. At the same time, while the RTL model is a powerful tool that allows fast description and simulation of complex circuits, it gives only estimations of real circuits performance, because a lot of informations on the circuit layout are

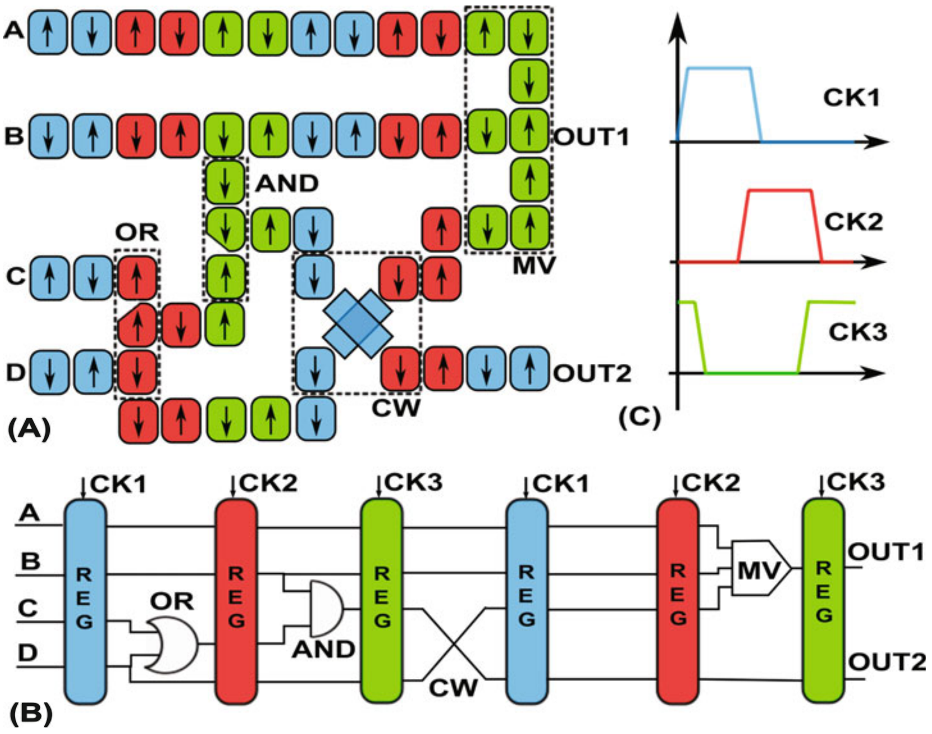

Fig. 5. VHDL model for NML circuits. (A) Example of NML circuit. (B) RTL model described with VHDL. Registers are used to simulate the propagation delay while ideal logic gates are used to model the logic function. (C) Clock signals applied to each register.

missed. We have therefore created our own tool, ToPoliNano, Torino Politecnico Nanotechnology tool [36], a tool targeted to design and simulation of Nano-Magnet Logic circuits. ToPoliNano emulates the top-down approach used in CMOS design, where circuits are described using the VHDL language and the layout is automatically generated. Circuits can be simulated and important information on the circuit behavior and the power consumption can be extracted, knowing exactly the circuit area and the precise placement of every element [37]. Mostly important the open and modular structure of the software allows to easily integrated others emerging technologies, like we have done with Silicon Nanowires NanoPLA [38,39], making it the ideal platform for the study of emerging technologies.

2.1 Tool Overview

ToPoliNano has been developed in C++ and is built around the idea to give to researcher the possibility to design NML circuits following the same top-down

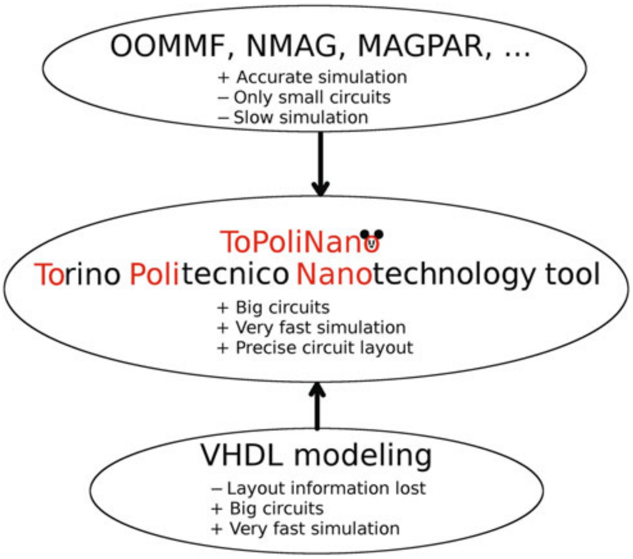

Fig. 6. Simulation of NML circuits. While physical simulators provide the most accurate simulation they can be used only on small circuits. At the same time, VHDL model gives inaccurate results for the loss of information regarding circuits layout. ToPoliNano was created to overcome these problems and to provide a tool that allows to have both accurate results and fast simulation of complex circuits.

methodology used in CMOS circuits. This means to describe circuits of any kind of complexity using VHDL language, to automatically generate the layout and to fast simulate the obtained circuit. For this emerging technology there are no tools available to perform these analysis, therefore it has been necessary to design a completely new system. The structure of Topolinano is shown in Fig. 7.

- The **Logic Synthesizer** is the first block encountered in the traditional CMOS design flow. Starting from a generic VHDL description it translates it on a specific logic gates set. In this case it takes an entry VHDL file and it generates another VHDL file with a structural description, that means the circuit is described only using the set of gates available in this technology (majority voter, and, or, inverter). The logic synthesizer is still partially in development.
- The **Parser** takes the structural VHDL file generated by the logic synthesizer and creates an in-memory representation of the circuit itself. The internal description is based on a hierarchical graph to efficiently handle the circuits in terms of both time and memory occupation on the host computation machine. The parser is complete and it is throughly described in Sect. 3.
- The **Place & Route** takes the graph generated by the parser and automatically creates the circuit layout. This block is still in development, as up to now it can handle only combinational circuits of any complexity but not

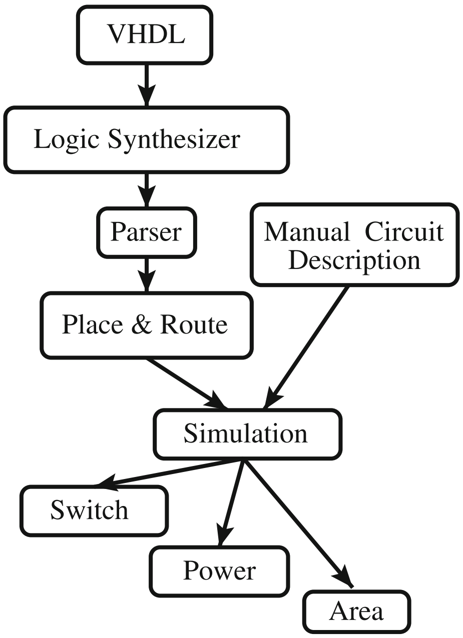

Fig. 7. ToPoliNano structure. Circuits are described through VHDL, a logic synthesizer maps the circuit on the technology library available and a parser generates the in-memory description. The layout can be generated automatically or manually, circuits are then simulated obtaining data on the circuit behavior, the area occupied and the power consumption estimation.

sequential. The circuit layout is based on the clock zones layout described in Sect. 1 and shown in Fig. 4(B). Section 4 provides more details on the Place & route in its current state.

- It is also possible to **Manually Describe** circuits with a full custom approach. This possibility is granted for two important reasons, because the Place & Route block is still in development and because, no matter what level of development the Place & Route block will reach, in certain cases the hand of a designer is requested to reach the maximum level of optimization. Up to now circuits can be described either directly writing the code that describes the circuit or using external vectorial graphic editors and then importing the circuit in ToPoliNano. Further details are given in Sect. 6.
- Once the circuit layout is generated, a **Simulator** is used to verify the correct behavior of the circuit. The algorithm used is based on a behavioral model extracted from low level simulations. This tool is designed for high complexity circuits (million of magnets), so only a behavioral algorithm allows a fast enough simulation. However, since the model is based on physical simulations it still gives accurate results. More details on the simulation algorithm are provided in Sect. 5.

– After the simulation the **Output Generator** allows to calculate the circuit area and to estimate the power consumption considering a magnetic field clock. More details can be found in Sect. 7.

3 Parser

One of the most common uses of a parser in computing is as a component of a compiler or an interpreter. This component usually parses the source code of a programming language in order to create an internal representation. The parsing process (i.e. syntax analysis) consists in the analysis of an input sequence to determine its grammatical structure with respect to a given formal grammar. The parsing process operates a transformation of the input text into a data structure (a tree in the present context), suitable for later processing. The data structure must be such to capture the implied hierarchy of the input, and a tree certainly is suitable for this purpose. The typical operation of parsers is in two stages: first, it identifies the meaningful tokens in the input. Then, it builds a data structure out of the tokens (Fig. 8).

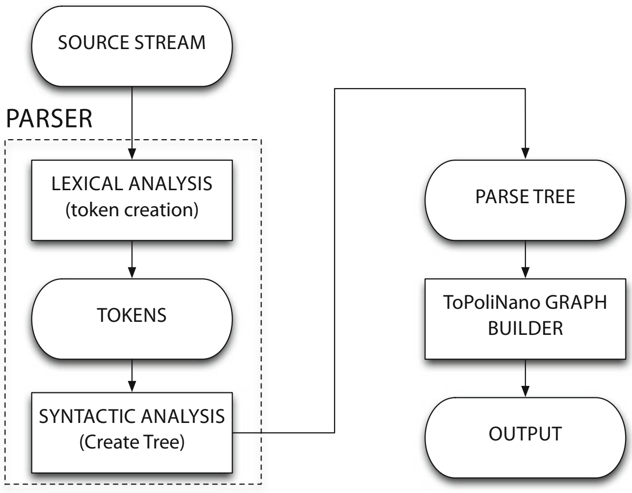
SOURCE STREAM
PARSER
LEXICAL ANALYSIS (token creation)
TOKENS
SYNTACTIC ANALYSIS (Create Tree)
PARSE TREE
ToPoliNano GRAPH BUILDER
OUTPUT

Fig. 8. Overview of the parsing process.

In ToPoliNano the use of the parser is related to the need to input circuit descriptions by means of a Hardware Description Language (HDL). In particular, the VHDL language is currently supported.

3.1 Lexical Analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program which performs the lexical analysis is called lexer or scanner. The input characters stream is split into meaningful symbols defined by a grammar. They define the set of possible character sequences used to form individual tokens. The term token, in the present context, refers to an abstraction for the smallest unit of VHDL code that is convenient when describing the syntax of the VHDL language. A token is a string of characters, categorized according to the rules as a symbol (e.g. identifier, number, comma, etc.). Starting from this time on, the interpreted data may be loaded into data structures for general use: in ToPoliNano data is used to build an internal representation of the circuit by means of a graph.

3.2 Syntactic Analysis

Syntactic analysis has the objective to determine the structure of the input stream and to build the data structure. Token are fed to the syntactic analyzer and, as output, in case of a tree-based data structure, one would get a node for each element. Basic elements are represented by leaf nodes, and other elements by composite nodes. This stage basically checks that the tokens form an allowed expression.

3.3 The Parse Tree

According to [40], a parse tree is an ordered and rooted tree that represents the syntactic structure of a string, text file, source code written in a given programming language according to some formal grammar. A parse tree for a source code is called Abstract Syntax Tree (AST). The syntax is 'abstract' in the sense that it does not represent every detail that appears in the real syntax.

3.4 Parsing Expression Grammar Definition

Parsing Expression Grammars (PEGs) are formal grammars that allow to describe a formal language in terms of a set of rules for recognizing tokens. The grammar encapsulates a set of rules, primitive parsers and sub-grammars. After being defined, rules can be used as parser components in more complex expressions in order to form a grammar. Grammar is basically a container for one or more rules allowing to encapsulate more complex parsers. A grammar has the same template arguments as a rule.

When the Parsing Expression Grammar to be defined is complex and nested, as in ToPoliNano, where we must define a PEG for VHDL93 language specifications, it may be useful to build user-defined parser components. The grammar for VHDL structural descriptions can be built starting from the grammar definition of the unique constructs, i.e. constructs that do not include any other.

3.5 VHDL Grammar

A VHDL file for structural description includes an Entity Declaration and an Architecture Body (in what follows, the libraries included at the begin of every VHDL file are not considered):

```
VHDL = EntityDeclaration
       > ArchitectureBody;
```

The Entity Declaration grammar could be written as:

```
EntityDeclaration = EntityKeyword > Identifier > IsKeyword
                    > ( GenericClause | *ascii::space )
                    >  PortClause
                    > EndKeyword > Identifier > lit(';');
```

Such a grammar may match a Generic Clause, if it is defined in the VHDL source file, otherwise zero or more white spaces, and a Port Clause every time an Entity Declaration is found.

The grammar for an Architecture Body is divided into two blocks:

– Architecture Declarative Part Grammar
– Architecture Statement Part Grammar

The custom grammar for the Architecture body could be written as:

```
ArchitectureBody = ArchitectureKeyword > ArchitectureIdentifier > OfKeyword
                 > EntityIdentifier > IsKeyword
                 > ArchitectureDeclarativePart
                 > BeginKeyword
                 > ArchitectureStatementPart
                 > EndKeyword >  ArchitectureIdentifier > lit(";");
```

In an Architecture Declarative Part there are one or more Component interfaces and zero or more Signal Declarations:

```
ArchitectureDeclarativePart = (+ ComponentDeclaration)
                              > (* SignalDeclaration);
```

The Component Declaration Grammar is very similar to the Entity Declaration, since the Component interface is the copy of the Entity interface. In an Architecture Statement there may be instantiated components with Generic Map and Port Map, assigned values to signals and generate well-patterned structures. These three statements can be found in any order and in any number within the Architecture Statement Part (here Generic Map and Port Map are joined for the sake of simplicity):

```
ArchitectureStatementPart = + ( GenericMapPortMap
                                | SignalAssignment
                                | GenerateStatement
                              );
```

In Fig. 9 the parser components have been defined, the grammars and the other parser components used are summarized.

Once the VHDL grammar has been defined, the need to store the information parsed from the VHDL projects arises. This information must be stored at parse time, and elaborated at a later time, possibly by other modules of the ToPoliNano

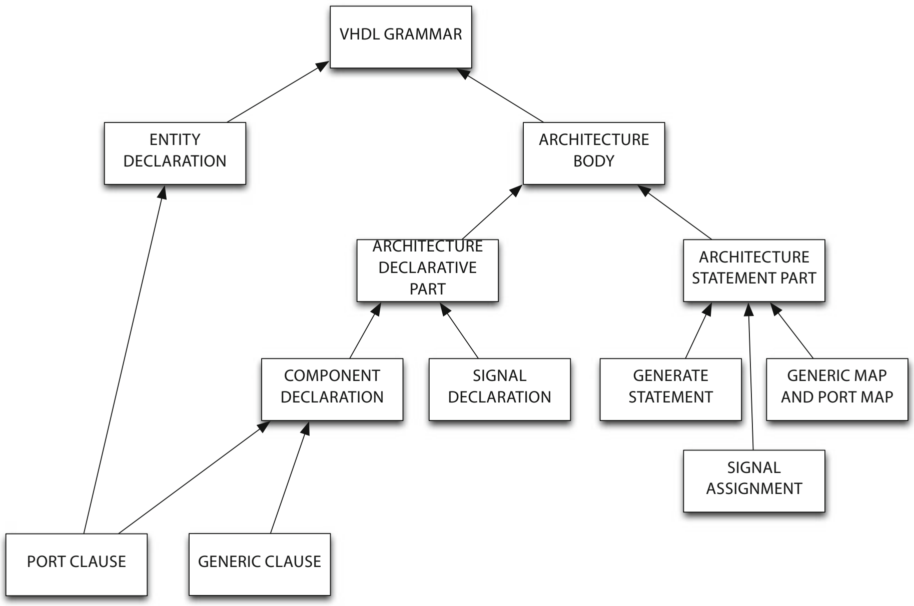

Fig. 9. Hierarchy of VHDL Grammar.

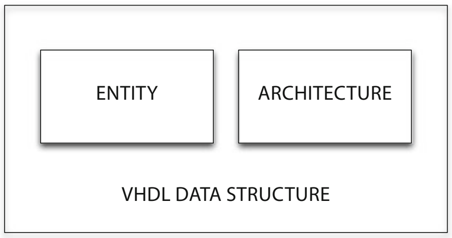

Fig. 10. VHDL Design Entity Data structure.

tool. Up to now, the Parser just recognizes data, but does nothing with it. A dynamic data structure is built and instantiate at runtime dynamic objects of certain classes just when they are needed, and populate them at parse time. The data structure used to store the parsed information from a VHDL design unit is a dynamic object of a main class called VhdlClass, which comprises pointers to an Entity object and to an Architecture object. It is shown in Fig. 10.

3.6 Intermediate Form Representation

Once all the VHDL design units have been parsed, a data structure that represents the digital circuit, component by component, has to be implemented. The idea is to create a hierarchical graph of nodes that represents the circuit and all

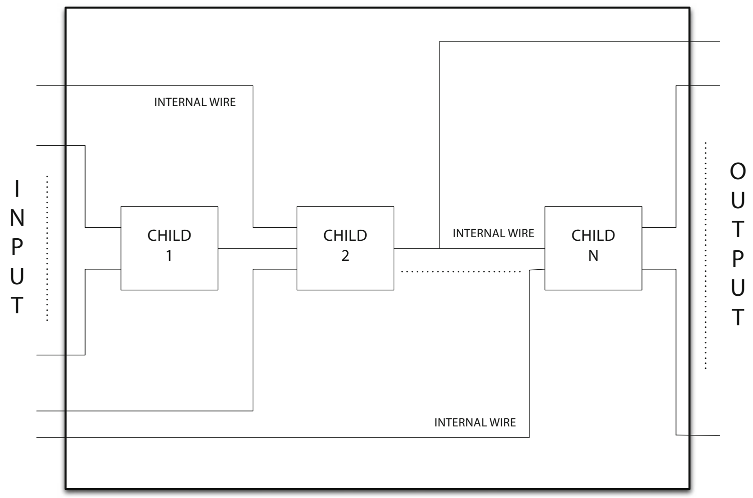

Fig. 11. Composite component.

its components; moreover, it has to capture how they are connected together. This graph is generated starting from the temporary data structure created during the VHDL code parsing. The graph must have as many nodes as the number of basic blocks plus the number of composite components in the circuit. All the nodes of the same grade represent the whole circuit seen at a certain level of abstraction. The lower the level, the higher the abstraction. The root has the maximum level of abstraction, because it depicts the circuit in just one block; all the graph leaves represent the basic blocks out of which the circuit is built. Every node contains all the information regarding a component (Input, Output, Bidirectional ports, internal Signals, Interconnections), and all of its children (if any) represent the components it is made of.

The basic element of the graph is an object of the abstract class Node. Two concrete classes are derived from the Node class: CompositeNode and LeafNode. Every node contains a set of Input, a set of Output, a potential set of Bidirectional I/O, a potential set of internal Signals, a potential set of child Nodes and the Interconnections. A Composite Node is a node that has at least one child. Composite nodes are used to represent the top level and all the intermediate level components. Composite nodes represent components like that shown in Fig. 11.

A Leaf Node is a node that does not have children. They are used to represent all the basic blocks in the circuit. All Inputs, Outputs, Bidirectional I/O and internal Signals can be represented by objects of the same class, i.e. the Wire class, because all of them are merely wires that carry the same kind of logic data. The class members are capable to differentiate from other Wires by means of a name within a design Entity and by means of a serial number within a set. The data structure built during the parsing phase is the starting point of the graph creation. It contains one object for each VHDL design unity. Starting from the output of the Parser, one has to calculate the hierarchy among the components in the circuit, then, for each component, to instantiate a Node and populate it.

The first step is to detect every instance of a component. For each VHDL design unity, we have to create as many objects as the number of components instantiated in the Port Map clauses. The objects in the VHDL design unities vector are copied and in case modified (if the component is instantiated with a generic interface different from its declaration) every time they are instantiated in a Port Map aspect. For this purpose a new object type is created in order to facilitate the work of the stage that instantiates the graph, that is an object of class Hierarchy. A vector containing all the composite and all the basic design unities is temporarily built.

Each object of class Hierarchy contains the following information:

- A pointer to the VHDL design unity object.
- The name of the Design Entity Declaration.
- The instantiation name, which is unambiguous among nodes with the same parent.
- A flag indicating if it is a basic block (Leaf Node).
- The index of the parent object in the vector.

The first item of the Hierarchy vector (position 0) contains the information about the top level (the root); it is the first item to be pushed into the vector. Then the population cycle starts (with only the top level in the vector). All the other items are pushed back in the vector during the cycle. For every design unity composed of some other components (i.e. for every composite), it takes the Port Map aspects, and, for each component instantiated, a new Hierarchy item is created, in which the VHDL design unity object with the correspondent Entity identifier, taken from the VHDL design unities vector, is copied and possibly modified, depending if it has a generic interface that differs from its declaration. Then, also the information about the name of the Entity, the instantiation name, the index of the parent object (which is merely the index of the FOR cycle) and the flag indicating if it is a basic block are inserted. After its population, the object is pushed back in the Hierarchy vector. The cycle ends when all the components in all the composite design unities are instantiated, i.e. when all the components instantiated in all the composite entities Port Map aspects are created. The graph is then instantiated and populated through an object of a concrete class, following the Visitor Pattern.

4 Place and Route

In recent years the rapid advances in fabrication technology have dramatically increased the complexity in VLSI circuits. Indeed, major focus is on development of tools for the design and analysis of heuristic algorithms for partitioning, placement, routing and layout compaction. The developed place and route engine [41] uses different algorithms from traditional technology, adapted in order to solve

NML technology issues. What is new is how these algorithms are used together in order to obtain a final layout compliant with the technology constrains and limitations. The proposed NML design flow can be divided in two main parts: graph elaboration and physical mapping, which will be discussed in the following sections. The algorithm is divided in two parts, *Graph Elaboration* where the graph generated by the parser is optimized (see Sect. 4.1) and *Physical Mapping* where the circuit layout is generated (described in Sect. 4.2).

4.1 Graph Elaboration

The flow diagram of the graph elaboration phase is shown in Fig. 12. Starting from a structural description of the circuit, mapped on a set of cells available (majority voter, and, or, inverter), the HDL parser generates a graph which is the primary input of the place and route engine.

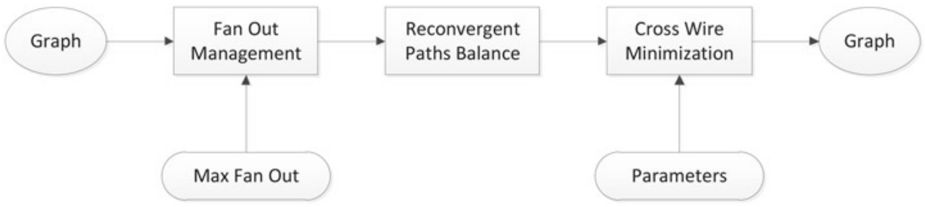

Fig. 12. Graph Elaboration flow diagram. The entry point is the graph generated by the parser while the output is an optimized graph used to create the circuit layout. Several optimizations are performed on the input graph, each part of the algorithm can be customized using appropriate parameters.

This data structure is then handled according to NML characteristics and the clock zones layout. The operations performed by these algorithms can be summarized as:

- *Fan-out management*, the graph is modified to take into account a limited fan-out, given as a parameter, for each graph node.
- *Reconvergent paths balance*, paths inside the graph are balanced to avoid "layout=timing" problems.
- *Wire crossing minimization*, the graph is elaborated using various algorithms (customizable with several input parameters) to reduce the number of wires-crossing.

At the end of this process the output graph is used as input for the physical mapping phase. In the following a detailed description of each algorithm is reported.

Fan-Out Management. As in the case of CMOS technology, also NML has a limitation on the fan-out that each cell can support. The main reasons are related to NML clock zones dimensions, to the physical space occupied by the wires and to the number of magnets that can be cascaded inside a clock zone. With a parallel clock zones layout organization, the vertical magnetic wires length should be limited in order to avoid propagation errors. Besides, considering that each wire is made by magnets there must be enough space to allow their physical placement. Therefore, the input graph is iteratively analyzed and mapped into a new one where the fan-out limit is satisfied.

Reconvergent Paths Balance. The graph generated starting from a circuit netlist can present many *reconvergent paths*. Two paths, in a direct acyclic graph (DAG), are called *reconvergent* if they diverge from and reconverge to the same blocks. This situation is common with traditional technology, but with NML it can generate some problems due to its intrinsic pipelined behavior that originates the so called "layout = timing" problem. In order to guarantee signals synchronization at the input of each logic gate, additional intermediate nodes must be added, so that all the reconvergent paths are composed by the same number of nodes.

Cross-Wires Minimization. Since NML is a planar technology, up to now just one layer can be used to build a circuit. A particular component, called *cross-wire*, is available to cross two wires on the same plane without interferences. Even if this block allows physical wires intersection a specific optimization is needed to reduce the number of cross-wire, therefore the wasted area. Different techniques are here implemented, such as *Barycenter*, *Fan-out Tolerance Duplication*, *Simulated Annealing* and *Kernighan-Lin*. They can be used alone or combining one or more of them together.

Barycenter. The basic idea behind this method is to rearrange nodes in order to place them directly above the nodes to which they are connected. The algorithm explores each rows of the graph two by two from inputs to outputs. For each couple of rows analyzed one is kept frozen while nodes in the other row are changed in position to reduce the number of cross-wires. This algorithm is quite simple and fast but leads to an unoptimized result. This is due because there can be situation in which multiple solutions satisfy the requisites of the algorithm. This solutions have however a different number of cross-wires, so the efficiency of the algorithm heavily depends on the policy chosen to solve this conflicting situations.

Fan-out Duplication. The job of the fan-out duplication algorithm is to integrate the *Barycenter* method in order to reduce wire crosses. As can be gathered from the name of this technique, graph nodes are duplicated trying to reduce the cross-wires number. The number of cross-wires can be theoretically reduced to 0, however the circuit area grows exponentially.

Kernighan-Lin. This algorithm is one of the most commonly used in the class of partition based methods. It heuristically divides the graph into sub-regions, trying to minimize the cut, i.e. the number of edges that connect one sub-region from the others.

Simulated Annealing. It is a stochastic algorithm that iteratively swap the position of nodes inside the graph trying to find the global minimum through consecutive solutions. The purpose of this technique is to minimize the number of cross-wires during the global placement. The algorithm needs three data as input:

- The graph nodes V.
- The minimum temperature that must be reached.
- The number of iteration that are necessary to obtain the final state.

The parameters used determine the efficiency of the simulated annealing algorithm, and these parameters must be chosen according to experience. While simulated annealing can lead to very good results, its a stochastic method that heavily relies on the parameters used, on the function used to generate random numbers and generally requires a huge time to converge.

References. Since the cross-wires minimization techniques here proposed are based on algorithm taken from the literature, here some references are included for interested readers. For the barycenter technique users can refer to [42,43] where these algorithms were already applied to QCA technology. For Kernighan-Lin technique the readers may refer to [44,45] while for simulated annealing readers may refer to [46–48]. These were the starting point, the algorithms have been reworked and adapted in order to be used on nanomagnet logic circuits.

4.2 Physical Mapping

During the physical mapping phase, the previous elaborated graph is transformed into the final circuit layout. Due to the high complexity of today ICs, the physical placement cannot be obtained in a single routing phase. For this reason, a three steps approach is followed:

- Placement, where graph nodes are mapped to their correspondent logic gate and they are initially placed on the layout.
- Global routing, where an iterative approach is used to find the optimum position for logic gates.
- Detailed routing, where interconnection wires among gates are routed.

In Fig. 13 a general flow diagram is shown.

Every node of the graph is mapped into its corresponding logic gate and it is placed into the circuit. Then, a global routing is performed. This process try to place each logic gate in order to minimize the occupied area. After the blocks positioning phase a detailed routing is performed among gates.

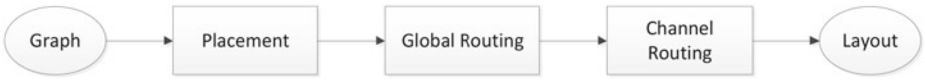

Fig. 13. Physical Mapping flow diagram. The circuit layout is generated starting from its graph following several optimization steps.

Placement. During the placement stage each node of the graph is mapped into its logic gate. As shown in Fig. 14, nodes are placed row by row starting from the top of the graph without any optimization. Thus, blocks are aligned with a minimum spacing equal to one equivalent magnet height.

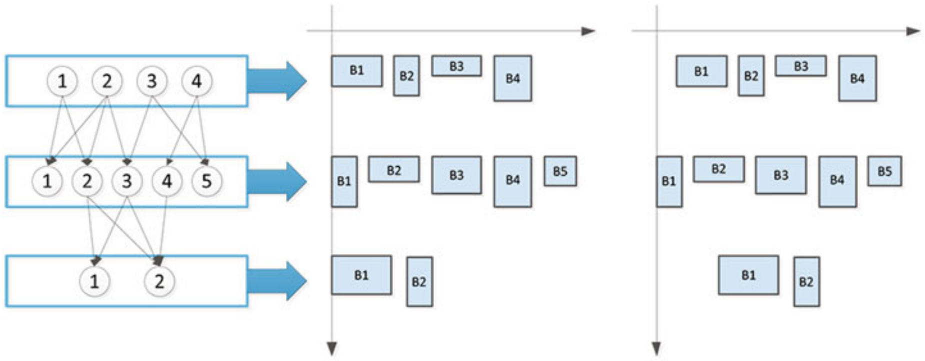

Fig. 14. *Center*: seed row placement for maximum with evaluation; *Right*: barycentered placement

This technique is used to evaluate the maximum width of the circuit, i.e the width of the largest row. At this point, a barycenter alignment is performed to shift placed blocks in order to reduce the overall wire length.

Global Routing. The final position of each gate is obtained with a fine shift performed during the global routing phase. The idea is to maximize the circuit compaction, therefore reducing the length of interconnection wires. Figure 15 shows the flow diagram of the global routing phase.

The implemented procedure is composed by the following main steps iteratively applied to each couple of rows:

- Logic gates of row i are shifted.
- Wires among row i and row $i+1$ are routed.
- Interconnection area is evaluated.

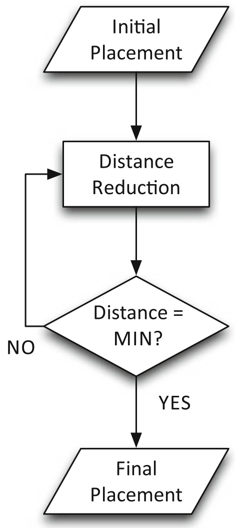

Fig. 15. Global routing flow diagram. It is an iterative process where the position of each block for every row is shifted with the aim of minimizing the interconnections length.

The goal is to find the global minimum area of the circuit, so an iterative approach is needed.

Channel Routing. When the final position of logic gates and pins is defined, the channel routing routine can start in order to obtain the final layout. In NML there is a limit to the maximum number of element that can be cascaded inside a clock zone. This feature coupled with the clock zones layout makes the signals propagation follow a “stair-like” path, as shown in Fig. 4(B). To model this behavior the channel routing algorithm is based on the mini-swap [49] method, which uses diagonal interconnections. Again though based on these algorithm, the method is a mix and largely adapted to the NML case.

Circuit Example. Figure 16 shows an example of circuit obtained after the whole process. It is a 6 bits ripple carry adder where the zoomed element represents a cross wire block.

References. The literature about placement and routing of VLSI circuits is quite wide. For interested readers an overview of the automatic placement and routing of CMOS VLSI circuits can be found in [50,51]. More detail on the wire and channel routing can instead be found in [49,52–54].

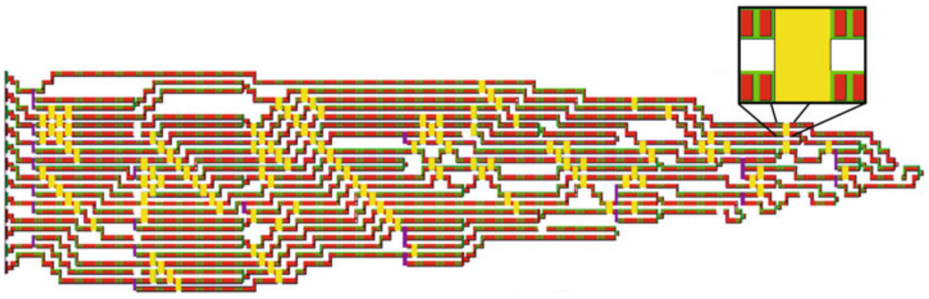

Fig. 16. Layout of a 6 bit Ripple Carry Adder.

5 Simulation Engine

After obtaining the detailed circuit layout it is important to simulate it to verify its behavior and to evaluate its performance. For this purpose a behavioral simulation engine was developed to be used in ToPoliNano. In order to simulate the circuit the graph used to represent the circuit must be modified. The hierarchical graph structure, which describes the circuit, is flattened by the simulation engine. A completely new data structure based on a dynamical matrix is created, in this way all physical information are kept (magnets positions) and the structural ones are discarded.

An example of simulation matrix is shown in Fig. 17. It represents a majority voter but the structural information are lost so it is seen simply as a matrix containing or not containing magnets in each cell. Three different type of magnets can be identified: input, output and normal. Every magnet is seen as a tristate device, because it can assume only three values, '0', '1' and RESET.

5.1 Simulation Algorithm

The behavior of the simulation algorithm has been captured and encapsulated in a finite-state machine (FSM). The clock waveforms Fig. 18(A) can be divided in 6 defined states and are therefore mapped to the FSM shown in Fig. 18(B).

The initial state is represented by the 0, while the first transition depends on the clock signals behavior. For example, the transition to the state 1 takes place when the first clock signal is equal to one and the second and the third are equal to 0. The FSM cycles through the states because of the periodic behavior of the clock waveforms. In order to manage transitions, the future step S_f can be calculated starting from the present state S_p (Eq. 1):

$$S_f = (S_p + 1) mod 6 \quad (1)$$

The general simulation algorithm follows the FSM switching periodically through its states. In each state, the value of each magnet must be calculated depending on the clock zone in which it is located. In particular, all magnets belonging to

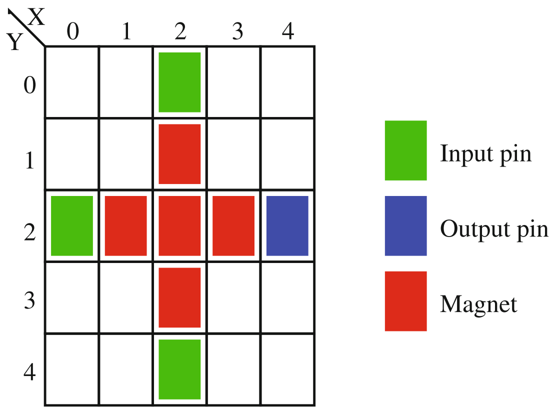

Fig. 17. Simulation matrix. Three categories of magnets can be identified: normal, input and output magnets. Every magnets is represented with a tristate approximation.

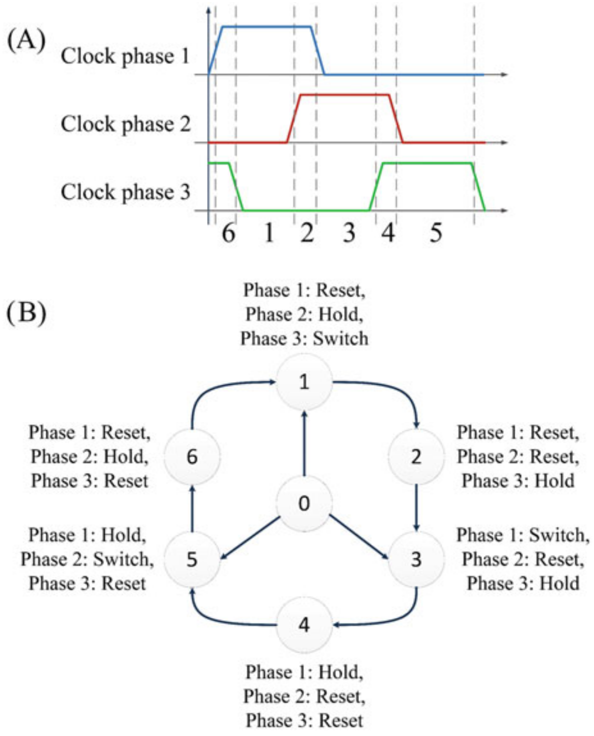

Fig. 18. (A) Clock signals. (B) Behavioral simulation finite state machine.

HOLD state are left untouched, while all magnets in RESET state are reset. In case of magnets belonging to a clock zone that is in the SWITCH state, the situation is more complex because the calculation of the state of each magnet requires a specific algorithm.

5.2 Matrix Exploration

For each clock zone where the SWITCH state is active, the simulation matrix must be scanned and the new value of magnetization of each magnet must be calculated. The matrix exploration algorithm is based on two nested loops which scan every column of a zone and for each of them scan every row.

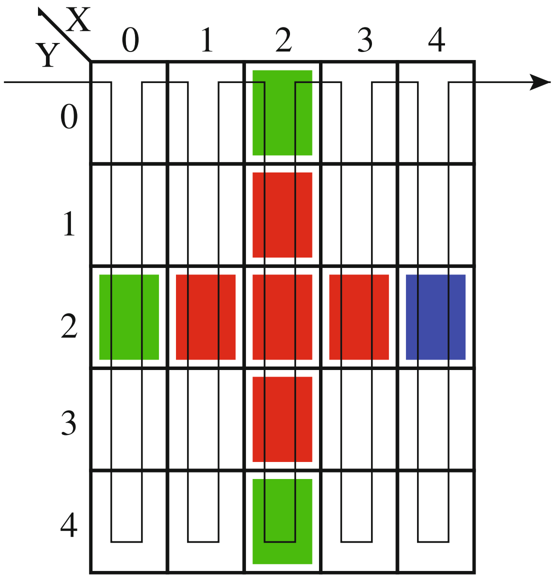

Fig. 19. Exploration matrix algorithm. Matrix columns are scanned one by one from left to right, each row of the column is scanned from up to down and then from down to up.

The behavior of the matrix exploration algorithm is depicted in Fig. 19. Each column of the matrix is scanned starting from the left border going toward the right border. For each column rows are scanned one by one from up to down and then from down to up. During the matrix visit for the new magnet state is evaluated following the magnetization algorithm (Sect. 5.3). This matrix exploration algorithm is chosen to reduce the usage of *if..then...else* constructs inside the programming code of ToPoliNano, to reduce their huge negative impact on performance when the code is executed in modern superscalar machines.

5.3 Magnetization Evaluation

During the matrix exploration the magnetization of every magnet is calculated according to the value of its neighbors, as shown in Fig. 20. The state of every magnets is calculated by the weighted sum of its neighbors.

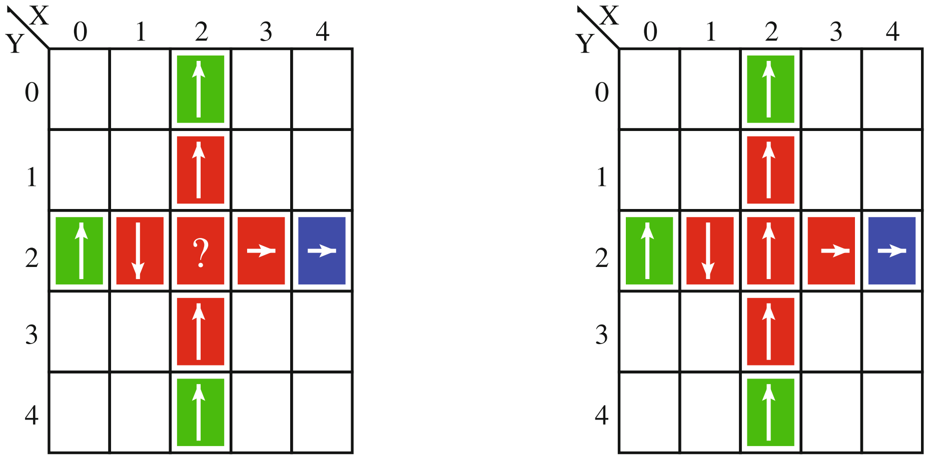

Fig. 20. Magnetization evaluation. The state of a magnet is calculated as a weighted sum of its neighbor elements.

6 Library Generator

The library generator is a tool that, together with the main application of ToPoliNano, allows the designer to describe new elementary components in order to be used within the internal library of the simulator. Its structure is shown in Fig. 21.

The library generator creates new components that can be used in ToPoliNano. However to create a component it needs three kind of information:

- *Ports* which contains information on the input/output pins. Every pin is identified by three parameters:
  - Direction: Input or Output.
  - Type: Standard Logic, Standard Logic Vector. This is required for compatibility with VHDL code, so that the created component can be identified by the parser.
  - Position within the circuit.
- *Layout* which contains information on the space occupied by the circuit.
- *Element Description* which contains the circuit layout, the gates which compose it and their position on the plane.

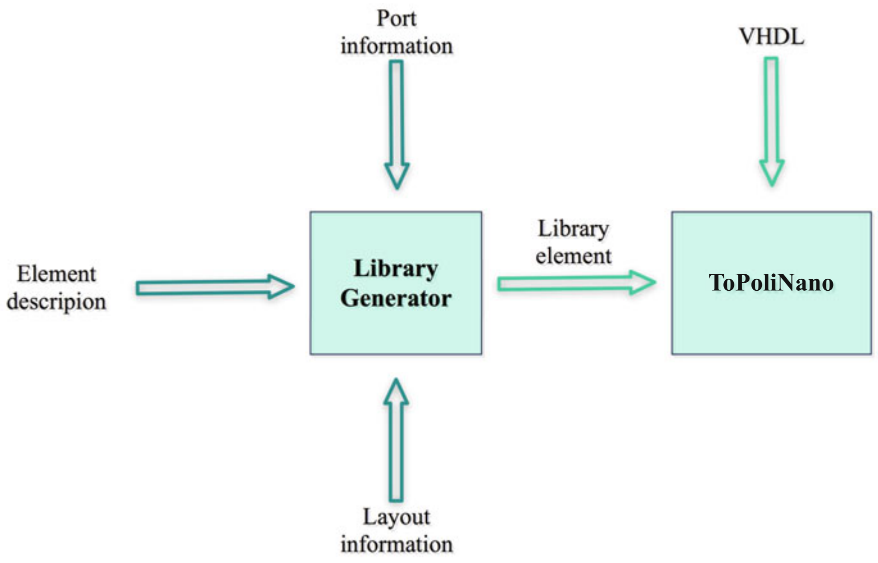

Fig. 21. Library Generation principle. To create a particular component the library generator requires three types of information. The component generated by the library generator can be used inside ToPoliNano.

The *Element Description* description of every component represents the circuit layout. It can be generated automatically by the *Place & Route* block or it can be created manually. There are two possibilities to manually describe a component. With the first possibility the user can draw circuit blocks using a vector graphic program called *Xfig*. The circuit drawn with *Xfig* is exported to an .svg file and loaded inside ToPoliNano. As a second option the circuit layout can be described manually writing the code which describes it. As a future work we should create an in-program editor of circuits. The layout is based on different logic gates defined as library components (e.g. majority voter, cross wire, wire, and, or).

7 Output Generation

At the end of the simulation, ToPoliNano offers two main ways to represent results.

- On the main program window displaying the output signal waveforms that can be saved on an Encapsulated PostScript file (EPS).
- A text file containing the timing samples that the depict the circuit behavior.

An example of output waveforms obtained after the simulation is reported in Fig. 22.

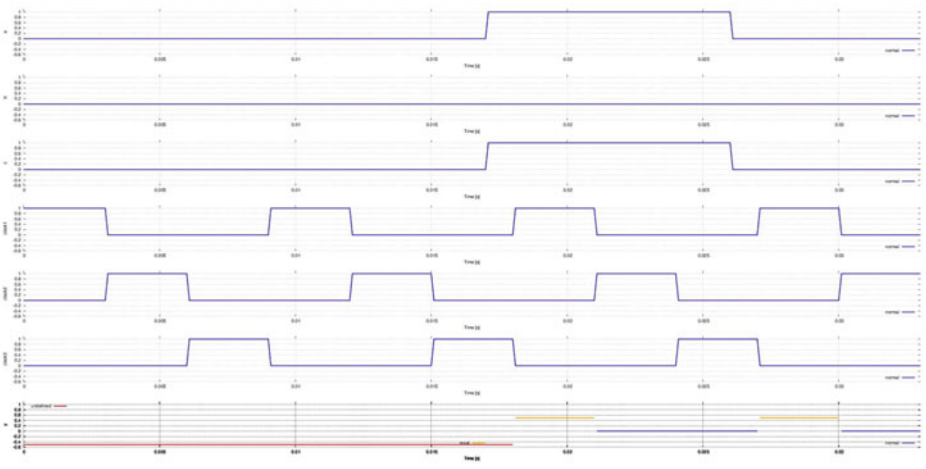

Fig. 22. Output waveforms example.

As well as the circuit timing behavior, additional information can be obtained, like circuit area, number of magnets, wasted area. Moreover a power estimation is also possible, up to now only considering the magnetic field clock.

8 Performance

One of our main targets during the development of ToPoliNano was to design a tool capable to handle high complexity circuits with reasonable execution times. While the software is still in development we were able to make some preliminary evaluations on performance. The tests are based on currently available machines, with *I-3*, *I-5* and *I-7* Intel processors, running both Linux and Mac-OS operating systems. As a benchmark we have used a simple Ripple Carry Adder, made by N full adders. Just for test purpose we have instantiated up to 10000 full adders. The placement of all magnets (around 2000000) took only 30 s. The in-memory occupation of such a circuit was just about 1.5 Gb. The simulation of one full adder for a time period of 80 µs with a simulation step of 1 ps, required just 0.3 s. To compare the performance of ToPoliNano with existing tools we have performed some simulations with two widely used micromagnetic simulators, NMAG [30] and OOMMF [29] on the same machine used for the testing of ToPoliNano. The simulated structure was a simple NML wire in three different cases, changing the length from 4 magnets, to 8 magnets and finally to 12 magnets. Results in terms of simulation time and memory usage are reported in Table 1. The time indicated is the machine time required to advance the state of the circuit of 1 ns. For example, in case of the 4 magnets wire simulated through NMAG, 43 s of machine-time are required to advance the state of the circuit of 1 ns. The memory usage and the simulation time increases with the circuit complexity. Starting from these values it is possible to get a rough estimation

Table 1. Simulation performance of two of the most widely used free micromagnetic tools, NMAG and OOMMF.

	Simulator	Memory [Mb]	Time [s/ns]
4 magnets wire	NMAG	49	43
	OOMMF	24	89
8 magnets wire	NMAG	79	75
	OOMMF	32	173
12 magnets wire	NMAG	104	127
	OOMMF	36	200

of the memory and the simulation time required to simulate a circuit made by around 2 milions of magnets. For NMAG a total of 12 TB of RAM memory and 70000 years of simulation time will be required. OOMMF instead requires only 4 TB of memory and just 35000 years of simulation times. These are just rough estimations but they clearly show the necessity of using a simulator like ToPoliNano to handle complex NML circuits. Comparing the performance of ToPoliNano with Modelsim [34], one of the most used VHDL simulators is more complex. However a simple comparison can be obtained looking at the microprocessor described in [26] and [27], or at the systolic array for biosequences analysis described in [55–57], which are also made by millions of magnets. In that case the simulation for a time of 100 µs required near 1–2 h of machine-time. Clearly ToPoliNano shows an advantage also over classical VHDL simulators.

Finally the Ripple Carry Adder was used also to test the performance of the Place & Route block, but only up to 32 bits. Using the most simple cross-wires minimization techniques, the *Barycenter*, the creation of the layout required only 20 ms. Optimizing the layout with the best technique available, the *Simulated Annealing*, the time required to synthesize a 32 bit adder was equal to few minutes.

9 General Code Structure

ToPoliNano has been developed in C++ language to be highly efficient in computation, thus providing ground to study multiple emerging technologies, demanding in terms of computational resources. With regard to the implementation, the C++ language was selected as programming language mainly for performance reasons, and the whole structure of the simulation is based on the use of classes grouped into macro-blocks:

– Controllers, which contains all the high-level logic of the application.
– GUI, which contains all the classes related to the management of graphical interface (configuration wizard, main window).
– HDL Graph Controller, about the logic of HDL parsing.
– Inputs And Clocks, for the generation of input and clock signals.

– NML, which contains all the classes for the implementation of the simulation algorithms and data structures of NML technology.
– NanoArray: which contains all the classes for the implementation of the simulation algorithms and data structures of NanoArray technology.

ToPoliNano has been designed as a single-source, cross-platform application. Currently, these are the supported platforms:

– Windows.
– Linux/X11.
– Mac OS X.

ToPoliNano is supported on a variety of 32-bit and 64-bit platforms.

– Ubuntu Linux 10.04 (32-bit).
– Ubuntu Linux 11.10 (64-bit).
– Microsoft Windows XP SP3 (32-bit).
– Microsoft Windows 7 (32-bit).
– Apple Mac OS X 10.6 “Snow Leopard” (64-bit).
– Apple Mac OS X 10.7 “Lion” (64-bit).
– Apple Mac OS X 10.8 “Mountain Lion” (64-bit).

ToPoliNano has been built on top of existing cross-platform frameworks. This enabled a development focused on core functionalities, rather then spending time re-creating commodity software. The framework used are well respected among developer communities. In particular, two are the cornerstones of ToPoliNano: the Qt framework and the Boost Library. Lots of resources are available on the web for both of them.

The Boost C++ Library is a collection of free libraries that extend the basic C++ functionality. It provides a wide range of platform agnostic functionality that the Standard Template Library (STL) missed. It can be regarded as a complement to STL rather than a replacement, according to its developers [58]. At the time of this writing, the current Boost release (1.54.0) contains over 80 individual libraries. The Boost C++ libraries are open-source and peer-reviewed.

The choice of the Boost library is due to the very high quality of the code and to its optimal performances. The use of such a library can speed-up the initial development: bugs are minimal, there is no need to “reinvent-the-wheel” and the maintenance cost is reduced to a bare minimum. The Boost libraries are mostly header-only, making them trivial to install, upgrade and configure. In most cases, installation and upgrading only requires the addition or the modification of the include path. Only few of them requires a full compilation. Moreover, Boost sub-libraries can be used independently of each other.

Nowadays computer architectures feature multicore designs, thus giving a calculator the opportunity to perform multiple tasks simultaneously. Not only operating systems can take advantage of the multiple cores in computers, but also applications. This is know as *multithreaded programming*. ToPoliNano supports multithreading to exploit the potential of modern microprocessor and to speed-up circuits simulation.

10 Other Supported Technologies

ToPoliNano was developed to support different technologies in one tool. Up to now it enables the study of NanoMagnet Logic and NanoArray, a promising technology that is briefly outlined in what follows. The NanoArray architecture is based on an intense research activity aimed at understanding the emerging nanotechnological devices and their constraints of realization. A NanoArray project is hierarchical, consisting of interconnected *tiles* that determine the architecture (Fig. 23) and it is optimized to meet the various constraints that come from the construction process, from materials and devices.

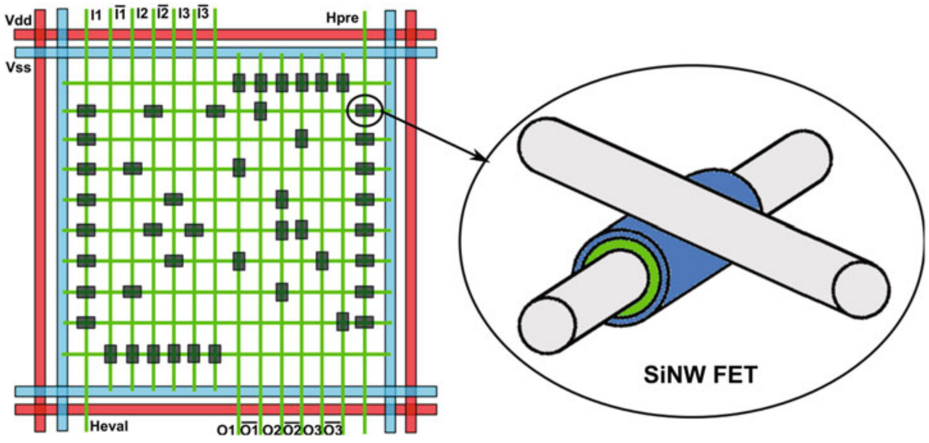

Fig. 23. Example of NanoArray made by Silicon Nanowires (SiNW) and SiNW FETs.

Among the devices that NanoArray architecture can use, there are FETs or diodes in 2-D structures based on crossed SiNWs to realize logical functions (Fig. 23); the various types of optimizations to overcome the limitations imposed by the layout, by the constructive process and by defects adopted in this approach differ from those enforced by other architectures.

The logic AND-OR (or equivalent according to DeMorgan) is carried out either statically, or dynamically. In a dynamic circuit style pipelining of circuits becomes possible. Signals like GND, Vdd, Vss and dynamic control signals necessary for operation are transported at micrometric scale by so-called microwires. The defects are masked directly into the circuit or architecture, without recurring to reconfigurability. The peculiarities of this approach can be summarized in four points:

- redundancy is added in each stage of AND-OR logic and outputs, in addition to the dual-rail redundancy; signals are intercalated and then combined as part of the AND & OR logic planes;

– modification of the logic of one stage to allow the masking of the defects in the current stage or, alternatively, in the following AND-OR stage;
– addition of wires with function of weak pull-up or pull-down in the device to maintain at a low logic level inputs potentially faulty (prior to an OR plane), or at a high logic level (prior to an AND plane);
– joint operation with majority voting circuits based on CMOS at key points of the architecture.

Nanotiles. The nanotiles represent the building blocks of NanoArray architecture. The crossing nanowires form a nanoarray, whose junctions (points of intersection) can be FETs or not connected. The nanoarrays are surrounded by microwires, that carry electrical power and, in this particular implementation, also signals for programming the interconnections. Each signal is present both in its original form and in complemented form.

Dynamic Pipelining. Due to design constraints (e.g. doping of the nanowire) and to topological constraints too, the latches are very difficult to implement within the nanoarray. Typically, latches or registers are used to obtain the functionalities of pipelining of data streams, for example in a datapath. This component is one of the most common within the microprocessor, so it is important to have an efficient way of pipelining. In NanoArray, when using circuits of the dynamic type, it is possible to obtain a temporary storage of information, without resorting to the explicit use of a latch. A pipelined NanoArray circuit can be realized by cascading dynamic nanoTiles, without requiring an explicit latching of the signals, which would entail a considerable reduction of density. A processor made in nanotechnology could have thousands of nanoTiles, so it is fundamental to have an efficient communication system, and this is one of the critical points in nanoarchitectures. In the NanoArray approach local communication occurs between nanowires, to maintain an optimum use of the area, while for the global communications microwires are used.

NanoArray Simulation. The NanoArray simulation engine belongs to the class of the event-driven simulators. It follows the information flow inside the structure under simulation and generates specific events, when necessary, to correctly handle the propagation of information. To better understand this process, we can imagine each sub-tile as a four-port device, with each port identified by a cardinal point.

A change in the information at a given port may need to be propagated inside the sub-tile, if there is an appropriate component to support propagation (e.g. a nanowire). We do not need to know anything about the electrical properties of the component to perform a logical analysis. As a function of the port at which the change in information happens, and the original direction of propagation of this piece of information, we can check whether there is support for further propagation and, if this is the case, to change the information on another port of the sub-tile by means of the supporting element. This, in turn, will trigger an update event over the sub-tile, if any, connected to the first one by means of the output port. By following the very same process, the information is propagated inside the structure, only where it is needed.

There could be an active device inside the sub-tile, and the propagation of information could lead to a change in its status. Should this happen, another kind of event would be enqueued in the event queue, waiting to be processed to take into account a possible change of information in a direction of propagation that is orthogonal with respect to the one that originated the event. This approach is very flexible, indeed, because it allows for different kind of control of dynamic circuits (number of phases) since the phase sequence is not embedded into the simulator but is coded in the input control sequence and the same approach can thereby be used in many different scenarios.

References. The literature about NanoArray is quite wide, here some references are reported for interested readers. In [59,60] some of the work of University of Massachusetts Amherst is reported. Other solutions were proposed by Likharev [61], Dehon [62], a group of HP [63] and the Carniege Melon University [64].

11 Conclusions

In this chapter we have described ToPoliNano, a tool which aim is to design and simulate NML circuits and other emerging technologies following the same top-down methodology used in the CMOS case. This tool allows to easily describe and simulate complex NML circuits without loosing important details like the placement of each magnets, as it happens in case of VHDL modeling, and without the limitations in terms of speed and circuit sizes of low level simulators.

The tool is still in development so it does not have all the planned functionalities but we are still working hard on completing and improving it. Particularly our efforts are oriented in three directions: Custom circuits description, Place & Route algorithm and simulation engine.

While the automatic layout generation is a very important feature, we acknowledge that also the manual custom circuit description plays an important role. We are therefore trying to improve the means available to describe circuits manually, for example providing an interface that allows to import circuits designed with QCADesigner, opportunely converted substituting magnets to the QCA cells used by QCADesigner. This can represent an important step in the tool development, since many researchers in the QCA community use QCADesigner. We are also working to modify the program to be released to other people that work in this field, so that they can fully exploit the advantages given by ToPoliNano.

The Place & Route algorithm is the program core and also the most complex part. We are working in two directions, to improve the algorithm allowing it to handle also sequential circuits and implementing a floorplanning algorithm to create the layout of complex circuits with a hierarchical structure. Sequential circuits require careful handling, because signals must be carefully synchronized. We would like to support in the future, multiple clock zones layout types and more NML types. Finally we are working with the aim of improving the simulation engine supporting other types of NML.

Overall it is possible to conclude that ToPoliNano has a great potential that must be still exploited, and we are doing our best toward this direction.

References

1. Porod, W.: Magnetic Logic Devices Based on Field-Coupled Nanomagnets. Nano and Giga, Tempe (2007)
2. Imre, A.: Experimental study of nanomagnets for quantum-dot cellular automata(MQCA)logic applications. Ph.D. thesis, University of Notre Dame, Notre Dame, Indiana (2005)
3. Lent, C., Tougaw, P., Porod, W., Bernstein, G.: Quantum cellular automata. Nanotechnology **4**, 49–57 (1993)
4. Csurgay, A., Porod, W., Lent, C.: Signal processing with near-neighborcoupled time-varying quantum-dot arrays. IEEE Trans. Circ. Syst. **47**(8), 1212–1223 (2000)
5. Awais, M., Vacca, M., Graziano, M., Masera, G.: FFT implementation using QCA. In: 2012 19th IEEE International Conference on Electronics, Circuits and Systems, ICECS, pp. 741–744 (2012)
6. Graziano, M., Chiolerio, A., Zamboni, M.: A technology aware magnetic QCA NCL-HDL architecture. In: 9th IEEE Conference on Nanotechnology. IEEE-NANO 2009, Genoa, Italy, July 2009, pp. 763–766. IEEE (2009)
7. Chiolerio, A., Allia, P., Graziano, M.: Magnetic dipolar coupling and collective effects for binary information codification in cost-effective logic devices. J. Magn. Magn. Mater. **324**(19), 3006–3012 (2012)
8. Pulimeno, A., Graziano, M., Piccinini, G.: Molecule interaction for QCA computation. In: IEEE International Conference on Nanotechnology, pp. 1–5 (2012)
9. Pulimeno, A., Graziano, M., Demarchi, D., Piccinini, G.: Towards a molecular QCA wire: simulation of write-in and read-out systems. Solid State Electron. **77**, 101–107 (2012)
10. Pulimeno, A., Graziano, M., Saginario, A., Cauda, V., Demarchi, D., Piccinini, G.: Bis-ferrocene molecular QCA wire: ab-initio simulations of fabrication driven fault tolerance. IEEE Trans. Nanotechnol. **12**(4), 498–507 (2013)
11. Das, J., Alam, S., Bhanja, S.: Ultra-low power hybrid CMOS-magnetic logic architecture. IEEE Trans. Circ. Syst. **59**(9), 2008–2016 (2012)
12. Karunaratne, D., Bhanja, S.: Study of single layer and multilayer nano-magnetic logic architectures. J. Appl. Phys. **111**, 07A928 (2012)
13. Niemier, M., et al.: Nanomagnet logic: progress toward system-level integration. J. Phys. Condens. Matter **23**, 493202 (2011)
14. Graziano, M., Vacca, M., Zamboni, M.: Magnetic QCA Design: Modeling, Simulation and Circuits. Cellular Automata - Innovative Modelling for Science and Engineering, InTech (2011). http://www.intechopen.com/articles/show/title/magnetic-qca-design-modeling-simulation-and-circuits
15. Vacca, M., Vighetti, D., Mascarino, M., Amaru, L., Graziano, M., Zamboni, M.: Magnetic QCA majority voter feasibility analysis. In: 2011 7th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME), pp. 229–232 (2011)

16. Vacca, M., Graziano, M., Zamboni, M.: Majority voter full characterization for nanomagnet logic circuits. IEEE Trans. Nanotechnol. **11**(5), 940–947 (2012)
17. Csaba, G., Porod, W.: Simulation of filed coupled computing architectures based on magnetic dot arrays. J. Comput. Electron. **1**, 87–91 (2002). (Kluwer)
18. Graziano, M., Vacca, M., Chiolerio, A., Zamboni, M.: A NCL-HDL snake-clock based magnetic QCA architecture. IEEE Trans. Nanotechnol. **10**(5), 1141–1149 (2011)
19. Niemier, M., Hu, X., Alam, M., Bernstein, G., Porod, W., Putney, M., DeAngelis, J.: Clocking structures and power analysis for nanomagnet-based logic devices. In: International Symposium on Low Power Electronics and Design, Portlan-Oregon, USA, pp. 26–31. IEEE (2007)
20. Ju, X., Niemier, M.T., Becherer, M., Porod, W., Lugli, P., Csaba, G.: Systolic pattern matching hardware with out-of-plane nanomagnet logic devices. IEEE Trans. Nanotechnol. **12**(3), 399–407 (2013)
21. Das, J., Alam, S., Bhanja, S.: Low power magnetic quantum cellular automata realization using magnetic multi-layer structures. J. Emerg. Sel. Top. Circ. Syst. **1**(3), 267–276 (2011)
22. Chung, T., Keller, S., Carman, G.: Electric-field-induced reversible magnetic single-domain evolution in a magnetoelectric thin film. Appl. Phys. Lett. **94**, 132501 (2009)
23. Csaba, G., Porod, W.: Behavior of nanomagnet logic in the presence of thermal noise. In: International Workshop on Computational Electronics, Pisa, Italy, pp. 1–4. IEEE (2010)
24. Ercan, I., Anderson, N.: Heat Dissipation Bounds for Nanocomputing: Theory and Application to QCA (2011)
25. Alam, M., Siddiq, M., Bernstein, G., Niemier, M., Porod, W., Hu, X.: On-chip clocking for nanomagnet logic devices. IEEE Trans. Nanotechnol. **9**(3), 348–351 (2010)
26. Graziano, M., Vacca, M., Blua, D., Zamboni, M.: Asynchrony in quantum-dot cellular automata nanocomputation: elixir or poison? IEEE Des. Test Comput. **28**(5), 72–83 (2011)
27. Vacca, M., Graziano, M., Zamboni, M.: Asynchronous solutions for nano-magnetic logic circuits. ACM J. Emerg. Tech. Comput. Syst. **7**(4), (2011)
28. Walus, K., Dysart, T., Jullien, G., Budiman, R.: Qcadesigner: a rapid design and simulation tool for quantum-dot cellular automata. IEEE Trans. Nanotechnol. **3**(1), 26–31 (2004)
29. Donahue, M., Porter, D.: OOMMF User's Guide, Version 1.0. Technical Report Interagency Report NISTIR 6376, National Institute of Standards and Technology, Gaithersburg, September 1999
30. Fischbacher, T., Franchin, M., Bordignon, G., Fangohr, H.: A systematic approach to multiphysics extensions of finite-element-based micromagnetic simulations: Nmag. IEEE Trans. Magn. **43**(6), 2887–2889 (2007)
31. Scholz, W., Fidler, J., Schrefl, T., Suess, D., Dittrich, R., Forster, H., Tsiantos, V.: Scalable parallel micromagnetic solvers for magnetic nanostructures. Comput. Mater. Sci. **28**, 366–383 (2003)
32. Vacca, M., Graziano, M., Zamboni, M.: Nanomagnetic logic microprocessor: hierarchical power model. IEEE Trans. VLSI Syst. **21**(8), 1410–1420 (2012)
33. Niemier, M., Varga, E., Bernstein, G., Porod, W., Alam, M., Dingler, A., Orlov, A., Hu, X.: Shape engineering for controlled switching with nanomagnet logic. IEEE Trans. Nanotechnol. **11**(2), 220–230 (2012)

34. Mentor graphics. http://www.modelsim.com
35. Awais, M., Vacca, M., Graziano, M., Masera, G.: Quantum dot cellular automata check node implementation for LDPC decoders. IEEE Trans. Nanotechnol. **12**(3), 368–377 (2013)
36. Graziano, M., Frache, S., Vacca, M., Turvani, G., Riente, F., RuoRoch, M., Zamboni, M.: Topolinano: Torino politecnico nanotechnology tool (2012). http://www.topolinano.polito.it
37. Vacca, M., Frache, S., Graziano, M., Zamboni, M.: ToPoliNano: a synthesis and simulation tool for NML circuits. In: IEEE International Conference on Nanotechnology, pp. 1–6, August 2012
38. Frache, S., Chiabrando, D., Graziano, M., Riente, F., Turvani, G., Zamboni, M.: ToPoliNano: nanoarchitectures design made real. In: IEEE International Symposium on Nanoscale Architectures NANOARCH, pp. 160–167 (2012)
39. Frache, S., Chiabrando, D., Graziano, M., Graziano, M., Boarino, L., Zamboni, M.: Enabling design and simulation of massive parallel nanoarchitectures. J. Parallel Distrib. Comput. **74**, 2530–2541 (2014)
40. Chiswell, I., Hodges, W.: Mathematical Logic. Oxford Texts in Logic, vol. 3. Oxford University Press, Oxford (2007)
41. Vacca, M., Frache, S., Graziano, M., Crescenzo, L., Cairo, F., Zamboni, M.: Automatic Place&Route of nano-magnetic logic circuits, pp. 58–63 (2013)
42. Ravichandran, R., et al.: Partitioning and placement for buildable QCA circuits. In: DAC, vol. 1 (2005)
43. Chung, W., et al.: Node duplication and routing algorithms for quantum-dot cellular automata circuits. IEEE Proc. Circ. Dev. Syst. **153**(5), 497–505 (2006)
44. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Syst. Tech. J. **49**, 291–307 (1970)
45. Ravikumar, Si.Pi., Ravikumar, C.P.: Parallel Methods for VLSI Layout Design. Greenwood Publishing Group, Boston (1995)
46. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **4598**(220), 671–680 (1983)
47. Cerny, V.: Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. J. Optim. Theory Appl. **45**, 41–51 (1985)
48. Sechen, C., Sangiovanni-Vincentelli, A.: The Timberwolf placement and routing package. IEEE J. Solid-State Circ. **20**(2), 510–522 (1985)
49. Wang, D.: Novel routing schemes for IC layout part I: two-layer channel routing. In: Design Automation Conference (1991)
50. Sherwani, N.: Algorithms for VLSI Physical Design Automation. Springer, Heidelberg (2002)
51. Kahng, A., Lienig, J., Markov, I., Hu, J.: VLSI Physical Design Automation: From Graph Partitioning to Timing Closure. Springer, Heidelberg (2011)
52. Deutsch, D.: A dogleg channel router. In: Proceedings of 19th Design Automation Conference (1976)
53. Yoshimura, T.: An Efficient Channel Router (1984)
54. Sau, A., Pal, A., Mandal, T., Datta, A., Pal, R., Chaudhuri, A.: A Graph based Algorithm to Minimize Total Wire Length in VLSI Channel Routing (2011)
55. Urgese, G., Graziano, M., Vacca, M., Awais, M., Frache, S., Zamboni, M.: Protein Alignment HW/SW Optimizations. In: The IEEE International Conference on Electronics, Circuits, and Systems (ICECS), pp. 145–148 (2012)
56. Wang, J., Vacca, M., Graziano, M., Zamboni, M.: Biosequences analysis on NanoMagnet Logic. In: International Conference on IC Design and Technology, pp. 131–134, May 2013

57. Graziano, M., Frache, S., Zamboni, M.: A hardware viewpoint on biosequence analysis: what's next? ACM J. Emerg. Technol. Comput. Syst. **9**(4), 29 (2013)
58. Free peer-reviewed portable C++ source libraries
59. Wang, T., Narayanan, P., Moritz, C.A.: Heterogeneous 2-level logic and its density and fault tolerance implications in nanoscale fabrics. IEEE Trans. Nanotechnol. **8**(1), 22–30 (2009)
60. Comparison of Analog and Digital Nano-Systems: Issues for the Nano-Architect (2008)
61. CMOL: devices, circuits, and architectures (2004)
62. DeHon, A., Wilson, M.: Nanowire-based sublithographic programmable logic arrays (2004)
63. Snider, G., Kuekes, P., Williams, R.: CMOS-like logic in defective, nanoscale crossbars. Nanotechnology **15**, 881–891 (2004)
64. Goldstein, S., Budiu, M.: NanoFabrics: spatial computing using molecular, electronics. In: Proceedings of the 28th Annual International Symposium on Computer Architecture (2001)