fiction: An Open Source Framework for the Design of Field-coupled Nanocomputing Circuits
(Extended Abstract)

Marcel Walter¹  Robert Wille²,³  Frank Sill Torres¹,³  Daniel Große¹,³  Rolf Drechsler¹,³

¹Group of Computer Architecture, University of Bremen, Germany
²Johannes Kepler University Linz, Austria
³Cyber Physical Systems, DFKI GmbH, Bremen, Germany
{m_walter, frasillt, grosse, drechsler}@uni-bremen.de, robert.wille@jku.at

https://github.com/marcelwa/fiction

ABSTRACT
As a class of emerging post-CMOS technologies, Field-coupled Nanocomputing (FCN) devices promise computation with tremendously low energy dissipation. Even though groundbreaking advances in several physical implementations like Quantum-dot Cellular Automata (QCA) or Nanomagnet Logic (NML) have been made in the last couple of years, design automation for FCN is still in its infancy and often still relies on manual labor. In this paper, we present an open source framework called fiction for physical design and technology mapping of FCN circuits. Its efficient data structures, state-of-the-art algorithms, and extensibility provide a basis for future research in the community.

1 INTRODUCTION AND BACKGROUND
Field-coupled Nanocomputing (FCN) [1] is a class of emerging technologies which conduct computations fundamentally differently from conventional systems relying e.g. on CMOS. Here, information is represented in terms of the polarity or magnetization of nanoscale cells and can be propagated to adjacent ones using repelling forces of local fields [7, 13]. This results in devices that allow to represent and process binary information without electrical current flow. Consequently, numerous contributions on their physical realization have been made in the past and several of some them in the last three to four years, e.g. molecular Quantum-dot Cellular Automata (mQCA) [12], atomic Quantum-dot Cellular Automata (aQCA) [2, 11], or Nanomagnet Logic (NML) [10].

Moreover, this way of representing and processing information is doable with highest processing performance and remarkably low energy dissipation – as confirmed by several theoretical and experimental studies (see e.g. [14, 18, 21]). This makes FCN a promising technology which conducts computations fundamentally differently from conventional systems relying e.g. on CMOS. Here, information is represented in terms of the polarity or magnetization of nanoscale cells and can be propagated to adjacent ones using repelling forces of local fields [7, 13]. This results in devices that allow to represent and process binary information without electrical current flow. Consequently, numerous contributions on their physical realization have been made in the past and several of some them in the last three to four years, e.g. molecular Quantum-dot Cellular Automata (mQCA) [12], atomic Quantum-dot Cellular Automata (aQCA) [2, 11], or Nanomagnet Logic (NML) [10].

Moreover, this way of representing and processing information is doable with highest processing performance and remarkably low energy dissipation – as confirmed by several theoretical and experimental studies (see e.g. [14, 18, 21]). This makes FCN a promising alternative to conventional integrated circuit technologies. However, no exhaustive automatic design flow is available for FCN technologies so far. Also, due to different design rules of CMOS VLSI, existing classical methods are not applicable to the FCN domain.

In this paper, we present fiction, a framework for the design of FCN circuits. The framework is written in C++ and uses the EPFL Logic Synthesis Libraries [17]. With this tool, we especially tackle the physical design steps of FCN like placement, routing, timing, and technology mapping under the domain specific constraints.

Logic synthesis however is taken as granted as it can be performed with existing tools like ABC [3]. Even though their physical implementations differ from each other, the structural models of most FCN technologies are nearly identical. Data structures in fiction are designed around this insight: whenever possible, fiction abstracts from physical implementations and conducts layout on a higher level. Only in the final step, a technology mapping is performed.

On that layer of abstraction, the design task boils down to the composition of tiles with assigned logic or wire elements. Such entities for an AND gate, an inverter, a straight wire, and a fanout are shown exemplarily in Figure 1. The shade of the tiles, the coloring of the logic elements, and the numbers in the bottom right corners represent redundant information about the clocking. For further information see [1, 9, 21, 24].

In the following, fiction is initially presented from the perspective of a standard user in Section 2, where an example layout flow is conducted and benchmarking is elaborated, followed by a description of the developer’s perspective in Section 3, where the implementation of a naive random placement is exemplarily shown. Section 4 concludes the paper.

2 THE USER’S PERSPECTIVE
In this section, two typical application scenarios within fiction are described. First, it is shown how interaction with the store-based command-line interface (CLI) alice [17] works by the use case of obtaining a routed layout which is prepared for physical simulation. Then, scripting, benchmarking and logging functionalities to easily generate statistical data are demonstrated.

2.1 The CLI
Starting point of all flows is a synthesized Verilog netlist file which exclusively uses the assign statement and logic primitives. Suitable files can be generated with ABC [3] using the following commands.

read <inputfile>
strash
write <outputfile>.v

Also, fiction comes with a set of sample netlist files which can be found in the benchmarks folder. These can be loaded using the command read. By this, the netlist is parsed and placed in a store where it can be accessed and (re-)used by other algorithms. So far, two state-of-the-art layout approaches are implemented which can be called via the commands exact [24] and ortho [26]. Algorithm exact utilizes the SAT solver Z3 [5] to generate minimal layouts in terms of area within the provided parameters. The approach is highly configurable and allows to toggle and set up several design criteria like the clocking scheme, the use of crossings, balanced
paths, synchronization elements [20], I/O pins, and wire length restrictions. However, exact is only applicable for rather small netlists due to the complexity of the tackled problem [25]. On the other hand, or tho is a heuristic algorithm which does not guarantee minimal layouts anymore and also is restricted to a fixed clocking scheme, but therefore can generate results in feasible runtime.

Eventually, both algorithms generate a gate-level abstraction of an FCN circuit grid. Using the command cell, a technology mapping is performed with a selected gate library, whose default is QCA-ONE [15]. Having a QCA circuit in store, it can be written as a simulation file for the QCADesigner [27], a standard tool for physical simulation of QCA structures, by entering qca <filename>.qca. Also, using the command show generates a scalable vector graphic to inspect the implemented circuit.

2.2 Benchmarking & Scripting

The flow shown in the previous section can easily be repeated by storing it in a fiction script file. We assume that two designs generated with different settings of the exact algorithm for the netlist c17.v shall be compared. Therefore, we create the following file compare.fs.

```bash
read ../../benchmarks/ISCAS85/c17.v
exact -ixbs 2ddwave4
ps -g
show
exact -ps use
ps -g
show
```

The first call to exact enables designated I/O pins (-1), allows crossings (-x), routes all I/Os to the grid borders (-b) and uses the 4-phase 2DDWave [23] scheme (-s 2ddwave4), while the second one allows for unbalanced (de-synchronized) paths (-p) and utilizes USE [4] as the clocking scheme (-s use).

To run this script, we enter ./fiction -f compare.fs. Not only do we get SVG images of both layouts but also, through the use of ps -g, some statistical information about the layouts are printed, i.e. the dimension of the resulting grid in tiles, the amount of gate (#G) and wire tiles (#W), crossings (#C), and latches (#L) used, the length of the critical path (CP) in tiles, and the throughput (TP) [19, 20]. Note that fan-outs and I/O pins are counted as gates since they are fixed by the input. This way, the displayed amount of wires represent the net costs [22]. The resulting graphics are shown in Figure 2.

![Figure 1: Tiles in QCA implementation](image)

![Figure 2: Two differently layouted variants of c17.v](image)

Using the -c flag, a semicolon-separated list of commands can be provided and the output is logged in a JSON file by the -1 flag. Note that when logging is activated, ps -g logs contain more in-depth data about the layout like its bounding box size, energy consumption [21], etc. For both, the physical models as well as the log files, the originally entered file name is used extended by the respective file extension.

3 THE DEVELOPER’S PERSPECTIVE

This section elaborates important design decisions for fiction’s data types and presents some sample code for a naive random placement in order to demonstrate their use.

Core of the implementation are the classes fcn_gate_layout and fcn_cell_layout – a gate level, tile-based abstraction and a physical FCN cell-based layout respectively. They are based on a boost::grid_graph, a highly memory efficient grid data structure from the Boost Graph Library (BGL) [16]. Following BGL’s paradigm, the grid topology is separated from the associated elements like wires, gates, or cells respectively. Such associations happen via defaulted maps that return standard values (mostly used for free grid positions), when an uninitialized access happen to save even more memory.

Also, regular clocking schemes like USE are stored in terms of a small cut-out which is then seamlessly extrapolated for larger
layouts as there is no need to store clock values for every single tile. Furthermore, all data structures provide convenience functions and iterators to shift attention away from implementation details and towards actual algorithms when working with fiction as a developer.

The following code snippet demonstrates how a simple function can be implemented to randomly place vertices of a logic_network (constructed from a parsed Verilog file) on layout tiles.

```cpp
void naive_random_placement()
{
    // fetch current logic network from store
    auto network = store<logic_network_ptr>().current();
    auto n = network->vertex_count();
    // create an empty 4-phase USE layout of size n x n
    auto layout = std::make_shared<fcn_gate_layout>(
        fcn_dimension_xy(n, n), use_4_clocking, network);
    // for all logic vertices v
    for (auto& v : network->vertices())
    {
        auto placed_successfully = false;
        do
        {
            // sample a random tile t in ground layer
            auto t = layout->random_tile(GROUND);
            if (layout->is_free_tile(t))
            {
                // place v at t
                layout->assign_logic_vertex(t, v);
                placed_successfully = true;
            }
        } while (!placed_successfully);
        // place resulting layout in a store
        store<fcn_gate_layout>().extend() = layout;
    }
}
```

The given function naive_random_placement can for instance be implemented as a new command in the file io_commands.h by following the scheme of existing ones or by considering the official alice documentation. Note that the function as given here only places gates but does neither take care of their orientation nor their routing. A custom router can fully benefit from the whole functionality offered by the BGL as most of their (path finding) algorithms work on boost::grid_graphs as well.

Assuming the routing step has happened as well, one might want to convert the gate level abstraction to an actual cell-based implementation to conduct lower level optimizations. We further assume, the QCA-ONE gate library should be utilized for the technology mapping. The following code snippet does the job.

```cpp
void technology_mapping()
{
    // fetch current gate layout from store
    auto gates = store<fcn_gate_layout>().current();
    // prepare a library object for technology mapping
    auto lib = std::make_shared<qca_one_library>(gates);
    // apply library to generate a cell-based layout
    auto cells = std::make_sharedfcn_cell_layout(1lib);
    // store cell layout
    store<fcn_cell_layout>().extend() = cells;
}
```

For further information, we refer the reader to io_commands.h and techfcn_gate_library.h.

Additionally to the already introduced functionalities, fiction supports the use of externally clocked synchronization elements [20].

the use of multiple wires elements in the same tile in the gate-level abstraction already, and the direct construction of (cell-wise clocked) physical cell layouts without the use of any gate library.

## 4 CONCLUSION

In this paper, we introduced fiction, an extensible open source framework written in C++ for the layout, optimization, and physical design of Field-coupled Nanocomputing Circuits. The framework comes with efficient data structures, state-of-the-art algorithms, as well as rich scripting and logging functionalities. We thereby provide a foundation for future research in the community.

## ACKNOWLEDGMENTS

We would like to thank Gregor Kuhn and Mario Kneidinger for code contributions and the authors of the EPFL Logic Synthesis Libraries for their inspiring work.

## REFERENCES

[1] N. G. Anderson and S. Bhanja. Field-coupled Nanocomputing, Paradigms, Progress, and Perspectives. Springer, New York, 1st edition, 2014.
[2] S. Bohloud, Q. Shi, R. A. Wolkow, and H. Guo. Quantum Transport in Gated Dangling-Bond Atomic Wires. Nano Letters, 17(1):322–327, 2017.
[3] R. Brayton and A. Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. Proc. of International Conference on Computer Aided Verification, pages 24–40, Springer, 2010.
[4] C. A. T. Campos, A. L. P. Mariano, O. P. V. Neto, and F. S. Torres. USE: A Universal, Scalable, and Efficient Clocking Scheme for QCA. TCAD, 35(5):515–517, 2016.
[5] D. De Souza and N. Bjørner. Z3: An Efficient SMT Solver. In Facas-Etaps, Berlin, Heidelberg, 2008.
[6] R. F. Freimond et al. BNCS: Bidirectional Alternating Nanomagnetic Clocking Scheme. In Sbcci, pages 1–6. IEEE, 2018.
[7] D. Giri, G. Causapruno, and F. Riente. Parallel and Serial Computation in Nanomagnet Logic: An Overview. TVLSI, pages 1–11, 2018.
[8] M. Goswami et al. An efficient clocking scheme for quantum-dot cellular automata. Electron. Lett., pages 1–14, 2019.
[9] K. Hennesey and C. S. Lent. Clocking of Molecular Quantum-dot Cellular Automata. J. Vac. Sci. Technol. B, 19(5):1752–1755, 2001.
[10] M. K. Hu, H. Dey, N. Liebing, G. Csaba, A. Orlov, G. H. Bernstein, W. Porod, P. Krzyztofek, S. Sieries, and H. W. Schumacher. Edge-Mode Resonance-Assisted Switching of Nanomagnet Logic Elements. Ieee Trans. Magn., 51(11):1–4, 2015.
[11] T. R. Huff, H. Labidi, et al. Atomic White-Out: Enabling Atomic Circuitry through Mechanically Induced Bonding of Single Hydrogen Atoms to a Silicon Surface. ACS Nano, 11(9):8636–8642, 2017.
[12] C. S. Lent et al. Molecular Cellular Networks: A non von Neumann architecture for molecular electronics. In ICRC, pages 1–7, 2016.
[13] C. S. Lent and P. D. Tougaw. A Device Architecture for Computing with Quantum Dots. Proceedings of the IEEE, 85(4):541–557, 1997.
[14] J. Pitters, L. Livadaru, M. B. Haider, and R. A. Wolkow. Tunnel Coupled Dangling Bond Structures on Hydrogen Terminated Silicon Surfaces. JCP, 134(6), 2011.
[15] D. A. Reis et al. A Methodology for Standard Cell Design for QCA. In ISCAS, pages 2114–2117, 2008.
[16] J. Sirk, A. Lumsdaine, and L.-Q. Lee. The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, 2002.
[17] M. Sookan, H. Riener, W. Haaswijk, and G. De Micheli. The EPFL Logic Synthesis Libraries, 2018. arXiv:1805.05211.
[18] J. Timmer and C. S. Lent. Power Gain and Dissipation in Quantum-dot Cellular Automata. J. Appl. Phys., 91(2):823–831, 2002.
[19] F. S. Torres et al. Exploration of the Synchronization Constraint in Quantum-dot Cellular Automata. In Dsd, pages 642–648, 2018.
[20] F. S. Torres, M. Walter, R. Wille, D. Große, and R. Drechsler. Synchronization of Clocked Field-Coupled Circuits. In IEEE-NANO, pages 1–6, IEEE, 2018.
[21] F. S. Torres, R. Wille, P. Niemann, and R. Drechsler. An Energy-aware Model for the Logic Synthesis of Quantum-dot Cellular Automata. TCAD, PP(99), 2018.
[22] F. S. Torres, R. Wille, M. Walter, P. Niemann, D. Große, and R. Drechsler. Evaluating the impact of interconnections in Quantum-dot Cellular Automata. In Dsd, pages 649–656, 2018.
[23] V. Vankamamidi, M. Ottavi, and F. Lombardi. Clocking and Cell Placement for QCA. In Proceedings of the 24th Asia and South Pacific Design Automation Conference, pages 197–202, ACM, 2019.
[24] M. Walter, R. Wille, D. Große, F. S. Torres, and R. Drechsler. Placement & Routing for Field-coupled Nanocomputing Circuits. In Proceedings of the 24th Asia and South Pacific Design Automation Conference, pages 197–202, ACM, 2019.
[25] K. Walus, T. J. Dyiart, G. A. Jullien, and R. A. Budiman. QCADesigner: A Rapid Design and Simulation Tool for Quantum-dot Cellular Automata. TNANO, 3(3):26–31, 2004.