

Magnetoelastic NanoMagnet Logic Circuits

BY

DAVIDE GIRI

B.S., Politecnico di Torino, Turin, Italy, 2012

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2015

Chicago, Illinois

Defense Committee:

Wenjing Rao, Chair and Advisor

Zhichun Zhu

Mariagrazia Graziano, Politecnico di Torino

*This thesis is dedicated to my family
and to my beloved grandparents.*

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors Prof. Graziano and Prof. Zamboni, for their guidance throughout this work. Without their constant support it would have been impossible for me to accomplish my research goals. My deepest gratitude goes also to my advisor at UIC, Prof. Wenjing Rao, who gave me the possibility to undertake this research work. I take this opportunity to express my gratitude to Dr. Marco Vacca and Giovanni Causapruno for their assistance and dedication. A special thanks goes to my mates at VLSI Laboratory. Eventually, I want to thank from the bottom of my heart my family and my friends for the unwavering support they always gave me during this experience in the United States.

DG

TABLE OF CONTENTS

<u>CHAPTER</u>		<u>PAGE</u>
1	INTRODUCTION	1
1.1	CMOS scaling	1
1.2	Quantum-Dot Cellular Automata (QCA)	3
1.3	Magnetic QCA	5
2	NML BACKGROUND	8
2.1	QCA basics	8
2.1.1	Logic Gates	9
2.1.2	Signal propagation and Clock	11
2.2	Nano-Magnets Logic (NML)	14
2.2.1	Logic Gates	16
2.2.2	Magnetic Clock NML	17
2.2.2.1	Snake Clock Layout	20
2.2.2.2	Working frequency	22
2.2.3	Magnetoelastic Clock NML (ME-NML)	23
2.2.3.1	Circuit Layout	26
2.2.4	Intrinsic Pipeline	28
2.2.4.1	Interleaving	30
3	VHDL MODEL FOR THE MAGNETOELASTIC NML	33
3.1	Standard Cell Library	34
3.2	VHDL of the Standard Cells	38
3.2.1	Generic parameters	39
3.2.2	Register plus logic function	40
3.2.3	Area and Energy	40
3.2.3.1	Area information	42
3.2.3.2	Energy information	44
3.2.4	Hierarchical model	47
3.3	Circuit layout	48
4	CASE STUDY I: GALOIS FIELD MULTIPLIER	51
4.1	Galois Fields arithmetic	52
4.1.1	Galois Field Multiplier scheme	54
4.2	CMOS Pipelined Implementation	57
4.2.1	Timing analysis	59
4.2.2	Circuit simulation	62
4.3	ME-NML Implementation	64

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
4.3.1	Circuit design	65
4.3.1.1	Basic blocks	65
4.3.1.2	4-bit GFM	67
4.3.1.3	4-bit GFM with synchronization circuitry	69
4.3.2	VHDL description and circuit simulation	70
4.4	Magnetic Clock NML Implementation	74
4.4.1	Galois Multiplier scheme	75
4.4.2	Timing analysis	77
5	CASE STUDY I: GFM RESULTS COMPARISON	78
5.1	CMOS Results	78
5.1.1	Occupied area	79
5.1.2	Power consumption	81
5.2	Magnetoelastic NML Results	82
5.2.1	Upper synchronization network	83
5.2.2	Lower synchronization network	85
5.2.3	Occupied area	86
5.2.4	Power consumption	87
5.3	Magnetic Clock NML Results	88
5.3.1	Number of clock zones and magnets	88
5.3.2	Occupied area	89
5.3.3	Power consumption	90
5.4	Results Comparison	92
6	CASE STUDY II: MULTIPLY ACCUMULATE UNIT (MAC)	99
6.1	Parallel Implementation	101
6.1.1	Array Multiplier and Ripple Carry Adder	102
6.1.2	Full Adder and Half Adder	104
6.1.3	Basic blocks	106
6.1.4	VHDL description and circuit simulation	109
6.1.5	Timing Analysis	111
6.2	Serial-Parallel Implementation	113
6.2.1	Circuit scheme	113
6.2.2	ME-NML implementation	116
6.2.3	Timing analysis	121
6.3	Serial Implementation	122
6.3.1	Serial MAC scheme	123
6.3.1.1	Multiplier	123
6.3.1.2	Adder	125
6.3.1.3	Accumulator	125
6.3.2	Serial MAC with shared Accumulator	125
6.3.3	ME-NML implementation	127

TABLE OF CONTENTS (Continued)

<u>CHAPTER</u>		<u>PAGE</u>
6.3.4	Timing analysis	130
7	CASE STUDY II: MAC RESULT COMPARISON	131
7.1	Parallel MAC Results	131
7.2	Serial-Parallel MAC Results	133
7.3	Serial MAC Results	134
7.4	Results Comparison	136
7.4.1	Comparison conditions	137
7.4.1.1	Interleaving	137
7.4.1.2	Equal throughput	137
7.4.2	Results exploiting interleaving	138
7.4.3	Results without exploiting interleaving	141
7.4.4	Final considerations	142
8	CONCLUSIONS	144
8.1	Future work	146
	APPENDIX	148
	CITED LITERATURE	205
	VITA	210

LIST OF TABLES

<u>TABLE</u>		<u>PAGE</u>
I	ADDITION AND MULTIPLICATION FOR GF(2)	52
II	POLYNOMIAL MAPPING AND MULTIPLICATION TABLE FOR GF(8). PRIMITIVE: $X^3 + X + 1$	53
III	TIMING PERFORMANCE OF THE CMOS GFM	60
IV	POLYNOMIAL MAPPING FOR GF(16). PRIMITIVE: $X^4 + X + 1$	62
V	MULTIPLICATION TABLE FOR GF(16). PRIMITIVE: $X^4 + X + 1$	63
VI	TIMING PERFORMANCE OF THE ME-NML GFM	72
VII	TIMING PERFORMANCE OF THE MAGNETIC NML GFM	77
VIII	AREA OCCUPATION OF CMOS GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	80
IX	POWER CONSUMPTION OF THE CMOS GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	81
X	NUMBER OF CELLS AND MAGNETS OF THE BASIC BLOCKS FOR THE UPPER INTERCONNECTIONS	84
XI	NUMBER OF MAGNETS AND CELLS OF ME-NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	86
XII	OCCUPIED AREA OF ME-NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	87
XIII	POWER CONSUMPTION OF ME-NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	89
XIV	DIMENSIONS AND NUMBER OF MAGNETS OF THE MAGNETIC NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	90

LIST OF TABLES (Continued)

<u>TABLE</u>		<u>PAGE</u>
XV	AREA OF THE MAGNETIC NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	90
XVI	POWER OF THE MAGNETIC NML GFM, BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.	92
XVII	RATIO BETWEEN RESULTS FOR ME-NML AND THOSE FOR CMOS AND MAGNETIC NML. THE THIRD TABLE SHOWS THE INTERCONNECTION OVERHEAD TRENDS OF EACH TECHNOLOGY.	93
XVIII	TIMING PERFORMANCE OF THE PARALLEL MAC	112
XIX	TIMING PERFORMANCE OF THE SERIAL-PARALLEL MAC	121
XX	TIMING PERFORMANCE OF THE SERIAL MAC.	130
XXI	PARALLEL MAC PERFORMANCE RESULTS.	132
XXII	SERIAL-PARALLEL MAC PERFORMANCE RESULTS.	134
XXIII	SERIAL MAC PERFORMANCE RESULTS.	135
XXIV	COMPARISON OF THE 3 MAC IMPLEMENTATIONS, WITH THE THROUGHPUT BEING EQUAL.	136

LIST OF FIGURES

<u>FIGURE</u>		<u>PAGE</u>
1	Possible states of a QCA cell: Stable states '0' and '1' and unstable NULL state.	3
2	A) Hysteresis cycle of a multidomain magnetic material. B) Hysteresis cycle of a single domain magnetic signal. C) The two stable states of the NML base cell.	6
3	Signal propagation through a 3 cells QCA wire. II) The first cell is forced to '1'. III) The second cell switches to '1' due to electrostatic interaction. IV) The third cell switches.	8
4	Logic gates of QCA. A) Wire. B) Crosswire. C) Inverter. D) Majority Gate.	9
5	Clock mechanism. A) Clock zones. B) Clock signals.	12
6	Simplest clock phases layout, the circuit's area is partitioned in vertical stripes.	13
7	A) Horizontal wire. B) Inverter. C) Vertical wire. D) Majority Voter. E) AND port. F) OR port. G) Crosswire.	16
8	NML with Magnetic Clock mechanism. The magnetic field H is generated by the current I flowing through the copper wire, which is placed under the magnets plane.	18
9	The clock phase sequence is RESET, SWITCH, HOLD. A) Functioning in space (horizontally) and time (vertically) of a horizontal NML wire. B) The 3 clock signals. They are applied to different zones in space and they are repeated over time. They are the same in magnitude but with a 120 phase shift.	19
10	Snake-clock. (A) Top 2-D layout. (B) 3-D layout. The nanomagnets are placed between the two planes. Magnets cannot be placed where wires 2 and 3 are twisted.	20

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
11	Example of a simple circuit based on the <i>Snake-Clock</i> system. Different background colours refer to different clock zones. The arrows show the signal flow direction.	21
12	Magnetoelastic NML clocking mechanism. A) No voltage applied. B) Voltage applied to the electrodes. The PZT substrate induces a strain on the nanomagnets forcing their magnetization to their intermediate state.	24
13	Clock zones of the ME-NML. A) Clock zone with AND logic function. B) Clock zone with OR logic function. C) Circuit layout example. D) Placement grid for ME-NML Cells	26
14	NML signal synchronization. The three inputs must arrive to the two AND ports simultaneously. To do so the input wires must pass through the same number of clock zones. (A) Not working routing. (B) Correct routing.	29
15	Data interleaving. In this example 3 operations are executed in parallel: $A + B + C$, $D + E + F$, $G + H + I$. At every clock cycle the input data comes from a different operation. Since the feedback loop is 3 registers long, data from the same operation are fed with 3 clock cycles of delay.	30
16	ME-NML cells. A) 3×3 size. B) 3×5 size.	34
17	Full 3×3 Standard Cell Library for ME-NML.	35
18	Detailed measures of the ME-NML 3×3 cell.	42
19	A) VHDL hierarchical model. The information on energy dissipation and area occupation are propagated hierarchically toward the top entity. B) <code>generic</code> inputs and outputs of a Standard Cell.	47
20	A) CMOS Half Adder. B) ME-NML Half Adder. C) Waveforms for the 4-phase overlapped clock system. A color is associated to each clock signal. D) VHDL counterpart of the ME-NML circuit, it is the circuit described by the VHDL model. E) Timing diagram of an example of signal propagation through the adder.	49
21	Scheme of a 4-bit bit-serial Galois Field Multiplier ($GF(2^4)$).	56

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
22	Scheme of the 4-bit fully pipelined Galois Field Multiplier.	57
23	CMOS implementation of the 4-bit fully pipelined GFM. (A) Preskew circuitry for <i>dataB</i> . (B) Circuit body.	58
24	3-inputs XOR function constructed with AND, OR and Inverter gates.	65
25	Basic blocks of the GFM. ME-NML blocks on top are matched with the correspondent CMOS blocks.	66
26	Magnetoelastic NML implementation of a 4-bit Galois Multiplier. .	67
27	Equivalent circuit for the ME-NML GFM.	68
28	ME-NML Galois Multiplier with additional preskew and deskew networks.	69
29	Timing diagram of the operation 9×10 with the ME-NML 4-bit Galois Multiplier.	73
30	The 2-bit Magnetic NML Galois Multiplier, comprehensive of preskew and deskew networks.	74
31	The 4-bit Magnetic NML Galois Multiplier, comprehensive of preskew and deskew networks. The circuit is split in left part (on top) and right part (below), to facilitate its comprehension.	76
32	Post-route layout of the GFM in its CMOS implementation.	79
33	Comparison of area occupation for the CMOS GFM both with and without synchronization circuitry.	80
34	Comparison of power consumption for the CMOS GFM both with and without synchronization circuitry.	82
35	Basic blocks for the upper interconnections.	83
36	Layout of the upper interconnections for the 8-bit GFM. The second table is the optimized layout.	84
37	Basic blocks for the lower interconnections.	85

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
38	Layout of the lower interconnections for the 7-bit, 8-bit and 9-bit GFM.	86
39	Comparison of area occupation for the ME-NML GFM both with and without synchronization circuitry.	88
40	Comparison of area occupation for the Magnetic NML GFM both with and without synchronization circuitry.	91
41	Area comparison between the three GFM implementations without synchronization networks.	95
42	Power comparison between the three GFM implementations without synchronization networks.	96
43	Area comparison between the three GFM implementations with synchronization networks.	97
44	Power comparison between the three GFM implementations with synchronization networks.	98
45	Multiply Accumulate unit scheme.	100
46	4-bit MAC scheme. Array Multiplier on the left and Ripple Carry Adder on the right.	102
47	Half Adder and Full Adder realized with both ME-NML and CMOS technologies. (A) Half Adder. (B) Full Adder.	104
48	4-bit parallel ME-NML MAC unit. Labels identify the base blocks of Multiplier and Adder.	105
49	Base blocks of the Array Multiplier for the parallel MAC.	107
50	Base blocks of the Ripple Carry Adder for the parallel MAC	108
51	Base blocks for the interconnections of the parallel MAC implementation	110
52	Critical paths of the parallel MAC. (A) Critical paths of multiplier's base blocks. (B) Feedback loop of adder's base blocks.	112
53	Body of the 4-bit serial-parallel MAC.	114

LIST OF FIGURES (Continued)

<u>FIGURE</u>		<u>PAGE</u>
54	Full scheme of the 4-bit serial-parallel MAC.	115
55	Full Adder block for the serial-parallel MAC. Three patterns underline horizontal crossing, vertical crossing and feedback loop.	117
56	ME-NML implementation of the serial-parallel MAC.	118
57	Basic blocks for each region of the serial-parallel MAC.	120
58	Scheme of the 4-bit serial MAC (preliminary implementation). . . .	122
59	Handmade 4-bit multiplication algorithm.	123
60	Scheme of the 8-bit serial MAC with shared Accumulator and Adder.	127
61	Base blocks of the 8-bit serial MAC with shared Accumulator and Adder.	128
62	ME-NML implementation of the 8-bit serial MAC with shared Accumulator and Adder.	129
63	Central blocks of Multiplier and Adder for an optimized version of the Parallel MAC.	133
64	Area comparison of the three MAC implementations exploiting interleaving and with the throughput being equal.	139
65	Power comparison of the three MAC implementations exploiting interleaving and with the throughput being equal.	140
66	Area comparison of the three MAC implementations without interleaving and with the throughput being equal.	141
67	Power comparison of the three MAC implementations without interleaving and with the throughput being equal.	142

SUMMARY

Among the several alternative technologies proposed for the post-CMOS scenario, Quantum-dot Cellular Automata (QCA) is one of most promising for its high level of integration and low power consumption. The magnetic based implementation of QCA, named NanoMagnet Logic (NML), is the only one that can both work at room temperature and is feasible with current fabrication processes. Also, its magnetic nature opens up to new possibilities, like developing logic circuits with an intrinsic memory ability.

The base cells of NML technology are nanomagnets, which can be arranged on a plane to create any logic circuit. There is no standby power consumption and the energy required for magnets switching is several orders of magnitude lower than latest CMOS transistors. However the network for controlling the cells' magnetization can nullify the advantages in terms of power losses. This is the case of the Magnetic Clock NML [1], which has been extensively and thoroughly studied in literature. A novel implementation of NML technology, the Magnetoelastic NML (ME-NML), drives the nanomagnets through an electric field instead of a magnetic field, highly reducing the power consumption. This solution has already been proved theoretically and experimentally, however up to now only elementary circuits have been studied.

The Magnetoelastic NML is the subject of this work. To fully understand its potential it is mandatory to analyze complex architectures keeping into account all the physical constraints related to the fabrication process.

SUMMARY (Continued)

First of all, because of the absence of a tool for design and simulation, we developed a RTL model for handling ME-NML circuits. The model also embeds the capability of evaluating area occupation and power consumption. Due to the strong regularity of the ME-NML circuits layout, we were able to define a Standard Cell library, which is a big step toward the creation of an aided design tool.

Secondly, through a case study we developed an accurate comparison of ME-NML with the Magnetic Clock NML and the state of the art CMOS transistor. ME-NML performances were excellent, enough to largely overcome both the other technologies. This was also the first approach to ME-NML from the architectural level, so it provided general information on circuit design. Nonetheless we could generalize the behavior of our case study to serial-parallel architectures.

Once the validity over other technologies was proven, it was mandatory to understand which kind of architectural organization maximizes the performance of the ME-NML. Therefore through a second case study we performed the first step of this investigation, comparing three different versions of a MAC unit: parallel, serial-parallel and serial. The parallel approach guarantees the best results, but it requires a certain level of interleaving.

In addition to attaining their specific goal, each one of the two case studies has been very resourceful in other fields. In fact they both helped identifying, from an architectural point of view, the major limitations of ME-NML technology as well as its strengths. Therefore this work also provides the first general guidelines for ME-NML design.

CHAPTER 1

INTRODUCTION

1.1 CMOS scaling

Over the past three decades the inexorable evolution of electronics had as foundation the ever-smaller device dimensions of silicon-based CMOS technology, which has been exponentially improving in both performance and density of integration. Today, however, the conventional physical scaling is experiencing asperities and, as forecasted in the International Technology Roadmap for Semiconductors [2], it is expected to reach its boundaries soon.

This decay counts several factors [3], physical and material limits above all. Basically, due to both electrostatics and tunneling mechanisms, ultra-small MOSFETs leakage currents begin to be comparable to the drain current. The increased leakage current, due to downsizing, forbids the threshold and supply voltages reduction, denying a speed increase. Correspondingly the higher electric field and the high concentration of dopants deeply impact electronic transport. These are some of the well known effects of down scaling: Drain Induced Barrier Lowering (DIBL), Short Channel Effect (SCE), Punch-Through and subthreshold inversion, mobility degradation, band-to-band tunneling [4][5]. Another challenge involves power consumption and thermal dissipation: The power density has been growing, as the supply voltage did not scale as much as the channel length. Furthermore some constraints come from economical aspects and the lithography-based fabrication techniques.

Due to all these factors, keeping up with the Moore's Law will most probably be a challenge that will not be answered by Silicon CMOS nanoelectronics. A lot of research on alternative technologies has been carried out to preserve the same rate of performance improvements. The efforts have been focused toward two main directions [2]:

- Innovation of CMOS materials and structures. Demonstrated examples are: SOI (Silicon On Insulator) transistors, with an insulator layer between substrate silicon body, and FinFET, where a multigate structure heavily reduces short channel effects.
- Creation of completely new nanoelectronic devices, called "Beyond CMOS Devices", able to replace CMOS technology. One of the most promising architectures is the Quantum-dot Cellular Automata (QCA). Nanotechnologies like QCA offer very high integration density, but they are still in a premature stage: A reliable and functional realization still requires extended study from the device up to the architectural level.

Current transistors exploit electronic charge to store information, therefore switching between logic levels involves charge movement, thus requiring a current flow and a consequent Joule dissipation. Energy losses are then an intrinsic characteristic of charge based electronics and, as explained before, highly scaled transistors will not be able to preserve the charge due to significant leakage. It is clear that charge based devices do not seem to be able to maintain the cost per function improvements of the last decades. The idea is to replace the charge with a new kind of information token such as for instance: Polarization of nanomagnets, change in molecular configuration, electron spin or position of a micromechanical object.

1.2 Quantum-Dot Cellular Automata (QCA)

Ever since the introduction of the Cellular Automata idea in 1993 [6], Quantum-Dot Cellular Automata (QCA) has been attracting an increasing interest. It is a valuable candidate for the post-CMOS era, because it effectively addresses the problems of device density and power dissipation.

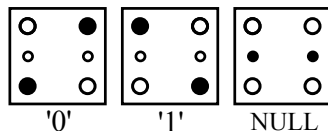


Figure 1. Possible states of a QCA cell: Stable states '0' and '1' and unstable NULL state.

QCA technology foundation is a bistable base cell; properly organized arrays of these cells can realize logic functions. The first proposed implementation used a square cell with 4 quantum dots in the corners. Since electrons repel each other, if two electrons are available for each cell, at equilibrium they will be found in two diagonal dots. Since there are only two diagonals, only two states are possible: '0' and '1' [7]. To allow a correct signal propagation we will see in Chapter 2 that a third unstable state (NULL state) is necessary, therefore two more dots need to be added (Figure 1). This is just the generic base cell, but the theoretical principle of the QCA can be realized in other ways, depending on the technology used.

Up to now the literature contains five physical implementations, while the first two present strong limitations, the others are way more promising.

- Metal QCA [8][9]. This was the first physical implementation and it had a purely demonstrative purpose. In fact it can work only at temperatures close to the absolute zero, unless the cell size is downscaled to atomic values. The base cell is composed by metallic structures on a SiO_2 substrate and the quantum dots are basically aluminum islands. The links between dots are made of Al_2O_3 tunnel junctions.
- Semiconductor QCA [10][11]. This approach exploits common electronic devices' structures, the cells and their dots are realized with GaAs and Si-Ge heterostructures. The electrons behavior is driven by a voltage applied to metal contacts. Compared to Metal QCA the operation temperature can be higher, but still it does not work at room temperature. Another limitation concerns the available fabrication processes, which cannot meet the requirement of very small and identical cells.
- Magnetic QCA or NanoMagnet Logic (NML) [12]. The base cell is a single-domain nanomagnet with dimensions lower than $100nm$, its two possible magnetizations correspond to '0' and '1' logic values [13]. About speed (hundreds of MHz) and dimensions this implementation is less interesting than the Molecular QCA, it is also slower than CMOS systems. What makes Magnetic QCA attractive lies in its magnetic nature, it has exceptionally low power consumption and a strong logic-in-memory predisposition [1][14][15]. But the most relevant advantage is the physical realization feasibility with current technol-

ogy, it allows to study and experiment on QCA based architectures on a higher abstraction than the single cell, facing directly design problems common to any QCA implementation.

- Molecular QCA [16]. The fundamental states of the Molecular QCA cell correspond to different charge distributions in a complex molecule, the charge movement can be triggered by electrons reacting with the oxide-reduction center of the molecule. Using molecules every QCA cell would be identical to the others and would have the very competitive dimension of a few nanometers. Moreover molecules reactions work perfectly at room temperature and are extremely fast, the expected switching speed of this implementation is of the order of THz [17][18][19]. This is the most promising approach, even though a functioning realization is still far: Current technology cannot manipulate single molecules as required yet. Another delicate issue is the transduction of electrical signals from and to information understandable by the molecule, up to now there is not any valid solution to this.
- Silicon Atomic QCA [20]. The QCA principle is implemented using atoms as quantum-dots. It has been proved that the dangling bond (DB) state of silicon atoms can be exploited as a quantum dot. Up to now the experimental results are promising and the electrostatic control over the charge within DB assemblies has been verified [21].

1.3 Magnetic QCA

Magnets have already been used in electronics for memory applications, the innovation of Magnetic Quantum dot Cellular Automata (MQCA), also called NanoMagnetic Logic (NML), is to use magnets to implement logic functions. The result are digital circuits with intrinsic

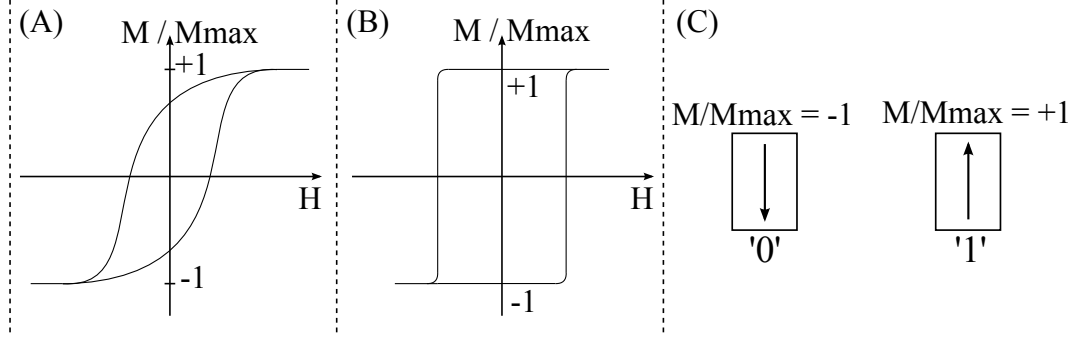


Figure 2. A) Hysteresis cycle of a multidomain magnetic material. B) Hysteresis cycle of a single domain magnetic signal. C) The two stable states of the NML base cell.

memory capability [22]. The current fabrication techniques allow to produce the NML base cells [23], which are nanomagnets with dimensions between $50nm$ and $100nm$. Magnets so small behave differently than bigger ones, they have only one magnetic domain, which means that the magnetization does not vary across the magnet, and the hysteresis cycle gets as in Figure 2.B. Hence nanomagnets smaller than $100nm$ can have two stable states only, which will be used to represent '0' and '1' values. The hysteresis cycle describes how magnetization (M) changes as a function of the magnetic field (H) applied.

As already anticipated there are several reasons that make the NML study worthy, even if the working frequency is limited:

- NML is the only QCA implementation that works at room temperature and it can be fabricated with current technology [23].

- Magnets do not dissipate static power and a single magnet switching absorbs around $30k_B T$. Therefore NML potentially has an extremely low power consumption.
- Since the difference between QCA and CMOS technologies is bottomless, to fully comprehend the potential of QCA, it is mandatory to investigate complex architectures, also considering all the working and fabrication constraints. Fortunately most of the architectural study on NML could probably be applied to other implementations like the molecular QCA, which seems far more promising than Magnetic QCA but it is still not supported by current technology.

CHAPTER 2

NML BACKGROUND

2.1 QCA basics

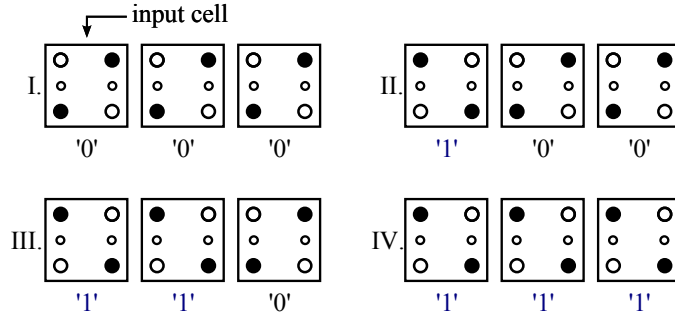


Figure 3. Signal propagation through a 3 cells QCA wire. II) The first cell is forced to '1'. III) The second cell switches to '1' due to electrostatic interaction. IV) The third cell switches.

The QCA base cell described in Section 1.2 contains six quantum dots, allowing to represent the '0' and '1' logic values and the NULL state (Figure 1). Placing cells one next to the other on the same plane it is possible to construct digital circuits, where the signal propagation through cells is due to electrostatic interaction. A series of adjacent cells is called wire, Figure 3 represents step by step the information propagation through a 3 cells wire. Forcing the first cell to '1' causes the switch of the nearby cell, due to electrons repulsion. In the same way

the second cell, after switching to ‘1’, will influence the last one. We can say that information propagates with a Domino-like effect.

2.1.1 Logic Gates

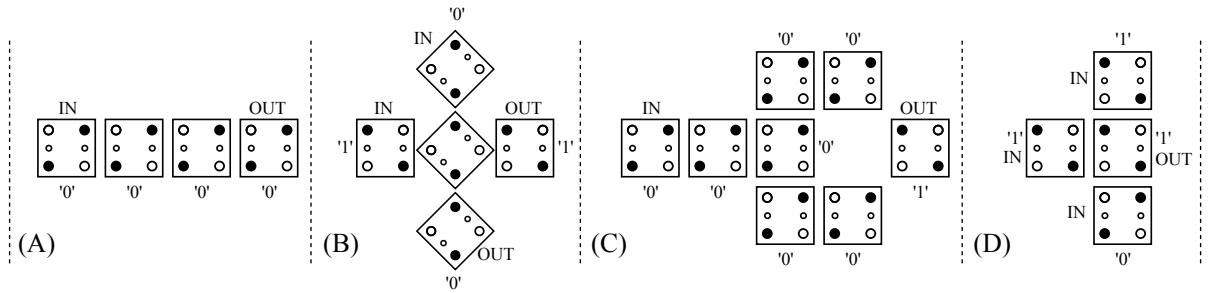


Figure 4. Logic gates of QCA. A) Wire. B) Crosswire. C) Inverter. D) Majority Gate.

QCA technology counts four basic logic blocks, they are depicted in Figure 4, where *IN* and *OUT* identify inputs and outputs. The blocks are: The wire (Figure 4.A), the crosswire (Figure 4.B), the inverter (Figure 4.C), the majority gate, also called majority voter (Figure 4.D). These are the standard gates of the theoretical QCA, keep in mind that each different QCA implementation has a slightly different ports set, even if the basic concepts remain unchanged.

Crosswire

The crosswire allows two independent signals to cross each other on the same plane

without interference. The one in Figure 4.B is just one example of crosswire, its actual realization strongly depends on the QCA implementation adopted. An alternative that has been proposed is to use multilayer structures, just like with CMOS technology. Even though it seems that this solution would be suitable for Magnetic QCA, unfortunately at the current time a multilayer structure is still not feasible due to fabrication complexity.

Inverter

Its logic function is a simple inversion, obtained through a diagonal coupling of cells. Notice that the signal gets duplicated before the inversion to strengthen the diagonal electrostatic interaction, which is weaker than the horizontal or vertical ones. Based on the QCA implementation employed, there are other possible configurations that provide inversion.

Majority Gate

This logic block is a peculiarity of QCA circuits, together with the inverter it allows to design any logic function. It is a three input port, where the output is equal to the majority of the input values. Referring to Figure 4.D, notice that the central cell is subject to the influence of the top, left and bottom cells. The output will be '1' if that is the value of at least two inputs, and the same works for '0'. The majority gate (or majority voter) logic function is:

$$F = AB + BC + AC.$$

2.1.2 Signal propagation and Clock

Despite what said above, the electrostatic interaction is not strong enough for a signal to propagate through a wire. The switching of a cell requires as much energy as the barrier between its two stable states, that is the energy keeping electrons trapped in the dots. Of course this amount of energy E_k (Kink Energy) is strictly related to the QCA implementation used, the cell size and the operating temperature. However this value is generally high enough not to allow autonomous data propagation. For this reason there is the need for an external mean able to control the signal propagation by acting on the energy barrier between the two stable states. Such barrier can be lowered by applying an electric field, as a consequence the electrons will be forced into the central dots leaving the cell in an unstable state, which is referred to as NULL state. Once removed the external field the cell will stabilize either at '0' or '1', depending on the state of neighbor cells.

So the main idea is that if we want a cell to assume the same value as its neighbor, we force such cell in an unstable state through an external electric field, and then we simply release the field. This control field is called *clock*. In principle this technique could work with an infinite number of cascaded cells, but practically the number has to be small. Otherwise there will be propagation errors mainly due to thermal noise [24]. Therefore a spatial flow control system is mandatory.

From the remarks above it is clear that a signal cannot pass through a whole circuit at once, the cells pattern would be too long. The solution is to break the circuit in small sections and let signals go over one section at a time, in a pipelined manner. So circuits are partitioned in

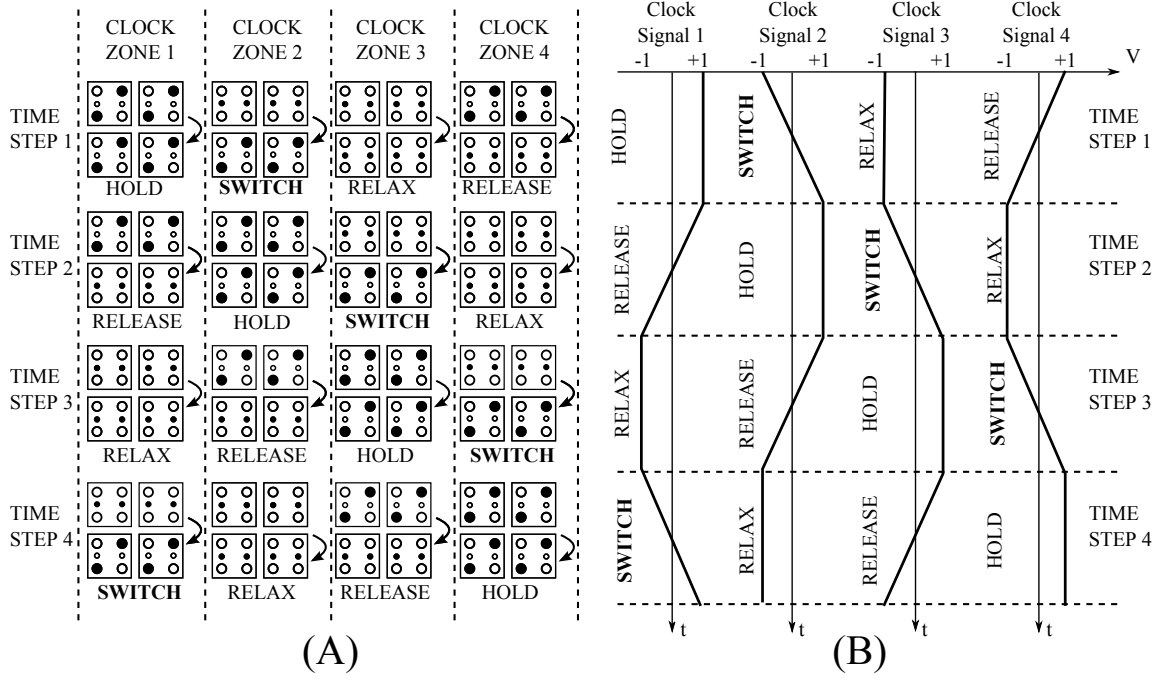


Figure 5. Clock mechanism. A) Clock zones. B) Clock signals.

small areas, where each area counts a limited number of cascaded cells; these areas will be called *clock zones*. In the classical scheme the spatial and timing control of the circuit is conferred to a four phases clocking system. There are then four clock signals with the same waveform but different phase. The 2^{nd} , 3^{rd} and 4^{th} clocks will have respectively 90° , 180° , 270° phase shift with respect to the 1^{st} clock. Each of the partitioned section will receive one of the four clocks, a correct assignment of the clocks will assure a correct circuit functioning.

Figure 5 shows the clock waveforms on the right and the functioning of a wire divided into four clock zones on the left. As explained we need a clock that can force cells in their unstable

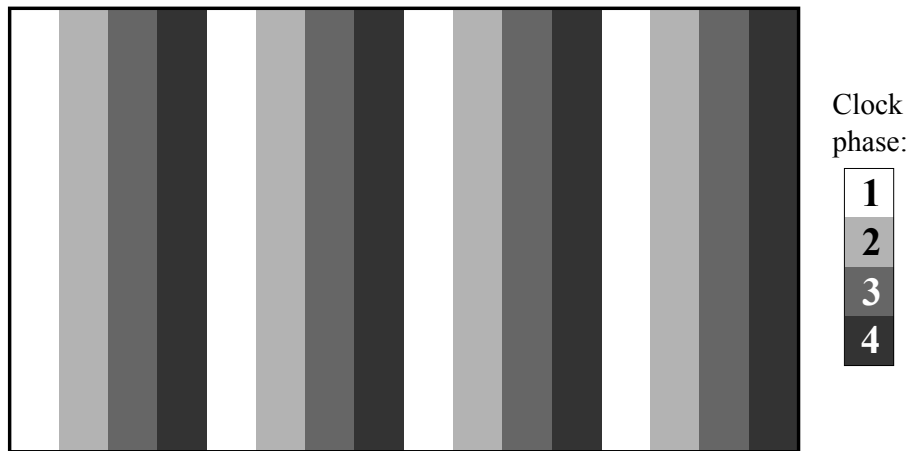


Figure 6. Simplest clock phases layout, the circuit's area is partitioned in vertical stripes.

state before the switching phase. The clock waveform has got four phases, as clearly pointed out by Figure 5.B:

Hold phase. The potential barrier is kept high by a high clock voltage. The cell cannot be influenced by neighbors.

Release phase. The clock voltage goes from high to low and so does the energy barrier. At the end of this phase the cell reaches its NULL state.

Relax phase. The potential barrier is kept low, so the barrier between stable states stall at its minimum. The cell is in the NULL state.

Switch phase. The clock voltage goes from low to high and so does the potential barrier. The cell will stabilize in one of the two states, depending on the neighbor cells.

In Figure 5.A the signal goes from left to right. When a clock zone is switching it is influenced by nearby cells. Cells on the left are in HOLD, they act as input, while cells on the right are in the RELAX phase, so they have no influence on the switching cells. In between of HOLD and RELEASE, the cells are either relaxing or latching. This methodology assures data propagation in a specific direction, it is thus fundamental to arrange the clock zones properly. For a correct functioning a signal must pass through the clock zones in order from 1 to 4 and then 1 again.

At this point the last issue is to decide how to arrange the clock zones. In principle the circuit area can be subdivided in clock zones with any shape, but of course technological limitations due to the clocking network must be always kept in mind. A straightforward arrangement of clock zones is represented in Figure 6, the circuit is divided in columns. The four shades of grey correspond to the different clock phases. The simple subdivision in columns has the strong disadvantage of allowing signal propagation in one direction only, following the clock phases order: 1,2,3,4,1,2,... To be able of dealing with any kind of circuit, the structure has to be more complex, it must allow propagation in any direction.

2.2 Nano-Magnets Logic (NML)

The most recent advancements in fabrication techniques, especially the lithography, allow to build logic circuits using magnets. While magnets have already been used in electronics for memory applications, the innovation of this implementation is to use magnets to implement logic functions. As a result NML circuits are digital circuits with intrinsic memory capability. The base element of NML is a very small bistable magnetic cell. Since it is not a permanent

magnet, its magnetization can be influenced by external means. Therefore nanomagnets placed side by side will arrange themselves in an antiferromagnetic manner, because of the attraction between opposite poles.

The nanomagnets dimensions must be between $50nm$ and $100nm$. The upper limit assures that the magnets only have one magnetic domain, which means that the magnetization does not vary across the magnet and the hysteresis cycle gets as in Figure 2.B. The two saturation values $M = +1$ and $M = -1$ are the only stable states, therefore they are associated to logic values '0' and '1'. The lower bound of $50nm$ is, instead, crucial to avoid the superparamagnetic effect, which would cause the magnetization to vary together with thermal fluctuations. To assure thermal stability the energy barrier between the two stable states must be at least $30k_B T$. As from Figure 2.C the two states have magnetizations in opposite directions, so they both lie on the same axis. At the equilibrium, if one side of the magnet is longer than the other, thanks to shape anisotropy, the magnetization will be forced along the longer axis (easy axis). Therefore it is important that in NML the ratio between the magnets dimensions (aspect ratio) is within the 1 : 1.2 range. For a correct signal propagation it is mandatory that every base cell is equal in shape to the others. Consequently, the more troublesome is the production, the higher will be the fault probability. That is why the rectangular and elliptical shapes are the most used, as they assure the best precision in the fabrication process.

The main advantage of Magnetic QCA is to be realizable with current technology (electron beam lithography or high end optical lithography) together with its ability to operate at room temperature. The fabrication feasibility was first proven by researchers of the University of

Notre Dame in Indiana (US). They built horizontal wires, vertical wires and majority gates [25]. A Magnetic QCA horizontal wire was also created by researchers of Politecnico di Torino.

2.2.1 Logic Gates

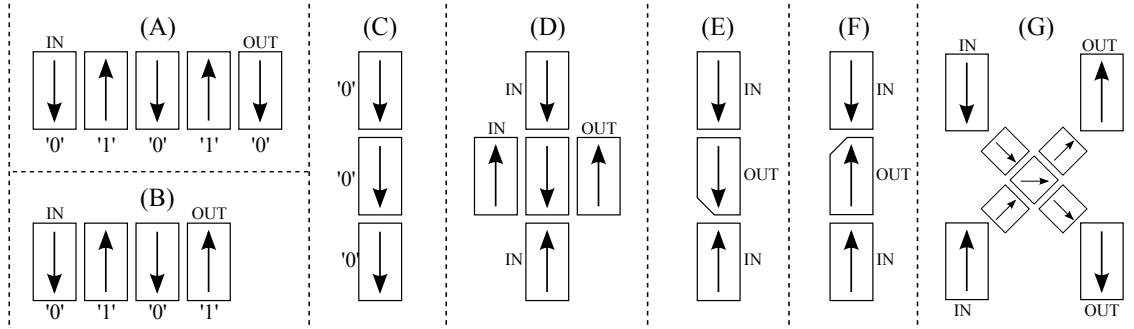


Figure 7. A) Horizontal wire. B) Inverter. C) Vertical wire. D) Majority Voter. E) AND port. F) OR port. G) Crosswire.

Even though the set of logic gates for the NML circuits recalls the generic QCA basic blocks (Figure 4), there are some differences and improvements. It is understandable that, moving from the general idea to the physical implementation, the general ports can be optimized based on the actual technology features. Figure 7 shows the complete set of logic blocks for NML circuits. The main difference with generic QCA is the horizontal coupling: Horizontally magnets align themselves antiferromagnetically, each magnet has inverted polarization with respect to the neighbors. So the inverter can be simplified to a simple horizontal wire with an even number of

magnets as in Figure 7.A. On the other hand an odd number of adjacent magnets would result in a buffer function, that is a simple wire (Figure 7.B). Vertically the coupling is ferromagnetic, so no inversion is possible (Figure 7.C). The majority voter, depicted in Figure 7.D, is pretty much the same as for general QCA.

Another disparity comes from the possibility of obtaining specific logic gates modifying the shape of a magnet: By making magnets with slanted edges it is possible to create *AND* and *OR* logic functions [26]. QCA would generally need a three inputs majority gate to obtain AND and OR logic ports, while only two inputs are needed for non-majority based gates, considerably optimizing area occupation and layout entanglements. The different-shaped magnets acquire a preferential state, which they will leave only when both inputs, from above and below, are up or down, implementing as a consequence an AND or OR logic function (Figure 7.E, Figure 7.F).

At the current time the NML crosswire realization does not have experimental proof of reliability yet. A possible implementation is the one represented in Figure 7.G, the crossing is made of five square cells ($50nm - 100nm$ of edge) that have four stable states instead of two. In this way they can let pass through two signals simultaneously.

2.2.2 Magnetic Clock NML

One solution for controlling the nanomagnets magnetization in NML circuits is the Magnetic clock, as proposed in [12] and verified experimentally in [23]. The magnetic field is generated by a current flowing through a wire positioned under the magnets plane (Figure 8). The material for the wire is copper, buried in a ferrite yoke envelope for field confinement. The wire's

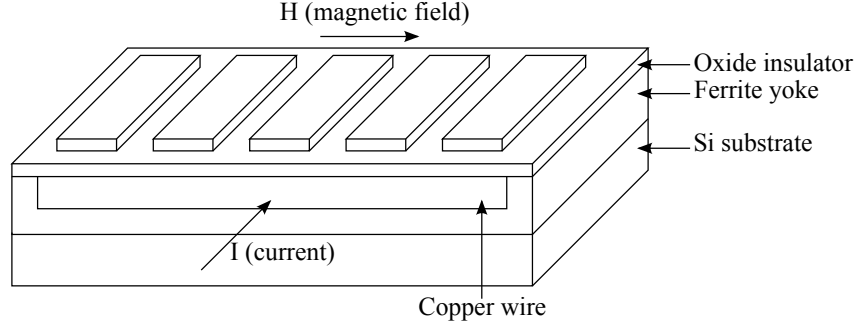


Figure 8. NML with Magnetic Clock mechanism. The magnetic field H is generated by the current I flowing through the copper wire, which is placed under the magnets plane.

thickness must be enough to generate a magnetic field able to force cells to the intermediate state (NULL state) [27].

As explained in Section 2.1.2 a multiphase clock system is required. The classic scheme has 4 phases, but also a 3-phase clock is feasible [28][29][30]. The Magnetic NML normally exploits a 3 phase clock system is normally exploited. Figure 9 shows the functioning of the 3-phase clock of a horizontal wire over time (vertical axis), just like in Figure 5 for the generic QCA.

Each clock zone undergoes three phases in the following temporal sequence: RESET, SWITCH and HOLD. The RESET ($clock = 1$) erases the information, leading cells to an intermediate state. In the SWITCH phase the clock goes to zero, so cells can assume a magnetic orientation. The orientation is influenced by the nearby cells being in HOLD state, as cells in the RESET state cannot affect the neighbors. When a group of cells, in the same clock zone, is in the HOLD phase, they have a stable magnetization.

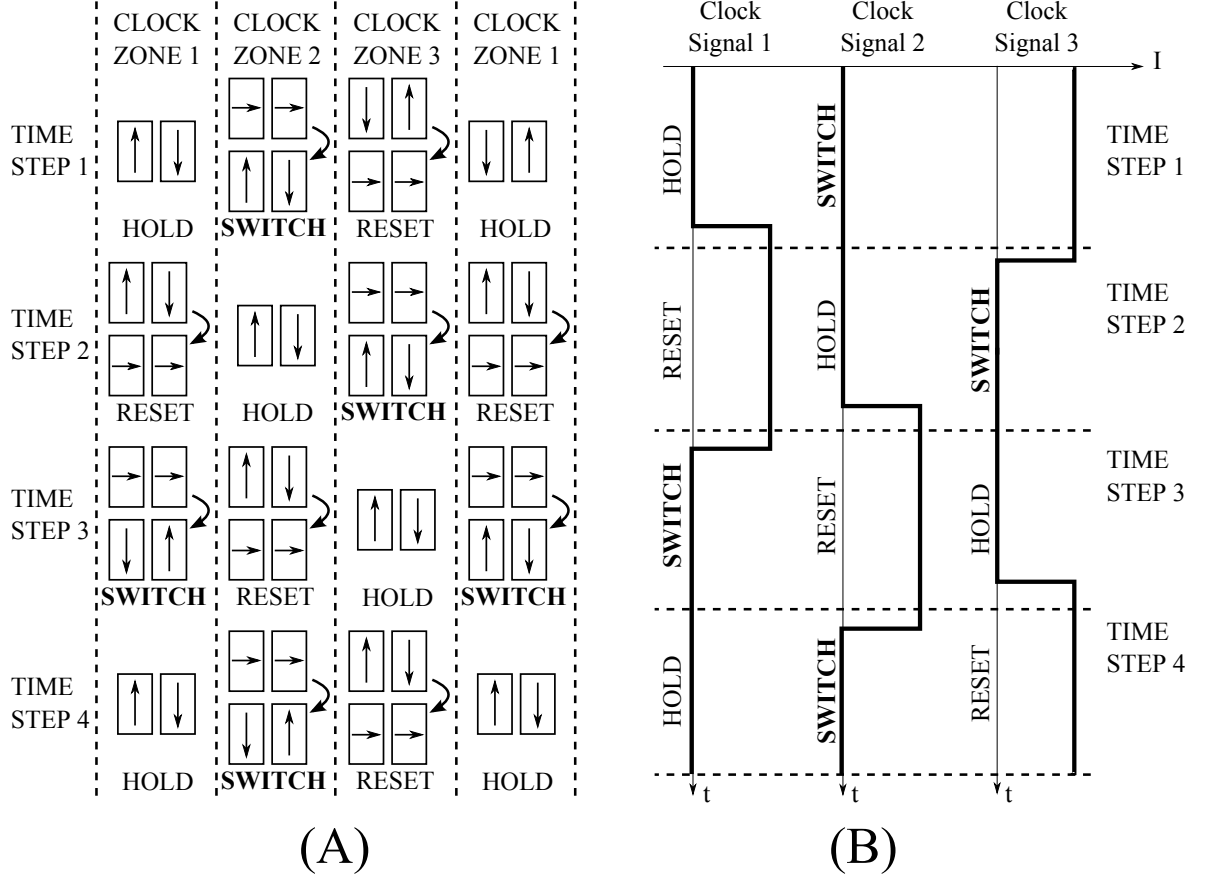


Figure 9. The clock phase sequence is RESET, SWITCH, HOLD. A) Functioning in space (horizontally) and time (vertically) of a horizontal NML wire. B) The 3 clock signals. They are applied to different zones in space and they are repeated over time. They are the same in magnitude but with a 120 phase shift.

To assure a correct signal propagation the RESET phase applied to different zones must overlap in time as in Figure 9.B, where the RESET state lasts slightly more than $2\pi/3$. The reason lies in the fact that when a zone is in the SWITCH phase, the two neighbor zones

must be respectively in HOLD and RESET phase. However if the field of the SWITCH zone is removed and the field is applied to the RESET zone at the same time, a back propagation phenomenon could take place. Initially, when the field is removed from the SWITCH zone, the RESET zone would still be in the HOLD state, as magnets need a finite time to switch from a stable polarization to the intermediate state. In Figure 9.A we can see how the value in *Time step 1* on the left is propagated step by step to magnets in the clock zone on the right.

2.2.2.1 Snake Clock Layout

The generic QCA is based on a 4-phase clock system, however it is also possible to use a 3-phase clock [12], given that the signals are overlapped. The clock network for Magnetic NML is a 3-phase overlapped system, called Snake-clock; its layout and 3D structure are depicted respectively in Figure 10.A and Figure 10.B.

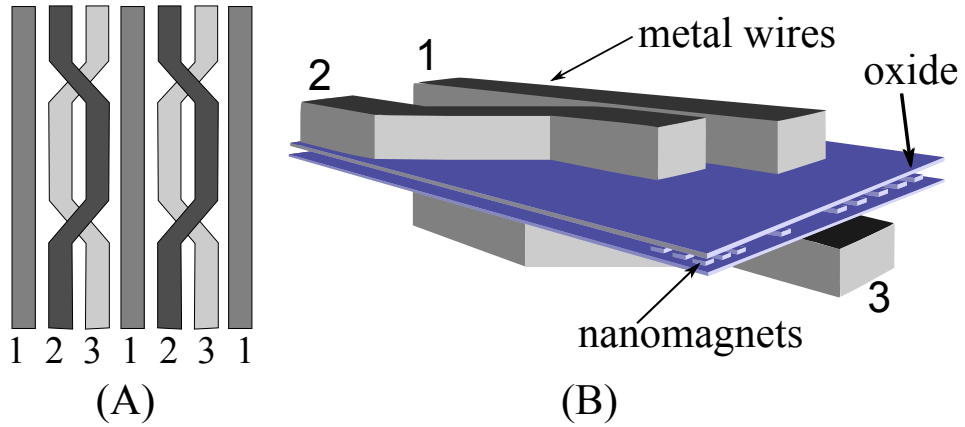


Figure 10. Snake-clock. (A) Top 2-D layout. (B) 3-D layout. The nanomagnets are placed between the two planes. Magnets cannot be placed where wires 2 and 3 are twisted.

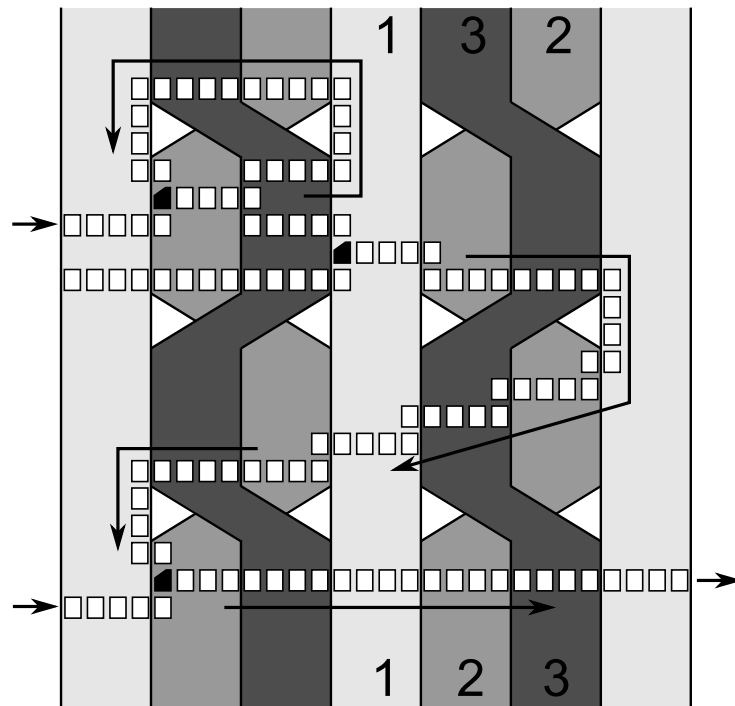


Figure 11. Example of a simple circuit based on the *Snake-Clock* system. Different background colours refer to different clock zones. The arrows show the signal flow direction.

The Snake-clock is based on the scheme in Figure 6, but with three phases only and with an expedient that allows propagation in both directions: left-right, right-left. The clock wires are basically simple metal wires parallel to the magnets plane, two positioned above and one below [28]. Two thin oxide layers provide separation between clock wires and nano-magnets. One clock wire is straight (number 1), while the other two have a complementary zig-zag shape. They are like twisted wires, but they do not display any interference, as they are on different

planes. In the case in Figure 10.B the wires 1 and 2 are routed on the same plane, while the clock 3 is on the other one.

Considering now the top view in Figure 10.A, it is straightforward to understand that magnets cannot be placed on areas corresponding to the wires twisting, as they would be affected by both clock wires 2 and 3. Moreover, in those regions, wires are not parallel to the magnets long side, hence the generated magnetic field would force them in the wrong state.

Figure 11 shows a very simple circuit based on the Snake-Clock system. The direction of the information flow is highlighted by arrows, signals propagate through clock zones in the order 1, 2, 3 and so on. The clock wires twisting divides the circuit area in horizontal stripes with alternate propagation directions. Furthermore, as required by this clock mechanism, there are no magnets placed over the twisting areas. The magnets with a slanted edge required for the AND logic function are highlighted in black.

2.2.2.2 Working frequency

The main limitation of NML technology is the maximum working frequency, which is intrinsically bounded. To obtain the highest possible clock frequency the clock zone width should be equal to that of a single magnet. However the usual width is 3-5 [24] because of several factors: fabrication limitations, thermal noise, latency, throughput. The more are the consecutive magnets in a clock zone the lower will be the clock frequency. The constraints on the clock frequency are mainly related to the clock mechanism chosen and the fall and rise time of the adiabatic switching of clock signals, mandatory to reduce power consumption. Less critical is instead the bound derived from the switching time of nanomagnets from the intermediate

(NULL) state to a stable one and viceversa. The NML circuit speed is expected to be of the order of $10 - 100MHz$ [31][32][33].

In the beyond-CMOS scenario, NML technology is a good solution but it cannot aim to completely substitute CMOS. Despite the clear benefits for what concern occupied area, power consumption and memory ability, NML's clock frequency cannot keep up with CMOS.

2.2.3 Magnetoelastic Clock NML (ME-NML)

Recently a valuable alternative to the Magnetic Clock NML has been proposed and studied: the Magnetoelastic Clock NML, also referred to as ME-NML [1][34].

In the previous section (2.2.2) the proposed external mean, responsible for the magnets switching, was the Magnetic Clock with a Snake-clock layout. The idea was to position clock wires below or above the magnets plane. A current flowing through the wires would generate a magnetic field able to control the cells magnetization. The generated field is then along the magnets' short side of the magnets, forcing cells in an intermediate unstable state.

The interest in Magnetic QCA is mainly due to the very low power consumption, several times lower than the latest CMOS transistors. While this is true for the magnets switching, unfortunately it does not apply to the clock generation system: $1\mu m$ copper wires with a required current of $545mA$ [35]. Due to Joule losses the power dissipation of the clocking system is very high, nullifying the advantage of a low-power magnets switching.

To solve this problem an alternative solution has been recently proposed [35][34], it is based on the Magnetoelastic effect: the magnetization of magnetic materials undergoing mechanical stress is bonded. Applying a mechanical stress with proper intensity and direction magnetic cells

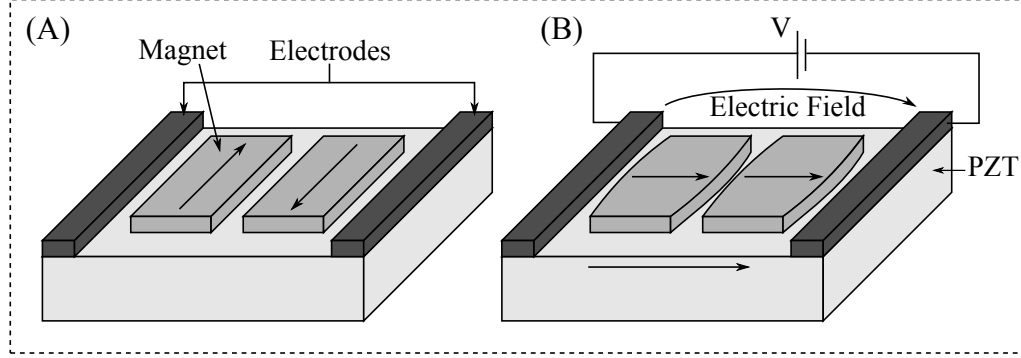


Figure 12. Magnetoelastic NML clocking mechanism. A) No voltage applied. B) Voltage applied to the electrodes. The PZT substrate induces a strain on the nanomagnets forcing their magnetization to their intermediate state.

can be forced into the RESET state. The magnetic cells ($10nm$ thick) are coupled with a $40nm$ thick PZT layer (Figure 12.A). To maximize the mechanical coupling, magnets are deposited directly onto the piezoelectric material. For a proper strain transfer, the PZT substrate has to be much thicker than the magnets. The magnetic material is then controlled by applying a voltage (few mW) to the piezoelectric. When the voltage is applied, the strain induced by the piezoelectric material, forces the magnetization of the magnets layer to the intermediate position, parallel to the short edges (see Figure 12.B).

The electrodes are deposited on top of the PZT, while the wires that drive the electrodes can be placed in additional layers, just as for CMOS. This makes this NML implementation compatible with CMOS fabrication.

This approach comes from a previous idea based on multiferroic structures instead of simple magnets [33][36]. The performances of the pure multiferroic structure are better, but there are two major fabrication problems. The aspect ratio is critical, there are only $2nm$ of difference between the length of the two cell's sides. Such a low resolution is hardly achieved with the Electron Beam Lithography. Moreover the electrodes should be only a few nanometers thick, a request that does not comply with the current technology. A pair of them is necessary for every element, to apply the required voltage. The advantage of the solution with the simple magnets is the feasibility with current fabrication techniques. Even if its performances are slightly worse than the multiferroic solution, they are anyway remarkably better than the previous NML solutions.

Since the clock system exploits a voltage instead of a current, the power consumption is extremely low, meeting the unmatched expectations for the initial Magnetic QCA concept. In [1], after a detailed analysis, the selected magnetic material is Terfenol, an alloy of Terbium, Dysprosium and Iron. The choice is mainly based on three parameters:

- maximum stress that can be applied to avoid permanent damage on the magnets;
- maximum value of electric field that can be tolerated by the piezoelectric material, since it is an insulator;
- minimum stress to force magnets in the RESET state;
- assure shape anisotropy equal of at least $30K_bT \approx 1.24 \cdot 10^{-19}J$, to have negligible effects of the thermal noise on the magnets stability;
- minimum aspect ratio for fabrication feasibility;

- tolerance to process variation of $\pm 20\%$, remaining within the working range.

2.2.3.1 Circuit Layout

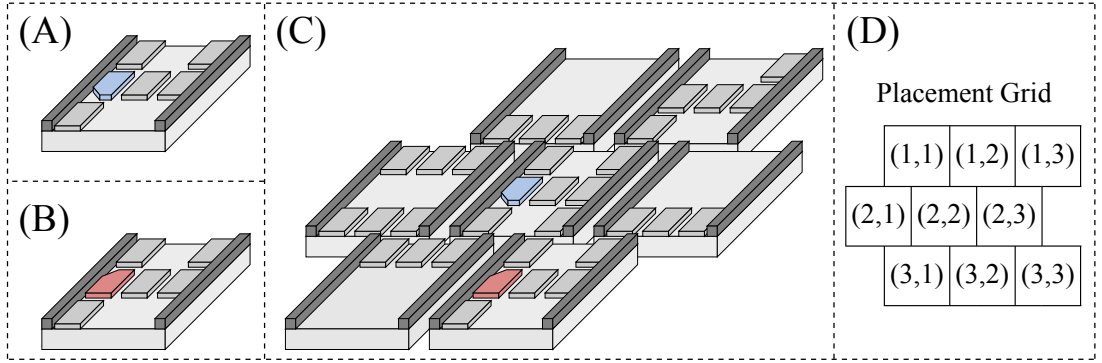


Figure 13. Clock zones of the ME-NML. A) Clock zone with AND logic function. B) Clock zone with OR logic function. C) Circuit layout example. D) Placement grid for ME-NML Cells

Starting from the structure just described in Section 2.2.3, MagnetoElastic clock NML (ME-NML) circuits are composed by mechanically isolated islands, like the one in Figure 13. Each island corresponds to a clock zone and it is driven by one of the clock signals, applied as a voltage on the Platinum electrodes. Notice that the electrodes position on top of the PZT is compatible with CMOS fabrication and leads to a uniform electric field distribution on the magnets plane.

The presence of the electrodes makes the clock zones communication on those sides impossible. The signal propagation among cells is allowed only through the top and bottom sides, which are free from electrodes. For this reason the Majority Voter port cannot be constructed. Therefore the basic logic gates exploited are inverter, AND (Figure 13.A) and OR (Figure 13.B) [26], so that any logic circuit can be implemented.

Figure 13.C shows how to put together the clock zones to create a circuit. As already said, the communication among cells can take place only through the top and bottom corners, because of the electrodes. For this reason the cells in a row are shifted with respect to the adjacent ones, to assure a correct signal propagation. In fact the cells are placed on a grid as in Figure 13.D, where the coefficients identify row and column of the cell's positioning within the circuit.

In the example of Figure 13.C the clock zones have both height and width equal to three nanomagnets. This is the solution adopted throughout the whole work, it has been chosen over the five magnets version. Thermal noise [24] and fabrication constraints allow cells dimensions to vary only between 3 and 5 nanomagnets. Small dimensions lead to smaller electrodes and cells, requiring then a very high resolution fabrication process. The minimum size feasible with current technology is 3. Bigger dimensions will relax the technology constraints, but will increase the error probability due to thermal noise and decrease the maximum circuit speed. If too many cascaded magnets are present in a clock zone, the signal propagation will be error prone.

The size of the electrodes varies according to the clock zones dimensions. They are $30-40nm$ for the three magnets cells, while $70-100nm$ for the five magnets case. This kind of electrodes are already available for CMOS technology.

Figure 13 does not highlight how and which clock signals are routed to the clock zones. It will be clarified later on in Section 3.3, where it will also be explained which kind of multi-phase clock system best suits the Magnetoelastic NML implementation.

2.2.4 Intrinsic Pipeline

In a N-phase clock system, signals need a clock period to propagate through N clock zones. As a consequence the delay of a signal depends on how many clock zones it has to cross. This is quite different from CMOS where wires with different lengths have very similar delays. Each clock zone crossed by a signal can be modelled as a register, as a result it is easy to understand that NML circuits (just like QCA) are intrinsically pipelined. Every group of N adjacent clock zones has an overall delay of a clock cycle.

For this reason signal synchronization is a very delicate issue in NML circuits. Figure 14 is useful for clarifying the problem, the input wires routing is correct in part B, while incorrect in part A. For a proper circuit functioning the three input signals must reach the two AND ports simultaneously, to do so the routing must assure that the input wires cross the same amount of clock zones. The example was presented for the Magnetic NML case, but the same concept applies to ME-NML as well as any QCA implementation.

The problem gets more complex when dealing with feedback signals, see for example the feedback in Figure 11 at the top left corner. While the external input of the AND port arrives

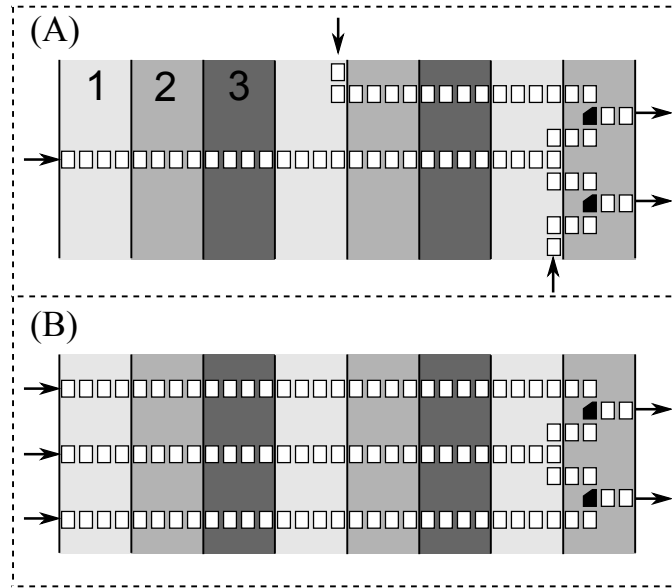


Figure 14. NML signal synchronization. The three inputs must arrive to the two AND ports simultaneously. To do so the input wires must pass through the same number of clock zones. (A) Not working routing. (B) Correct routing.

at every clock cycle, the second one (the feedback) arrives later. The output of the AND port needs two clock cycles to be fed back. Therefore at every clock cycle the AND operation is performed between the new input and the output result obtained 2 cycles before. The proper result will arrive at the next time step. Notice that the longer the feedback wire, the longer the delay. The input must then be delayed long enough to match the length of the feedback loop. In conclusion the inputs have to be fed with a delay equal to the feedback length, reducing then the throughput, particularly in case of long loops. If, for instance, a circuit has a feedback 5 cycles long, only an input every 5 cycles can be fed. Therefore the throughput is $1/5$ of what it

could be if the input was continuous. In fact, at any time, only 1/5 of the magnets will contain useful data.

2.2.4.1 Interleaving

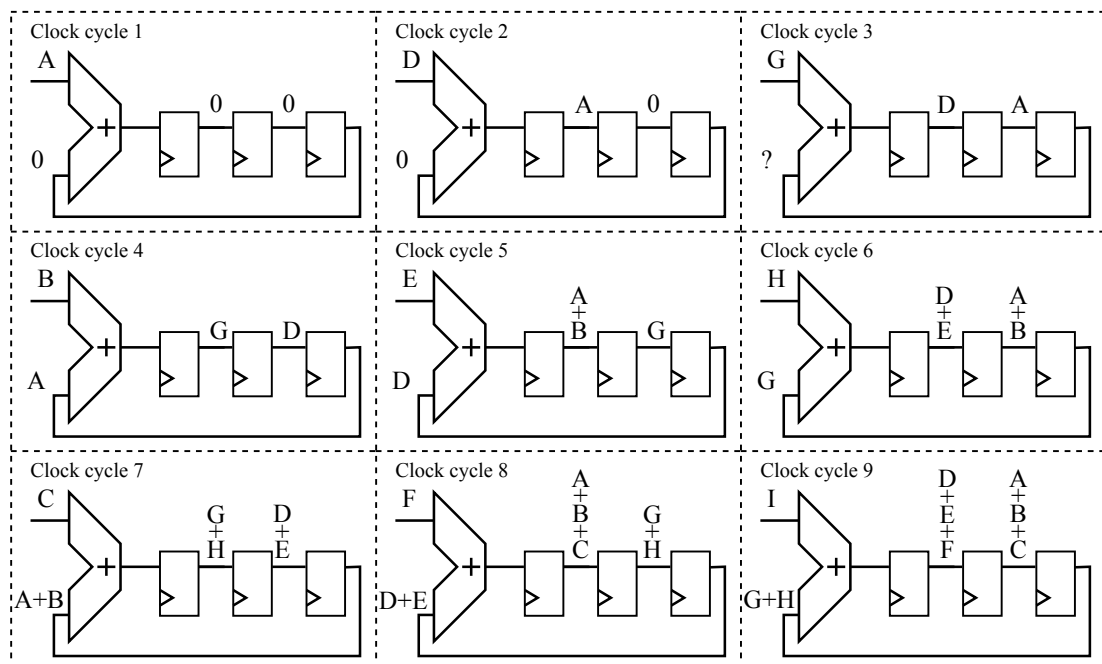


Figure 15. Data interleaving. In this example 3 operations are executed in parallel: $A + B + C$, $D + E + F$, $G + H + I$. At every clock cycle the input data comes from a different operation. Since the feedback loop is 3 registers long, data from the same operation are fed with 3 clock cycles of delay.

The problem of pipelining in CMOS sequential circuits is very complex and delicate. Unfortunately it is even worse for the NML (QCA) technology, as it is not possible to control the

pipeline level. Since the pipeline is intrinsic to the technology, it cannot be eliminated, it can only be reduced by optimizing the circuit layout.

The usual improvement techniques for CMOS pipelining are jump prediction and instruction reordering, but for NML (QCA) they can only reduce the problem, they are not able to solve it. A radical solution is the *Data Interleaving* [22], which allows to reach the maximum throughput. The idea is to have a continuous flow of input data. Multiple non correlated set of operations are executed in parallel, so that the delay time between an input and the next is filled with other operations.

Figure 15 shows an example of data interleaving mechanism. Three operations are executed in parallel: $A + B + C$, $D + E + F$, $G + H + I$. At clock cycle 1 the first data of the first operation, A , is given as input. For a correct synchronization, B has to be fed when A reaches the end of the feedback loop, which is 3 clock cycles long. Therefore we give A , B and C as inputs respectively at clock cycles 1, 4 and 7. In the intermediate time steps we can execute in parallel the other two operations, to reach the maximum throughput. This is possible only if the three operations are uncorrelated. So at clock cycle 2 the input is not the number data of operation 1, but D : the first data of operation 2. And in the same way we will input G , the first data of operation 3, at clock cycle 3. The same goes for the next time steps; the input order is the following: $A, D, G, B, E, H, C, F, I$. That is: OP.1 DT.1, OP.2 DT.1, OP.3 DT.1, OP.1 DT.2, OP.2 DT.2, OP.3 DT.2, OP.1 DT.3, OP.2 DT.3, OP.3 DT.3 (where OP. stands for operation and DT. for data).

Data interleaving is a simple expedient that can solve the deep pipeline problems, but if the required number of operations to execute in parallel is too high then it might not be a feasible solution anymore. The number of required parallel operations is equal to the delay (in terms of clock cycle) of the longest loop inside the circuit. During the circuit design phase for NML circuits it is then extremely important to keep loops as short as possible.

CHAPTER 3

VHDL MODEL FOR THE MAGNETOELASTIC NML

The main purpose of this work is to study for the first time the Magnetoelastic Clock NML (ME-NML) from the architectural point of view, taking into account physical and technological constraints. The work directly concerns ME-NML, but some aspects could be easily generalized to other QCA implementations. The Magnetoelastic clock system has been verified [1], but no design and architectural study is present in literature. As for now there is no automated tool for properly simulating and synthesizing NML circuits. For this reason researchers at Politecnico di Torino developed a VHDL model (preliminary done in [37][38][39]) and a design tool, named ToPoliNano [40]. This tool is specifically constructed for the Magnetic clock NML.

Based on this idea we developed a RTL model in VHDL language which allows to:

- easily simulate any ME-NML circuit, verifying its functioning;
- hierarchically estimate the circuit performance in terms of area occupation and power consumption.

The model keeps consideration of all the relevant technology constraints. The result will be a circuit with an embedded evaluation function for power and area. Thanks to the clock network, each clock zone samples one data per clock cycle, therefore it can be modeled with a register as they have the same behavior.

3.1 Standard Cell Library

In Section 2.2.3.1 Figure 13 shows that the ME-NML layout is based on mechanically isolated islands, which will be referred to as cells or clock zones, as they receive their own clock signal. It has been already mentioned that, for fabrication and physical limitations, the height and width of a cell can be of either 3 or 5 magnets. For this work we chose the 3×3 cell dimension, as it is the smallest size feasible with current lithographic resolution. Compared to bigger cells, it has a shorter critical pattern (number of cascaded magnets) leading to both an higher working speed and a better signal propagation reliability. Based on our choice all the drawings and circuits from now on will exploit 3×3 clock zones, but the VHDL model is generalized for any cell size.

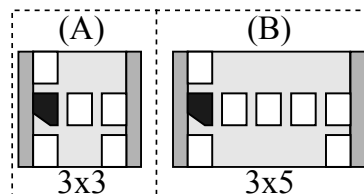


Figure 16. ME-NML cells. A) 3×3 size. B) 3×5 size.

We noticed that, due to the small size of this ME-NML cells, there is a limited number of possible magnets configurations. Hence the totality of the conceivable clock zones is reasonably small. This interesting feature of ME-NML triggered the idea of designing a finite set of standard

cells: a Standard Cell Library, where each element is described in VHDL language. The result is that, assembling cells from the library, any digital circuit can be designed. This standard cell approach confers to ME-NML a propensity for design automation, making this technology very much suitable for having its own simulation and synthesis tool.



























Standard Cells									
Wire	'0'		'1'		Crosswire				
					Inverter	'0'		'1'	
									
									
	"00"	"01"	"10"	"11"	Double Inverter				
					AND				
Double Wire					OR				

Figure 17. Full 3×3 Standard Cell Library for ME-NML.

The full 3×3 Standard Cell Library is tabulated in Figure 17. The logic gates are basically the same as for the Magnetic clock NML (Figure 7): Wire, Crosswire, Inverter, AND, OR. But

here they must be distinguished also by layout and orientation, not only by their logic function. The main reason is that the whole library is thought in the perspective of a future automated tool for circuit design.

Cells lying within the same row of Figure 17 can be derived from each other by horizontal and/or vertical flipping. Since they represent different orientations of the same cell, they are described by the same VHDL **entity**. The binary numbers in the table will be given as **generic** parameters to state the cell orientation. The only exceptions are *Double Wire*, *AND*, *OR*: These cells are put in the same row to get a more compact image, but they have to be defined with different VHDL **entity**.

Each cell is modeled as a CMOS register plus, if needed, an ideal logic port.

Wire. The word *wire* in NML technology refers to a series of adjacent magnets. With a proper clock system a *wire* can propagate signals with a domino-like behavior. In section 2.1.1 we explained that the horizontal alignment of magnets is antiferromagnetic, while vertically each magnet has the same polarization of its neighbors. Therefore, for a clock zone to have a Wire function, the number of horizontal magnets must be odd. Since wires do not carry any logic function they are simply described as registers. As clear from Figure 17, there are four different wires in the library:

- Vertical Wire. There are two possible orientations: *left* and *right*.
- Horizontal Wire. There are two possible orientations: *up* and *down*.
- Long Wire. From one corner to the opposite one. There are two possible orientations.

- 2 Outputs Wire. This wire covers three corners, so there are two outputs, as there cannot be more than one input. It is the only cell with 4 possible orientations.

Double Wire. It contains two independent wires with length of three magnets. In the model this cell is described with two different registers. From the logic function point of view it is just like putting together two single wire cell. There are two Double Wire cells in the library, described by two different VHDL **entities**. Notice that the horizontal and the vertical Double Wire are not two different orientations of the same cell.

Crosswire. It is modeled similarly to the Double Wire, but physically the wires cross each other. This interference-immune crossing is vital, since for now NML is still a planar technology.

Inverter. The horizontal alternate alignment of magnets is exploited to obtain the inverter function: Any even number of adjacent horizontal magnets generates an inversion. The VHDL model only has a small difference compared to the Wire case. To implement the inversion an ideal CMOS inverter has to be added at the input of the registers. Just like for the wires two inverters can be present within the same cell, but only horizontally. The vertical coupling is ferromagnetic, so the inversion does not take place. The library also contains a cell with both an inverter and a horizontal wire.

- Inverter. It is horizontal only. There are two possible orientations: *up* and *down*.
- Inverter plus wire. There are two possible orientations: inverter *up* and inverter *down*.

- Double Inverter. Beside the fact that it implements the inversion, it is the same as the horizontal Double Wire.

AND. In Section 2.2.1 it is explained how AND and OR gates can be obtained by modifying the shape of a magnet [26]. For visual clarity the magnets with the slanted edges are filled with black. A cut on the bottom left corner provides the AND function. None of the six AND cells in the library can be derived from another one by flipping, even if they look like they could. Notice that the slanted edge is always on a left corner of the magnet. Therefore each AND cell is described by a different VHDL `entity`. The first four cells have one output, while the others have two outputs.

- AND. There are four different AND cells with only one output. The inputs can be either both on the left or both on the right, while the output on the other side can be either at the top or the bottom.
- AND with two outputs. There are two different AND cells with two outputs. The inputs can be either on the left or on the right, while the outputs are on the other side.

OR. The only difference from AND cells is the position of the slanted edge, which is on the upper left corner.

3.2 VHDL of the Standard Cells

In this section we will see how the actual VHDL for standard cells works. The Listing 3.1 is used as an example, it contains the complete code for the *Inverter plus Wire*. The inverter (4

adjacent magnets) and the wire (3 adjacent magnets) are horizontal, so the cell can be flipped around its horizontal axis.

3.2.1 Generic parameters

Each VHDL `entity` has many `generic` parameters that allows to differentiate clock zones belonging to the same type of cell and their relative positioning within the circuit (see lines 11-16 of listing 3.1). In Figure 19.B they are represented as inputs of the Standard Cell. These parameters do not affect the logic or the functioning of the circuit, indeed they provide information useful for performance estimation or for a future possible aided design tool.

- **PHASE.** For ME-NML we chose a 4-phase clocking system. This `generic` defines which one of the four clock signals will be connected to the clock zone. This information is redundant, as the required clock signal is directly connected to the *clk* port, but we included it to assure a better suitability of this model to a design tool.
- **ROW and COLUMN.** ME-NML circuits are composed by cells disposed in a grid-like fashion, just like depicted in Figure 13.D. ROW and COLUMN refer to the relative position of a cell within the circuit described by the upper level entity. It will be explained in section 3.2.4 that the model is hierarchical. If single cells are considered as layer 1, an entity in layer 2 will assemble them to create the final circuit or part of it.
- **ORIENTATION.** As represented in Figure 17, when cells can be obtained from each other by a simple flipping, they are described by the same VHDL file. The ORIENTATION parameter says which one to use. Once again, this does not affect the logic or the circuit performance: It is just a matter of layout.

- **H and L.** The choice for this work has been to exploit 3×3 clock zones. So the height and width (in terms of nanomagnets) of a cell are always equal to 3. Anyway the model is as generic as possible, so the height and width are parameters: H and L.

3.2.2 Register plus logic function

The Inverter plus Wire cell is composed by two parallel series of magnets: 4 for the inverter and 3 for the wire. Therefore it is modeled by two D Flip Flop registers, plus an ideal inverter applied to one of the outputs. Lines 37-38 of Listing 3.1 contain the registers instantiations, while the inversion function is at line 35.

3.2.3 Area and Energy

In this section we refer once again to the Listing 3.1. Each cell described with VHDL evaluates and gives as output its own number of magnets (*n_mag*), its area occupation (*area_eff*, *area_tot*) and power consumption (*Er*, *Ec*) (Figure 19.B). The number of magnets is evaluated at line 34, while the other values are calculated by a component named **area_and_energy** (lines 23-31 and 40-41). This component, starting from the number of magnets, height and width of a cell, provides as output the required information on area and power. For the number of magnets evaluation, the central part of the Crosswire (the cross) is considered equivalent to 3 magnets.

Listing 3.1. Inverter plus Wire

```

1  entity inv_with_wire is
    generic (PHASE: std_logic_vector(1 downto 0); — Clk phase.
              ROW: natural; — Relative cell position (row)
              COLUMN: natural; — Relative cell position (col)
              ORIENTATION: std_logic;
              H: natural; — Height (# of magnets)
              L: natural; — Width (# of magnets)
    port (
        d1,d2: in std_logic; — Inputs
        clk: in std_logic; — Depends on the phase
        q1_n,q2: out std_logic; — Outputs
        n_mag: buffer natural; — # of magnets
        n_zones: out natural := 1; — # number of cells
        area_eff: out natural; — Total magnets area
        area_tot: out natural; — Cell area
        Er: out natural; — Switching energy
        Ec: out natural; — Clock network losses
    end inv_with_wire;

19 architecture behavior of inv_with_wire is
    component reg is — D FlipFlop (1bit)
        ...
    end component;
    component area_and_energy is
        generic (H: natural; — Height (# of magnets)
                  L: natural; — Width (# of magnets)
        port (
            n_mag: in natural; — # of magnets
            area_eff: out natural; — Total magnets area
            area_tot: out natural; — Cell area
            Er: out natural; — Switching energy
            Ec: out natural; — Clock network losses
        end component;
    signal q1: std_logic;
begin
34  n_mag <= L*2+1; — Evaluate the number of magnets using H and L.
    q1_n <= not q1; — Inversion

37  Wire1: reg port map(d => d1, clk => clk, q => q1);
    Wire2: reg port map(d => d2, clk => clk, q => q2);

40  Evaluate_area_energy: area_and_energy generic map(H,L)
    port map(n_mag, area_eff, area_tot, Er, Ec);
end behavior;

```


3.2.3.1 Area information

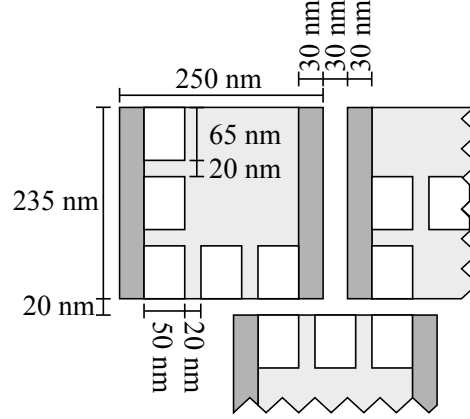


Figure 18. Detailed measures of the ME-NML 3×3 cell.

Figure 18 reports the complete clock zone measures and the distance from nearby cells.

Here is the list of the relevant measures:

- Magnets. Height: $H_{mag} = 65nm$, width: $W_{mag} = 50nm$.
- Magnets separation. Both horizontal and vertical separation: $Sep_{mag} = 20nm$.
- Electrodes. Width: $W_{electrode} = 30nm$.
- Cells separation. Horizontal: $Sep_{horiz_{cell}} = 30nm$, vertical: $Sep_{vert_{cell}} = 20nm$.

The fixed values above are assigned to the proper constants in the model (Listing 3.2), so that the component **area_and_power** will be able to evaluate the correct area information for each cell. Each cell gives as output two data related to area occupation:

Magnets Area. It is the area of one magnet multiplied by the number of magnet on the cell.

$$A_{magnets} = n_{mag} \cdot (H_{mag} \cdot W_{mag}) \quad (3.1)$$

Cell Area. It is the area of the cell, including the electrodes and the separation space among cells. It will be used to evaluate the total area of the circuit. Since in this work the cell dimension is fixed to 3×3 , the Cell Area will be the same for every cell.

$$H_{cell} = 3 \cdot (H_{mag} + Sep_{mag}) = 255nm \quad (3.2)$$

$$W_{cell} = 3 \cdot W_{mag} + 2 \cdot (Sep_{mag} + W_{electrode}) + Sep_{horiz_{cell}} = 280nm \quad (3.3)$$

$$A_{cell} = H_{cell} \cdot W_{cell} \quad (3.4)$$

From now on, for an easier design procedure, the vertical separation between cells will be null. The height of the substrate (and electrodes) will be depicted $20nm$ higher, occupying then the area previously devoted to the separation.

Listing 3.2. Nanomagnets and Cell measures

```

3  — CELL and MAGNET size — All values are expressed in [nm] —
6  constant HMAG:          natural := 65; — Nanomagnets height
   constant WMAG:          natural := 50; — Nanomagnets width
   constant SEP_MAG:        natural := 20; — Nanomagnets separation
   constant WELECTRODE:     natural := 30; — Electrode width
   constant SEP_HORIZ_CELL: natural := 20; — Vertical separation between cells
   constant SEP_VERT_CELL:  natural := 30; — Horizontal separation between cells

```

3.2.3.2 Energy information

The `area_and_power` component actually estimates the energy dissipation E and not the power. Knowing the working frequency f_{clk} , which for this work was chosen equal to $100MHz$, the power P can be easily derived:

$$P = E \cdot f_{clk} \quad (3.5)$$

The VHDL contains the definition of all the constants needed for this section, they are shown in Listing 3.3. The main sources of energy dissipation in NML circuits are basically two:

Magnets Switching. It is the intrinsic energy loss required to force magnets in the NULL state (E_r in Listing 3.1). The switching can be either adiabatic or abrupt: For the Magnetic clock NML the difference in term of losses was extremely wide, so the switching had to be adiabatic. But ME-NML behaves differently: The energy consumption is still equal to $30K_bT$ if adiabatic, but only $180K_bT$ (the whole energy barrier for $50 \times 65 \times 10nm^3$ nanomagnets) if abrupt. Since in both cases the consumption will be negligible

compared to the second component, the choice is the abrupt switching, which reaches better performance.

After defining how much energy is dissipated by the switching of a single magnet (E_{mag}), to calculate the energy consumption of a cell the only information needed is the number of magnets (n_{mag}) on that cell:

$$E_{cell} = n_{mag} \cdot E_{mag} \quad (3.6)$$

Clock Network. It is the energy dissipated by the clock network mainly due to Joule losses (E_c in Listing 3.1). Since PZT (piezoelastic materials in general) is an insulator, a ME-NML cell behaves as a capacitor. Therefore the main contribution to clock losses (for a $100MHz$ frequency) is the charge of such capacitor. The capacitance is estimated in equation Equation 3.7 [35].

$$C = \frac{\epsilon_0 \cdot \epsilon_r \cdot t_{PZT} \cdot H_{cell_eff}}{W_{cell_eff}} \quad (3.7)$$

The first three constants are the absolute dielectric constant (ϵ_0), the relative dielectric constant of PZT (ϵ_r), the thickness of the PZT substrate ($t_{PZT} = 40nm$ [35]). The other two values are the effective dimensions of a clock zone, without the inclusion of the separation between cells. Hence $H_{cell_eff} = 235$ and $W_{cell_eff} = 250$ (Figure 18).

Equation 3.8 evaluates the voltage that must be applied to a clock zone to force it into the RESET state.

$$V = \frac{W_{cell_eff} \cdot \sigma}{Y \cdot d_{33}} \quad (3.8)$$

Listing 3.3. Constants for Energy estimation

```

1  ——— CONSTANTS FOR ENERGY EVALUATION ———
2
3  — For switching energy evaluation
4  constant Kb:          real := 13.8065e-23; —Boltzmann const.(m^2*kg*s^-2*K^-1)
5  constant T:           real := 300.0;      —Room temperature (K)
6  constant EMAG:        real := 180*Kb*T;    —
7  — For clock energy evaluation
8  constant VACUUMPERM:   real := 8.854e-12; —Vacuum permittivity (F/m).
9  constant RELPERM:      real := 1300.0;     —Substrate relative perm. (-)
10 constant T_PZT:        real := 40e-9;      —Electrodes thickness (m)
11 constant STRESS:       real := 28e+6;      —Applied stress (Pa)
12 constant YOUNG.MODULUS: real := 80e+9;     —Young modulus for Terfenol (Pa)
13 constant PZT.CONST:    real := 150e-12;    —Substrate const., piezo coeff. (m/V)

```

In this formula we have the applied stress ($\sigma = 28MPa$), the Young modulus for Terfenol ($Y = 80GPa$) and the coefficient for strain and applied voltage coupling in the PZT substrate ($d_{33} = 150pm/V$). Normally for our cells the applied voltage should be in the range of $0.7 - 1.3V$ [35]. Finally the energy required to charge the capacitance of one cell is listed in equation Equation 3.9.

$$E_{clk} = \frac{1}{2} \cdot C \cdot V^2 \quad (3.9)$$

In this work the clock will be always chosen equal to $f_{clk} = 100MHz$. The clock period T_{clk} depends on technological constraints, not on the logic, as the critical path for signals is fixed, no matter which logic has been implemented. The constraints on the clock duration are manifold, all of them derive from technology choices:

- maximum number of magnets per clock zone;
- number of clock phases (3 or 4 for the implementations studied in this work);

- usage of either adiabatic or abrupt switching.

The power contribution of the circuit for clock generation is negligible, as the circuit counts a limited number of transistors [12]. Therefore this component will not be taken into account.

3.2.4 Hierarchical model

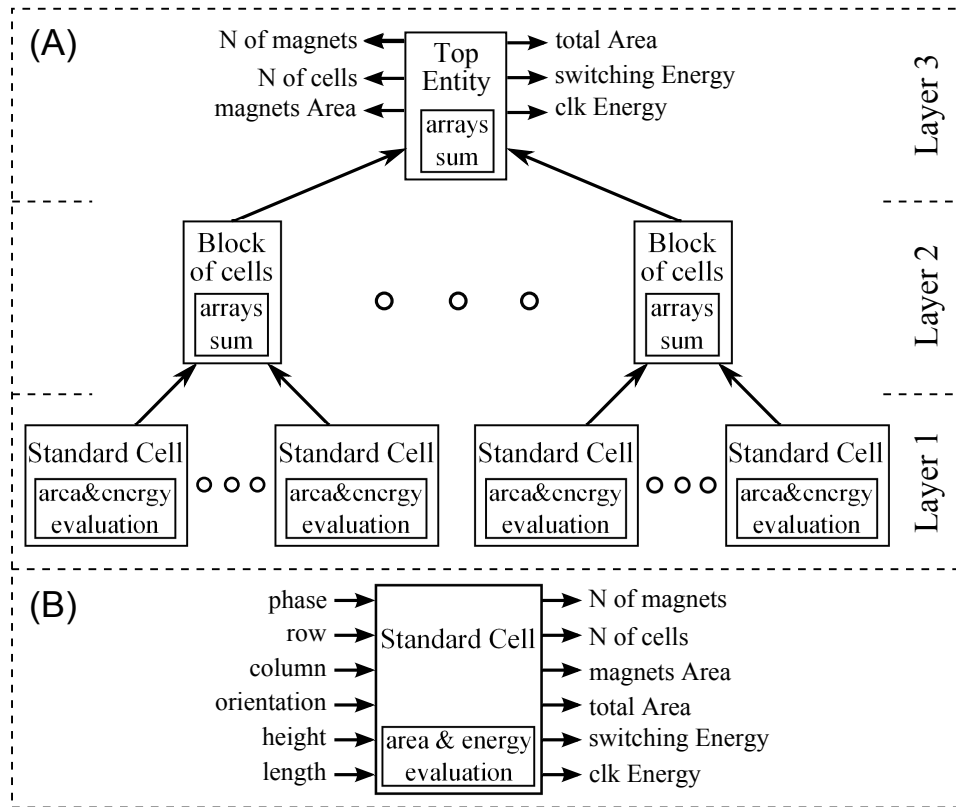


Figure 19. A) VHDL hierarchical model. The information on energy dissipation and area occupation are propagated hierarchically toward the top entity. B) generic inputs and outputs of a Standard Cell.

Since VHDL language is hierarchical, the same is true for our model. The standard cells form the bottom layer, while components in the upper layer can assemble them together to create circuits. These blocks of cells themselves can be instantiated by bigger circuits and so on up to the top entity. Figure 19.A depicts a generic 3-layers hierarchy. The Top Entity (layer 3) is composed by many Block of cells (layer 2), while each block of cells encloses the required standard cells (layer 1).

This hierarchy is exploited for a bottom-up evaluation of the number of magnets, number of cells and performance in terms of area and power. As explained in section 3.2.3, each Standard Cell gives as output all this information about itself thanks to the `area_and_power` component. The elements in the upper layer sum up the data received from every element in the lower layer (with what is called *arrays sum* in Figure 19), outputting then the results. This mechanism goes on recursively up to the Top Entity, which gives as output the total results for the whole circuit. Notice that the model provides exact results, as there is no approximation in the hierarchical evaluation and the circuit design for ME-NML provides a layout correspondent with the actual physical mapping.

3.3 Circuit layout

We have seen how cells are described by the model and how performance is evaluated. In this section we will see the first example of a ME-NML circuit, focusing on many general aspects of the design: the circuit layout, the CMOS circuit described by the model, the multiphase clocking system, the timing of signal propagation. This quick glance will be very useful when dealing with more complex systems in the following chapters. The small circuit studied in this section

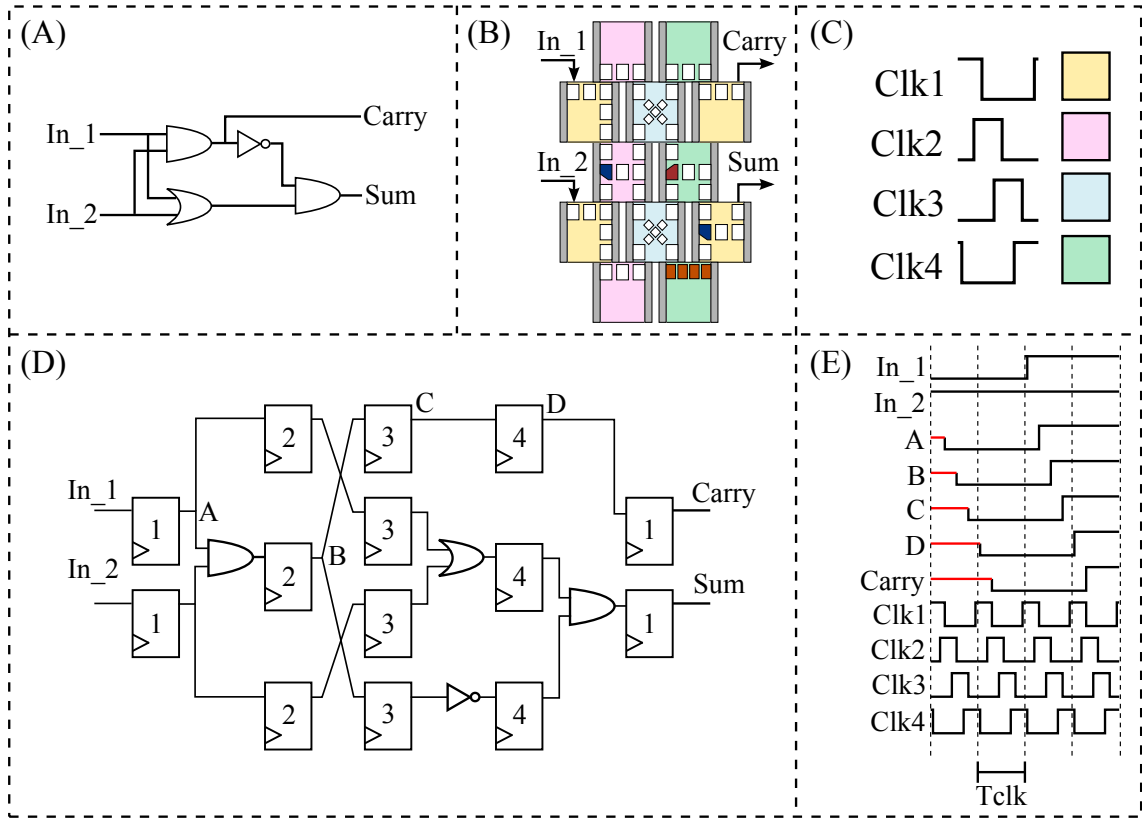


Figure 20. A) CMOS Half Adder. B) ME-NML Half Adder. C) Waveforms for the 4-phase overlapped clock system. A color is associated to each clock signal. D) VHDL counterpart of the ME-NML circuit, it is the circuit described by the VHDL model. E) Timing diagram of an example of signal propagation through the adder.

is a Half Adder (HA). Since the only logic ports available are Inverter, AND, OR, to design the ME-NML Half Adder in Figure 20.B we started from the configuration in Figure 20.A.

Cells are placed on a grid-like scheme (Figure 13.D). The pattern from inputs to outputs is 5 clock zones long. For an easier visual comprehension, the AND, OR and inverter magnets are highlighted respectively in blue, red and orange, while the substrate coloration identifies the

clock phase of a cell, namely the clock signal driving such clock zone. The clock system choice for ME-NML is a 4-phases overlapped clock, the 4 waveforms, with their assigned colors, are listed in Figure 20.C. Notice that clock signals are slightly overlapped, to avoid back propagation.

The model presented in this chapter maps each clock zone to one/two registers, plus a logic gate if needed. The VHDL code for the ME-NML HA describes the CMOS circuit as in Figure 20.D. Notice that the path from input to outputs counts 5 registers (5 pipeline stages), just like the 5 clock zones needed to pass through the ME-NML version. The numbers marking registers define their clock phase.

For a better comprehension of the circuit functioning, the timing graph in Figure 20.E shows a simple propagation example. The signals follow the pattern from the inputs to the Carry output, passing through the nodes marked as A-B-C-D. All clock signals have the same period T_{clk} , but they are shifted by 90° . It is quite clear from the timing that a signal needs one clock cycle T_{clk} to cross 4 clock zones (registers in the VHDL counterpart of the ME-NML circuit). Hence a signal has a latency of $T_{clk}/4$ to cross a clock zone.

CHAPTER 4

CASE STUDY I: GALOIS FIELD MULTIPLIER

The aim of the first case study is to answer the most critical question about MagnetoElastic NML (ME-NML) technology: Does it offer significant improvements over state-of-the-art CMOS transistors? Is the power dissipation much less than for the Magnetic Clock NML? To prove the benefits of ME-NML, it is presented an accurate comparison of performances between three different implementations of the same circuit: $28nm$ CMOS, Magnetic Clock NML and ME-NML.

The circuit chosen as case study is a Galois Field Multiplier (GFM). It has got several applications in cryptography, digital signal processing, coding theory and computer algebra. This circuit shows strong modularity, because of its systolic array structure: It is composed by arrays of identical elements able to communicate only with their adjacent neighbors [41][42]. Since the usage of long interconnection wires is avoided, systolic arrays are very much suitable for NML circuits (QCA in general). NML technology is indeed still planar, it is not possible to use additional layers for interconnections, so the circuit complexity explodes with the increase of interconnection overhead. Therefore it is strongly advised, for any QCA implementation, to design circuits with a systolic array layout, as it is the only way to fully exploit their capabilities. If designed otherwise, NML circuits would lose to CMOS performances. The example in [43] clearly proves how with the wrong architectural choices the interconnection overhead can occupy as much as the 99% of circuit area.

4.1 Galois Fields arithmetic

A Galois Field $\text{GF}(q)$ encloses a finite number q of elements, together with the definition of addition and multiplication operations on pair of elements [44]. When $q = p^m$, with m positive integer and p prime number, the field exists and is unique. For this work we are exclusively interested in Binary Galois Fields ($\text{GF}(2^m)$, $p = 2$), as they perfectly suit digital systems. A XOR function implements the addition, while an AND port can perform the multiplication. In general, when $m = 1$ the operations are defined as the common modulo p addition and multiplication. So $\text{GF}(2^1)$, the smallest possible Binary Galois Field, only has the two elements $\{0, 1\}$ and modulo 2 operations. Table I shows the addition and multiplication results for $\text{GF}(2^1)$.

However, when $m > 1$, modulo operations between polynomials are required instead of ordinary modulo operations. A polynomial with degree up to $m - 1$ can be associated to each element of a field $\text{GF}(2^m)$. Its coefficients are elements of the field $\text{GF}(2)$, that is 0 or 1, so each polynomial can be represented by a binary number composed by its own coefficients. In Table II we can see the polynomial mapping and the corresponding binary representation for

TABLE I

ADDITION AND MULTIPLICATION FOR $\text{GF}(2)$

+	0	1	·	0	1
0	0	1	0	0	0
1	1	0	1	0	1

TABLE II

POLYNOMIAL MAPPING AND MULTIPLICATION TABLE FOR GF(8). PRIMITIVE:
 $X^3 + X + 1$.

Element	Polynomial	Binary Repr.	·	0	1	A	B	C	D	E	F
0	0	000	X	0	0	0	0	0	0	0	0
1	1	001	1	0	1	A	B	C	D	E	F
A	x	010	A	0	A	C	E	B	1	F	D
B	$x + 1$	011	B	0	B	E	D	F	C	1	A
C	x^2	100	C	0	C	B	F	E	A	D	1
D	$x^2 + 1$	101	D	0	D	1	C	A	F	B	E
E	$x^2 + x$	110	E	0	E	F	1	D	B	A	C
F	$x^2 + x + 1$	111	F	0	F	D	A	1	E	C	B

the field GF(2³). Its elements are eight: $\{0, 1, A, B, C, D, E, F\}$. This representation has as primitive polynomial $x^3 + x + 1$, which guarantees an efficient hardware implementation. A different choice of $p(x)$ generates a different polynomial representation.

But how to obtain the product results in Table II? The algorithm for multiplication of two polynomials $a(x)$ and $b(x)$ modulo an irreducible polynomial $p(x)$ (called primitive) is reported in listing 4.1. It is called the Montgomery Multiplication Algorithm [45]. For GF(2 ^{m}) the primitive polynomial has degree equal to m . The algorithm can perform modular multiplication without requiring division, which would be very costly. The multiplication is performed by *sum-and-shift* of partial products, while the modulo operation is obtained by subtracting the irreducible polynomial whenever the degree of the intermediate result gets equal to m . The $a_i \cdot b(x)$ term is either equal to 0 or to $b(x)$, respectively when $a_i = 0$ and $a_i = 1$. So one

Listing 4.1. Montgomery multiplication algorithm.

```

r(x) := 0
2 for i = m-1 downto 0 do
    r(x) := x*r(x) + a_i*b(x)
    if degree(r(x)) = m then r(x) := r(x)-p(x)
5 return r(x)

```

coefficient of $a(x)$ at a time is multiplied (carry free) with all the coefficients of $b(x)$. Then the current result is shifted left (multiplying by x) before adding the new partial result.

4.1.1 Galois Field Multiplier scheme

Translating the Montgomery algorithm into an actual circuit, we obtained a MSB-first bit-serial Galois Field multiplier. *MSB-first* and *bit-serial* refer to how the coefficients of $a(x)$ are fed to the circuit: Serially and starting from the MSB. Figure 21 shows the scheme of the multiplier for $\text{GF}(2^4)$. 1-bit registers are exploited to hold inputs and partial results, while the \times and $+$ symbols stand for multiplication and addition. The steps of the algorithm are mapped to the circuit scheme:

- **Shift:** $x \cdot r(x)$

Implemented with a 1-bit shift register toward the MSB. This operation provides the alignment with the next partial product. The 4 central registers form a shift register that moves the intermediate result $r(x)$ to the right.

- **Partial product:** $a_i \cdot b(x)$

Implemented with m bit-wise multiplications. This multiplications will be realized with 2-inputs AND ports. Data $a(x)$ has to be fed serially, while data $b(x)$ is a parallel input.

- **Intermediate result:** $r(x) = x \cdot r(x) + a_i \cdot b(x)$

The partial products addition is performed by 4 bit-wise additions, which can be obtained using XOR ports.

- **Subtrahend selection:** if $\text{degree}(r(x)) = m$

When this is true the primitive polynomial must be subtracted from the intermediate result, while when false the subtrahend will be 0. To generate the proper subtrahend ($p(x)$ or 0), $p(x)$ is multiplied bit-wise with r_{m-1} , which is the MSB of the intermediate result. As already mentioned multiplication can be implemented by AND ports.

- **Modulo operation:** $r(x) = r(x) + p(x)$

To subtract the selected subtrahend from the intermediate result $r(x)$ the two values are added (GF addition) bit-wise. This addition can be implemented by XOR ports.

The addition symbols in Figure 21 have three inputs, they perform two of the operation just described: *Intermediate result* and *Modulo operation*.

$$r(x) = x \cdot r(x) + a_i \cdot b(x)$$

$$r(x) = r(x) \cdot p(x)$$

In Figure 21 the systolic array organization is evident, multiple entities of the same basic block (circled with a dashed line) are combined to form the multiplier. A N -bit GFM requires N identical basic blocks, the only exception are the first and last which are slightly different from the others. Simply connecting a different number of this blocks it is possible to obtain any parallelism. Therefore a generalized GFM can be designed defining only three blocks, which will be referred to as *first*, *central* and *last*. This characteristic will be valid for any GFM implementation explored throughout the whole work.

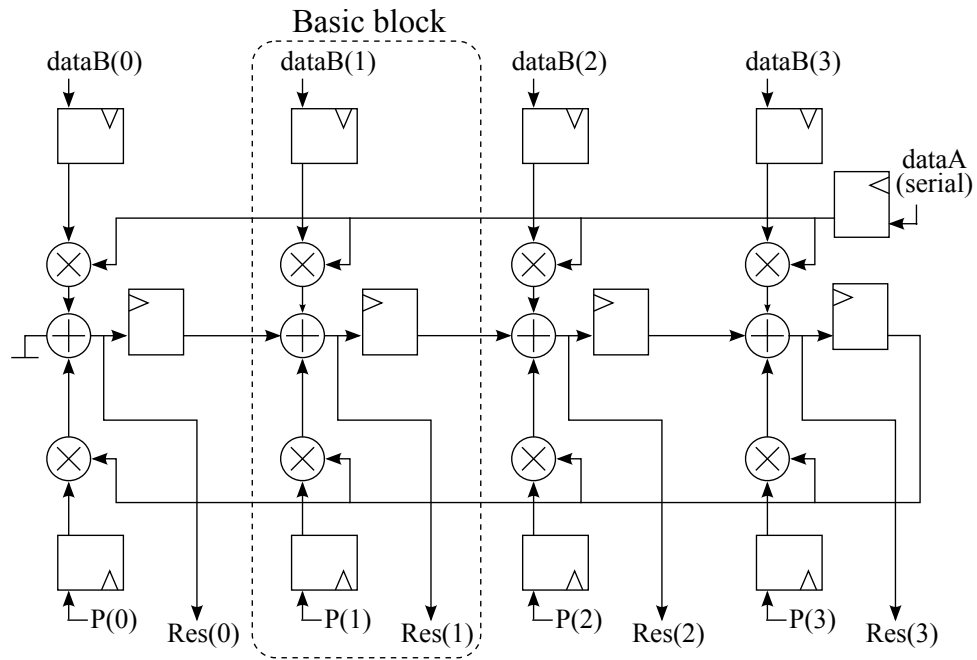


Figure 21. Scheme of a 4-bit bit-serial Galois Field Multiplier (GF(2⁴)).

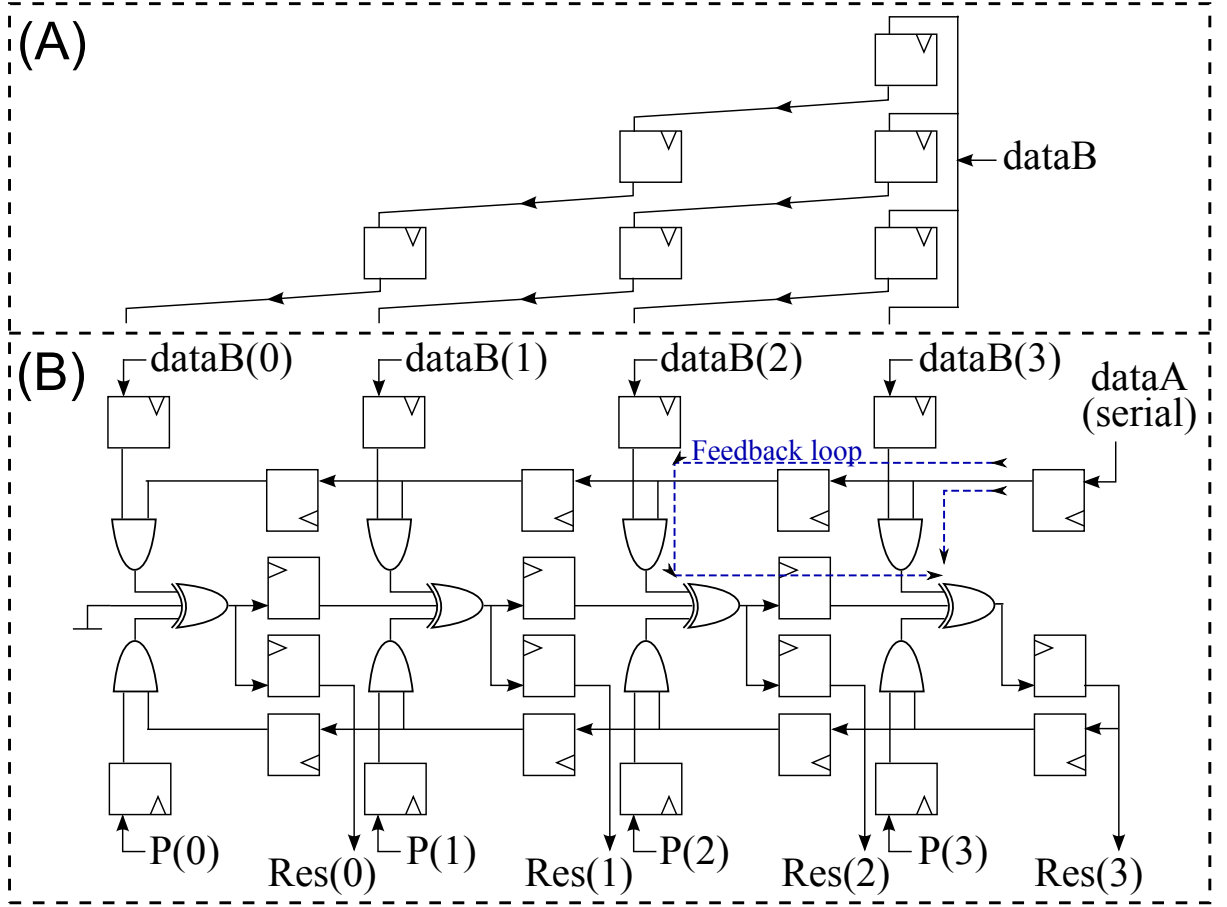


Figure 23. CMOS implementation of the 4-bit fully pipelined GFM. (A) Preskew circuitry for *dataB*. (B) Circuit body.

Moreover this is also the architecture chosen for the CMOS implementation, so that it assures an accurate and straightforward comparison between the two NML technologies considered in this work. Figure 23.B shows the fully pipelined CMOS circuit: It is exactly like in Figure 22, with additions performed by XOR gates, and multiplication by AND gates. Just

like for the scheme in Figure 21, the CMOS implementation is composed by standard blocks. The N -bit multiplier is formed by N adjacent blocks, which are all identical beside the slight differences of the first and last ones.

Because of this strong modularity, once defined the three basic blocks (first, central, last), it is straightforward to create a GFM with any parallelism just by tuning the number of central blocks ($N_{bit} - 2$ central blocks). For example a 4-bit multiplier, like in Figure 23, counts 2 central blocks. Increasing the parallelism the circuit layout will simply grow horizontally.

The generalized N -bit CMOS GFM has been described with VHDL language. The top entity, called `GaloisMultiplier`, instantiates N `basic_block` components. The `basic_block` has slightly different configurations, depending on its position within the circuit: first, last or center. This exact organization has been used also for the two NML implementations.

4.2.1 Timing analysis

Consider from now on a generic N -bit GFM. Due to both the Montgomery algorithm and the full pipeline, the inputs must be fed to the circuit in a peculiar way. The feedback path highlighted in blue in Figure 23 determines the input protocol. For a correct alignment of partial products' sum, $DataA(n-1)$ must arrive when $DataA(n)$ reaches the end of the feedback loop. Since the blue loop is two clock cycles long, $DataA$ bits must be fed with a delay of 2 clock cycles starting from the MSB. Therefore the overall time for $DataA$ to be inputted is $2N \cdot T_{clk}$, leading to a throughput of $1/(2N \cdot T_{clk})$.

$DataB$ (just like the primitive polynomial P) is a parallel input, but to generate the correct partial products with $DataA$ its bits cannot arrive simultaneously. According to the circuit

TABLE III

TIMING PERFORMANCE OF THE CMOS GFM

N bit	Interleaving	Throughput	Result: 1st bit out	Result: last bit out
4	2 op.	$1/(8T_{clk})$	$8T_{clk}$	$11T_{clk}$
8	2 op.	$1/(16T_{clk})$	$16T_{clk}$	$23T_{clk}$
N	2 op.	$1/(2N \cdot T_{clk})$	$(2(N - 1) + 2) \cdot T_{clk}$	$(3(N - 1) + 2) \cdot T_{clk}$

in Figure 23, *DataA* bits require a single clock cycle to pass through a basic block. Then the delay between *DataB* bits is of one clock cycle (T_{clk}), and each bit has to be hold for the whole operation: $2N \cdot T_{clk}$. The same is true for *P* because the feedback propagates as *DataA*. The result *Res* behaves just like *DataB* and *P*. Although this protocol remains unchanged for any circuit parallelism, inputs with higher number of bits need more time to be fed to the circuit. Table III lists the timing information for a generic multiplier and for two specific parallelisms: 4-bit and 8-bit.

Three major issues derive from the required protocol:

- There is an unused clock cycle between a *DataA* bit and the next. This means that meaningful inputs are fed only for half of the time, so at any time half of the registers in the circuit would contain useless data.
- It is not possible to supply all bits of *DataB* simultaneously. The same is true for *P* and for acquiring *Res*.
- To guarantee a continuous data flow, the inputs of an operation are fed right after the ones from the previous one. Therefore the new operation starts while the previous one

is still processing. The first partial product has to be summed to 0, so the central shift register would be required to contain zero when the new data arrives. However it would still be carrying the final result from the previous operation.

The solutions adopted applies also to the two nanomagnetic implementations:

- **Interleaving.** To have a continuous flow of input data multiple non correlated sets of operations can be executed in parallel, so that the delay time between an input and the next is filled with other operations (2.2.4.1).
- **Preskew and deskew networks.** A full set of additional registers must be added to the multiplier's body, in order to form preskew (for *DataB* and *P*) and deskew (for *Res*) networks. Figure 23.A shows the additional circuitry so that all bits of *DataB* can be served simultaneously. The same network has been used for *P* and *Res*. We will see in Chapter 5 how they affect the circuit area growth as a function of the number of bits.
- **Shift Register Reset.** Each register of the central row has to be reset (set to '0') when data from a new operation arrives. Since in that moment it will contain the final result of the previous operation, such result will be erased. Therefore a line of additional registers, with the same input as the shift registers, is added right below. In this way the final result can be preserved, allowing to execute a continuous flow of operations. The reset of the shift register is applied in the same way as *DataB* is fed to the circuit. A 1 clock cycle reset is applied to each register when a new data is fed to its correspondent *DataB*

TABLE IV

POLYNOMIAL MAPPING FOR GF(16). PRIMITIVE: $X^4 + X + 1$.

Element	Polynomial	Binary Representation
0	0	0000
1	1	0001
2	x	0010
3	$x + 1$	0011
4	x^2	0100
5	$x^2 + 1$	0101
6	$x^2 + x$	0110
7	$x^2 + x + 1$	0111
8	x^3	1000
9	$x^3 + 1$	1001
10	$x^3 + x$	1010
11	$x^3 + x + 1$	1011
12	$x^3 + x^2$	1100
13	$x^3 + x^2 + 1$	1101
14	$x^3 + x^2 + x$	1110
15	$x^3 + x^2 + x + 1$	1111

register. The first register of the feedback (bottom-right corner) must be reset as well anytime a new $DataB(3)$ bit is applied.

4.2.2 Circuit simulation

The purpose of creating a CMOS version of the GFM is to compare its performances with those of NML technology. First, the circuit has been described with VHDL and verified through simulation with Modelsim 6.4. Then the circuit performances have been estimated through a physical place&route with Cadence Encounter 13.1, using a 28nm library of low power CMOS transistors.

TABLE V

MULTIPLICATION TABLE FOR GF(16). PRIMITIVE: $X^4 + X + 1$.

X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0	2	4	6	8	10	12	14	3	1	7	5	11	9	15	13
3	0	3	6	5	12	15	10	9	11	8	13	14	7	4	1	2
4	0	4	8	12	3	7	11	15	6	2	14	10	5	1	13	9
5	0	5	10	15	7	2	13	8	14	11	4	1	9	12	3	6
6	0	6	12	10	11	13	7	1	5	3	9	15	14	8	2	4
7	0	7	14	9	15	8	1	6	13	10	3	4	2	5	12	11
8	0	8	3	11	6	14	5	13	12	4	15	7	10	2	9	1
9	0	9	1	8	2	11	3	10	4	13	5	12	6	15	7	14
10	0	10	7	13	14	4	9	3	15	5	8	2	1	11	6	12
11	0	11	5	14	10	1	15	4	7	12	2	9	13	6	8	3
12	0	12	11	7	5	9	14	2	10	6	1	13	15	3	4	8
13	0	13	9	4	1	12	8	5	2	15	11	6	3	14	10	7
14	0	14	15	1	13	3	2	12	9	7	6	8	4	10	11	5
15	0	15	13	2	9	6	4	11	1	14	12	3	8	7	5	10

We described with VHDL the CMOS fully pipelined version of the N-bit Galois Multiplier. At first only the multiplier body was tested, both with and without exploiting interleaving. Then the preskew and deskew additional circuitry have been added to the multiplier itself for another simulation session. The simulation without synchronization circuitry requires a quite complex testbench, while after adding the additional registers the timing protocol gets much simpler. The parallelism is defined by a **generic** parameter called N.BIT, which in the simulation is in the range 4 : 64 (GF(16) to GF(2^{64})).

The circuit verification was carried out comparing simulation results to expected results evaluated through a proper Matlab script. Every testbench prints the results into a text file,

using functions from the `std.textio` library for VHDL. On the other side Matlab has a set of functions for handling Galois Field arithmetic. The function `gf` creates the required array of Galois Field elements, then any operation on those elements is performed within the Galois Field specified. It is then trivial to generate the product matrix (as in Table V) that will be used to write the expected results into a text file. This work uses the default primitive polynomials defined by Matlab, which identifies them with a number corresponding to the binary representation of polynomials' coefficients. The simulation evaluates only a limited number of randomly determined multiplications, because for high number of bits the product table is extremely vast.

4.3 ME-NML Implementation

The central part of the study on the Galois Field Multiplier (GFM) has been the design and optimization of its MagnetoElastic NML implementation. This work presents, for the first time in literature, the design of a ME-NML circuit, also keeping into account the technological and physical constraints of this newly proposed technology. Chapter 3 explained how a Standard Cell Library and a RTL model have been developed for this technology, starting from the base cell derived from the MagnetoElastic Clock idea [1]. Section 3.3 introduced the ME-NML design methodology, also providing in Figure 20 a small design example. However only through the study of complex architectures it is possible to fully understand the potentialities and limitations of a novel technology.

4.3.1 Circuit design

The register function is intrinsic in ME-NML technology. So, while designing circuits, all that counts is the combinational logic. Since the only available ports are AND, OR and Inverter the 3-inputs XOR has been realized as in Figure 24.

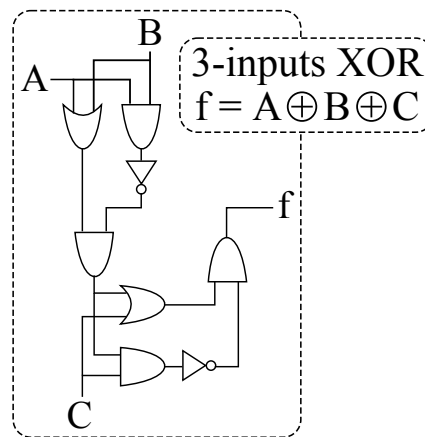


Figure 24. 3-inputs XOR function constructed with AND, OR and Inverter gates.

4.3.1.1 Basic blocks

The basic block of GFM contains two AND and one XOR gates, plus a certain number of registers. Through several steps of manual design and optimization, the final basic blocks for the GFM came out as in Figure 25, where the newly designed ME-NML blocks are matched with the correspondent CMOS blocks of the circuit in Figure 23. The in/out signals for each block are indicated for an easier comparison with the CMOS circuit.

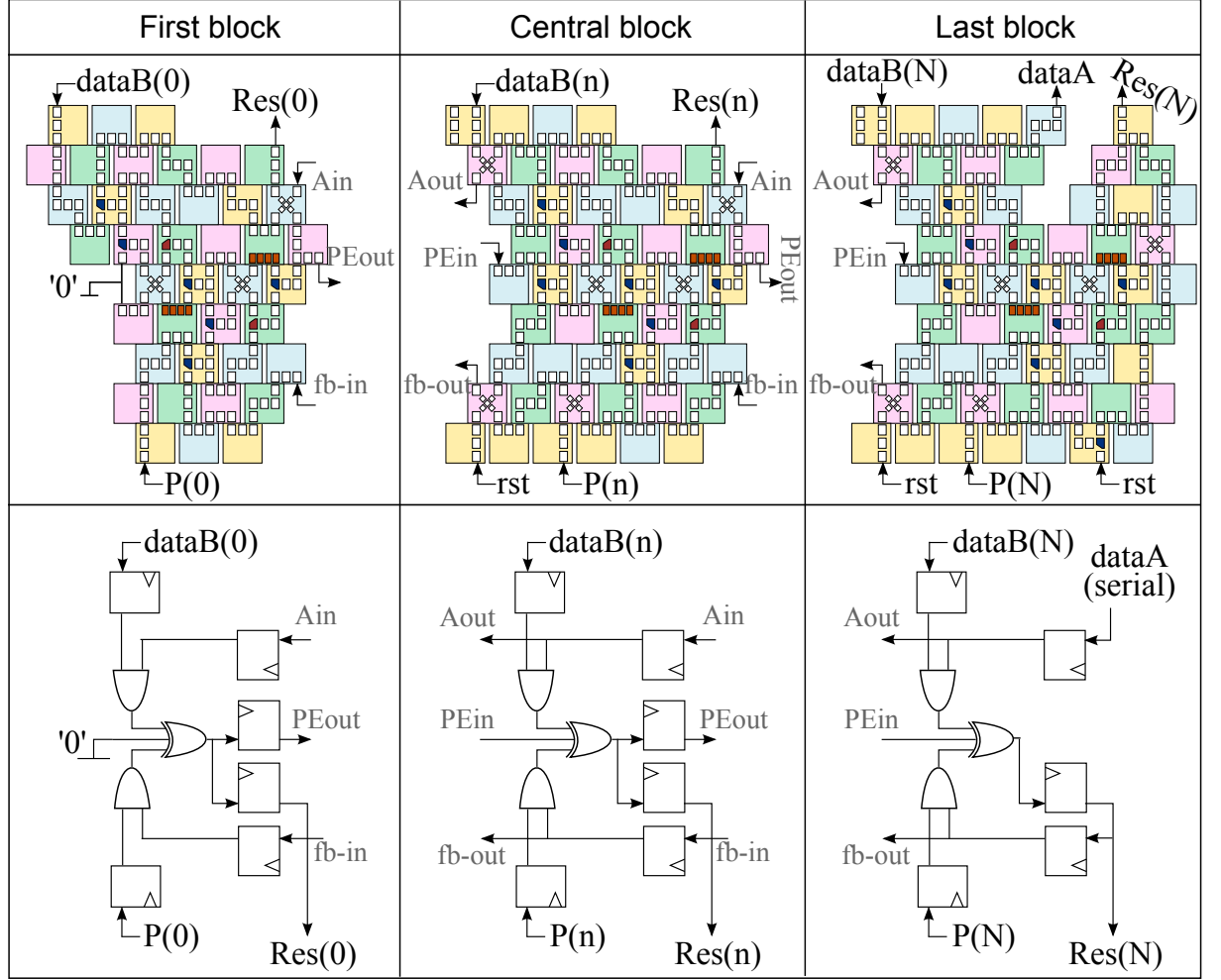


Figure 25. Basic blocks of the GFM. ME-NML blocks on top are matched with the correspondent CMOS blocks.

The reset network is not shown for CMOS, but its functioning was explained in 4.2.1. In the ME-NML implementation the reset (*rst*), treated just like any other signals, is applied to the signal (*PEin*) that propagates the temporary result from a block to the next one. The reset

is obtained through an AND gate with as inputs $PEin$ and the reset signal itself. The same is true for the reset applied to the feedback wire in the *Last* block (bottom-right corner).

For the sake of clarity the electrodes were omitted and there is no vertical separation between cells. The cell's color identify the clock phase: yellow for phase 1, pink for phase 2, light blue for phase 3, green for phase 4. A N -bit multiplier requires N adjacent blocks: 1 *First* block, $N - 2$ *Central* blocks, 1 *Last* block. Notice that the right border of the n block has the same shape as the left border of the $n+1$ block.

4.3.1.2 4-bit GFM

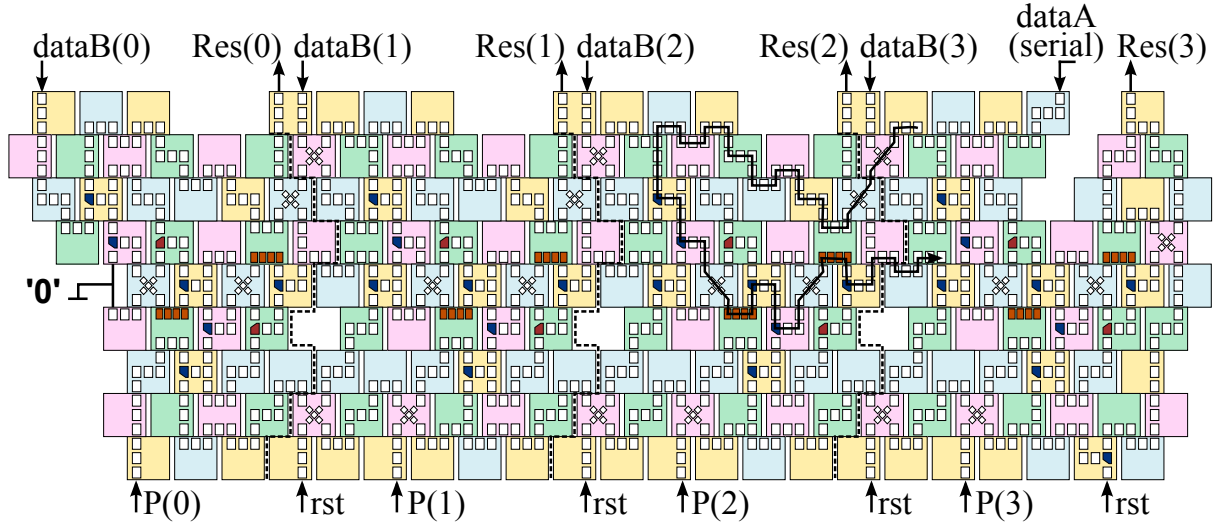


Figure 26. Magnetoelastic NML implementation of a 4-bit Galois Multiplier.

In Figure 26 the basic blocks have been pulled together to form the 4-bit GFM. Figure 27 contains a circuit which is equivalent to the ME-NML version in terms of timing. This scheme allows to easily comprehend how the ME-NML implementation works. Each register of Figure 27 represents four consecutive phases, so it is crossed in one clock cycle, which is the time needed to pass through four ME-NML cells. A feedback path is highlighted in both drawings: It is 6 clock cycles long, that is the time for crossing 24 ME-NML cells. The delay between *DataA* bits has to correspond to this critical path's length. This delay is much longer compared to the CMOS circuit, because of the intrinsic pipeline nature of NML. The blue arrow is also useful to indicate how signals propagate through this kind of circuit.

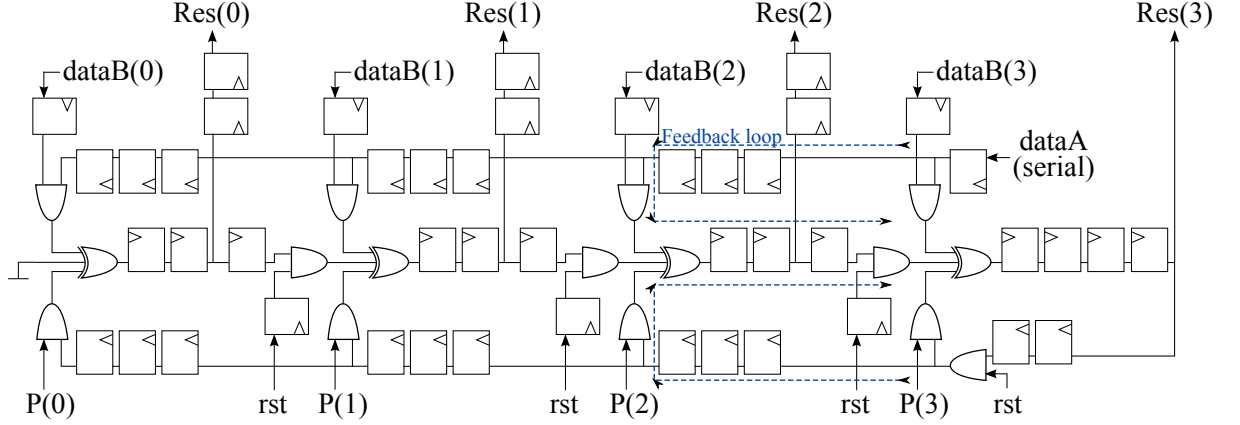


Figure 27. Equivalent circuit for the ME-NML GFM.

4.3.1.3 4-bit GFM with synchronization circuitry

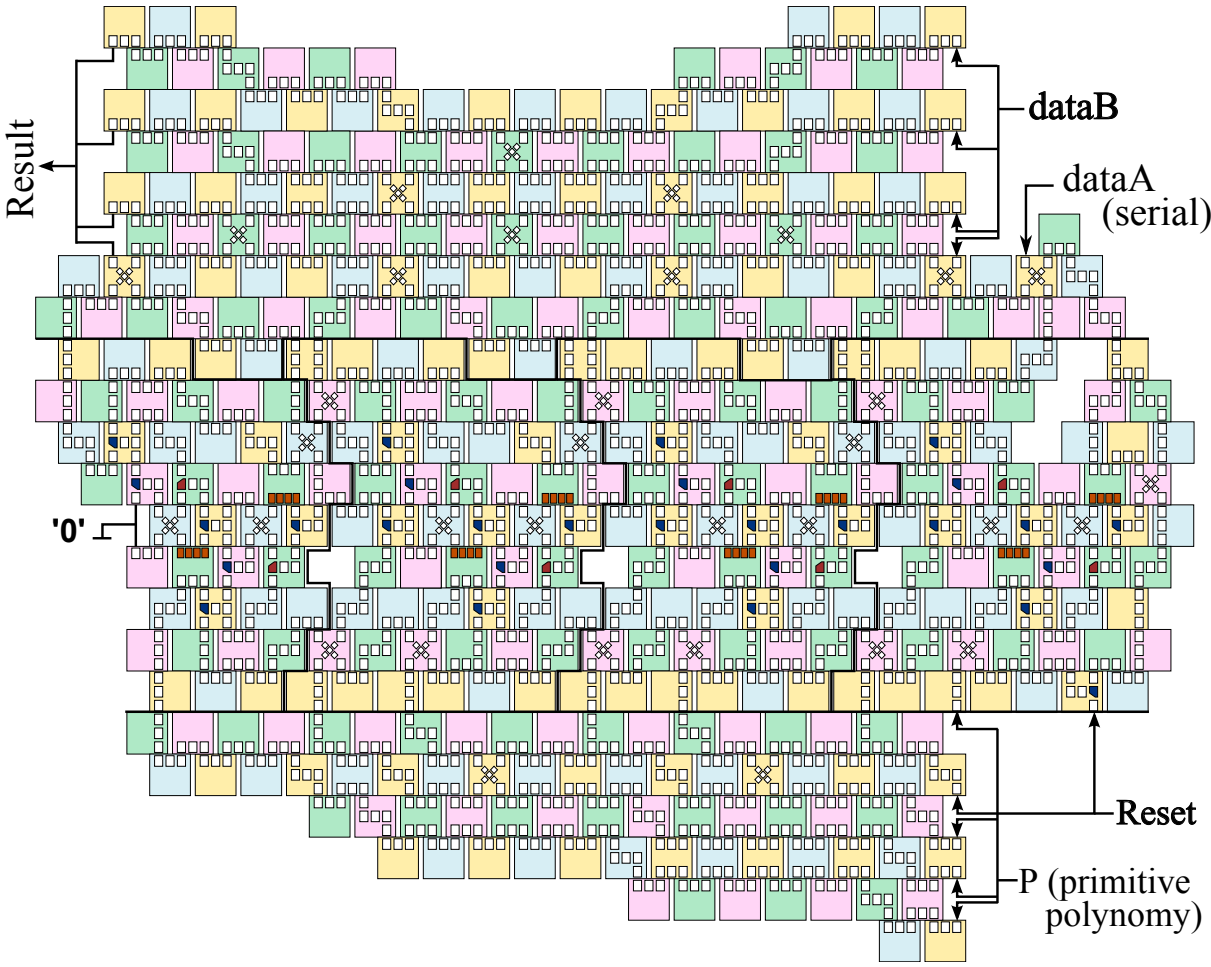


Figure 28. ME-NML Galois Multiplier with additional preskew and deskew networks.

One criticality described in Section 4.2.1 is the introduction of a preskew/deskew network, so that all bits of *DataB*, *P* and *Res* can be served/acquired simultaneously. The additional circuitry has been designed and added to the GFM body. Figure 28 is divided into three horizontal stripes. The central one is the GFM's body (Figure 26) and the top and bottom ones are the required synchronization networks. The preskew/deskew circuitries can also be decomposed in basic blocks and described with VHDL generically for any number of bits, even though they are not as regular as the central section. They do not contain any logic, only interconnections.

4.3.2 VHDL description and circuit simulation

To verify the circuit functioning and to evaluate performances, the ME-NML Galois Multiplier has been described with the RTL model presented in Chapter 3. The top entity `Galois_Multiplier` instantiates and connects the required number of basic blocks (Figure 25), which are defined by another entity called `Base_Blocks` (see Listing 4.2). Any circuit parallelism can be obtained assigning the desired number of bits to the constant `N_BIT`.

The simulation was performed only on the circuit body, as it was enough to verify the circuit functioning. The synchronization circuitry do not add any logic, anyway a full synchronization network, related to the second case study of this work, has been modeled and simulated (Chapter 6). Since the additional network is not considered, the testbench is very delicate from the timing point of view. The simulation procedure is the same used for CMOS: Results from the simulation are compared with the expected results evaluated by a Matlab script.

Listing 4.2. VHDL entities of Galois Multiplier: full circuit and basic block.

```

1  -- Galois Multiplier entity
entity Galois_Multiplier is
4  port(DataB,P,rst: in std_logic_vector(N_BIT-1 downto 0); -- DataB,P(x),reset
      rst_fb,DataA: in std_logic; -- feedback reset, DataA serially fed
      Res: out std_logic_vector(N_BIT-1 downto 0); -- Result
      clkA, clkB, clkC, clkD: in std_logic; -- Clock signals
7  n_mag: out natural := init_natural; -- # of magnets
      n_zones: out natural := init_natural; -- # of cells used
      AREA_EFF: out natural; -- Total magnets area
10     AREA_TOT: out natural; -- Total area occupied by the cells
      Er: out natural; -- Energy consumption of nanomags
      Ec: out natural); -- Energy consumption of clock
13 end Galois_Multiplier;

-- Base Block entity
16 entity Base_Block is
    generic(ELEMENT: integer); -- Identifies one among N_BIT basic blocks
    port(
19     A_in, B, P, fb_in, PE_in: in std_logic;
        mrbit_out, fb_out, Res, PE_out: out std_logic;
        rst,rst_fb: in std_logic; -- reset signals
22     clk, clkA, clkB, clkC, clkD: in std_logic; -- Clock signals (all phases)
        n_mag: out natural := init_natural; -- # of magnets
        n_zones: out natural := init_natural; -- # of cells used
25     AREA_EFF: out natural; -- Total magnets area
        AREA_TOT: out natural; -- Total area occupied by the cells
        Er: out natural; -- Energy consumption of nanomags
28     Ec: out natural); -- Energy consumption of clock
end Base_Block;

```

The timing protocol is very similar to the CMOS case but with 3 times longer delay, because the critical path is not 2 anymore, but 6. The result is a 6 clock periods delay between *DataA* bits, and 3 clock cycles of delay for the others: *DataB*, *P*, *Res*. To reach the maximum throughput 6 uncorrelated operations should be interleaved. Table VI contains the timing information concerning the ME-NML implementation. To properly understand this table refer to the equivalent circuit in Figure 26, rather than the original one in Figure 27.

The timing diagram resulting from the simulation of a simple operation is reported in Figure 29. The operation executed is $Res = DataA \cdot DataB = 10 \times 9$. The result can be found in Table V: $10 \times 9 = 5$, “1010” \times “1001” = “0101”. Prior to the operation execution all cells are

TABLE VI

TIMING PERFORMANCE OF THE ME-NML GFM				
N bit	Interleaving	Throughput	Result:1st bit out	Result:last bit out
4	2 op.	$1/(24T_{clk})$	$23T_{clk}$	$32T_{clk}$
8	2 op.	$1/(48T_{clk})$	$45T_{clk}$	$66T_{clk}$
N	2 op.	$1/(6N \cdot T_{clk})$	$(6(N - 1) + 5) \cdot T_{clk}$	$(9(N - 1) + 5) \cdot T_{clk}$

considered in an undefined state, so that it will be easier to understand how inputs are given to the circuit, because signals stay undefined until they are assigned a value. The whole timing protocol strictly depends on the physical layout of the circuit.

Let's analyze the diagram in detail:

DataA *DataA* is fed serially one bit every 6 clock cycles starting from the MSB.

DataB *DataB* is fed in parallel, one bit every 3 clock cycles starting from the MSB. Its values change every $6N_{bit}$ clock cycles.

Primitive polynomial It should be applied like *DataB*, but since the polynomial is usually kept fixed it is treated as a constant. The polynomial chosen is $x^4 + x + 1$ and it is mapped to binary as "10011", but the MSB is not used by the Galois Multiplier.

Result The result must be acquired one bit every 3 clock cycles, starting from the MSB.

Reset signals The *rst* signal is applied to all the blocks but the first one, so *rst*(0) is always equal to 1. Each *rst*(*i*) bit is applied together with its corresponding *DataB*(*i*) and kept low for 6 clock cycles. The *rst2* controls the feedback and it is applied 1 clock cycle after the beginning of the operation.

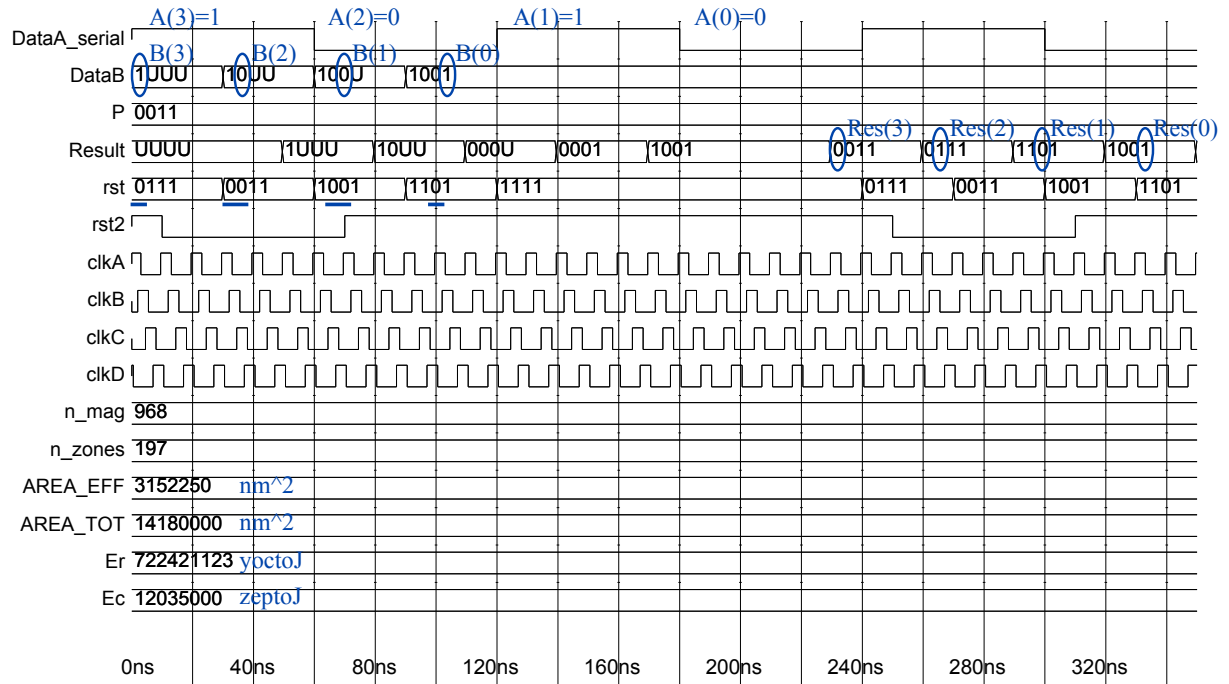


Figure 29. Timing diagram of the operation 9×10 with the ME-NML 4-bit Galois Multiplier.

Clock signals There are 4 overlapped clock signals. The phase shift between one signal and the next is then 90° .

Area and Power The six **natural** signals at the bottom contain the results of the embedded performance evaluation: Number of nanomagnets, number of cells, total area occupied by nanomagnets, total area occupied by cells, energy required for magnets switching, energy dissipated by the clock network.

It may seem that in the diagram in Figure 29 only one operation is executed, but that is not totally true. The interleaving technique is not exploited, so the only operation evaluated by

the circuit is 10×9 , but a close look reveals that such multiplication is executed 6 consecutive times. Notice that each bit of the final result keeps its value for 6 clock cycles. The reason is that instead of applying inputs and resets only for a single clock period out of six, they are kept active for 5 more.

4.4 Magnetic Clock NML Implementation

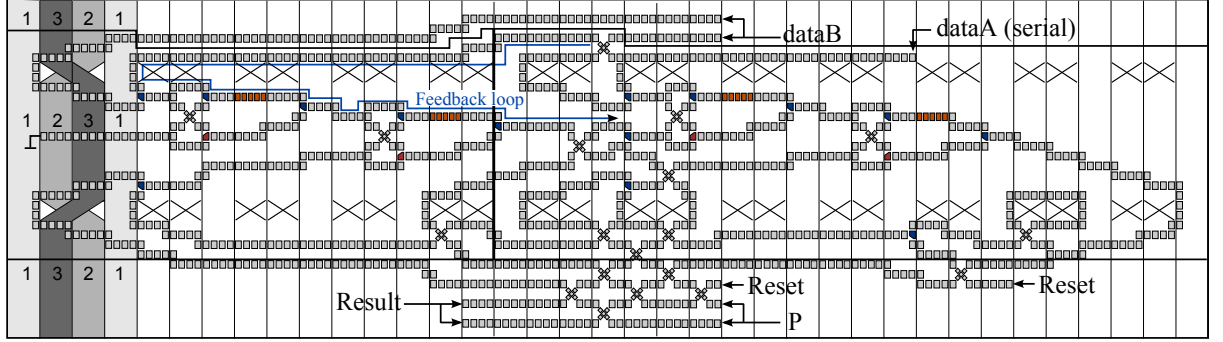


Figure 30. The 2-bit Magnetic NML Galois Multiplier, comprehensive of preskew and deskew networks.

The introduction of the MagnetoElastic Clock technology [1], was mainly triggered by the too high energy dissipation of the Magnetic Clock system. Therefore for an exhaustive study of the ME-NML, we provide a comparison with the Magnetic NML (described in 2.2.2), which is based on a magnetic field clock and a snake-clock mechanism [28].

This section illustrates an implementation of the Galois Multiplier based on the Magnetic NLM technology. Performance data will be extracted and compared with the other two technologies considered in this work. Magnetic Clock NML has already been studied from the physical and architectural point of view developing an *ad-hoc* RTL model [37]. So to say that the validity of this technology has been already proved, here we will just design the generic N-bit Galois Multiplier and compute directly from the circuit schematic all the information regarding timing, occupied area and power dissipation.

The two small examples presented in Figure 11 and Figure 14 provided an insight on how Magnetic NML circuits look like and how signal propagation works. We also explained how to address synchronization and feedback issues derived from the circuit layout, which is strongly dependent on the snake-clock system. This preliminary knowledge can be easily applied also to more complex structures, such as the Galois Multiplier.

4.4.1 Galois Multiplier scheme

Despite all the similarities among different NML implementations, the snake-clock approach leads to a unique circuit organizations. What remains unchanged is the systolic array nature of the bit-serial Galois Multiplier: Three basic blocks are defined for the Magnetic NML too.

We enclosed two drawings of the Magnetic NML Galois Multiplier including the synchronization networks: the 2-bit version in Figure 30 and the 4-bit version in Figure 31. The latter has been divided in two parts to allow a better visual comprehension: The right side of the cut on top should be connected to the left side of the other one. In both figures the circuit body (central stripe) is separated by the preskew/deskew networks (top and bottom). Furthermore

vertical blue lines mark the division among basic blocks: First, Central, Last. Once again any parallelism can be obtained by combining these blocks.

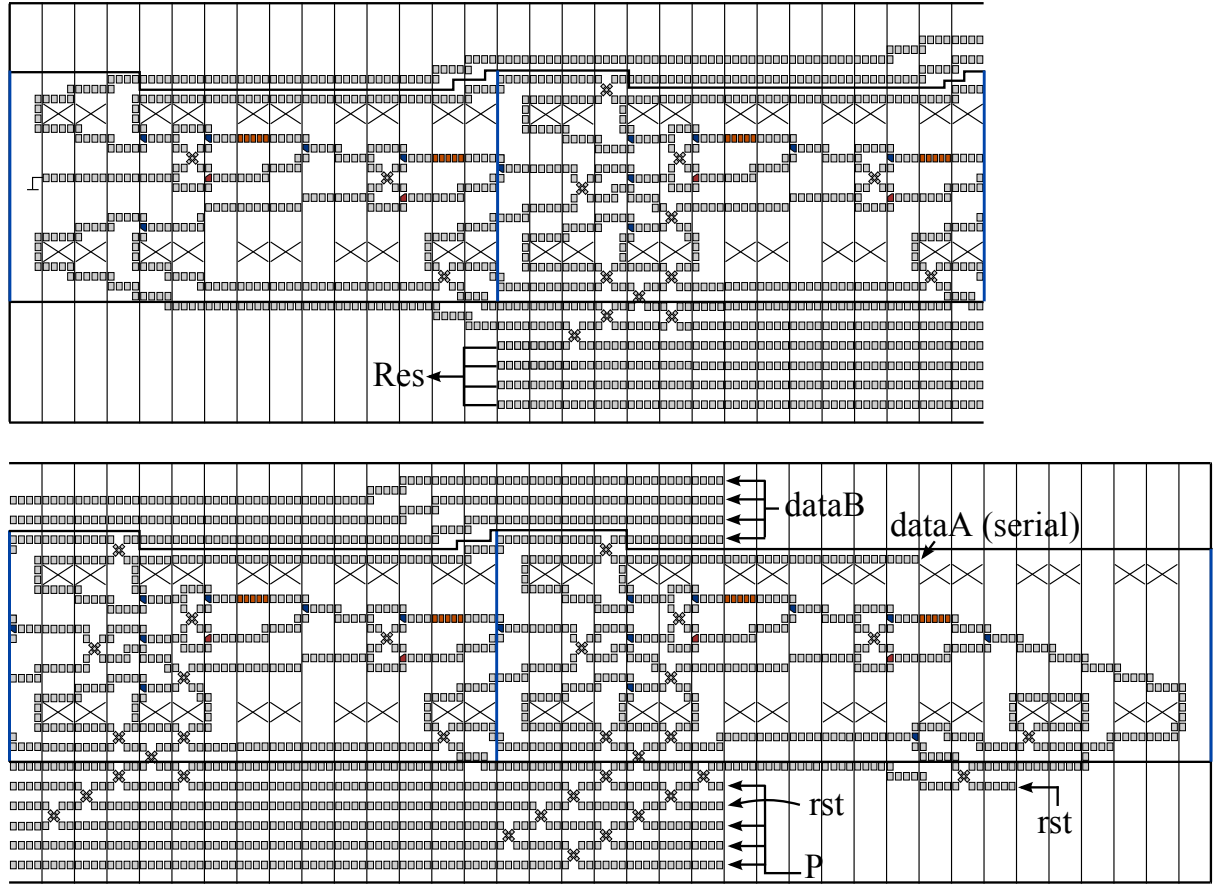


Figure 31. The 4-bit Magnetic NML Galois Multiplier, comprehensive of preskew and deskew networks. The circuit is split in left part (on top) and right part (below), to facilitate its comprehension.

A small area on the left in Figure 30 shows the exact layout of snake-clock wires and the signal propagation directions, in the rest of the drawing the forbidden areas are simply marked by black crosses. Notice also the feedback critical path for this implementation, it is highlighted with blue. Its length is 30 clock zones, which correspond to 10 clock cycles, since the snake-clock is a 3-phase clocking system.

4.4.2 Timing analysis

A new bit of *DataA* can be sent to the circuit every 10 clock cycles. The basic block depth is instead equal to 15 clock zones (5 clock periods), so that will be the delay between bits of *DataB*, *P* and *Res*. Table VII gives the main timing information on this implementation.

TABLE VII

TIMING PERFORMANCE OF THE MAGNETIC NML GFM				
N bit	Interleaving	Throughput	Result:1st bit out	Result:Last bit out
4	2 op.	$1/(40T_{clk})$	$40T_{clk}$	$55T_{clk}$
8	2 op.	$1/(80T_{clk})$	$80T_{clk}$	$115T_{clk}$
N	2 op.	$1/(10N \cdot T_{clk})$	$(10(N - 1) + 10)$	$(15(N - 1) + 10) \cdot T_{clk}$

CHAPTER 5

CASE STUDY I: GFM RESULTS COMPARISON

This chapter is devoted to performance evaluation of the three GFM implementations in terms of occupied area and power consumption. First of all the results produced for each technology are discussed separately, providing details on their evaluation. Then the three versions are placed side by side, presenting an accurate comparison. NML circuits are handled keeping into account technological constraints and the exact details on the clock network chosen.

The outcomes demonstrate the effectiveness of ME-NML for power and area performances. For each implementation the results are evaluated for 4 to 64 bits, both with and without the preskew/deskew circuitry for input and outputs signals. The additional synchronization networks are a factor generally neglected in literature, even though they bring a significant increase of circuit area.

5.1 CMOS Results

The CMOS version of the GFM has been presented in Section 4.2. All the results are extracted after finalizing the physical layout through Cadence Encounter 13.1. For the place&route we exploited a low power CMOS 28 nm FDSOI standard cell library, with the following working conditions: $V = 0.9V$, $T = 25^{\circ}C$. The working frequency was set to $f = 100MHz$ even though the CMOS implementation could reach up to $7GHz$. The reason was to assure a fair comparison with the NML implementations, which are limited to a $100MHz$ frequency.

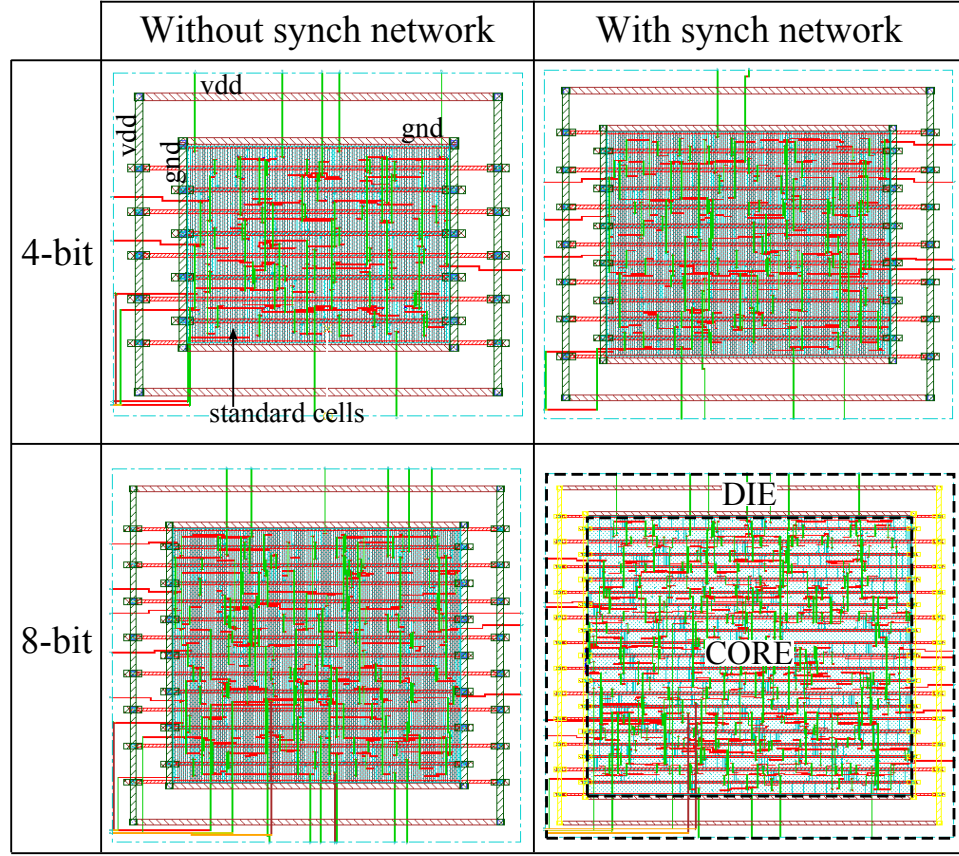


Figure 32. Post-route layout of the GFM in its CMOS implementation.

5.1.1 Occupied area

Figure 32 puts side by side the postroute layout of the GFM with and without synchronization circuitry, in its 4-bit and 8-bit implementations. The area has been calculated from the values of height and width of the core, without considering the die. Table VIII contains all the results of area occupation for the CMOS GFM.

TABLE VIII

AREA OCCUPATION OF CMOS GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.

CIRCUIT AREA		Number of bits				
		4	8	16	32	64
No Synch	Width (μm)	14,31	19,08	28,37	37,35	52,95
	Height (μm)	10,80	16,80	22,80	34,80	49,20
	AREA (μm^2)	154,6	320,6	646,9	1299,7	2605,3
With Synch	Width (μm)	18,21	30,69	54,47	103,88	202,22
	Height (μm)	14,40	26,40	50,40	96,00	187,20
	AREA (μm^2)	262,3	810,1	2745,2	9972,5	37856,0
Interconnection Overhead		1,7	2,5	4,2	7,7	14,5

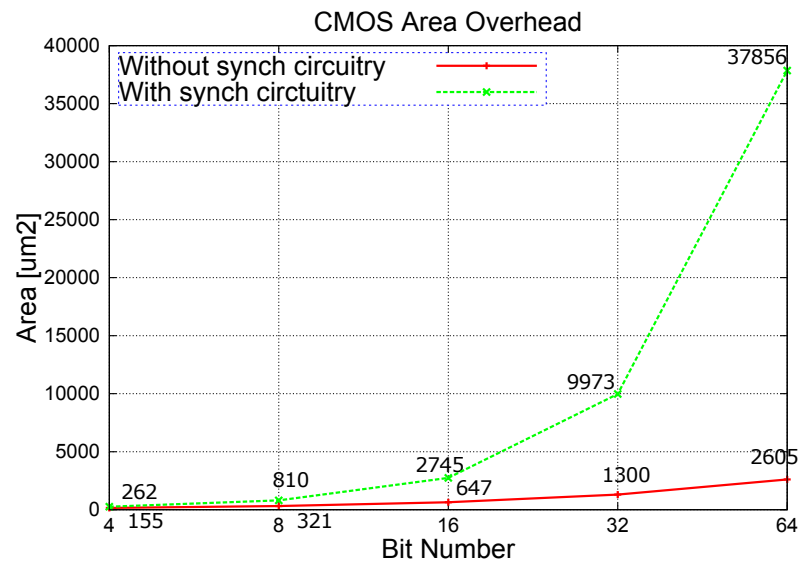


Figure 33. Comparison of area occupation for the CMOS GFM both with and without synchronization circuitry.

The interconnection overhead is simply evaluated as the ratio between values with and without preskew/deskew networks, and it can also be observed in Figure 33. The impact of the additional circuitry goes from 1.7 (4 bit) to 14.5 (64 bit). Which means that it goes from adding the 70% of the area for the 4 bit circuit, to increasing the 64 bit circuit (the highest parallelism considered) of 14.5 times.

5.1.2 Power consumption

TABLE IX

POWER CONSUMPTION OF THE CMOS GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.

POWER CONSUMPTION (μW)		Number of bits				
		4	8	16	32	64
No Synch	Internal	12,09	28,21	57,28	116,95	245,87
	Switching	1,21	3,38	7,21	14,99	31,52
	Leakage	1,00	2,05	4,13	8,30	16,63
	TOTAL	14,30	33,63	68,62	140,24	294,03
With Synch	Internal	20,40	70,38	243,90	855,50	3240,00
	Switching	1,63	5,75	17,07	56,72	200,70
	Leakage	1,69	5,25	17,85	64,99	247,10
	TOTAL	23,72	81,37	278,82	977,21	3687,80
Interconnection Overhead		1,7	2,4	4,1	7,0	12,5

The post-route power estimation gave the results in Table IX. The losses increase due to interconnection overhead is also disclosed by Figure 34. The additional circuitry affects the

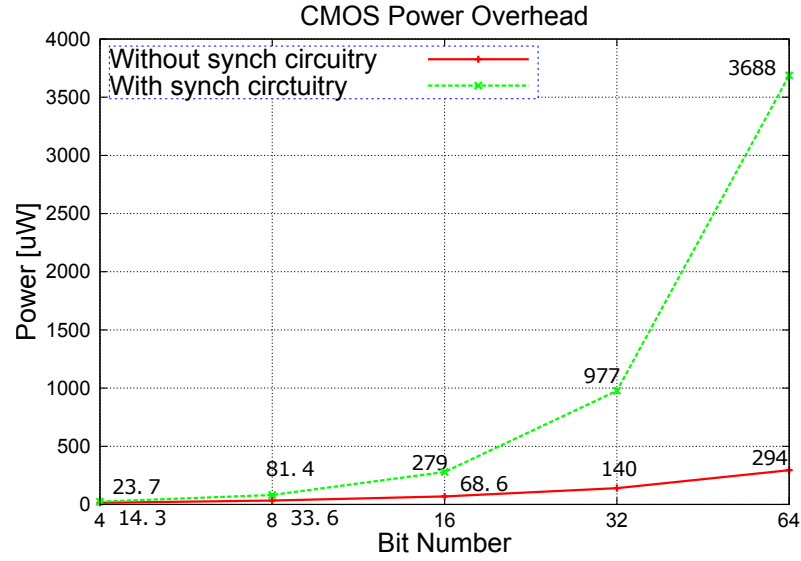


Figure 34. Comparison of power consumption for the CMOS GFM both with and without synchronization circuitry.

power consumption less than the area occupation, reaching a maximum increase of 12.5 times with respect to the power required by the GFM body itself.

5.2 Magnetoelastic NML Results

For what concerns the area and power estimation for the ME-NML implementation, the methodology and formulas have been detailed in Section 3.2.3. The results for the GFM body are directly evaluated by the VHDL model. Total area and energy components are given as output of the top entity `Galois_Multiplier` during simulation, just like in the timing diagram of Figure 29. On the other hand the preskew/deskew parts have not been described with the model, their performance has been evaluated directly from the drawings. They can be generalized to any number of bits isolating some basic blocks. However the generalization

Col = 1		Col = 2 : N-1		Col = N	
Row $\overline{3:N}$	G.	Row \overline{N}	H.	Row $\overline{3:N}$	I.
Row $\overline{2}$	D.	Row $\overline{2:N-1}$	E.	Row $\overline{2}$	F.
Row $\overline{1}$	A.	Row $\overline{1}$	B.	Row $\overline{1}$	C.

Figure 35. Basic blocks for the upper interconnections.

is much more complex, because the interconnections grow also vertically, requiring then the definition of more basic blocks.

5.2.1 Upper synchronization network

Figure 35 contains the nine blocks from which it is possible to compose, for any parallelism, the synchronization circuitry above the GFM's body. For example with a certain combination of these blocks it is possible to create the interconnections above the circuit's body in Figure 28. There would actually be a few differences between Figure 28 and the circuit realized with the standard blocks, because the blocks will have some additional cells and magnets, useless to the circuit functioning. The reason is that the base blocks have been generalized as much as possible. For our purposes the simplification is not a problem, the final results of area and power will just be slightly higher than they should.

8-bit layout	G	H	H	H	H	H	H	I
	G	E	E	E	E	E	E	I
	G	E	E	E	E	E	E	I
	G	E	E	E	E	E	E	I
	G	E	E	E	E	E	E	I
	G	E	E	E	E	E	E	I
	D	E	E	E	E	E	E	F
	A	B	B	B	B	B	B	C

8-bit layout optimized	G	H					H	I	
	G	E	H				H	E	I
	G	E	E	H	H	E	E	E	I
	G	E	E	E	E	E	E	E	I
	G	E	E	E	E	E	E	E	I
	G	E	E	E	E	E	E	E	I
	D	E	E	E	E	E	E	E	F
	A	B	B	B	B	B	B	B	C

Figure 36. Layout of the upper interconnections for the 8-bit GFM. The second table is the optimized layout.

TABLE X

NUMBER OF CELLS AND MAGNETS OF THE BASIC BLOCKS FOR THE UPPER INTERCONNECTIONS

	Cells	Magnets		Cells	Magnets		Cells	Magnets
G	8	26	H	7	23	I	5	15
D	8	40	E	12	74	F	5	21
A	13	48	B	14	56	C	13	56

Figure 35 tries to explain how to create the upper interconnections for a N -bit GFM, starting from the blocks from A to I . They result in a $N \times N$ matrix of blocks. The left part of Figure 36 shows how blocks would be placed in the 8-bit case. However some blocks in the top-central region are useless. The layout can then be optimized as in Figure 36 on the right, where the empty boxes correspond to empty regions. The number of rows and columns will be the same, but the block E will not be present $N - 2$ times in each column anymore. The central columns

($col = 2$ to $col = N - 1$) will have the following number of E blocks (the fractions have *integer* results):

$$\left| col - \frac{N_{bit} + 1}{2} \right| + \frac{N_{bit}}{2}$$

Table X lists the total number of cells and nanomagnets for each of the nine blocks. These values are used to evaluate the occupied area and power consumption according to the organization described above. Blocks are identified by the capital letters assigned in Figure 35.

5.2.2 Lower synchronization network

Conn. Below	Col = 1	Col = EVEN	Col = ODD	Col = N
Row $\overline{1}$	A.	B.	C.	D.
Row $\overline{2 : N-1}$		E.	F.	G.
Row \overline{N}			H.	I.

Figure 37. Basic blocks for the lower interconnections.

The synchronization circuit below the GFM's body has been treated just like the interconnections on top. As before the basic blocks have been organized in a table (Figure 37). The central columns, excluding then the first and last, have an alternate behavior. Odd columns

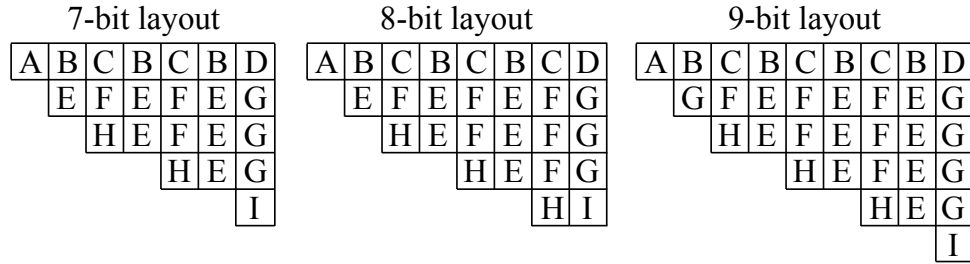


Figure 38. Layout of the lower interconnections for the 7-bit, 8-bit and 9-bit GFM.

enclose different blocks than the even ones. The three examples in Figure 38 help understand the circuit organization.

5.2.3 Occupied area

TABLE XI

NUMBER OF MAGNETS AND CELLS OF ME-NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.

MAGNETS and CELLS		Number of bits				
		4	8	16	32	64
No	N of magnets	974	1990	4022	8086	16214
Synch	N of cells	199	403	811	1627	3259
With	N of magnets	2007	6431	22287	82509	297710
synch	N of cells	427	1273	4117	14547	50235

First, the number of nanomagnets and cells have to be determined, the values are listed in Table XI. Eventually the results concerning area occupation, both with and without the

preskew/deskew circuits, are organized in Table XII and plotted in Figure 39. Where, apart from the individual results, the interconnection overhead can be observed as well. The overhead due to the upper and lower interconnections behaves similarly to the CMOS implementation. It grows quadratically with the number of bits, going from 2.1 (4 bit) to 15.4 (64 bit).

TABLE XII
OCCUPIED AREA OF ME-NML GFM BOTH WITH AND WITHOUT
SYNCHRONIZATION CIRCUITRY.

CIRCUIT AREA (μm^2)		Number of bits				
		4	8	16	32	64
No Synch	Magnets	3.2	6.5	13	26	53
	Cells	14	29	58	116	233
With synch	Magnets	6.5	21	72	268	968
	Cells	31	91	294	1040	3590
Interc. overhead		2.1	3.2	5.1	8.9	15.4

5.2.4 Power consumption

The power consumption is proportional to the area occupation, because both measures have the number of cells as factor. Therefore the interconnections overhead is the same as for the occupied area. The detailed results are in Table XIII. In fact the magnets switching energy is negligible (20 times smaller) compared the clock network dissipation.

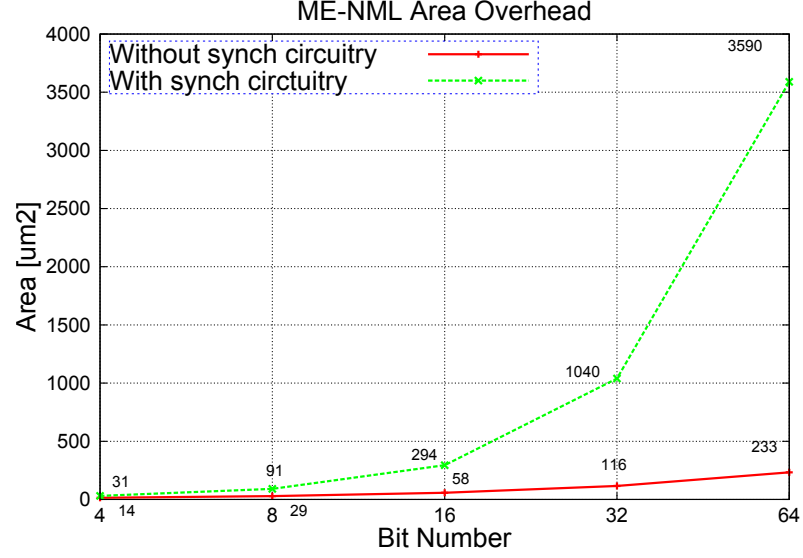


Figure 39. Comparison of area occupation for the ME-NML GFM both with and without synchronization circuitry.

5.3 Magnetic Clock NML Results

It has not been discussed yet how to evaluate the performance of Magnetic Clock NML circuits, so it is done in this section before providing the results.

5.3.1 Number of clock zones and magnets

The evaluation of area and power performances requires: the number of clock zones, the length of the clock zones (circuit height) and the total number of magnets. These values are at first computed for each basic block and then put together to obtain results for each parallelism and with or without the upper and lower interconnections parts. The final results are directly presented in Table XIV. The number of clock zones is nothing less than the circuit horizontal

TABLE XIII

POWER CONSUMPTION OF ME-NML GFM BOTH WITH AND WITHOUT
SYNCHRONIZATION CIRCUITRY.

POWER CONSUMPTION (μW)		Number of bits				
		4	8	16	32	64
No Synch	Switching	0.07	0.15	0.30	0.60	1.21
	Clock	1.21	2.45	4.92	9.88	19.8
	TOTAL	1.28	2.60	5.22	10.5	21.0
With synch	Switching	0.15	0.48	1.66	6.15	22.2
	Clock	2.59	7.73	25.0	88.3	305
	TOTAL	2.74	8.21	26.7	94.5	327
Inter. overhead		2.1	3.2	5.1	9.0	15.6

width, while the circuit height is for now measured in terms of magnets, the actual dimension can be evaluated knowing the magnets height and their vertical separation.

5.3.2 Occupied area

The Magnetic Clock NML exploits $90 \times 60nm^2$ magnets with separation $Sep_{mag} = 20nm$. Horizontally the clock zone contains four magnets, therefore its width is $W_{zone} = 4 \cdot (W_{mag} + Sep_{mag}) = 320nm$. These data, together with those in Table XIV, allow to evaluate the total area of magnets and the rectangle circumscribed to the circuit, the latter is shown in Table XV. Such table as usual encloses information on the preskew/deskew networks overhead, which is the lowest among the three technologies considered. We will see that the interconnection overhead is the same for both area and power estimation. Figure 40 gives an idea of the GFM behavior increasing the number of bits, with and without the additional synchronization circuits.

TABLE XIV

DIMENSIONS AND NUMBER OF MAGNETS OF THE MAGNETIC NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.

		Number of bits				
		4	8	16	32	64
No Synch	Number of magnets	1818	3678	7398	14838	29718
	Width (clock zones)	67	127	247	487	967
	Height (magnets)	24	24	24	24	24
With Synch	Number of magnets	3154	7388	18880	53960	172504
	Width (clock zones)	67	127	247	487	967
	Height (magnets)	40	56	88	152	280

TABLE XV

AREA OF THE MAGNETIC NML GFM BOTH WITH AND WITHOUT SYNCHRONIZATION CIRCUITRY.

CIRCUIT AREA	Number of bits				
	4	8	16	32	64
Area without synch (μm^2)	57	107	209	411	817
Area with synch (μm^2)	94	250	765	2610	9530
Interconn. overhead	1.7	2.3	3.7	6.3	11.7

5.3.3 Power consumption

The power dissipation, as for the ME-NML, has two sources: magnets switching and clock wires. The average energy required by the switching of a single nanomagnets is equal to $\delta E = 30K_b T = 1.24 \cdot 10^{-19} J$, since an adiabatic switch has to be exploited. The switching energy is obtained multiplying this value for the total number of magnets. However the main contribution is due to the clock network losses, because the current needed to generate the

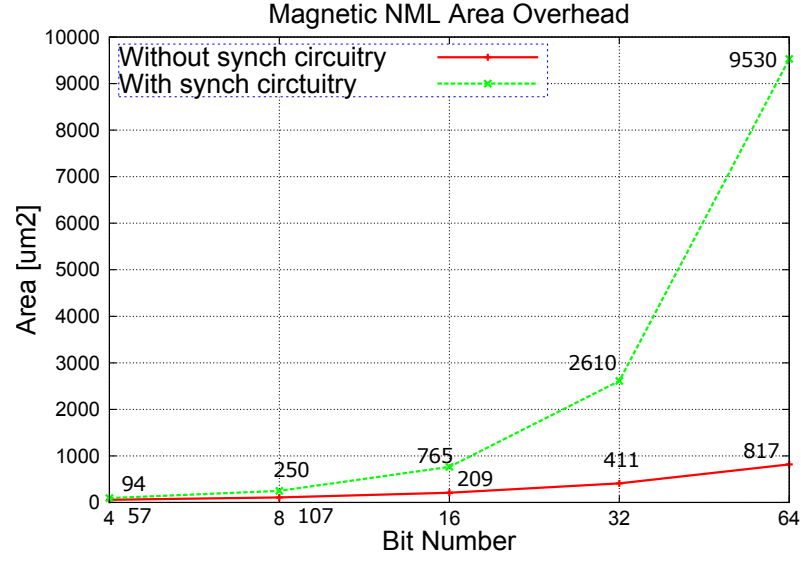


Figure 40. Comparison of area occupation for the Magnetic NML GFM both with and without synchronization circuitry.

magnetic field is very high: $I = 3mA$. The power consumption is therefore the dissipation of the current I flowing through a copper wire, which has resistivity $\rho = 16.8n\Omega \cdot m$. For each clock zone we consider a copper wire with width $W_{clk} = W_{zone} = 320nm$ and thickness of $T_{clk} = 400nm$, so its section is $S_{clk} = W_{clk} \cdot T_{clk}$. At any instant, only one third of the clock zones is active, since only one of the clock wires at a time is active. Summing the length H_{zone} of one third of the clock zones N_{zones_eff} we obtain the length $L_{clk} = N_{zones_eff} \cdot H_{zone}$ to assign to the copper wire, that will model the clock dissipation of the whole circuit. The power consumption is derived from the following formula:

$$P = I^2 \cdot \rho \frac{L_{clk}}{S_{clk}}$$

The power consumption results are in Table XVI. For further information on the Magnetic NML model refer to [37].

TABLE XVI
POWER OF THE MAGNETIC NML GFM, BOTH WITH AND WITHOUT
SYNCHRONIZATION CIRCUITRY.

POWER CONSUMPTION		Number of bits				
μW		4	8	16	32	64
No Synch	Magnets Switching	0.023	0.046	0.092	0.18	0.37
	Clock Wires	70	132	257	506	1010
	TOTAL	70	132	257	506	1010
With Synch	Magnets Switching	0.040	0.092	0.24	0.67	2.14
	Clock Wires	116	308	941	3210	11700
	TOTAL	116	308	942	3210	11700
Interconn. overhead		1.7	2.3	3.7	6.3	11.7

5.4 Results Comparison

Now that all the results have been presented, we compare the performances of the three implementations in terms of area and power. The main interest is the ratio between the results for ME-NML and those for CMOS and Magnetic NML. Nonetheless the interconnection overhead trends of each technology are put side by side. The purpose of Table XVII is to collect in one place all these data. The first table concerns the GFM's body only, the second table shows the results for the whole circuit, including the synchronization networks. Finally the third table reports once again the interconnection overhead, which is the ratio between area and power for

TABLE XVII

RATIO BETWEEN RESULTS FOR ME-NML AND THOSE FOR CMOS AND MAGNETIC NML. THE THIRD TABLE SHOWS THE INTERCONNECTION OVERHEAD TRENDS OF EACH TECHNOLOGY.

No Synch		Number of bits				
		4	8	16	32	64
Area	CMOS / ME-NML	11,0	11,1	11,2	11,2	11,2
	Mag.NML / ME-NML	4,1	3,7	3,6	3,5	3,5
Power	CMOS / ME-NML	11,2	12,9	13,1	13,4	14,0
	Mag.NML / ME-NML	54,7	50,8	49,2	48,2	48,1

With Synch		Number of bits				
		4	8	16	32	64
Area	CMOS / ME-NML	8,5	8,9	9,3	9,6	10,5
	Mag.NML / ME-NML	3,0	2,7	2,6	2,5	2,7
Power	CMOS / ME-NML	8,7	9,9	10,4	10,3	11,3
	Mag.NML / ME-NML	42,3	37,5	35,3	34,0	35,8

Interconnection Overhead		Number of bits				
		4	8	16	32	64
Area	CMOS	1,7	2,5	4,2	7,7	14,5
	Mag.NML	1,6	2,3	3,7	6,4	11,7
	ME-NML	2,2	3,1	5,1	9,0	15,4
Power	CMOS	1,7	2,4	4,1	7,0	12,5
	Mag.NML	1,7	2,3	3,7	6,3	11,6
	ME-NML	2,1	3,2	5,1	9,0	15,6

the whole circuit and those related to the body itself. For the whole analysis the number of bits has been varied from 4 to 64.

The table shows that ME-NML owns the best performance in all the cases. First, consider the area without synchronization circuitry: CMOS circuit is 11 times larger than ME-NML, Magnetic NML instead is 3.5-4.1 times bigger. The additional interconnections have a slightly

stronger impact on ME-NML than on the others. The ratio between technologies lowers to 8.5-10.5 for CMOS and 2.5-3.0 for Magnetic NML. This decrease is confirmed by the interconnection overhead table, where ME-NML has the highest values for any number of bits.

Let's switch now to the power consumption data, ME-NML is still the best technology. First consider the results without synchronization circuitry: CMOS consumes 11-14 times more energy than ME-NML, Magnetic NML instead requires around 50 times more than ME-NML. Just like for the area, when considering the full circuit, ME-NML performance suffers more for the additional interconnections. However this does not weaken its leadership significantly. The ratios decrease to 8.7-11.3 for CMOS and 42-36 for Magnetic NML. Notice once more that this behavior is also shown by the interconnection overhead values, which for ME-NML are always slightly higher than for Magnetic NML and CMOS.

Referring now to the third table only, notice that the synchronization networks have a huge impact, particularly for high number of bits. The area and power increase up to 15.6 times in the ME-NML case, 14.5 times for CMOS and 11.7 for the Magnetic NML.

For a better visual comprehension four comparison graphs have been enclosed:

1. Area comparison without synchronization networks (Figure 41);
2. Power comparison without synchronization networks (Figure 42);
3. Area comparison with synchronization networks (Figure 43);
4. Power comparison with synchronization networks (Figure 44).

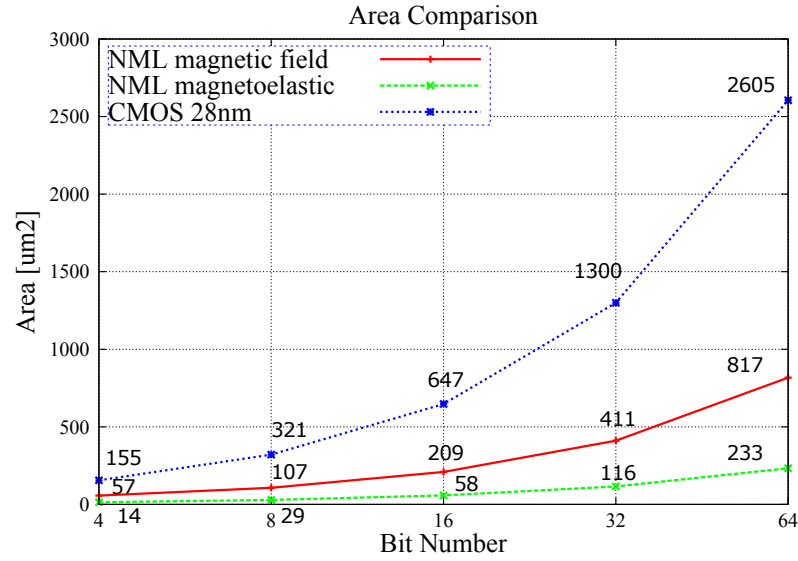


Figure 41. Area comparison between the three GFM implementations without synchronization networks.

Data on occupied area, without considering the additional networks, is plotted in Figure 41. Of course the area increases with the number of bits, the interesting outcome is that the CMOS implementation has the worst performance, while the smallest area belongs to the ME-NML circuit. The CMOS library chosen is the most scaled that we have, but there currently exist transistors smaller than $28nm$. However, even considering a $14nm$ library, it would result in a CMOS scaling of 4 times, so that the ME-NML still has a considerable margin. Moreover NML magnets can be scaled too.

Figure 42 depicts instead the power comparison, still neglecting the upper and lower interconnections. The curves are similar to the graphs of the circuit area. However, while ME-NML confirms itself as the best technology, the Magnetic NML is now definitely the worst one. It is

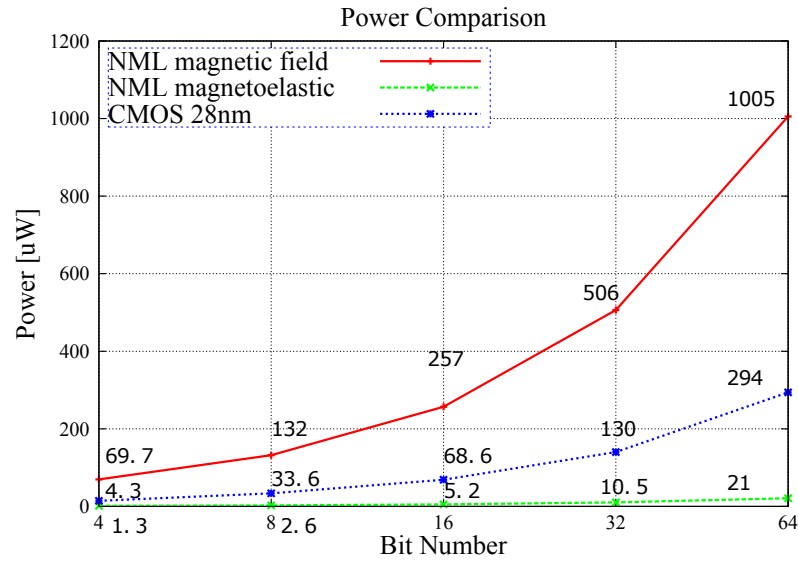


Figure 42. Power comparison between the three GFM implementations without synchronization networks.

though what expected, as the Magnetic Clock network requires a very high current to generate the magnetic field.

For what concerns the synchronization networks, simply notice that the circuit body only grows horizontally, while the upper and lower networks grow also vertically, hence they grow quadratically. This additional cost is often neglected in literature, even though such circuitry is essential to properly interface our module with others. This is a recurring problem of QCA circuits [22], because of their intrinsic pipeline nature.

Figure 43 shows the occupied area for the three GFM versions after adding the preskew/des skew modules. All the curves have similar trends and ME-NML and CMOS have respectively the best and worst performance, like when considering the area of the GFM's body only.

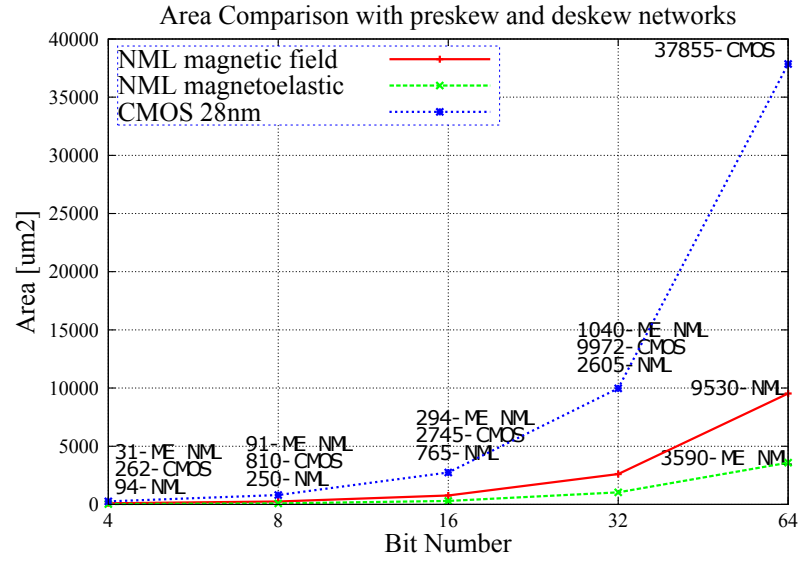


Figure 43. Area comparison between the three GFM implementations with synchronization networks.

Figure 44 shows instead the power consumption for the three GFM versions after adding the preskew/deskew modules.

The final considerations are mostly three. First, the MagnetoElastic NML has confirmed its potentialities. With a proper architectural choice it leads to a great reduction of circuit area and power losses of the clock network, which was the insuperable drawback of previous NML implementations. Second, the synchronization networks have a huge impact on performances, thus it is imperative to take them into consideration when they are required. Third, even with these excellent results, NML technology is not meant as a replacement for CMOS technology, since its speed is intrinsically limited. For this very circuit, with the 28nm library exploited, CMOS technology would be able to work at $7GHz$, 70 times faster than the NML maximum

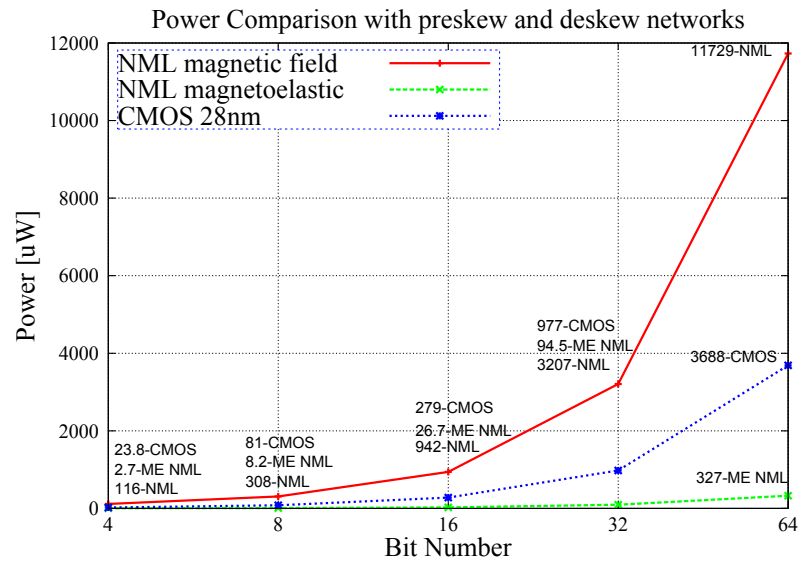


Figure 44. Power comparison between the three GFM implementations with synchronization networks.

frequency: 100MHz . The benefits of NML technology are bounded to circuit area and power consumption, together with its intrinsic memory ability.

CHAPTER 6

CASE STUDY II: MULTIPLY ACCUMULATE UNIT (MAC)

It has been proved that Magnetoelastic NML overcomes both Magnetic Clock NML and CMOS technologies in terms of circuit area and power consumption (Chapters 4 and 5). The ME-NML implementation of the bit-serial Galois Multiplier, organized as a systolic array, turned out to be extremely compact and easily scalable. However not all kinds of architectures are suitable for ME-NML technology. In this chapter we start investigating which architectures are best suited for this technology and why. The final goal is to develop some general guidelines for identifying which circuit organizations and design approaches can boost ME-NML performances.

Our inquiry focuses on the dualism between serial and parallel structures, trying to determine which one of the approaches gets the best out of ME-NML. The case study chosen is a generalized Multiply Accumulate unit (MAC), which will be realized in three different versions: fully parallel, serial-parallel, fully serial. The three generalized MAC will be designed, modelled, simulated and compared in terms of area, power, throughput and latency.

The MAC unit is composed by a multiplier, an adder and an accumulator: The main scheme is depicted in Figure 45. The operation performed by this circuit is the following:

$$\begin{aligned}
 t_0 : Res_0 &= A_0 \cdot B_0 \\
 t_1 : Res_1 &= Res_0 + (A_1 \cdot B_1) \\
 t_2 : Res_2 &= Res_1 + (A_2 \cdot B_2) \\
 &\dots \\
 t_N : Res_N &= \sum_{i=0}^N A_i \cdot B_i
 \end{aligned}
 \tag{6.1}$$

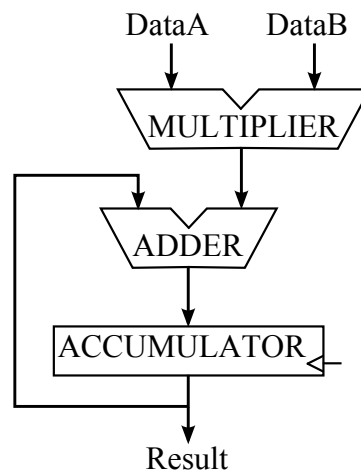


Figure 45. Multiply Accumulate unit scheme.

6.1 Parallel Implementation

The first implementation presented is a parallel version of the MAC unit. It is basically composed by a parallel multiplier and an adder with feedback. The accumulator function is instead embedded, as ME-NML is intrinsically pipelined. The array multiplier and the ripple carry adder (RCA) have been chosen as components of the parallel MAC, because they both have a systolic array architecture. They are composed by blocks that communicate only with their neighbors, avoiding long interconnections and feedback. The feasibility of ME-NML circuit design strongly depends on those properties.

It is crucial to point out that the best circuits for CMOS usually maximize performances in terms of working frequency, at the cost of an higher complexity. However such optimizations do not necessarily have the same advantages when designed with ME-NML. The intrinsic pipeline sets a fixed maximum working frequency that depends on the technology itself and not on the architecture adopted. Therefore the optimization for ME-NML cannot improve the circuit speed, it has to be aimed elsewhere:

- Reduce area occupation and consequently also the power consumption;
- Minimize internal delays limiting the pipeline stages of feedback loops, affecting positively the overall circuit latency. The throughput instead does not depend on circuit layout if the interleaving technique can be exploited properly.

From this considerations and from the previous case study we can state that when handling ME-NML technology, the plainer the layout the better the performance. To double check this deductions we also designed a multiplier and an adder different from the Array Multiplier and

RCA. Both the Booth's multiplier and the Carry Look-ahead Adder proved to be much more complex and big, especially the latter.

6.1.1 Array Multiplier and Ripple Carry Adder

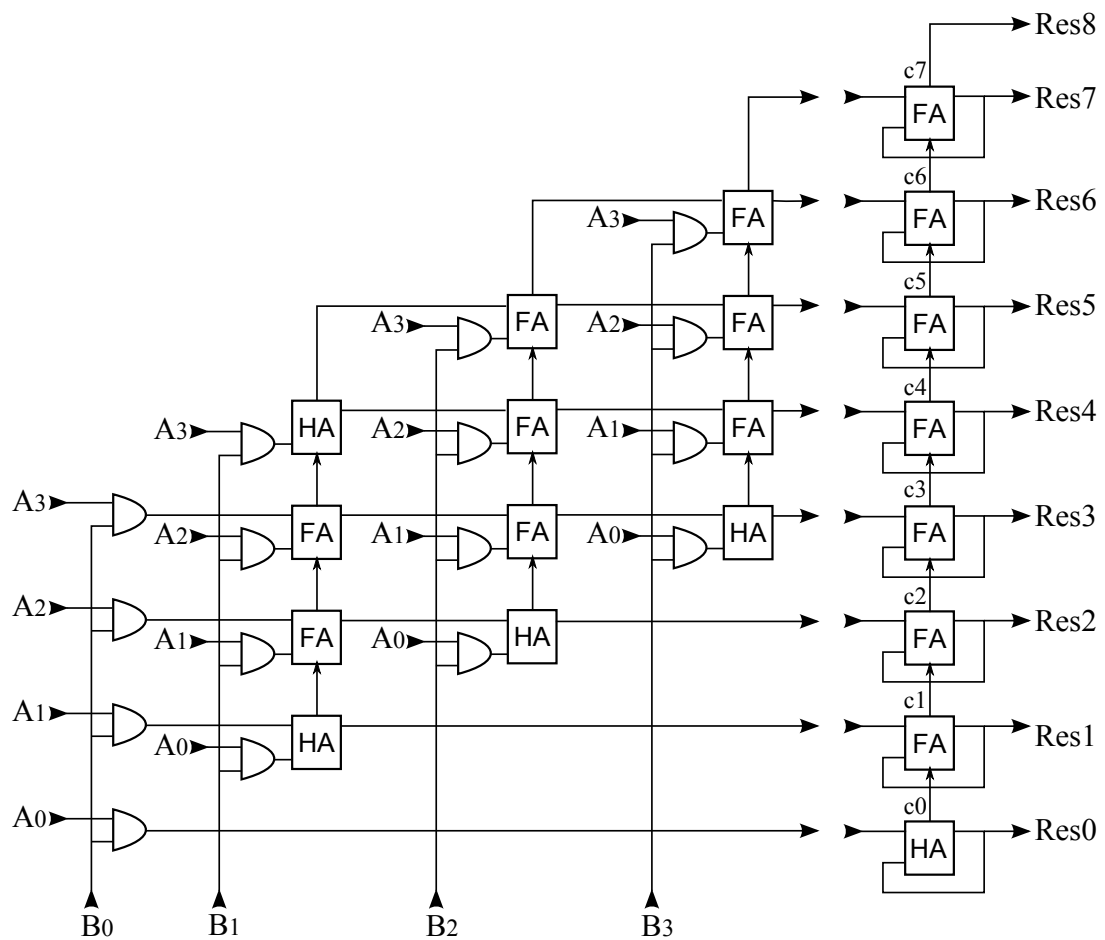


Figure 46. 4-bit MAC scheme. Array Multiplier on the left and Ripple Carry Adder on the right.

The scheme of the 4-bit Array Multiplier (left) and the 8-bit Ripple Carry Adder (right) are drawn in Figure 46, where *FA* and *HA* stand for Full Adder and Half Adder. The two inputs *A* and *B* are parallel, just like the output *Res*. Let's consider a MAC unit with N_{bit} inputs *A* and *B*. The result of the N-bit multiplication is a $2N_{bit}$ number, therefore the adder will have $2N_{bit}$ inputs. In fact in Figure 46 we have a 4-bit multiplier and a 8-bit adder.

Notice that the multiplier is basically a matrix of Full Adders, so it is two-dimensional and its area grows quadratically with the circuit parallelism. The Array Multiplier's algorithm is the simplest one, it follows step by step the handmade multiplication. Partial products are shifted and added to an intermediate result. Each AND ports column in the drawing evaluates a partial product, which is then added to the intermediate result by the Full Adders. Moreover every AND column has a 1-bit shift with respect to the previous column to assure the proper alignment of the partial products sum. The final product goes to the RCA, which sums it with the accumulator's value, which is stored in the RCA's feedbacks. Within the adder the carry propagates vertically from one FA to the next.

The circuit arrangement and orientation imitates the ME-NML implementation that will be presented shortly, to guarantee an easy visual comparison between the two circuits. However there are some differences. The scheme in Figure 46 does not have any pipeline stage, while the ME-NML MAC will be fully pipelined.

For the two circuits to be more similar, each row and column in Figure 46 should represent a pipeline stage. Furthermore the free space in the bottom part of the circuit will be removed

to optimize the circuit area, placing half of the Adder's FA modules horizontally under the Multiplier.

6.1.2 Full Adder and Half Adder

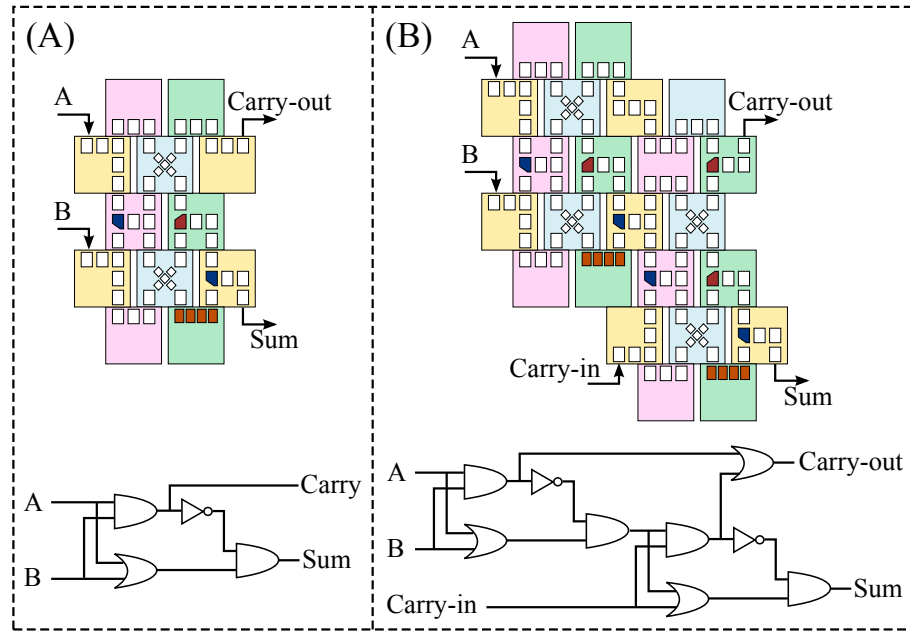


Figure 47. Half Adder and Full Adder realized with both ME-NML and CMOS technologies.
(A) Half Adder. (B) Full Adder.

The basic modules of the MAC unit are Full Adder (FA) and Half Adder (HA), they represent the first step of the ME-NML MAC design. These modules can be arranged in many different ways, one version of the Half Adder has already been depicted in Figure 20. Here we present the FA and HA that have been exploited to create the parallel MAC. Figure 47.A

shows once again the ME-NML HA together with its CMOS scheme, Figure 47.B encloses the FA instead.

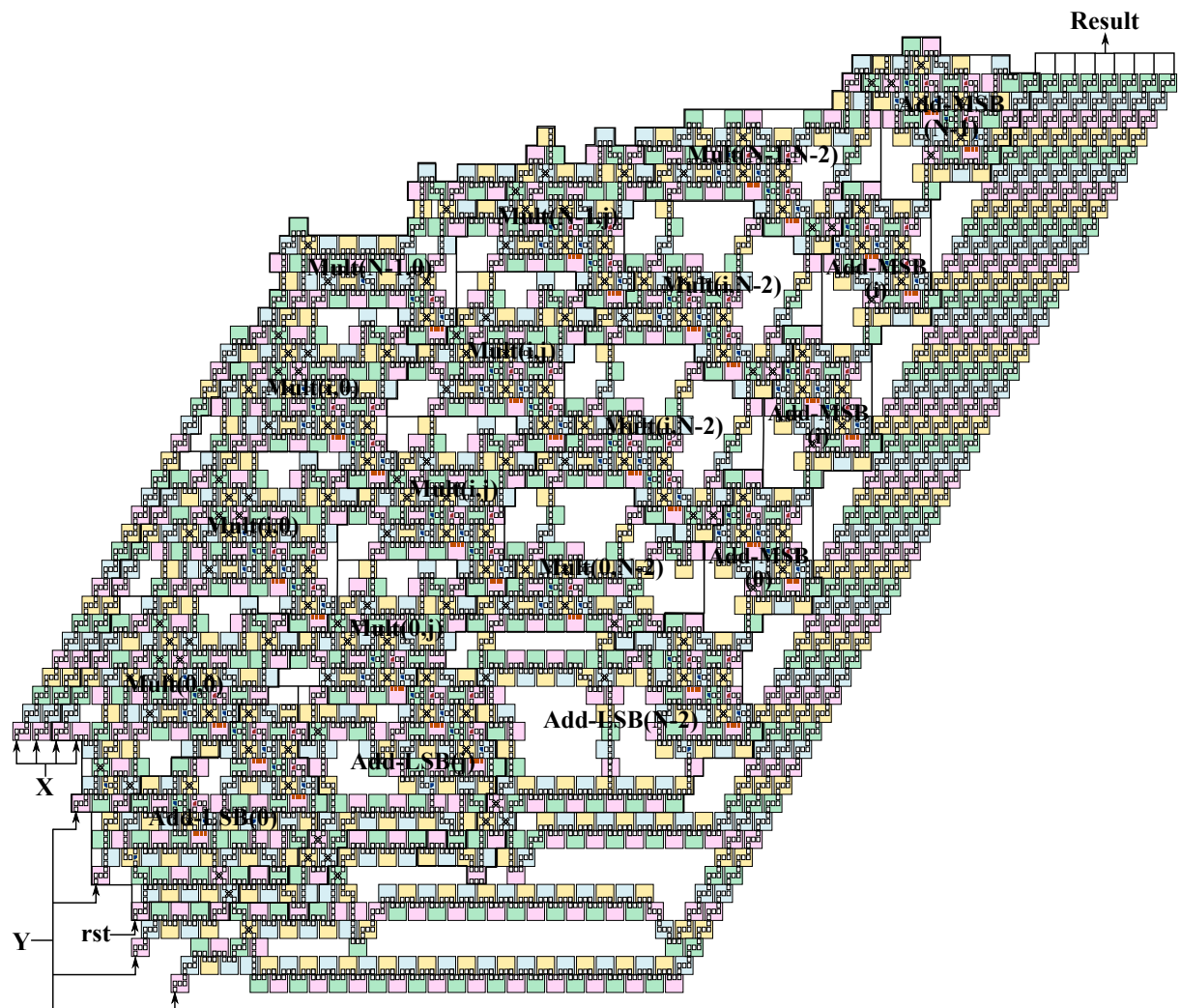


Figure 48. 4-bit parallel ME-NML MAC unit. Labels identify the base blocks of Multiplier and Adder.

Basically the whole parallel MAC has been designed exploiting these blocks only, providing them with a properly routed network of interconnections. A mandatory reset signal that propagates toward all the feedback loops of the RCA. The reset sets to '0' the feedbacks for the first operation and whenever the accumulator needs to be zeroed.

6.1.3 Basic blocks

The circuit organization is the same as for the Galois Field Multiplier. A set of basic blocks is defined so that they can be assembled to create a generic N-bit MAC. The blocks can be divided in three groups:

Multiplier blocks There are 9 base blocks and they are represented in Figure 49. The indexes of $Mult(-,-)$ refer to their position and occurrences within the matrix of a generic N-bit Array Multiplier. The main inputs and outputs are all labeled. X and Y are the multiplier inputs. The internal carry and partial sum signals are referred to as c and S . The reset signal rst does not concern the multiplication, it is simply passing through the Array Multiplier on its way toward the Adder. The S_{out} outputs of the blocks in the Row 0 and in Column N-1 have the multiplication result bits, they will be connected to the input of the RCA. Row 0 has the results from $Res(0)$ to $Res(N-1)$, while Column N-1 ($Mult(0,N-1)$ excluded) has the results from $Res(N)$ to $Res(2N-1)$, where $Res(2N-1)$ is the signal c_{out} of the block $Mult(N-1,N-1)$. The Adder's base blocks are also labeled in Figure 48, which shows the whole 4-bit parallel MAC.

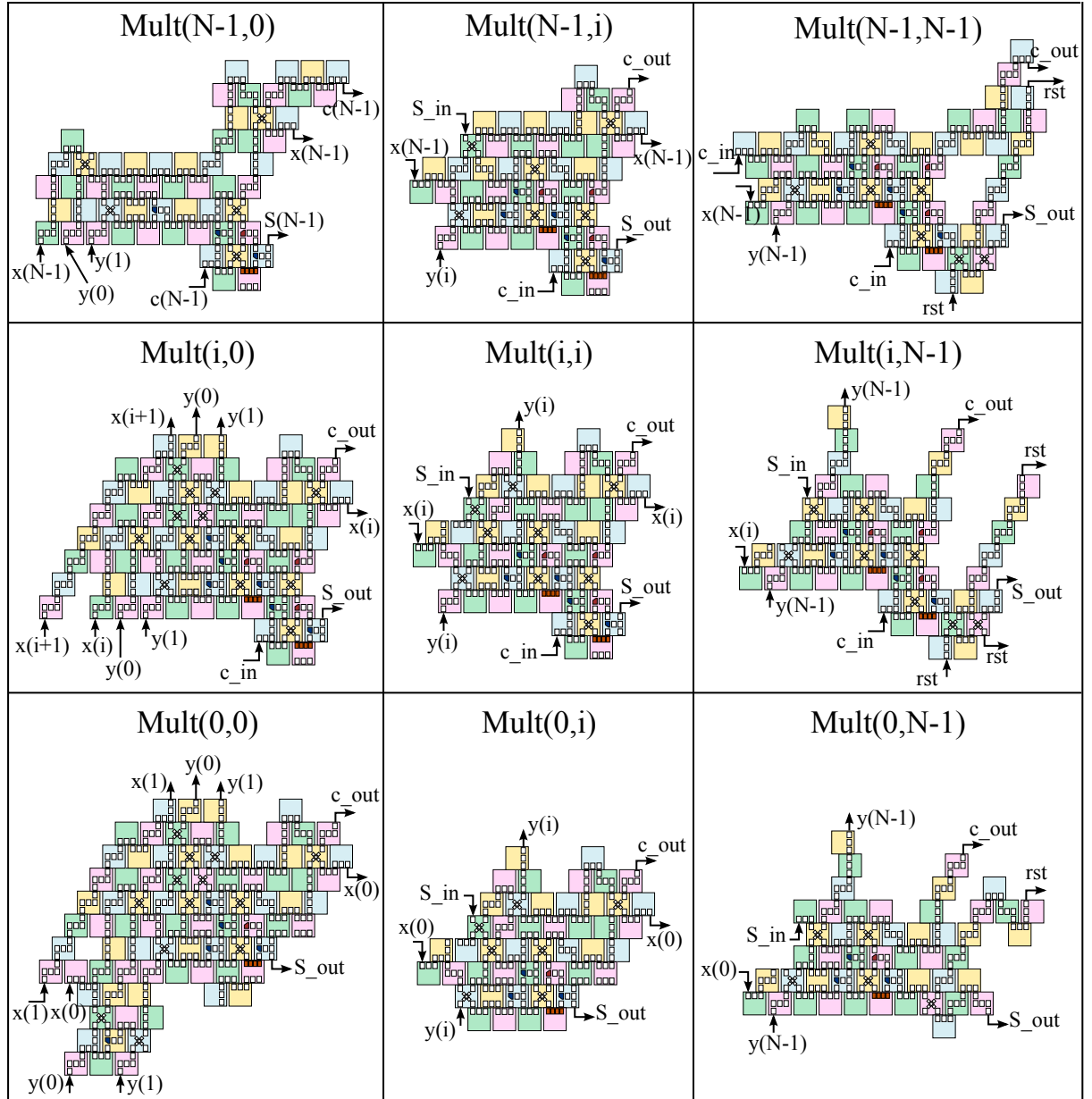


Figure 49. Base blocks of the Array Multiplier for the parallel MAC.

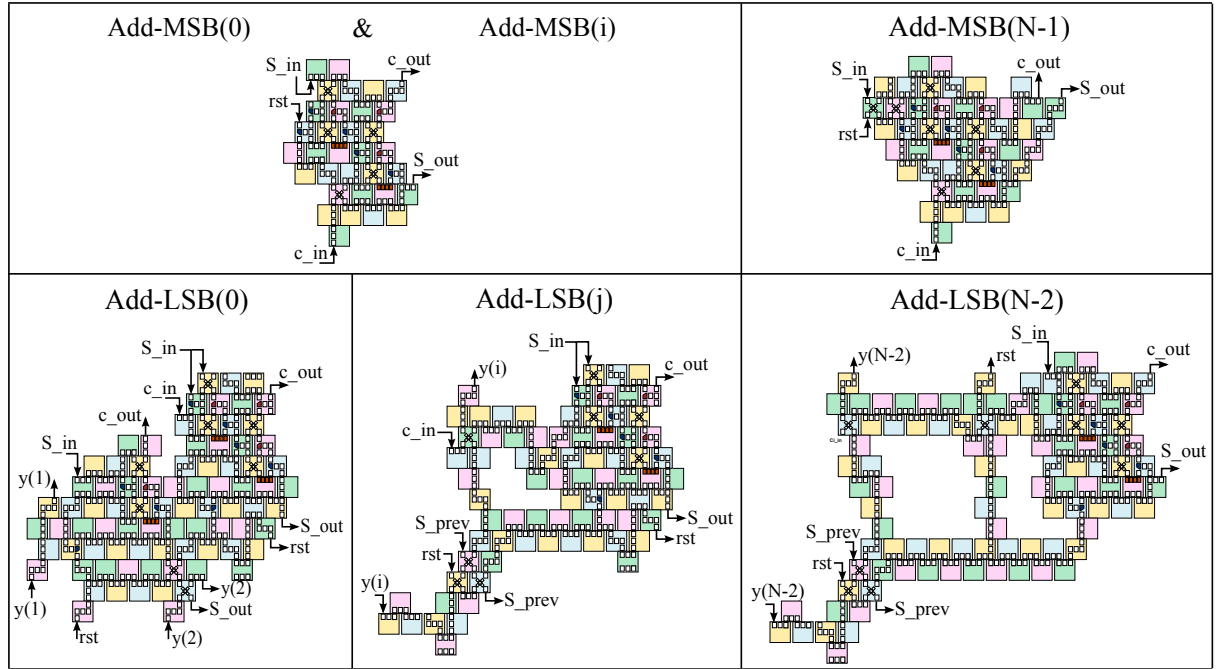


Figure 50. Base blocks of the Ripple Carry Adder for the parallel MAC

Adder blocks The 5 base blocks for the RCA are in Figure 50. As clear from Figure 48, half blocks are placed horizontally under the multiplier (*Add-LSB*), while the other half is placed vertically on the right side of the multiplier (*Add-MSB*). *Add-LSB(0)* contains the first two modules of the RCA: an Half Adder and a Full Adder. All the other blocks enclose a single FA with its feedback loop. Once again the main inputs and outputs are labeled:

S_in* and *S_out *S_in* is one bit of the multiplication result, while *S_out* is one bit of the final MAC result.

c_in* and *c_out *c_in* and *c_out* are simply the carry in and carry out that respectively come from the previous FA and go to toward the next.

rst Each FA receives a reset signal and splits it in two branches. The first acts on the feedback loop while the second is forwarded to the next block.

y The *Add-LSB* blocks lie below the multiplier, therefore the *Y* bits have to pass through them.

Interconnections blocks To describe with the VHDL model a generic MAC also the interconnections have been divided in base blocks (Figure 51). 9 blocks are needed to build the interconnections for any circuit parallelism. The 7-bit MAC is the first one that requires all the 9 blocks, implementations smaller than 7-bit only require some of them. One of the functions of interconnection regions is the inputs and outputs synchronization. Just like for the Galois Multiplier, they assure that bits of the same signal can be fed and acquired simultaneously, guaranteeing the easiest possible interface protocol with other devices.

6.1.4 VHDL description and circuit simulation

The VHDL model and simulation procedure is the same as for the Galois Multiplier. The generic parallel MAC has been modeled with the usual components' hierarchy and tested up to 64 bits. Thanks to Matlab, for each parallelism to be tested, we created a set of 1000 random

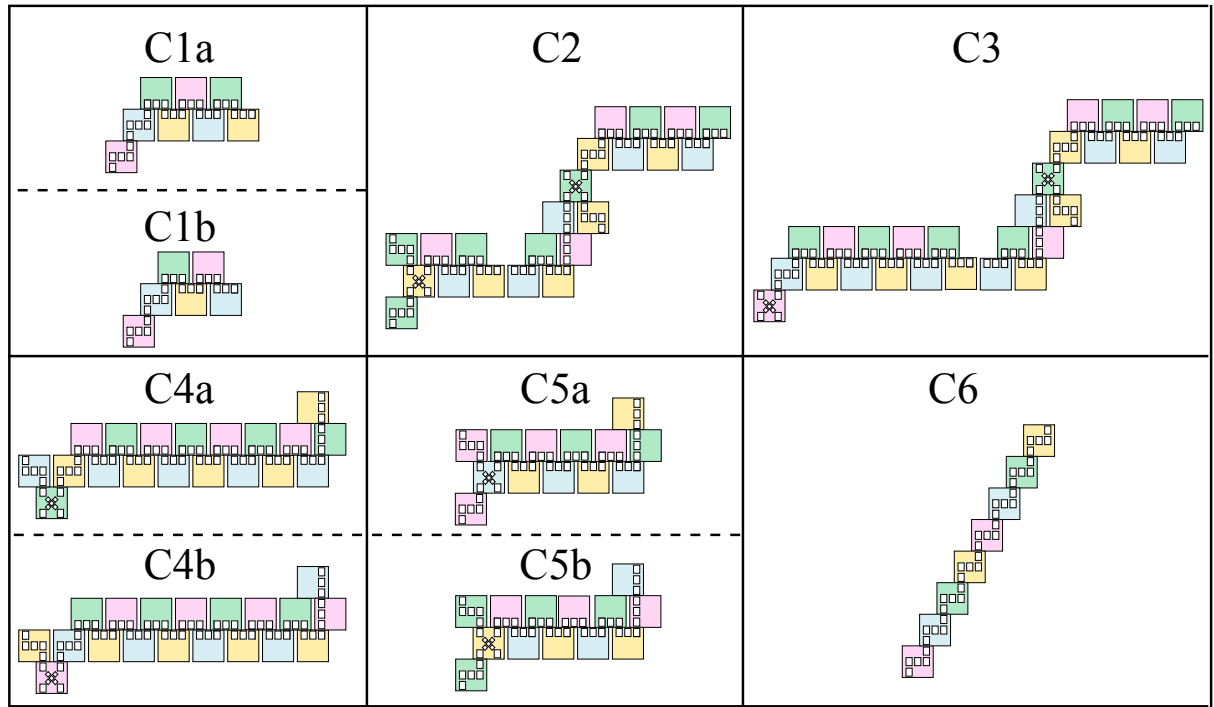


Figure 51. Base blocks for the interconnections of the parallel MAC implementation

inputs and related results. The VHDL testbench acquires those random inputs and writes the simulation results into another file, which is to be compared to the expected results.

The top entity `MAC_N_bit` (see Listing 6.1) instantiates the Multiplier, the Adder and various interconnections entities. Each of these entities will instantiate its own base blocks introduced in Section 6.1.3. In Listing 6.1, beside the performance `natural` signals and the clocks, there are the N-bit inputs `X`, `Y` and the 1-bit `reset`. The outputs are the 2N-bit `MAC_results` and the carry out of the MSB Full Adder of the RCA (`MAC_Co`).

Listing 6.1. VHDL top entity of the parallel MAC

```

entity MAC_N_BIT is
2   port(X,Y:in std_logic_vector (N_BIT-1 downto 0);
      reset:in std_logic;
      MAC_result:out std_logic_vector (2*N_BIT-1 downto 0);
5   MAC_Co:out std_logic;
      clkA, clkB, clkC, clkD:in std_logic; — Main clock and clock zones
      n_mag:out natural := init_natural; — # of magnets
8   n_zones:out natural := init_natural; — # of cells used
      AREA_EFF:out natural; — Total magnets area
      AREA_TOT:out natural; — Total area occupied by the cells
11  Er:out natural; — Energy consumption of nanomags
      Ec:out natural); — Energy consumption of clock
end MAC_N_BIT;

```

6.1.5 Timing Analysis

The Array Multiplier is composed by a matrix of $N \times (N - 1)$ base blocks. Increasing the circuit parallelism the matrix will get bigger, affecting the overall circuit latency. On the other hand for any number of bits the RCA will always be only one column thick, having a constant impact on the latency. Every block of the Multiplier requires 5 clock cycles to be crossed horizontally (signal X) and 2 vertically (signal Y). Therefore the inputs (bottom-left) need $(5(N - 1) + 2N + 5) \cdot T_{clk}$ to reach the result. The additional 5 clock cycles are fixed and mainly refer to the time needed to pass through the RCA. The critical paths are highlighted with blue in Figure 52.A.

In a MAC each multiplication's result is added to the value in the accumulator. Since each block of the adder has a 5 clock long feedback loop (Figure 52.B), the operations cannot be fed to the MAC in a continuous flow. Two operations must be fed with 5 cycles of delay in order for them to be added to each other. Therefore to reach the maximum throughput 5 uncorrelated

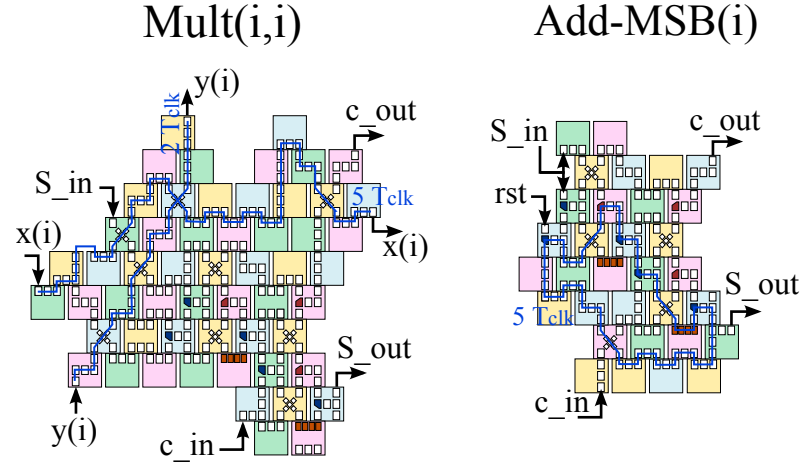


Figure 52. Critical paths of the parallel MAC. (A) Critical paths of multiplier's base blocks. (B) Feedback loop of adder's base blocks.

operations should be interleaved. With the interleaving the throughput is of one operation per clock cycle. All information regarding timing performance of the parallel MAC are listed in Table XVIII. The table indicates the throughput when the interleaving technique is exploited, hence the maximum possible throughput.

TABLE XVIII

TIMING PERFORMANCE OF THE PARALLEL MAC			
N bit	Interleaving	Throughput	Latency: 1st Result out
4	5 op.	$1/(T_{clk})$	$28T_{clk}$
8	5 op.	$1/(T_{clk})$	$56T_{clk}$
N	5 op.	$1/(T_{clk})$	$5(N - 1) + 2N + 5 \cdot T_{clk}$

6.2 Serial-Parallel Implementation

The parallel MAC described in the previous section has a 2D layout. The idea for the second version of the MAC was to create a circuit organized as a 1D array of elements. This section presents the best circuit we were able to obtain. It is referred to as serial-parallel MAC, because it has serial inputs and parallel output. While the design of the parallel MAC was trivial, in this case it was not possible to design a simple circuit able to keep up with the parallel implementation. The circuit's body itself has excellent characteristics, but its input/output protocol is unique, it would be very difficult to interface it directly with other devices. Moreover additional interconnections are required, as in the case of the Galois Multiplier (Chapter 4), terribly spoiling the performances.

6.2.1 Circuit scheme

The scheme in Figure 53 is the body of the 4-bit serial-parallel MAC, but to have serial inputs and parallel output it requires additional registers. Let's discard for now the preskew/deskew networks. The circuit counts $2N_{bit}$ 1-bit adders. Each adder has its own feedback, so that the array of FAs can function as an accumulator. A reset signal allows to reset the accumulator whenever necessary. As usual the scheme is fully pipelined to imitate ME-NML behavior. The timing protocol follows the handmade multiplication procedure, where the N partial products are evaluated one by one and summed together.

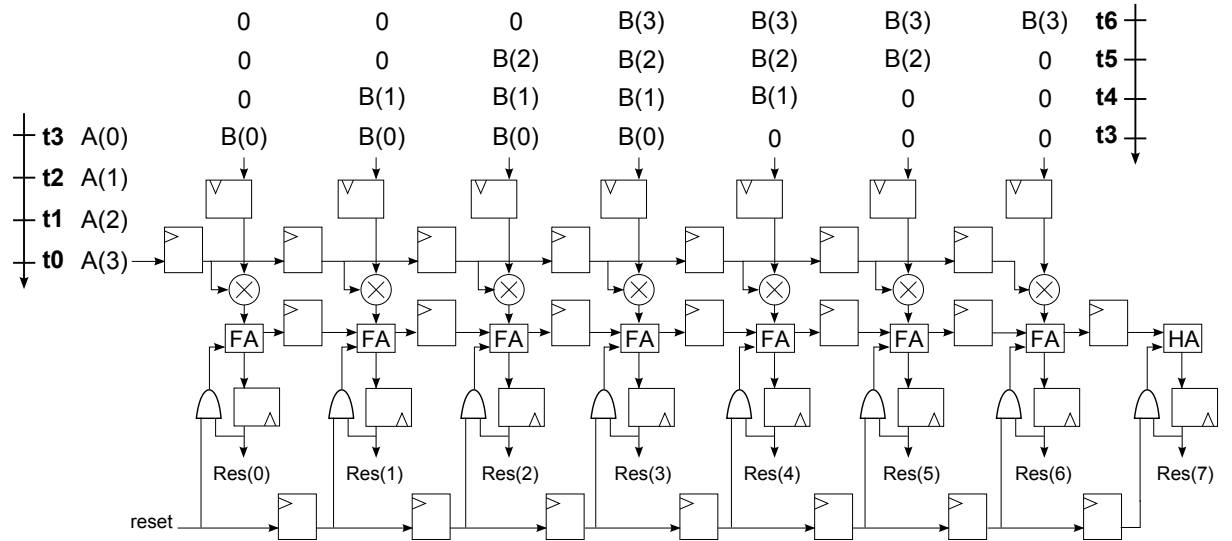


Figure 53. Body of the 4-bit serial-parallel MAC.

Figure 53 also shows a timeline that explains the inputs protocol to execute a 4-bit operation. At t_0 A is fed serially starting from the MSB. After all 4 bits of A enter the shift register, they are multiplied bitwise with $B(0)$, which has been applied in the meantime. This gives the first 4-bit partial product which goes in the first four Full Adders, while the remaining three Adders receive '0'. Data B always has $N - 1 = 3$ bits equal to '0', because partial products have a N -bit width. After the first partial product is evaluated data A bits shift to the right and are multiplied with data $B(1)$ which arrives right after $B(0)$ but shifted of one step toward the MSB (right). In this way the second partial product is correctly aligned to the first one, so that they are added properly.

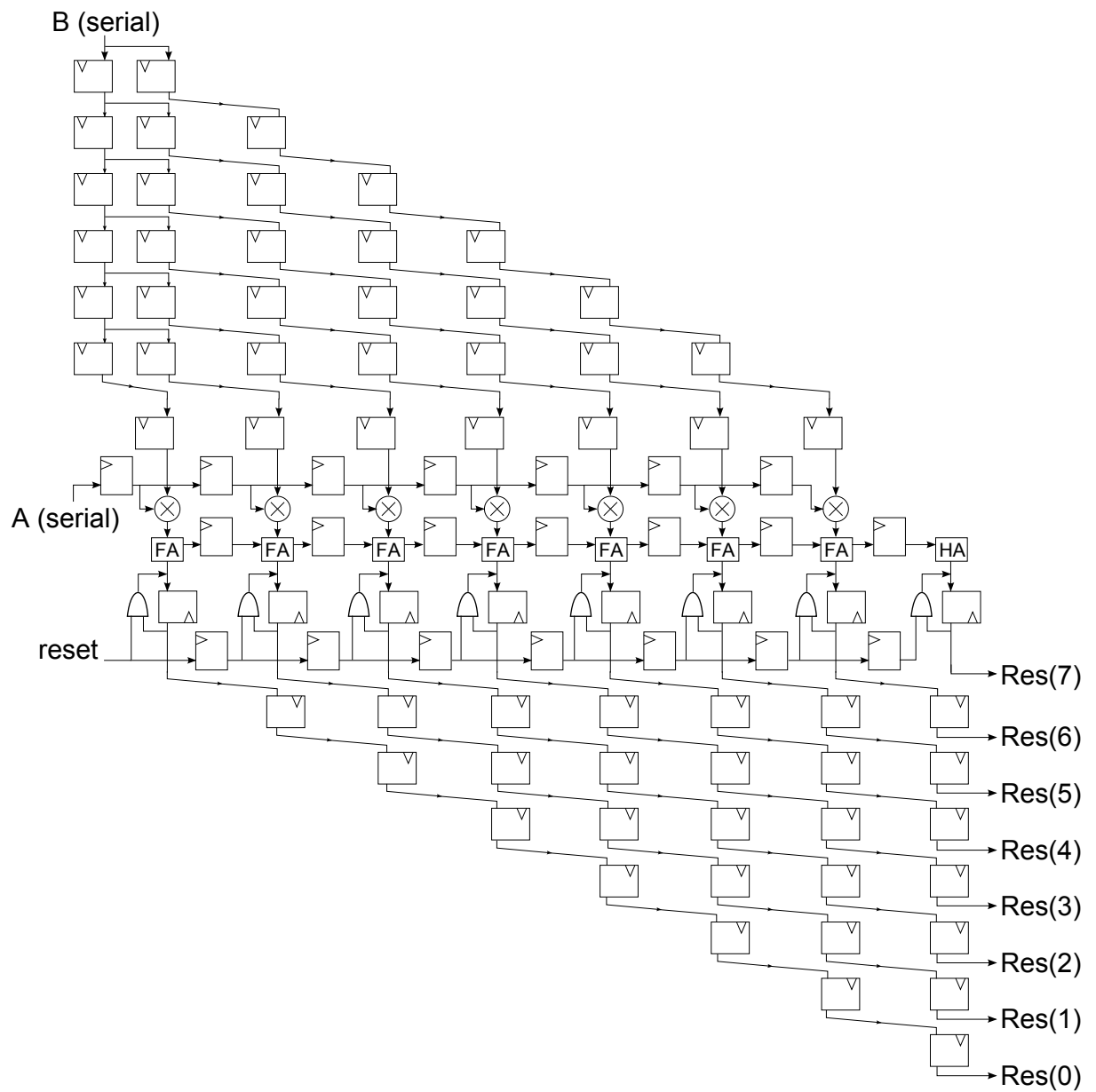


Figure 54. Full scheme of the 4-bit serial-parallel MAC.

Evaluating all the partial products only requires N clock cycles. But another N cycles have to be spent feeding '0s' to prepare the circuit for the next operation. The Full Adders' carry-out signals are propagated to the carry-in of the next FA on the right. It is now evident that input B enters the circuit in a way that would make it difficult to interface this circuit with others. The same applies to the result, whose bits need to be synchronized, just like for the Galois Multiplier in chapter 4. In Figure 54 the preskew (for B) and deskew (for Res) networks are added on top and bottom of the circuit body. It is immediately clear their great impact.

Also, the input B is distributed to all FA blocks, while it should be given only to 4 blocks at a time, assigning '0' to the others. As a consequence, for the circuit to work properly, the '0s' must come from data A . Input A , after giving the N bits of data A , will give N '0s'. In this way the time to execute a single operation doubles.

6.2.2 ME-NML implementation

The main element is a Full Adder with a feedback loop for the result. The ME-NML FA used for our serial-parallel MAC is drawn in Figure 55. The feedback loop, highlighted in blue, is 3 clock periods long. Like the previous cases, the feedback is the critical path that decides the delay required between inputs. In this case a input bit has to be served every 3 clock cycles, hence the maximum throughput can be reached with a 3-operations interleaving. The two other patterns point out that the base block takes 2 clock cycles to be crossed horizontally, and 3 cycles vertically.

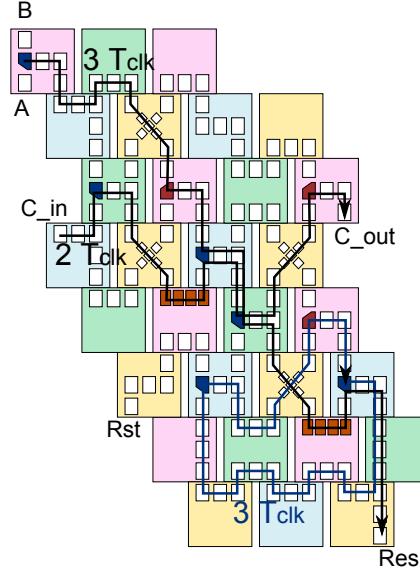


Figure 55. Full Adder block for the serial-parallel MAC. Three patterns underline horizontal crossing, vertical crossing and feedback loop.

The full adders in the scheme of Figure 54 only have $1T_{clk}$ latencies, therefore the timing is slightly different than the final ME-NML implementation. The two circuits are exactly the same apart from the internal delays. For example consider the input conditioning structure for B in Figure 54, each register of the column at the top-left corner is realized in ME-NML with a 3 cycles delay.

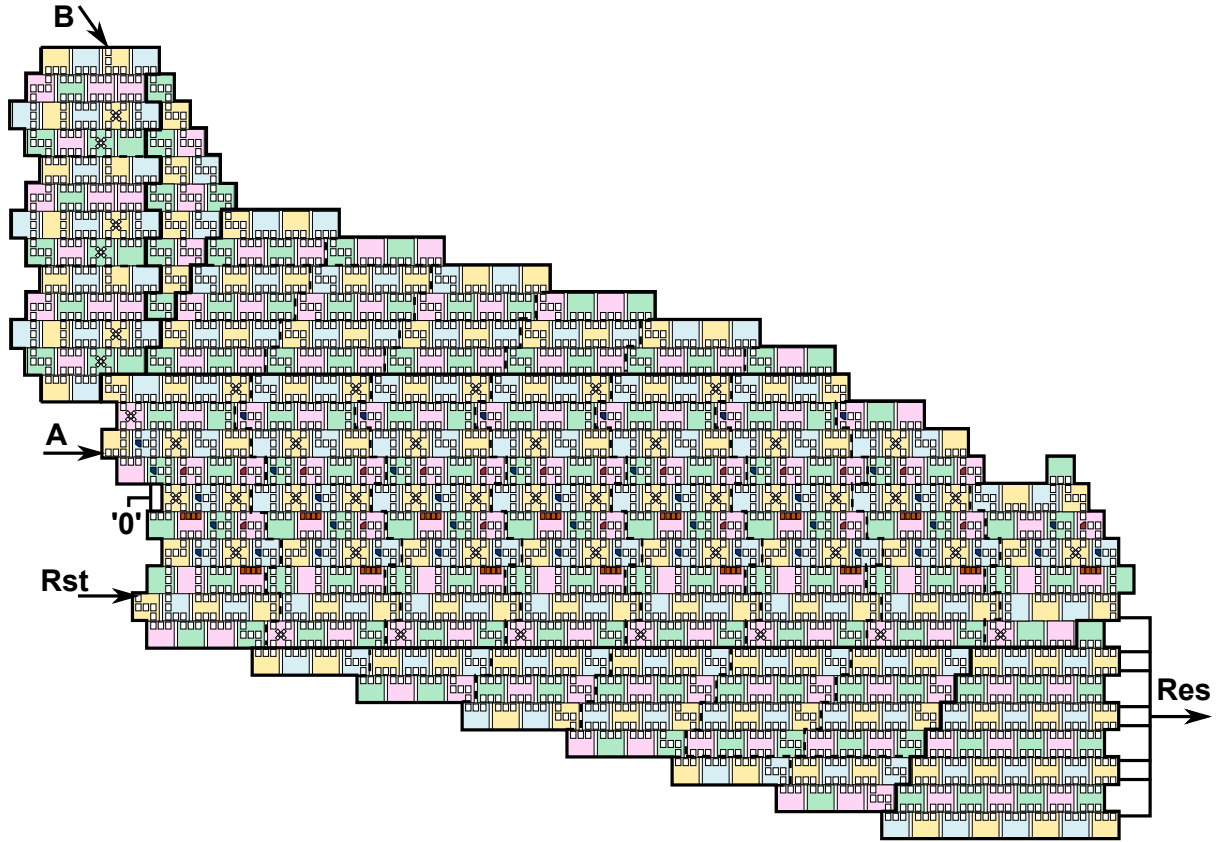


Figure 56. ME-NML implementation of the serial-parallel MAC.

The ME-NML final circuit of the 4-bit MAC is in Figure 56. The circuit is divided into four main regions and within each region the dashed lines identify the basic blocks. To construct the generic MAC each region has been treated separately. First, we selected the set of recurrent blocks, then we investigated how to organize them so that combining them properly it is possible

to create a MAC with any number of bits. The full set of blocks are enclosed in Figure 57. A different VHDL entity defines each region:

MAC_1D_body The central part is composed by 4 different blocks. It contains all the logic functions, while the other regions are exclusively interconnections.

MAC_1D_conn_above Describes the two regions pointed in Figure 56. The triangular region contains only one kind of cell, so it has been generated directly without the need of defining base blocks. The other part has been divided into 5 types of block. Their organization (described in the VHDL model) is quite tricky, but they still can recreate the required interconnections for any circuit parallelism.

MAC_1D_conn_below The left part is very similar to the right part of the connections above. The 5 base blocks of the two regions are lightly different. On the other hand the right part is composed only by two type of cells, therefore it has been described directly without requiring the definition of basic blocks.

MAC_1D_input_cond This conditioning network simply models a shift register, it allows to provide simultaneously the same bit of data B to multiple FAs of the **MAC_1D_body** region.

The whole circuit has been described with the RTL model we developed for ME-NML technology. A substantial effort was devoted to the generic description of the interconnection networks. The top entity **MAC_1D** instantiates the four entities reported above. Notice in Listing 6.2 that the inputs A and B are serial, while the *Result* is parallel.

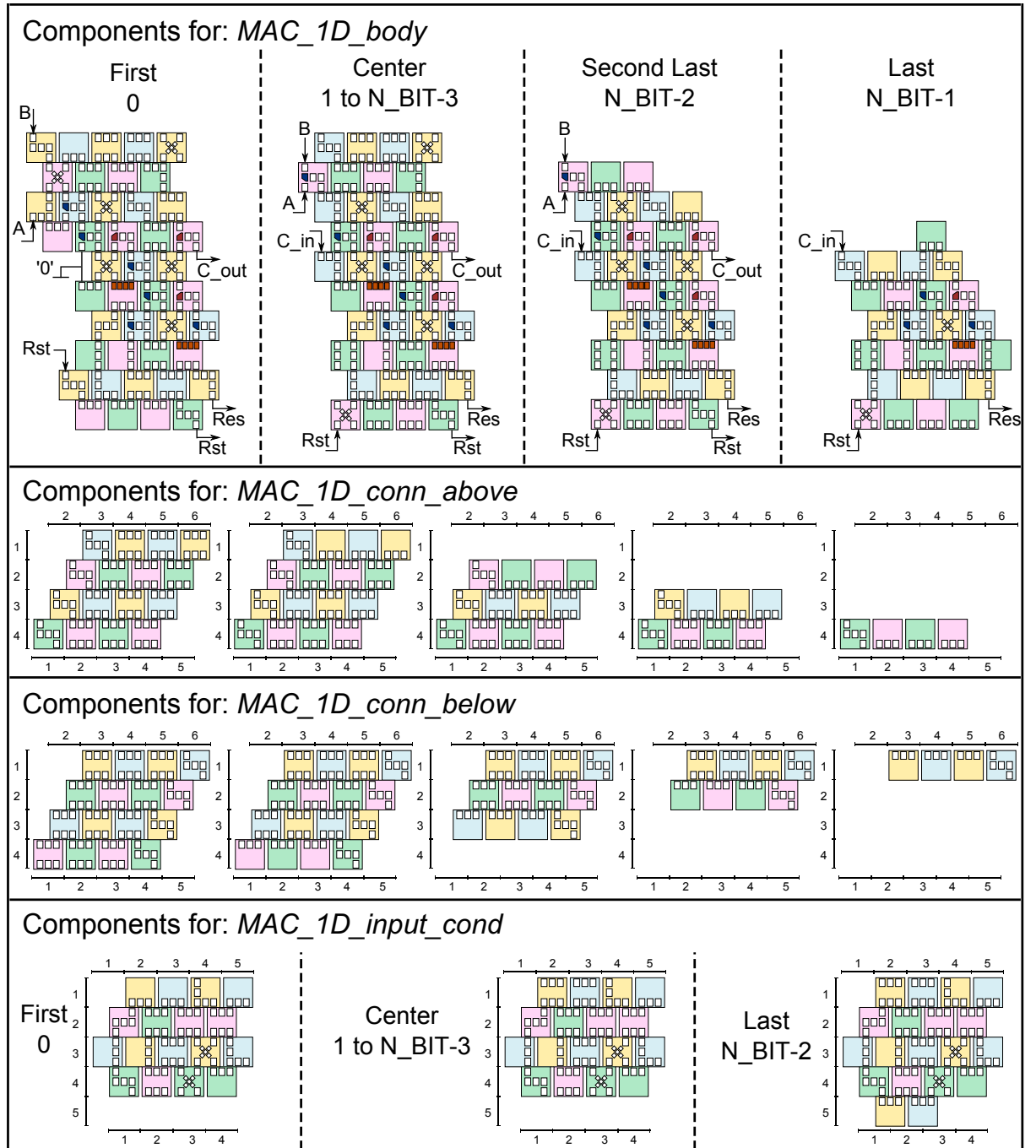


Figure 57. Basic blocks for each region of the serial-parallel MAC.

Listing 6.2. VHDL top entity of the serial-parallel MAC

```

2  entity MAC_ID is
    port (A,B,Rst: in std_logic;
          Result: out std_logic_vector(2*N_BIT-1 downto 0);
          [...] — Omitted clock and area-power signals
          )
5  end MAC_ID;
```

6.2.3 Timing analysis

Data A and data B give their bits serially with a delay of 3 clock cycles between them. Then the time required to provide all the bits is $3N_{bit} \cdot T_{clk}$. After that for another $3N_{bit} \cdot T_{clk}$ the inputs are set to '0', until a new operation starts. The throughput would be equal to one operation every $3 \cdot 2N_{bit}$ clock cycles, but exploiting the interleaving technique it goes up to $1/(2N_{bit} \cdot T_{clk})$. Table XIX reports these results and also evaluates the overall circuit latency. Data A arrives directly at the MAC's body, data B instead has to cross the preskew network first. Also, data B must reach the MAC's body when all the bits of data A have entered the circuit. As a consequence data B must be fed earlier than data A . More precisely the two inputs must be applied with a time difference of $3(N_{bit} - 1) \cdot T_{clk}$.

TABLE XIX

TIMING PERFORMANCE OF THE SERIAL-PARALLEL MAC			
N bit	Interleaving	Throughput	Latency: 1st Result out
4	3 op.	$1/(8T_{clk})$	$36T_{clk}$
8	3 op.	$1/(16T_{clk})$	$76T_{clk}$
N	3 op.	$1/(2N \cdot T_{clk})$	$(6(N - 1) + 4N + 2) \cdot T_{clk}$

6.3 Serial Implementation

The third and last implementation analyzed in this work is the Serial MAC, which has both serial inputs and output. The starting idea was to create a circuit exploiting only two 1-bit Full Adder, one for the multiplier and one for the adder.

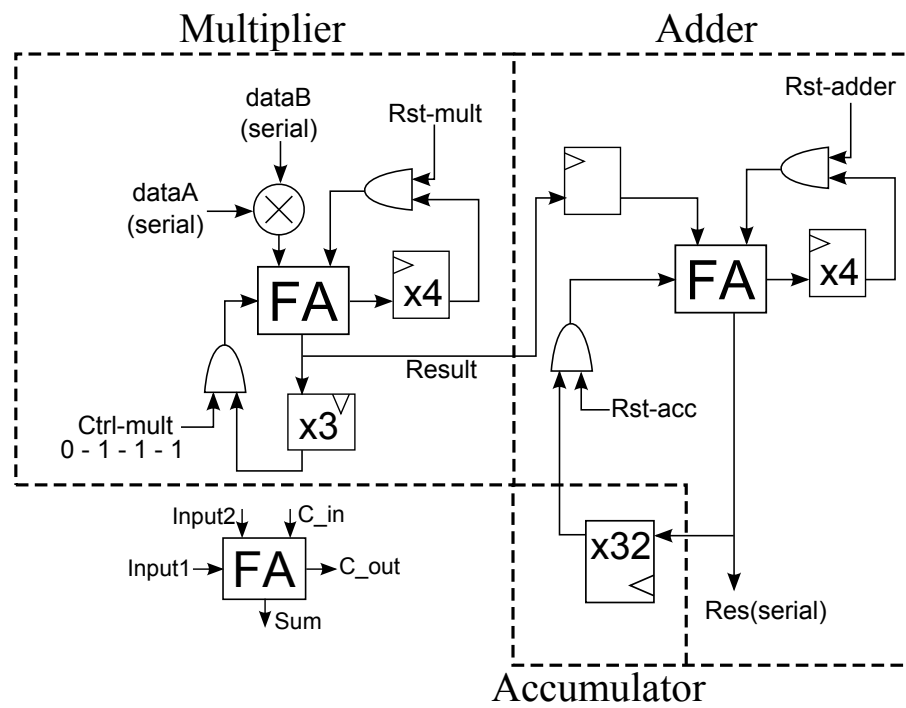


Figure 58. Scheme of the 4-bit serial MAC (preliminary implementation).

6.3.1 Serial MAC scheme

The architecture that best suited our demands is represented in Figure 58 in its 4-bit version. It consists of a serial multiplier, a serial adder and an accumulator, which is nothing less than the adder's feedback loop. Registers with the $x3, x4, x32$ labels represent multiple cascaded registers (respectively 3, 4, 32) that have been combined together for a sharper visual understanding.

6.3.1.1 Multiplier

The multiplier accurately imitates the handmade multiplication algorithm (Figure 59 shows the 4-bit case). The serial inputs A and B are multiplied and then fed to the first Full Adder. Their products must produce all the 1-bit partial products of the form $A_i \cdot B_j$ (see Figure 59). To do so the inputs protocol for a 4-bit multiplication is the following:

				A ₃	A ₂	A ₁	A ₀	x
				B ₃	B ₂	B ₁	B ₀	=
				A ₃ B ₀	A ₂ B ₀	A ₁ B ₀	A ₀ B ₀	
		A ₃ B ₁	A ₂ B ₁	A ₁ B ₁	A ₀ B ₁	—	—	
	A ₃ B ₂	A ₂ B ₂	A ₁ B ₂	A ₀ B ₂	—	—	—	
A ₃ B ₃	A ₂ B ₃	A ₁ B ₃	A ₀ B ₃	—	—	—	—	
S ₇	S ₆	S ₅	S ₄	S ₃	S ₂	S ₁	S ₀	

Figure 59. Handmade 4-bit multiplication algorithm.

Data A bits are given in the order $\{A_0, A_1, A_2, A_3\}$ for 4 times (N_{bit} times) and then data A is set to '0' until the end of the operation. To generate the partial product properly, each bit of data B must be multiplied with all the data A bits. Therefore the elapsed time to generate all the $A_i \cdot B_i$ products is $16 \cdot T_{clk}$ (in general $N_{bit}^2 \cdot T_{clk}$). In the 4-bit case B is fed in the following order: $\{B_0, B_0, B_0, B_0, B_1, B_1, B_1, B_1, B_2, B_2, B_2, B_2, B_3, B_3, B_3, B_3\}$. After that data B is set to '0' until the end of the operation.

So the Full Adder of the multiplier sums the partial products one bit at a time. It has two feedbacks, one for the result S and one for the carry-out, so that the whole multiplication can be carried out by a single FA module. For a correct alignment of the partial products' sum the carry feedback has to be N_{bit} registers long, while only $N_{bit} - 1$ are required for the result's loop. The multiplier produces one bit of the result every N_{bit} clock cycles, therefore the whole operation takes $2N_{bit}^2 \cdot T_{clk}$, as the result counts $2N$ bits. The result is then forwarded to the adder, but only 1 bit out of N is meaningful.

Notice that the multiplier's feedbacks both demand a control signal. The **Rst-mult** simply resets the carry-in before starting a new operation. The **Ctrl-mult** has instead a more complex function. We said that the output of the FA contains a bit of the final result every $N_{bit} \cdot T_{clk}$, all the other data are intermediate results. For a correct circuit functioning (see the algorithm in Figure 59), the bits of the final result must not be fed back to the FA. **Ctrl-mult** is supposed to mask those bits, setting the feedback to '0'.

6.3.1.2 Adder

The adder sums up the multiplication result to the value in the accumulator starting from the LSB and puts the result back into the accumulator. It also has to keep track of the carry bits. `Rst-adder` resets the carry loop when the LSB of a new result arrives. The other reset signal `Rst-acc` allows to set the accumulator to 0.

6.3.1.3 Accumulator

The accumulator works as a shift registers, its data is always moving. Its length is equal to the duration of a multiplication: $2N_{bit}^2 \cdot T_{clk}$. Because of the circuit functioning, at any instant only $2N$ cells ($1/N$) of the accumulator registers will contain useful data. A lot of space is then wasted by registers (or cells in ME-NML) that for most of the time do not contain meaningful data. The solution we propose to reduce the great impact of the accumulator on the circuit area is to let multiple MAC units share the same accumulator.

6.3.2 Serial MAC with shared Accumulator

The accumulator of the first serial MAC proposed (Figure 58) is too long and costly. Even though the data to be stored is $2N_{bit}$ long, the accumulator has a length of $2N_{bit}^2$ registers. At every instant $2N_{bit} \cdot (N_{bit} - 1)$ register contain meaningless data. This means that ideally the same accumulator could be shared by N different MAC units.

The circuit with the common accumulator was designed, described with the RTL model and simulated with positive results. But afterwards a further optimization came up: Notice that the Adder block can be shared as well, as it processes useful data only once every N_{bit} clock cycles. So here we will present only the latest version.

The final scheme is shown in Figure 60, where eight 8-bit serial MAC units are represented together. They all share the same Accumulator and Adder. The *Mult.* and *Adder* are simplified as boxes, but they refer to the 8-bit circuit in Figure 58.

Below the multiplier blocks there are four rows of shift registers. The top line is meant to carry the results from all the multipliers to the Adder, but only the meaningful values are allowed to enter that shift register. The output of each multiplier has to pass first through a multiplexer which is controlled by `Ctrl_results`. This signal makes sure that the intermediate results of the multiplication will not enter the shift register, given that all the multipliers output the result bits at the same time. The length of 7 registers assures that the results of one MAC do not overwrite those of another one. The signal `Ctrl_results` is set to '1' once every N clock cycles, otherwise it is '0'.

The Adder works as described before, but this time it will be exploited to its best, processing useful data all the time. The MAC result, stored in the accumulator, can be extracted serially at any point of it. If one acquires all the results from the same point of the accumulator, those of the MAC units closer to the Adder will arrive earlier.

Even though the eight MAC units are connected together, one might want to consider them as independent from each other, with their own inputs and output. Also, one might want all of them to have the same latency from inputs to result. In Figure 60 each MAC block has its own arrow that extracts the information in the accumulator. If the results are acquired in that way, every MAC has the same latency. Furthermore, inputs and output of each MAC are spatially separated from other MAC's signals.

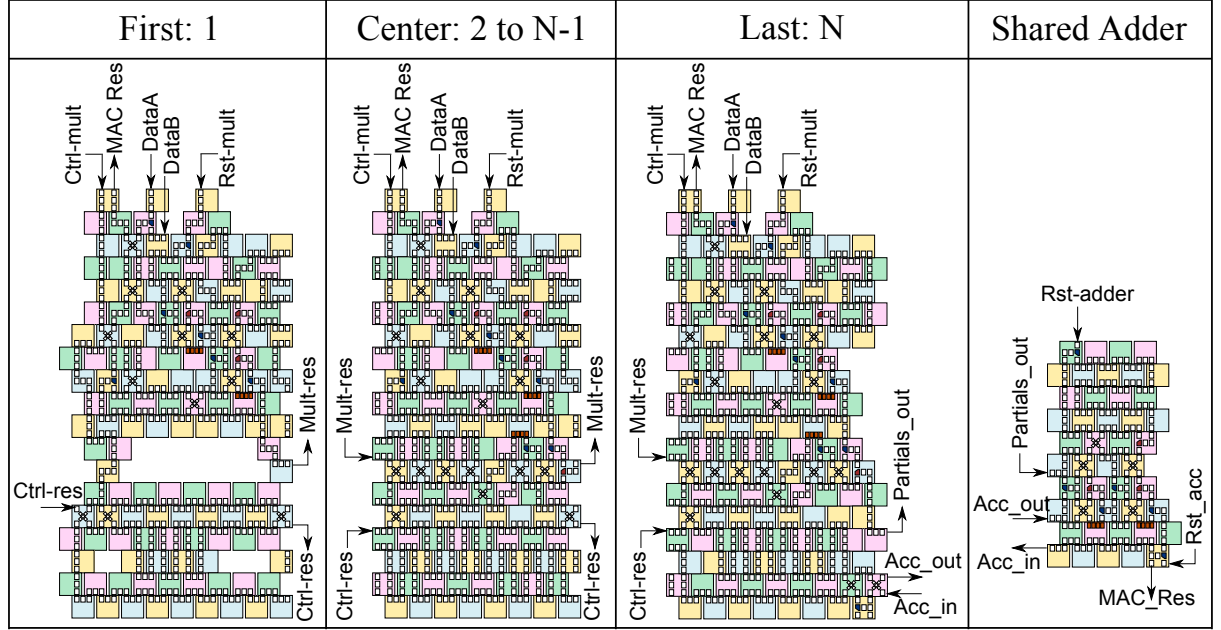


Figure 61. Base blocks of the 8-bit serial MAC with shared Accumulator and Adder.

Changing the parallelism, the MAC requires to be redesigned from scratch, so we only designed and simulated the 8-bit serial MAC. The full 8-bit serial MAC is in Figure 62, it contains 8 different MAC units working in parallel and sharing both Accumulator and Adder. The results are going out from the top of each block, while in Figure 60 they were outputted at the bottom. Anyway both cases have the same timing. As usual this architecture has been described with our RTL model for ME-NML.

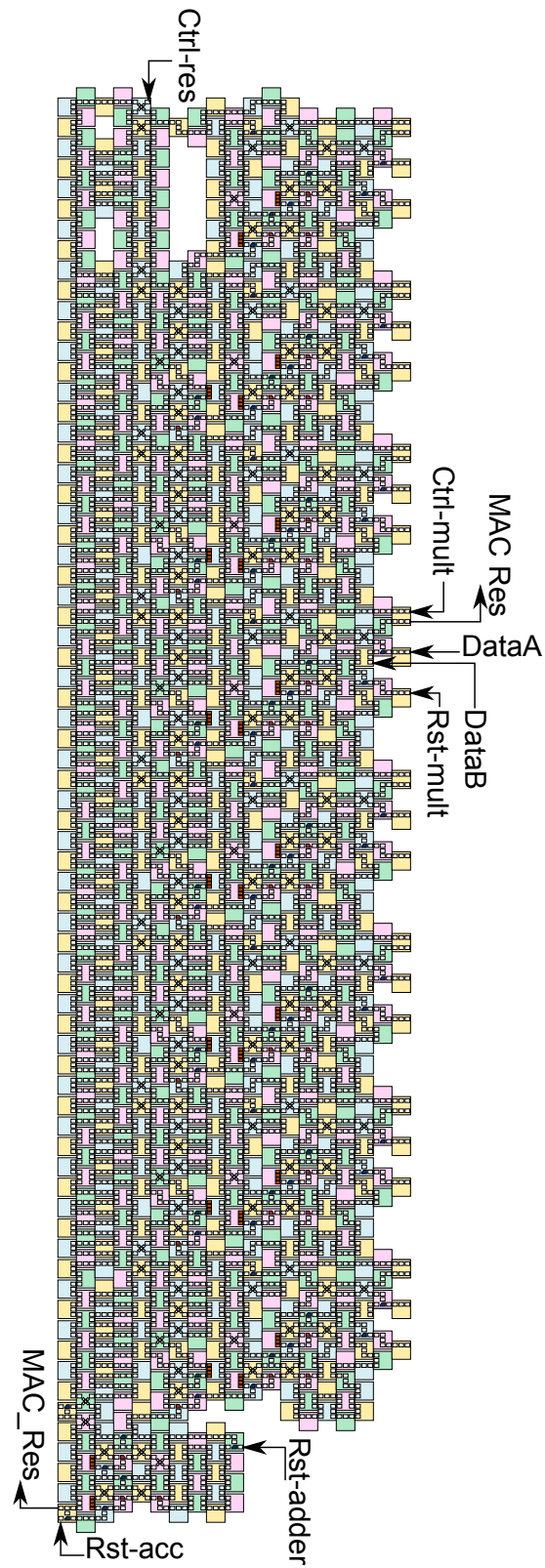


Figure 62. ME-NML implementation of the 8-bit serial MAC with shared Accumulator and Adder.

6.3.4 Timing analysis

Since the Full Adders of the Multiplier processes a continuous flow of data, for this implementation it is not necessary to use the interleaving technique. Table XX contains the main information concerning timing. The throughput is the inverse of the execution time of one operation: $1/(2N^2 \cdot T_{clk})$. Since the proposed circuit requires N MAC to be linked together, the throughput for the entire shared-accumulator serial MAC (8 MACs) is $1/(2N)$.

TABLE XX

TIMING PERFORMANCE OF THE SERIAL MAC			
N bit	Interleaving	Throughput	Latency: LSB of Result out
4	1 op.	$1/(32T_{clk})$	$45T_{clk}$
8	1 op.	$1/(128T_{clk})$	$85T_{clk}$
N	1 op.	$1/(2N^2 \cdot T_{clk})$	$(2N^2 + 9) \cdot T_{clk}$

The latency from the beginning of an operation to when the LSB of the result reaches the output is $2N^2 \cdot T_{clk}$. This value of latency applies if the results are taken as in Figure 62, from the top of each block. In this way all the blocks have the same latency, even if some of them are closer than others to the Adder. However the result could also be acquired much closer to the Adder, so to minimize the latency. For example right after the Adder's output, just like **MAC_Res** in Figure 62 at the bottom-right corner.

CHAPTER 7

CASE STUDY II: MAC RESULT COMPARISON

This chapter presents the performance outcomes for the MAC unit implementations proposed in Chapter 6. Here the three architectures are examined in terms of occupied area and power consumption, while the throughput and latency information have already been exhibited. At last the different MAC versions are placed side by side, offering a rigorous comparison. The results estimation follows the main guidelines adopted for the case study on the Galois Multiplier.

It will be proved the superiority of the parallel MAC over the other two architectures. For a fair comparison of area and power, each implementation should have the same throughput, but that it is not the case. Therefore we combined as many MAC modules as needed to reach a throughput equal to 1. For example since the serial MAC has throughput $1/(2N^2)$, the area and power of a single serial MAC have been multiplied by $(2N^2)$ as if $(2N^2)$ MAC units were working together to achieve a 1/1 throughput.

7.1 Parallel MAC Results

The simulation of the VHDL model for the parallel MAC tells us the number of nanomagnets and cells, the occupied area and the value of the two energy components. The complete set of results has been arranged in Table XXI:

- **Area.** The layout of this circuit, as clear from Figure 48, has many empty internal regions.

So the area evaluated by the model (*Cells area* in the table) is smaller than it should,

TABLE XXI

PARALLEL MAC PERFORMANCE RESULTS.

Parallel MAC		Number of bits			
		4	8	16	32
NUMBER OF CELLS		1507	5913	23413	93165
AREA	Cells area(μm^2)	96	422	1670	6650
	Height (# of cells)	53	105	209	417
	Width (# of cells)	43	87	175	351
	TOT (μm^2)	150	601	2410	9630
	Increase rate	-	4.01	4.00	4.00
POWER	TOT (μW)	9.7	38	150	600
	Increase rate	-	3.92	3.96	3.98

because it only considers the space occupied by cells. The value actually assigned to the parallel MAC is rounded up to the parallelogram circumscribed to the circuit (TOT in the table). To obtain the parallelogram's area we derived a generic equation for evaluating height and width (in terms of cells) for any number of bits.

- **Power.** The model evaluates the two power components, which have been added together to get the total consumption.
- **Increase rate.** The increase rate simply shows the growth of area and power when the number of bits doubles. So the increase rate in the $N_{bit} = 16$ column is the result for $N_{bit} = 16$ divided by the result for $N_{bit} = 8$.

As expected the increase rate is quadratic and regular, because that is how both the Multiplier and the interconnection regions grow. The wasted space because of the empty inner regions has a big effect on the area occupation, while it does not affect the power consumption.

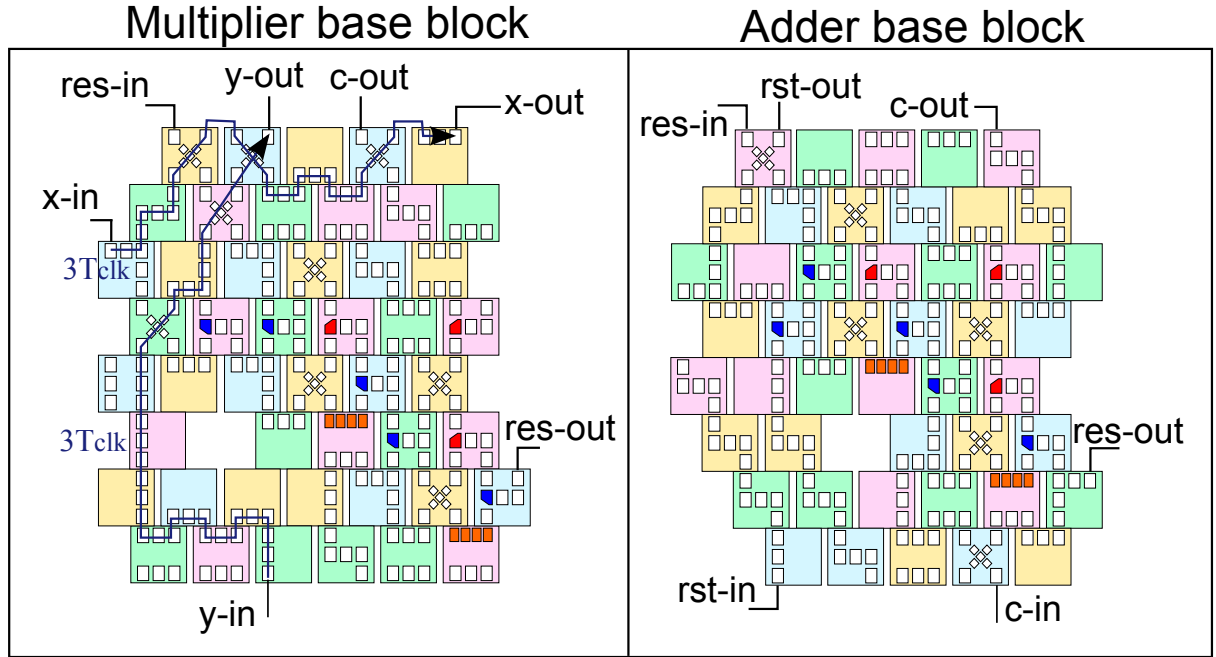


Figure 63. Central blocks of Multiplier and Adder for an optimized version of the Parallel MAC.

Right now we are working on an optimized version that is completely compact. The new circuit was obtained by rearranging the base block of both the multiplier and the adder as in Figure 63. The expected improvements concern area and latency reduction ($3T_{clk}$ both vertically and horizontally). Anyway precise results are not available yet.

7.2 Serial-Parallel MAC Results

Since the Serial-parallel MAC layout is very compact, the area calculated by the VHDL model corresponds to the actual space occupied by the circuit. So increase rates of area and power are pretty much the same as they are both proportional to the number of cells. Actually

TABLE XXII

SERIAL-PARALLEL MAC PERFORMANCE RESULTS.

Serial-parallel MAC		Number of bits				
		4	8	16	32	64
NUMBER OF CELLS	Body	303	623	1263	2543	5103
	Conn. above	98	510	2294	9702	39878
	Conn. below	125	541	2237	9085	36605
	Input cond.	53	121	257	529	1073
	TOT	579	1795	6051	21859	82659
	TOT / Body	1.9	2.9	4.8	8.6	16.2
AREA (μm^2)		41	128	432	1560	5900
POWER (μW)		3.7	12	39	140	530
Increase rate		-	3.10	3.37	3.61	3.78

the switching energy is only proportional to the number of nanomagnets, but for big circuits it is also in some way proportional to the number of cells. Furthermore this component is at least 15 times lower than the clock network losses.

All the results are displayed in Table XXII. Looking at the number of cells of each region, it is possible to see how different parts grow as the number of bits increases. As expected the body and the input conditioning expand linearly, while the interconnection regions grow quadratically. The *TOT/Body* slot gives precise intel on how much area is occupied by input/output preskew/deskew networks.

7.3 Serial MAC Results

Even if only the 8-bit serial MAC has been designed and simulated, it was trivial to obtain a projection of the number of cells for the other parallelisms. What varies with the number of bits are the two feedback loops of the multiplier block (Figure 58), the Accumulator and

TABLE XXIII

SERIAL MAC PERFORMANCE RESULTS.

Serial MAC		Number of bits				
		4	8	16	32	64
NUMBER OF CELLS	Entire shared circuit	510	1138	3514	12106	44650
	Effective MAC	128	142	220	378	698
AREA (μm^2)		9.1	10.2	15.7	27	50
POWER (μW)		0.82	0.92	1.4	2.4	4.5
Increase rate		-	1.12	1.54	1.72	1.84

the shift register that brings the products to the Adder (Figure 60). Also the loop of the adder gains length. In each single MAC block, to obtain the 2N-bit circuit from the N-bit one, the multiplier's loops must get N clock periods longer. The same is true for the segment of the products' shift register and for each of the two segments of the accumulator. From this considerations it was possible to predict with good approximation the growth of the serial MAC with the number of bits.

In Table XXIII the first row of data refers to the whole circuit with shared adder and accumulator. But such circuit contains N MAC units. To get an idea of the weight of a single MAC, the total number of cells has been divided by the number of bits, which is also the number of MAC blocks enclosed by the entire circuit. The result are in the *Effective MAC* row. Area and power are also the effective values for a single MAC, not those for the whole structure containing many MAC units. Their behavior is the same, as usual for compact circuits.

The throughput of the serial MAC decreases quadratically with the number of bits. Therefore ideally, to keep up with the parallel MAC performance, the increase rate of a single MAC should be equal to 1. Unfortunately this is clearly not the case.

7.4 Results Comparison

TABLE XXIV
COMPARISON OF THE 3 MAC IMPLEMENTATIONS, WITH THE THROUGHPUT
BEING EQUAL.

			Number of bits			
			4	8	16	32
With Interleaving	AREA	0D	291	1300	8030	55,300
		1D	331	2050	13,800	99,900
		2D	150	601	2410	6930
		0D/2D	1.94	2.16	3.34	5.74
		1D/2D	2.21	3.41	5.75	10.4
	POWER	0D	26.3	117	724	4990
		1D	29.8	185	1250	9010
		2D	9.71	38.1	151	600
		0D/2D	2.71	3.08	4.8	8.32
		1D/2D	3.07	4.86	8.27	15
No Interleaving	AREA	0D	58	260	1610	11,100
		1D	198	1230	8300	59,000
		2D	150	601	2410	6930
		0D/2D	0.39	0.43	0.67	1.15
		1D/2D	1.32	2.05	3.45	6.22
	POWER	0D	5.3	23.5	145	998
		1D	17.9	111	748	5410
		2D	9.71	38.1	151	600
		0D/2D	0.54	0.61	0.96	1.66
		1D/2D	1.84	2.91	4.96	9.01

The three architectures have been analyzed in terms of throughput, latency (Chapter 6), circuit area and power consumption (Sections 7.1-2-3). Up to now the results of each MAC implementation have been presented singularly, here they are placed side by side. The complete set of data required for the comparison are enclosed in Table XXIV, where the labels $0D$, $1D$, $2D$ refer respectively to Serial MAC, Serial-parallel MAC, Parallel MAC.

7.4.1 Comparison conditions

7.4.1.1 Interleaving

To reach their maximum throughput, both Parallel MAC and Serial-Parallel MAC, necessitate the interleaving technique. The parallel circuit requires a 5 operations interleaving, otherwise its throughput would be $1/5T_{clk}$ and not $1/T_{clk}$. The serial-parallel version requires instead 3 operations only. The Serial MAC does not require any interleaving. The comparison is carried out in two different situations, at first without considering the interleaving possibility, then assuming that the interleaving is exploited to its best.

7.4.1.2 Equal throughput

To obtain a meaningful comparison, area and power performance should be referred to circuits with the same throughput. The output rate of the Parallel MAC has been used as reference for both cases: With and without interleaving. So in Table XXIV the results concerning the parallel MAC are simply those of a single unit. On the other hand the results of the other two implementations have been multiplied by a coefficient, which is the number of units that should work in parallel to reach the same throughput as the Parallel MAC. They have to arrive at a $1/5T_{clk}$ rate without interleaving, and a $1/T_{clk}$ with interleaving.

- **Serial MAC.** Throughput always is $1/(2N^2 \cdot T_{clk})$. $2N^2$ MAC units required to reach $1/T_{clk}$, $2N^2/5$ MAC units required to reach $1/5T_{clk}$.
- **Serial-parallel MAC.** Exploiting interleaving its throughput is $1/(2N \cdot T_{clk})$, so $2N$ units required to reach $1/T_{clk}$. Without using the interleaving technique output rate is $1/(3 \cdot 2N \cdot T_{clk})$. So $3/5 \cdot 2N$ units to arrive at $1/5T_{clk}$.

7.4.2 Results exploiting interleaving

Let's take now a closer look at Table XXIV, starting from the *With interleaving* part. The same results are also depicted in Figure 64 and Figure 65. The 2D implementation is undoubtedly the most efficient, while the 1D (serial-parallel) has the worst outcomes. The *0D/2D* and *1D/2D* rows are meant to give a sharper impression of the comparison. There are two main trends to be noticed.

Firstly the parallel MAC, with respect to the other implementations, is the best both for area and power, but in different ways. The power performance leads on the other architectures definitely more than the area occupation. This fact is to be attributed to the empty regions within the parallel MAC layout (serial and serial-parallel MAC do not have any), which largely increase the area but do not affect the power consumption. Furthermore, the area has been rounded up to the circumscribed parallelogram, including then also some empty space outside of the multiplier. So to say that the actual circuit area is slightly less than what reported.

From the alternative solution we are working on for the Parallel MAC it seems that the problem of free inner regions can be solved. Therefor the power results presented here are more

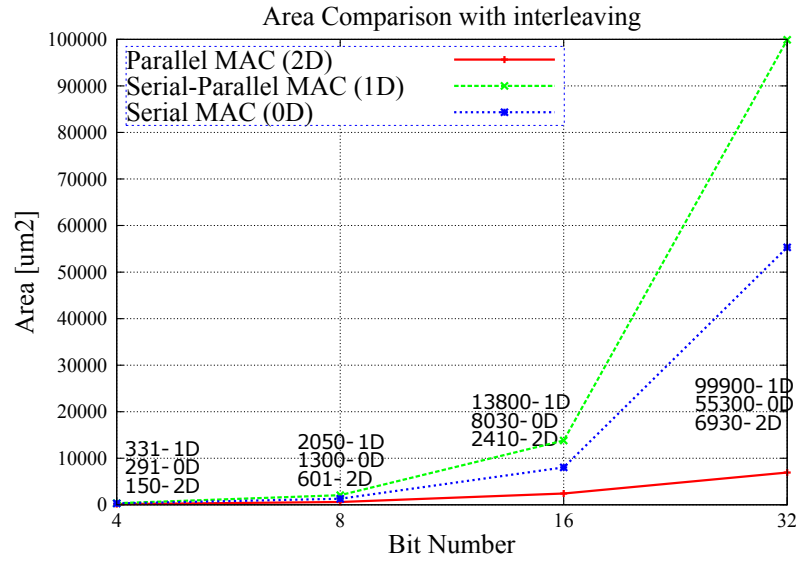


Figure 64. Area comparison of the three MAC implementations exploiting interleaving and with the throughput being equal.

significant than those for the area. Because with a compact layout the area would improve up to the power performance.

Secondly, the results of the three implementations fall apart from each other as the number of bits increases. Notice how for the power the 4-bit case gives $0D/2D = 2.71$, $1D/2D = 3.07$, while the 32-bit case has $0D/2D = 8.32$, $1D/2D = 15$. Ideally how should the three MAC units grow to keep the same relationships among each other together with a constant throughput?

- Parallel MAC: if N_{bit} doubles it grows 4 times and keeps the same throughput. Let's see how the other should behave to reach a 4 times area increase.
- Serial-parallel MAC: if N_{bit} doubles the throughput gets halved, so twice the number of MAC units are required to maintain the same throughput as before. Therefore for the

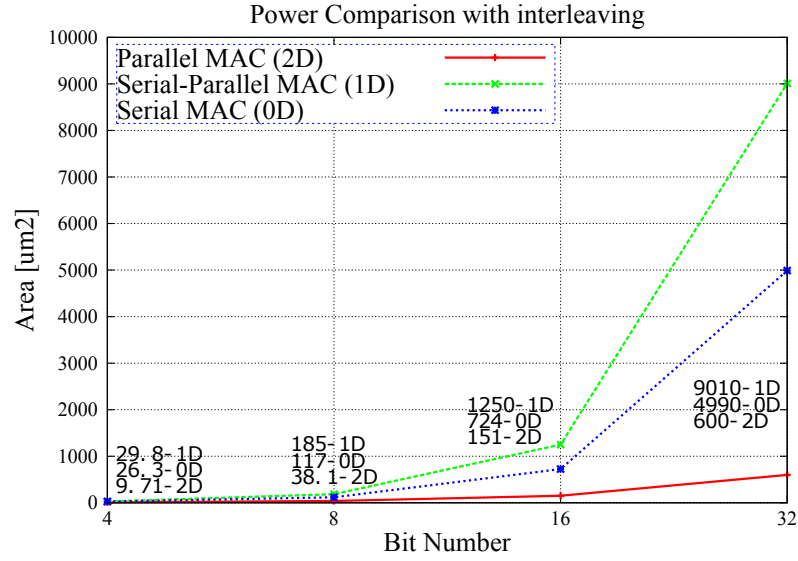


Figure 65. Power comparison of the three MAC implementations exploiting interleaving and with the throughput being equal.

overall area to grow only 4 times, the area of a single MAC should become at most twice as much as before.

- Serial MAC: if N_{bit} doubles the throughput gets 4 times lower, so 4 times the number of MAC units are required to maintain the same throughput as before. Therefore for the overall area to grow only 4 times, the area of a single MAC cannot become any larger. However the serial-parallel MAC grows almost quadratically with the number of bits (section Table XXII), because of the preskew/deskew networks. The parallel MAC also has those kind of interconnections, but their quadratic growth do not affect the MAC performance. It has a 2-D structure, so every part of it grows quadratically. So also the serial MAC do not follow the ideal behavior required to keep up with the parallel

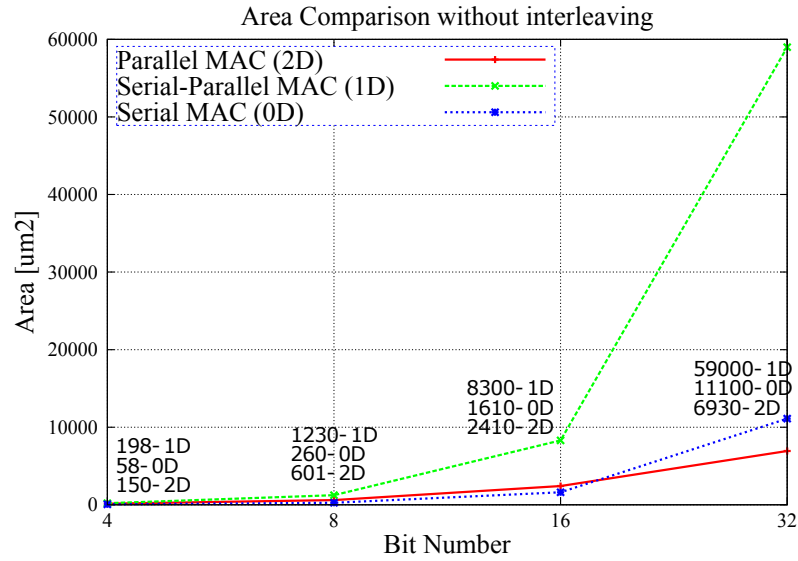


Figure 66. Area comparison of the three MAC implementations without interleaving and with the throughput being equal.

implementation. Its MAC unit increases with the number of bits up to the 545% (64-bits), the reason are the several feedback loops increasing linearly with the number of bits (Table XXIII).

7.4.3 Results without exploiting interleaving

In a situation where the interleaving technique could not be used, the hierarchies among the three MAC versions undergo slight changes. Look again at Table XXIV and also refer to Figure 66 and Figure 67. The performance of the parallel and serial-parallel MAC units worsen respectively of 5 and 3 times, according to their previous interleaving usage. The serial MAC gains a lot in this situation because it cannot exploit interleaving anyway.

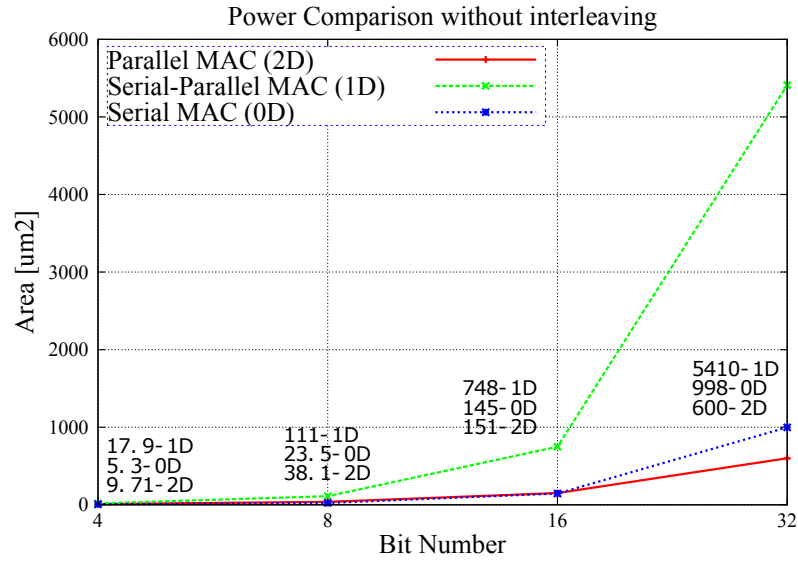


Figure 67. Power comparison of the three MAC implementations without interleaving and with the throughput being equal.

In fact the serial MAC becomes the leading architecture up to a 16 bits parallelism. But since, as explained before, none of the implementations can keep up with the parallel one when the number of bits increases, finally the parallel MAC takes back its lead for 32 or higher number of bits.

7.4.4 Final considerations

The serial MAC can then be a fine alternative when unable to provide to the circuit interleaved operations. The idea of sharing accumulator and adder boosted the performance, but it can also be a setback, as it requires N_{bit} multiple MAC modules to be connected together. It is not possible for one of them to function without the presence of the others. Anyway the parallel MAC has by far the most promising architecture organization. Its strength is twofold:

Advantage over the serial-parallel MAC The circuit's body is a 2D array, just like all the synchronization interconnections networks. So, while extremely bothersome for 1D structures, the additional skew networks do not affect the 2D circuits growth trends. Unfortunately the serial-parallel MAC greatly suffers the input/output conditioning networks. The only way for this circuit to be very competitive would be a system where it could interface itself with other modules without the need for its additional preskew/deskew networks.

Advantage over the serial MAC Long interconnections and feedbacks are the main problem with ME-NML technology. The 2D systolic array organization of the parallel MAC keeps them to the minimum, avoiding for example the long feedback required for serial multiplication, which in the serial MAC also affect the loops of adder and accumulator.

CHAPTER 8

CONCLUSIONS

The Magnetoelastic clock brought a great enhancement to NML technology because it consumes remarkably low power. However only a proper choice of architecture can preserve the benefits of this clock solution. This work is meant to be the starting point for circuit design with MagnetoElastic technology, presenting a series of achievements to deal with ME-NML circuits.

Standard Cell library

First of all the definition of the Standard Cell library, together with the high regularity of circuit layout, will be the foundation for the creation of a design tool, that could greatly improve future research in this field. The design and simulation methodology proposed for this work consists of a hierarchical RTL model, based of the set of Standard Cells. The model contains all the information concerning the physical placement and orientation of cells. Furthermore the embedded capability of exact performance evaluation makes the model an advanced stand-alone tool.

Case Study I

The case study on the generic N-bit Galois Multiplier provided an assessment of the true value of this new technology. Results proved that the MagnetoElastic clock system solves the power losses issue of the clock network for the Magnetic NML, achieving a 50 times reduction of power absorption. Nonetheless its area turns out to be 4 times more compact. However

the most impressive outcome is the advantage over the state-of-the-art CMOS transistors. The comparison has been performed mapping the CMOS circuit with Encounter by Cadence exploiting a low power 28nm FDSOI standard cell library. The ME-NML Galois Multiplier overcomes CMOS transistors of 10 times both for area occupation and power consumption.

Anyway even with these excellent results NML technology is not meant as a replacement for CMOS technology, since its speed is intrinsically limited to 100MHz. ME-NML could be introduced alongside CMOS for some specific applications that can benefit from its high integration, low power and logic-in-memory ability. A radical change of technology would require an enormous cost for retooling an entire industry. To even consider such a change the performance improvements of the new technology must be measured in orders of magnitude.

Apart from the comparison with other technologies this case study also provides other essential structural information, as it is the first design example of a complex circuit. It demonstrates that systolic array architectures are particularly suited for ME-NML design. Furthermore it highlights a huge limitation of serial-parallel circuits, as they require preskew and deskew networks that greatly affect the area occupation. The interconnection overhead gets worse increasing the circuit parallelism. In fact the body of serial parallel circuits grows linearly with number of bits, while the synchronization networks increase their area quadratically. The effect is that half of the 4-bit Galois Multiplier's area is devoted to the additional circuitry, while in the 64-bit Galois Multiplier the synchronization networks are around 14 times bigger than the circuit's body.

Case Study II

Once the potentialities of ME-NML technology have been ascertained, an extensive work has to be carried out to discover which kinds of architectural organization lead to the best performance. To do so the second case study put side by side three main classes of circuits: parallel (2D structure), serial parallel (1D structure), serial (0D structure). The parallel MAC results are overall better than the other two implementations in particular for high parallelisms, while for small circuits the performances are more similar. When it is not possible to exploit interleaving, the serial MAC implementation is able to reach the performance of the parallel MAC and even outdo it for small parallelisms. The reason is that the serial MAC implementation does not require the interleaving technique to function at its best.

Even though this is just one example, after investigating the reasons behind the result we can say that most likely this outcome represents the general trend for ME-NML. The serial-parallel structure requires additional synchronization circuits that strongly affect its performance, just like for the Galois Multiplier. Notice also that even though the same interconnections are required for the parallel MAC, in that case the MAC body grows with the number of bits just like the interconnections. On the other hand, the drawbacks of the fully serial MAC are the feedback loops, that get longer increasing the parallelism of the incoming data.

8.1 Future work

Beside the mere case studies results, there are some relevant side considerations that lead to possible future goals:

- The circuit design was done entirely manually using a graphic design software. Any small adjustment or modification, even if just for interconnections, requires a considerable amount of time. Therefore it is strongly advised to create an *ad-hoc* automated design tool. Researchers at Politecnico di Torino developed one [40] for the Magnetic clock NML, and already started to work on extending it to MagnetoElastic NML.
- The architectural study begun with the second case study should be carried on to reach a complete knowledge over MagnetoElastic circuits, so that this novel technology can be exploited to its best.
- Since ME-NML is intrinsically pipelined, any parallel input or output requires an additional synchronization circuitry to interface other modules. While this issue greatly affect the performance of 1D array structures (e.g. Galois Multiplier, serial-parallel MAC), it does not have too much of an impact on 2D arrays (parallel MAC). The only way to improve this aspect is to aim toward systems where all the modules share the same in-out protocol for parallel signals. We already devoted efforts in this direction with excellent outcomes because in ME-NML circuits the input or output protocols for parallel signal are the same in most of the cases.
- Every circuit with at least a feedback loop requires the interleaving technique to reach its maximum throughput. It would be interesting to design and study the circuitry required for handling input signals.

APPENDIX

VHDL LISTINGS

This chapter contains the main VHDL listings developed during the thesis work. They are organized in four sections:

- RTL model for Magnetoelastic NML technology;
- Case study I: Galois Field Multiplier;
- Case study II: MAC unit;
- Testbench template for ME-NML circuits.

A.1 ME-NML model

A.1.1 Constants package for ME-NML

Listing A.1. ME-NML constants package: `MENML_package`

```
package MENML_package is
3  -- Parallelism -----
  constant N_BIT: integer := 4;
6  -- Time constants -----
  constant CLK_PERIOD: time:= 10 ns; -- Clock period
9  -- MAGNETS, ELECTRODES AND CELLS DIMENSIONS -----
  -- All the values are expressed in [nm] --
12 -- Primary constants
  constant LMAG: natural := 50; -- Nanomagnets length
  constant HMAG: natural := 65; -- Nanomagnets height
15  constant TMAG: natural := 10; -- Nanomagnets thickness
  constant LSEP_MAG: natural := 20; -- Nanomagnets horizontal separation
  constant HSEP_MAG: natural := 20; -- Nanomagnets vertical separation
18  constant NMAG_CROSS: natural := 3; -- # of magnets for the Crosswire cross
  -- Derived constants
  constant LELECTRODE: natural := (LSEP_MAG*2 + LMAG)/3; -- Electrode length
21  constant LSEP_CELL: natural := (LSEP_MAG*2 + LMAG)/3; -- Cells separation
end package;
```

```

24  -- ENERGY EVALUTATION CONSTANTS -----
27  -- Primary constants
constant Kb: real := 13.8065; -- Kb = 1.38065e-23 m2*kg*(s^-2)/K
constant T: real := 300.0; -- Room temperature [K]
-- This value is 10^24 greater. Expressed in [yocto Joule]
30  constant MAG_CONSUMPTION: natural := natural(180.0*Kb*T);

constant VACUUMPERM: real := 8854.0; -- Vacuum permittivity: 8.854e-12 F/m.
33  constant RELPERM: real := 1300.0; -- Substrate relative permittivity
constant T_PZT: real := 4.0; -- Electrodes thickness: 40e-9 m.
constant STRESS: real := 28.0; -- Applied stress: 26e+6 Pa.
36  constant YOUNGMODULUS: real := 8.0; -- Young modulus for Terfenol: 80 GPa.
constant PZT_CONST: real := 15.0; -- Piezoelectric coefficient: 150 pm/V.
-- Derived constants
39  constant Ec_const: natural := natural((VACUUMPERM*RELPERM*T_PZT*(STRESS)**2)/((
    YOUNGMODULUS*PZT_CONST)**2)*(1.0e-6));

-- # of CELLS -----
42  constant N_ZONES_BASE_ELEMENT: natural := 1; -- Useful for counting the total number
    of cells used
constant N_ZONES_PE: integer := 72; -- Max # of cells in a PE: 9*8=72
-- Orientation -----
45  constant LX: std_logic := '0'; -- Left
constant RX: std_logic := '1'; -- Right
constant UP: std_logic := '0';
48  constant DOWN: std_logic := '1';
constant LX_UP: std_logic_vector(1 downto 0) := "00";
constant LX_DOWN: std_logic_vector(1 downto 0) := "01";
51  constant RX_UP: std_logic_vector(1 downto 0) := "10";
constant RX_DOWN: std_logic_vector(1 downto 0) := "11";
-- Phase -----
54  constant A: std_logic_vector(1 downto 0) := "00";
constant B: std_logic_vector(1 downto 0) := "01";
constant C: std_logic_vector(1 downto 0) := "10";
57  constant D: std_logic_vector(1 downto 0) := "11";
-- Dimensions -----
constant ZONE_H: natural := 3;
60  constant ZONE_L: natural := 3;
-- Other constants -----
constant init_natural : natural := 0; -- Initialization natural port

63  end MENML_package;

66  package body MENML_package is
end MENML_package;

```

A.1.2 Area and Energy evaluation

Listing A.2. Area and energy evaluation for a single cell: `area_and_energy`

```

library ieee;
2  use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
5  use work.MENML_package.all;

```

```

8  entity area_and_energy is
    generic (H: natural;           — Height (# of magnets)
             L: natural);         — Width (# of magnets)
    port (   n_mag: in  natural;    — # of magnets
            area_eff: out natural; — Total magnets area
            area_tot: out natural; — Cell area
            Er: out  natural;      — Switching energy
            Ec: out  natural);     — Clock network losses
14  end area_and_energy;

17  architecture behavior of area_and_energy is
    signal L_CELL_EFF: natural:= L*L_MAG + (L-1)*L_SEP_MAG; —Length of the cell
    signal H_CELL: natural:= H*(H_MAG+H_SEP_MAG); —Height of the cell
20  signal H_ELECTRODE: natural:= H*(H_MAG+H_SEP_MAG); —Height of the electrodes
    begin
    — Area evaluation —————
23  AREA_EFF <= n_mag*L_MAG*H_MAG; —Total area occupied by the magnets themselves
    AREA_TOT <= L_CELL_EFF*H_CELL + 2*H_ELECTRODE*L_ELECTRODE + H_CELL*L_SEP_CELL;
    — Energy evaluation —————
26  Er <= n_mag * MAG.CONSUMPTION; — Magnets energy consumption
    Ec <= (L_CELL_EFF/2)*H_CELL*Ec_const; — Clock energy consumption
    end behavior;

```

A.1.3 Standard Cells

Only the VHDL code for one standard cell has been completely enclosed, the `short_wire_horiz`.

All the others are very similar, so the common parts have been deleted.

Listing A.3. Short horizontal wire: `short_wire_horiz`.

```

— CELL ORIENTATIONS —————
2  —
— UP   DOWN  (o = magnet, x = void)
— ooo  xxx
5  — xxx  xxx
— xxx  ooo

8  library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
11  use ieee.std_logic_arith.all;
   use work.MENML_package.all;

14  entity short_wire_horiz is
    generic (PHASE: std_logic_vector(1 downto 0); — Clk phase.
             ROW: natural;                       — Relative cell position (row)
             COLUMN: natural;                    — Relative cell position (col)
             ORIENTATION: std_logic;
             H: natural;                         — Height (# of magnets)
             L: natural);                       — Width (# of magnets)
20  port (   d: in  std_logic;                  — Inputs
            clk: in  std_logic;                — Depends on the phase
            q: out std_logic;                  — Outputs
            n_mag: buffer natural;             — # of magnets
23

```

```

26         n_zones: out natural := 1; — # number of cells
        area_eff: out natural; — Total magnets area
        area_tot: out natural; — Cell area
        Er: out natural; — Switching energy
29        Ec: out natural); — Clock network losses
end short_wire_horiz;

32 architecture behavior of short_wire_horiz is
    — D FlipFlop (1 bit)
    component reg is
35        port(d: in std_logic;
              clk: in std_logic;
              q: out std_logic);
38    end component;
    — Component for evaluating area and energy of a standard cell
    component area_and_energy is
41        generic(H: natural; — Height (# of magnets)
                 L: natural); — Width (# of magnets)
        port( n_mag: in natural; — # of magnets
              area_eff: out natural; — Total magnets area
44              area_tot: out natural; — Cell area
              Er: out natural; — Switching energy
47              Ec: out natural); — Clock network losses
    end component;
begin
50    n_mag <= L; — Evaluate the number of magnets using H and L.

    — This cell is modelled as a register.
53    Wire: reg port map( d => d,
                        clk => clk,
                        q => q);

56    — Area and energy evaluation for this cell.
    Evaluate_area_energy: area_and_energy generic map(H,L)
59    port map(n_mag, area_eff, area_tot, Er, Ec);
end behavior;

```

Listing A.4. Short vertical wire: `short_wire_vert`.

```

— CELL ORIENTATIONS —
—
3 — LX  RX  (o = magnet, x = void, a = and magnet, r = or magnet)
— oxx  xxo
— oxx  xxo
6 — oxx  xxo

begin
9    n_mag <= H; — Evaluate the number of magnets using H and L.
    — This cell is modelled as a register.
    Wire: reg port map( d => d,
12                      clk => clk,
                      q => q);

    [...]
15 end behavior;

```

Listing A.5. Long diagonal wire: `long_wire`.

```

3  -- CELL ORIENTATIONS -----
--
-- LX   RX   (o = magnet, x = void, a = and magnet, r = or magnet)
-- oxx   xxo
-- ooo   ooo
6  -- xxo   oxx
--
begin
9  n_mag <= H+L-1; -- Evaluate the number of magnets using H and L.
-- This cell is modelled as a register.
Wire: reg port map( d => d,
12      clk => clk,
      q => q);
[...]
```

```

15 end behavior;
```

Listing A.6. Angle wire with 2 outputs: `wire_2outputs`.

```

-- CELL ORIENTATIONS -----
--
-- (o = magnet, x = void, a = and magnet, r = or magnet)
3  -- LX_UP  LX_DOWN  RX_UP  RX_DOWN
-- ooo      oxx      ooo      xxo
-- oxx      oxx      xxo      xxo
6  -- oxx      ooo      xxo      ooo
--
begin
9  n_mag <= H+L-1; -- Evaluate the number of magnets using H and L.
q1 <= q;
q2 <= q;
12  -- This cell is modelled as a register.
Wire: reg port map( d => d,
      clk => clk,
15      q => q);
[...]
```

```

end behavior;
```

Listing A.7. Double horizontal wire: `double_wire_horiz`.

```

1  -- CELL ORIENTATIONS -----
--
-- ooo (o = magnet, x = void, a = and magnet, r = or magnet)
4  -- xxx
-- ooo
--
7  begin
n_mag <= L*2; -- Evaluate the number of magnets using H and L.
-- This cell is modelled as 2 registers.
10 Wire1: reg port map(d => d1,
      clk => clk,
      q => q1);
13 Wire2: reg port map(d => d2,
      clk => clk,
      q => q2);
16 [...]
```

```

end behavior;
```

Listing A.8. Double vertical wire: `double_wire_vert.`

```

1  -- CELL ORIENTATIONS -----
   --
   --  o x o   (o = magnet, x = void, a = and magnet, r = or magnet)
4  --  o x o
   --  o x o
   -----
7  begin
   n_mag <= H*2; -- Evaluate the number of magnets using H and L.
   -- This cell is modelled as 2 registers.
10 Wire1: reg port map(d => d1,
                      clk => clk,
                      q => q1);
13 Wire2: reg port map(d => d2,
                      clk => clk,
                      q => q2);
16 [...]
end behavior;

```

Listing A.9. Crosswire: `crosswire.`

```

1  -- CELL ORIENTATIONS -----
   --
   --  o x o   (o = magnet, x = void, a = and magnet, r = or magnet)
4  --  x o x
   --  o x o
   -----
7  begin
   n_mag <= N_MAG_CROSS + (H-1)*2 + (L-3)*2; -- Evaluate the number of magnets using H
   and L.
   -- This cell is modelled as 2 registers.
10 Wire12: reg port map(d => d1,
                       clk => clk,
                       q => q2);
13 Wire21: reg port map(d => d2,
                       clk => clk,
                       q => q1);
16 [...]
end behavior;

```

Listing A.10. Inverter (always horizontal): `inv_horiz.`

```

1  -- CELL ORIENTATIONS -----
   --
   --  UP   DOWN   (o = magnet, x = void, a = and magnet, r = or magnet)
4  --  o o o o   x x x
   --  x x x     x x x
   --  x x x     o o o o
   -----
7  begin
   n_mag <= L+1; -- Evaluate the number of magnets using H and L.
   q_n <= not q; -- Logic function
   -- This cell is modelled as a register plus its logic function.
10 Wire: reg port map( d => d,
                      clk => clk,
                      q => q);
13 [...]

```



```
16 end behavior;
```

Listing A.11. Parallel inverter and wire: `inv_with_wire_horiz.`

```

2  — CELL ORIENTATIONS —————
—
— UP      DOWN  (o = magnet,x = void , a = and magnet , r = or magnet)
— oooo    ooo
5  — xxx    xxx
— ooo     oooo

8  begin
  n_mag <= L*2+1; — Evaluate the number of magnets using H and L.
  q1_n <= not q1; — Logic function
11 — This cell is modelled as 2 registers plus its logic function.
  Wire1: reg port map(d => d1,
                      clk => clk,
14                      q => q1);
  Wire2: reg port map(d => d2,
                      clk => clk,
17                      q => q2);
  [...]
end behavior;
```

Listing A.12. Double inverter: `double_inv_horiz.`

```

2  — CELL ORIENTATIONS —————
—
— oooo    (o = magnet,x = void , a = and magnet , r = or magnet)
— xxx
5  — oooo

8  begin
  n_mag <= L*2+2; — Evaluate the number of magnets using H and L.
  q1_n <= not q1; — Logic function
  q2_n <= not q2; — Logic function
11 — This cell is modelled as 2 registers plus its logic function.
  Wire1: reg port map(d => d1,
                      clk => clk,
14                      q => q1);
  Wire2: reg port map(d => d2,
                      clk => clk,
17                      q => q2);
  [...]
end behavior;
```

Listing A.13. AND on the left with output going up: `and_wire_lx_up.`

```

2  — CELL ORIENTATIONS —————
—
— oxo    oxx  (o = magnet,x = void , a = and magnet , r = or magnet)
— aoo    aoo
5  — oxx    oxo

8  begin
  n_mag <= H + (H-1)/2 + L - 1; — Evaluate the number of magnets using H and L.
```

```

and_res <= d1 and d2; -- Logic function
-- This cell is modelled as a register plus its logic function.
11 Wire: reg port map( d => and_res ,
                    clk => clk ,
                    q => q );
14 [...]
end behavior;

```

Listing A.14. AND on the left with 2 outputs: `and_2outputs_lx`.

```

-- CELL ORIENTATIONS -----
--
3 --   oxo   (o = magnet,x = void , a = and magnet , r = or magnet)
--   aoo
--   oxo
6 -----
begin
  n_mag <= H*2 + L - 2; -- Evaluate the number of magnets using H and L.
  and_res <= d1 and d2; -- Logic function
9   q1 <= q;
  q2 <= q;
12 -- This cell is modelled as a register plus its logic function.
  Wire: reg port map( d => and_res ,
                    clk => clk ,
15                    q => q );
  [...]
end behavior;

```

Listing A.15. OR on the left with output going up: `and_wire_lx_up`.

```

1 -- CELL ORIENTATIONS -----
--
--   oxo   oxx (o = magnet,x = void , a = and magnet , r = or magnet)
4 --   roo   roo
--   oxx   oxo
7 -----
begin
  n_mag <= H + (H-1)/2 + L - 1; -- Evaluate the number of magnets using H and L.
  or_res <= d1 or d2; -- Logic function
10 -- This cell is modelled as a register plus its logic function.
  Wire: reg port map( d => or_res ,
                    clk => clk ,
13                    q => q );
  [...]
end behavior;

```

Listing A.16. OR on the left with 2 outputs: `or_2outputs_lx`.

```

-- CELL ORIENTATIONS -----
--
3 --   oxo   (o = magnet,x = void , a = and magnet , r = or magnet)
--   roo
--   oxo
6 -----
begin
  n_mag <= H*2 + L - 2; -- Evaluate the number of magnets using H and L.

```

```

9   or_res <= d1 or d2; — Logic function
   q1 <= q;
   q2 <= q;
12  — This cell is modelled as a register plus its logic function.
   Wire: reg port map( d => or_res ,
                      clk => clk ,
15                      q => q);
   [...]
end behavior;

```

A.2 Galois Multiplier (GFM)

Both the CMOS and the ME-NML versions of the GFM have been described with VHDL and enclosed in this section.

A.2.1 CMOS

For what concerns the CMOS GFM, the listings of the top entity `GFM_synch` and the basic blocks component `Basic_Block` are reported below. All the basic blocks are defined within the same entity.

Listing A.17. Top entity of the CMOS GFM (includes synch networks): `GFM_synch`.

```

1  library IEEE;
   use IEEE.Std_Logic_1164.all;

4  entity GFM_synch is
     generic (N_BIT : integer:=8);
     port (clk,rst,mrbit: in std_logic;
          p_synch,b_synch: in std_logic_vector (N_BIT-1 downto 0);
          r_synch: out std_logic_vector (N_BIT-1 downto 0));
10 end GFM_synch;

10 architecture struct of GFM_synch is
     component Basic_Block is
13         port (clk, rst : in std_logic;
              in_b, in_p, mrbit, in_reg, en_p: in std_logic;
              out_result,out_PE : out std_logic);
16     end component;
     component reg is
         port (clk : in STD_LOGIC;
              rst : in STD_LOGIC;
              reg_in : in STD_LOGIC;
              reg_out : out STD_LOGIC);
19     end component;
22     signal mrbit_vect: std_logic_vector(N_BIT downto 0);
     signal en_p_vect : std_logic_vector(N_BIT downto 0);
25     signal parallel_data: std_logic_vector(N_BIT downto 0);

```

```

signal p,b,r: std_logic_vector (N_BIT-1 downto 0);
begin
28 parallel_data(0) <= '0';
   en_p_vect(N_BIT) <= parallel_data(N_BIT);
   mrbit_vect(N_BIT) <= mrbit;
31 mrbit_vect(N_BIT-1) <= mrbit_vect(N_BIT);

Mult_gen: for I in (N_BIT-1) downto 0 generate
34 begin
   last_Basic_Block: if I=(N_BIT-1) generate
       signal p_synch_prop, b_synch_prop, r_synch_prop: std_logic_vector(N_BIT downto 0);
37
       begin
           pe: Basic_Block
           port map
40 (clk=>clk, rst=>rst,
            in_b => b(I), in_p=> p(I), mrbit=> mrbit_vect(I),
43 in_reg => parallel_data(I), en_p => en_p_vect(I),
            out_result => r(I),
            out_PE => parallel_data(I+1));
46
           reg2_pipe_enp: reg port map(clk => clk, rst => rst, reg_in => en_p_vect(I+1),
reg_out => en_p_vect(I));
           p(I) <= p_synch_prop(0);
49 b(I) <= b_synch_prop(0);
           r_synch_prop(0) <= r(I);
           p_synch_prop(N_BIT-1-I) <= p_synch(I);
52 b_synch_prop(N_BIT-1-I) <= b_synch(I);
           r_synch(I) <= r_synch_prop(I);

           res_synch: for j in 1 to I generate
           begin
               reg_synch_r: reg port map(clk=>clk, rst=>rst, reg_in => r_synch_prop(j-1),
reg_out => r_synch_prop(j));
58 end generate;
       end generate; -- end if I= (N_BIT-1)

61 center_Basic_Block: if I>0 and I<N_BIT-1 generate
       signal p_synch_prop, b_synch_prop, r_synch_prop: std_logic_vector(N_BIT downto
0);
       begin
           pe: Basic_Block
           port map
64 (clk=>clk, rst=>rst,
            in_b => b(I), in_p=> p(I), mrbit=> mrbit_vect(I),
67 in_reg => parallel_data(I), en_p => en_p_vect(I),
            out_result => r(I),
70 out_PE => parallel_data(I+1));

           reg1_pipe_mrbit: reg port map(clk =>clk, rst=>rst, reg_in=>mrbit_vect(I+1), reg_out=>
mrbit_vect(I));
73 reg2_pipe_enp: reg port map(clk => clk, rst => rst, reg_in => en_p_vect(I+1),
reg_out => en_p_vect(I));
           p(I) <= p_synch_prop(0);
           b(I) <= b_synch_prop(0);
76 r_synch_prop(0) <= r(I);
           p_synch_prop(N_BIT-1-I) <= p_synch(I);
           b_synch_prop(N_BIT-1-I) <= b_synch(I);
79 r_synch(I) <= r_synch_prop(I);

```

```

82     res_synch: for j in 1 to I generate
      begin
        reg_synch_r: reg port map(clk=>clk, rst=>rst, reg_in => r_synch_prop(j-1),
reg_out => r_synch_prop(j));
      end generate;
85
      sig_synch: for j in 1 to N_BIT-1-I generate
        begin
88          reg_synch_p: reg port map(clk=>clk, rst=>rst, reg_in => p_synch_prop(j), reg_out
=> p_synch_prop(j-1));
          reg_synch_b: reg port map(clk=>clk, rst=>rst, reg_in => b_synch_prop(j), reg_out
=> b_synch_prop(j-1));
          end generate;
91      end generate; — End if I = medium

      Basic_Block_first: if I=0 generate
94          signal p_synch_prop, b_synch_prop, r_synch_prop: std_logic_vector(N_BIT downto
0);
          begin
            pe: Basic_Block
97              port map
                (clk=>clk, rst=>rst,
100                in_b => b(I), in_p=> p(I), mrbit=> mrbit_vect(I),
                in_reg => parallel_data(I), en_p => en_p_vect(I),
                out_result => r(I),
                out_PE => parallel_data(I+1));
103
            reg1_pipe_mrbit: reg port map(clk=>clk, rst=>rst, reg_in => mrbit_vect(I+1),
reg_out => mrbit_vect(I));
            reg2_pipe_enp: reg port map(clk=>clk, rst=>rst, reg_in => en_p_vect(I+1), reg_out
=> en_p_vect(I));
106            p(I) <= p_synch_prop(0);
            b(I) <= b_synch_prop(0);
            r_synch_prop(0) <= r(I);
109            p_synch_prop(N_BIT-1-I) <= p_synch(I);
            b_synch_prop(N_BIT-1-I) <= b_synch(I);
            r_synch(I) <= r_synch_prop(I);
112
            sig_synch: for j in 1 to N_BIT-1-I generate
              begin
115                reg_synch_p: reg port map(clk=>clk, rst=>rst, reg_in => p_synch_prop(j), reg_out
=> p_synch_prop(j-1));
                reg_synch_b: reg port map(clk=>clk, rst=>rst, reg_in => b_synch_prop(j), reg_out
=> b_synch_prop(j-1));
                end generate;
118            end generate; — end if I = 0
          end generate; — end for I
        end struct;

```

Listing A.18. Basic blocks of the CMOS GFM: Basic_Block.

```

library IEEE;
2 use IEEE.Std_Logic_1164.all;
  use IEEE.math_real.all;
  use ieee.numeric_std.all;
5
entity Basic_Block is

```

```

      port (clk, rst : in std_logic;
8         in_b, in_p, mrbit, in_reg, en_p: in std_logic;
          out_result: out std_logic;
          out_PE: out std_logic);
11 end Basic_Block;

architecture struct of Basic_Block is
14   component reg is
      port (clk, rst, reg_in: in STD_LOGIC;
          reg_out : out STD_LOGIC);
17   end component;
   component and_and is
      port ( in_0, in_1 : in std_logic;
20         out_and : out std_logic) ;
   end component;
   component xor3in is
23   port( in_prod, in_reg, in_sub: in std_logic;
          out_xor : out std_logic);
   end component;

26   signal b_to_in_0 : std_logic;
   signal mrbit_to_in_1 : std_logic;
29   signal out_and_to_in_prod : std_logic;
   signal out_p_to_in_sub : std_logic;
   signal out_r : std_logic;
32   signal out_sub_to_in_xor : std_logic;
   signal xor_to_r : std_logic;

35 begin

   B: reg port map (clk => clk ,
38         rst => rst ,
          reg_in => in_b ,
          reg_out => b_to_in_0);
41   R: reg port map (clk => clk ,
          rst => rst ,
          reg_in => out_r ,
          reg_out => out_result);
44   P: reg port map (clk => clk ,
          rst => rst ,
          reg_in => in_p ,
          reg_out => out_p_to_in_sub);
47   PE_NEXT: reg port map (clk => clk ,
          rst => rst ,
          reg_in => out_r ,
          reg_out => out_PE);
50   ANDB: and_and port map( in_0 => b_to_in_0 ,
          in_1 => mrbit ,
          out_and => out_and_to_in_prod);
53   ANDP: and_and port map (in_0 => out_p_to_in_sub ,
          in_1 => en_p ,
          out_and => out_sub_to_in_xor);
56   XOR1: xor3in port map (in_prod=> out_and_to_in_prod ,
          in_reg=> in_reg ,
          in_sub=> out_sub_to_in_xor ,
          out_xor=> out_r);
59   end struct;
62

```

A.2.2 Magnetoelastic NML (ME-NML)

Here are reported the listings of the top entity `Galois_Multiplier` and the the basic blocks component `Basic_Block`. All the basic blocks are defined within the same entity.

Listing A.19. Top entity of the ME-NML GFM: `Galois_Multiplier`.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML_package.all;

6
entity Galois_Multiplier is
  port(B_in,P,reset: in std_logic_vector(N_BIT-1 downto 0);
9     reset2,A_in: in std_logic;
    Res: out std_logic_vector(N_BIT-1 downto 0);
    clkA, clkB, clkC, clkD: in std_logic;
12    n_mag: out natural := init_natural;
    n_zones: out natural := init_natural;
    AREA_EFF: out natural;
15    AREA_TOT: out natural;
    Er: out natural;
    Ec: out natural);
18 end Galois_Multiplier;

architecture behavior of Galois_Multiplier is
21   component Basic_block is
    [...]
  end component;
24   signal A_vect,en_p_vect, results_vect: std_logic_vector(N_BIT downto 0);
  -- Vectors of natural for magnets and cell count, area and energy evaluation
  type natural_vector is array (natural range <>) of natural;
27   signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
    natural_vector (N_BIT downto 1) := (others => init_natural);

begin
30   -- SUM OF ARRAYS OF NATURAL ELEMENTS -----
  -- This process sums up the values of n_mag,n_zones,area_eff,area_tot,Er,Ec of every
  -- PE instantiated.
  -- Results are given as outputs of this "Galois_Multiplier" component.
33   N_mag_sum: process (n_mag_vect,n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
    Ec_vect)
    variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
    natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
    variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
    := init_natural;
36   variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
    sum_tot_Er, sum_tot_Ec: natural := init_natural;
  begin
    n_nat_mag := n_mag_vect;
    n_nat_zones := n_zones_vect;
39    nat_area_eff := area_eff_vect;
    nat_area_tot := area_tot_vect;
    nat_Er := Er_vect;
42    nat_Ec := Ec_vect;

```

```

45  sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
    sum_Ec:= 0;
    for i in 0 to n_mag_vect'length -1 loop
48      sum_n_mag := sum_n_mag + n_nat_mag(i);
      sum_n_zones := sum_n_zones + n_nat_zones(i);
      sum_area_eff := sum_area_eff + nat_area_eff(i);
      sum_area_tot := sum_area_tot + nat_area_tot(i);
51      sum_Er := sum_Er + nat_Er(i);
      sum_Ec := sum_Ec + nat_Ec(i);
    end loop;
54  sum_tot_n_mag := sum_n_mag * INTERCONNECT.OVERHEAD;
    sum_tot_n_zones := sum_n_zones * INTERCONNECT.OVERHEAD;
    sum_tot_area_eff := sum_area_eff * INTERCONNECT.OVERHEAD;
57  sum_tot_area_tot := sum_area_tot * INTERCONNECT.OVERHEAD;
    sum_tot_Er := sum_Er * INTERCONNECT.OVERHEAD;
    sum_tot_Ec := sum_Ec * INTERCONNECT.OVERHEAD;

60  n_mag <= sum_tot_n_mag;
    n_zones <= sum_tot_n_zones;
63  area_eff <= sum_tot_area_eff;
    area_tot <= sum_tot_area_tot;
    Er <= sum_tot_Er;
66  Ec <= sum_tot_Ec;
end process;

69  results_vect(0) <= '0';
    en_p_vect(N_BIT) <= results_vect(N_BIT);
    A_vect(N_BIT)<=A_in;

72  -- The Multiplier is obtained assembling as many PEs as the N_BIT. There are only 3
    different PE,
    -- beside the first one and the last one all the others are the same.
75  Mult_gen:
    for i in 0 to N_BIT-1 generate
    begin
78      PE: Basic_block
          generic map(ELEMENT=>i)
          port map(
81          A_in => A_vect(i+1),
            b_in=>B_in(i),
            p=>P(i),
84          en_p_in => en_p_vect(i+1),
            PE_in => results_vect(i),
            A_out => A_vect(i),
87          en_p_out => en_p_vect(i),
            res => Res(i),
            PE_out => results_vect(i+1),
90          reset=> reset(i),reset2=>reset2, clk=>clk, clkA=>clkA, clkB=>clkB, clkC=>clkC,
            clkD=>clkD,
            n_mag=>n_mag_vect(i+1),
            n_zones=>n_zones_vect(i+1),
93          AREA_EFF=>area_eff_vect(i+1),
            AREA_TOT=>area_tot_vect(i+1),
            Er=>Er_vect(i+1),
96          Ec=>Ec_vect(i+1)
          );
    end generate;
99

```



```
end behavior;
```

Listing A.20. Basic blocks of the ME-NML GFM: Basic_Block.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML-package.all;

6
entity Basic_block is
  generic (ELEMENT: integer); -- Identifies one among N.BIT processing elements
9  port (
    A_in, b_in, p, en_p_in, PE_in: in std_logic;
    A_out, en_p_out, res, PE_out: out std_logic;
12    reset, reset2, clkA, clkB, clkC, clkD: in std_logic;
    n_mag: out natural := init_natural;
    n_zones: out natural := init_natural;
15    AREA_EFF: out natural;
    AREA_TOT: out natural;
    Er: out natural;
18    Ec: out natural);
end Basic_block;

21 architecture behavior of Basic_block is
  [...] -- Define as components all the standard cells.

24  -- Vectors of natural for magnets and cell count, area and energy evaluation
  type natural_vector is array (natural range <>) of natural;
  signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
    natural_vector (N_ZONES_PE downto 1) := (others => init_natural);

27  -- Connections among cells
  signal A1_2d, C1_3d, A1_4d, A1_6d: std_logic;
  signal B2_1d, B2_1d2, D2_2d, B2_3d, B2_3d2, D2_4d, B2_5d, D2_6d: std_logic;
30  signal C3_2d, A3_3d, A3_3d2, C3_4d, C3_5d, A3_6d, C3_7d2, D4_2d, D4_2d2: std_logic;
  signal B4_3d, B4_3d2, D4_4d, D4_4d2, B4_5d, D4_6d, D4_6d2, B4_7d: std_logic;
  signal C5_2d, A5_3d, A5_3d2, C5_4d, C5_4d2: std_logic;
33  signal A5_5d2, C5_6d, C5_6d2, A5_7d, A5_7d2: std_logic;
  signal D6_2d, B6_3d, D6_4d, D6_4d2, B6_5d, B6_5d2, D6_6d, D6_6d2: std_logic;
  signal C7_2d, C7_3d, C7_4d, A7_5d, A7_5d2, C7_6d, C7_7d: std_logic;
36  signal B8_1d, B8_1d2, D8_2d, B8_3d, B8_3d2, D8_4d, B8_5d, B8_5d2, D8_6d: std_logic;
  signal A9_2d, A9_4d, C9_5d, A9_6d: std_logic;
  signal zero_in: std_logic; -- connected to PE_in of the first PE. Then set to '0'.
39  -- Signals added for Last PE for the cells in positions:
    15, 17, 18, 27, 28, 38, 58, 67, 87, 97.
  signal C1_5d, C1_7d, A1_8d, B2_5d2, D2_6d2, B2_7d, D2_8d, A3_5d: std_logic;
  signal C3_6d, A3_7d, C3_8d, B4_7d2, C5_8d, D6_7d, A7_7d, B8_7d, C9_7d: std_logic;
42
begin
  zero_in <= '0';

45  -- SUM OF ARRAYS OF NATURAL ELEMENTS
  -- This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec of every
  -- standard cell instantiated.
48  -- Results are given as outputs of this "Basic_block" component.
  N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
    Ec_vect)

```

```

variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
51 variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
:= init_natural;
variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
sum_tot_Er, sum_tot_Ec: natural := init_natural;
begin
54 n_nat_mag := n_mag_vect;
n_nat_zones := n_zones_vect;
nat_area_eff := area_eff_vect;
57 nat_area_tot := area_tot_vect;
nat_Er := Er_vect;
nat_Ec := Ec_vect;

60 sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
sum_Ec:= 0;
for i in 0 to n_mag_vect'length -1 loop
63 sum_n_mag := sum_n_mag + n_nat_mag(i);
sum_n_zones := sum_n_zones + n_nat_zones(i);
sum_area_eff := sum_area_eff + nat_area_eff(i);
66 sum_area_tot := sum_area_tot + nat_area_tot(i);
sum_Er := sum_Er + nat_Er(i);
sum_Ec := sum_Ec + nat_Ec(i);
69 end loop;
sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
72 sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
75 sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

n_mag <= sum_tot_n_mag;
78 n_zones <= sum_tot_n_zones;
area_eff <= sum_tot_area_eff;
area_tot <= sum_tot_area_tot;
81 Er <= sum_tot_Er;
Ec <= sum_tot_Ec;
end process;

84
-- STANDARD CELLS INSTANTIATIONS -----
-- Labels have the following form: (letter)(number)_(number)
87 -- The letter is the clock phase, first number the relative row within the PE,
-- second number the relative column within the PE.
-- In order to assign signals *d or *d2 to a cell follow this sequence:
90 -- up_left, up_right, down_left, down_right.
C1.3: short_wire_horiz generic map(C,1,3,DOWN,ZONE_H,ZONE_L)
port map(C1.3d,clkC,D2.2d,n_mag_vect(3),n_zones_vect(3),area_eff_vect(3),
area_tot_vect(3),Er_vect(3),Ec_vect(3));
93 B2.3: double_wire_horiz generic map(D,2,3,ZONE_H,ZONE_L)
port map(B2.3d,B2.3d2,clkB,C1.3d,C3.4d,n_mag_vect(11),n_zones_vect(11),
area_eff_vect(11),area_tot_vect(11),Er_vect(11),Ec_vect(11));
C3.2: long_wire generic map(C,3,2,LX,ZONE_H,ZONE_L)
96 port map(C3.2d,clkC,D4.2d,n_mag_vect(18),n_zones_vect(18),area_eff_vect(18),
area_tot_vect(18),Er_vect(18),Ec_vect(18));
A3.3: and_2outputs_lx generic map(A,3,3,ZONE_H,ZONE_L)
port map(A3.3d,A3.3d2,clkA,B2.3d2,B4.3d,n_mag_vect(19),n_zones_vect(19),
area_eff_vect(19),area_tot_vect(19),Er_vect(19),Ec_vect(19));
99 C3.4: long_wire generic map(C,3,4,LX,ZONE_H,ZONE_L)

```

```

    port map(C3_4d, clkC, D4_4d, n_mag_vect(20), n_zones_vect(20), area_eff_vect(20),
    area_tot_vect(20), Er_vect(20), Ec_vect(20));
102 D4.4: or_wire_lx generic map(D,4,4,DOWN,ZONE_H,ZONE_L)
    port map(D4_4d, D4_4d2, clkD, A5_5d, n_mag_vect(28), n_zones_vect(28), area_eff_vect(28)
    , area_tot_vect(28), Er_vect(28), Ec_vect(28));
B4.5: short_wire_horiz generic map(B,4,5,DOWN,ZONE_H,ZONE_L)
    port map(B4_5d, clkB, C5_6d, n_mag_vect(29), n_zones_vect(29), area_eff_vect(29),
    area_tot_vect(29), Er_vect(29), Ec_vect(29));
105 C5.4: crosswire generic map(C,5,4,ZONE_H,ZONE_L)
    port map(C5_4d, C5_4d2, clkC, D4_4d2, D6_4d, n_mag_vect(36), n_zones_vect(36),
    area_eff_vect(36), area_tot_vect(36), Er_vect(36), Ec_vect(36));
A5.5: and_2outputs_lx generic map(A,5,5,ZONE_H,ZONE_L)
108 port map(A5_5d, A5_5d2, clkA, B4_5d, B6_5d, n_mag_vect(37), n_zones_vect(37),
    area_eff_vect(37), area_tot_vect(37), Er_vect(37), Ec_vect(37));
C5.6: crosswire generic map(C,5,6,ZONE_H,ZONE_L)
    port map(C5_6d, C5_6d2, clkC, D4_6d, D6_6d, n_mag_vect(38), n_zones_vect(38),
    area_eff_vect(38), area_tot_vect(38), Er_vect(38), Ec_vect(38));
111 D6.4: inv_with_wire_horiz generic map(D,6,4,UP,ZONE_H,ZONE_L)
    port map(D6_4d, D6_4d2, clkD, A5_5d2, A7_5d, n_mag_vect(44), n_zones_vect(44),
    area_eff_vect(44), area_tot_vect(44), Er_vect(44), Ec_vect(17));
B6.5: and_wire_lx generic map(B,6,5,UP,ZONE_H,ZONE_L)
114 port map(B6_5d, B6_5d2, clkB, C5_6d2, n_mag_vect(45), n_zones_vect(45), area_eff_vect
    (45), area_tot_vect(45), Er_vect(45), Ec_vect(45));
D6.6: or_wire_lx generic map(D,6,6,UP,ZONE_H,ZONE_L)
    port map(D6_6d, D6_6d2, clkD, A5_7d2, n_mag_vect(46), n_zones_vect(46), area_eff_vect
    (46), area_tot_vect(46), Er_vect(46), Ec_vect(46));
117 C7.4: long_wire generic map(C,7,4,RX,ZONE_H,ZONE_L)
    port map(C7_4d, clkC, D6_4d2, n_mag_vect(52), n_zones_vect(52), area_eff_vect(52),
    area_tot_vect(52), Er_vect(52), Ec_vect(52));
A7.5: and_2outputs_lx generic map(A,7,5,ZONE_H,ZONE_L)
120 port map(A7_5d, A7_5d2, clkA, B6_5d2, B8_5d, n_mag_vect(53), n_zones_vect(53),
    area_eff_vect(53), area_tot_vect(53), Er_vect(53), Ec_vect(53));
C7.6: long_wire generic map(C,7,6,RX,ZONE_H,ZONE_L)
    port map(C7_6d, clkC, D6_6d2, n_mag_vect(54), n_zones_vect(54), area_eff_vect(54),
    area_tot_vect(54), Er_vect(54), Ec_vect(54));
123 B8.5: double_wire_horiz generic map(B,8,5,ZONE_H,ZONE_L)
    port map(B8_5d, B8_5d2, clkB, C7_6d, C9_5d, n_mag_vect(61), n_zones_vect(61),
    area_eff_vect(61), area_tot_vect(61), Er_vect(61), Ec_vect(61));
C9.5: short_wire_horiz generic map(C,9,5,UP,ZONE_H,ZONE_L)
126 port map(C9_5d, clkC, D8_4d, n_mag_vect(69), n_zones_vect(69), area_eff_vect(69),
    area_tot_vect(69), Er_vect(69), Ec_vect(69));

Last: if (ELEMENT = N.BIT-1) generate
129 begin
    A1.4: long_wire generic map(A,1,4,RX,ZONE_H,ZONE_L)
        port map(A_in, clkA, B2_3d, n_mag_vect(4), n_zones_vect(4), area_eff_vect(4),
        area_tot_vect(4), Er_vect(4), Ec_vect(4));
132 C1.5: short_wire_horiz generic map(C,1,5,DOWN,ZONE_H,ZONE_L)
        port map(C1_5d, clkC, D2_4d, n_mag_vect(5), n_zones_vect(5), area_eff_vect(5),
        area_tot_vect(5), Er_vect(5), Ec_vect(5));
A1.6: short_wire_horiz generic map(A,1,6,DOWN,ZONE_H,ZONE_L)
135 port map(A1_6d, clkA, B2_5d, n_mag_vect(6), n_zones_vect(6), area_eff_vect(6),
        area_tot_vect(6), Er_vect(6), Ec_vect(6));
C1.7: short_wire_horiz generic map(C,1,7,DOWN,ZONE_H,ZONE_L)
        port map(C1_7d, clkC, D2_6d, n_mag_vect(7), n_zones_vect(7), area_eff_vect(7),
        area_tot_vect(7), Er_vect(7), Ec_vect(7));
138 A1.8: wire_2outputs generic map(A,1,8,RX,DOWN,ZONE_H,ZONE_L)
        port map(A1_8d, clkA, res, B2_7d, n_mag_vect(8), n_zones_vect(8), area_eff_vect(8),
        area_tot_vect(8), Er_vect(8), Ec_vect(8));

```

```

141 D2.4: short_wire_vert generic map(D,2,4,RX,ZONE_H,ZONE_L)
      port map(D2.4d,clkD,A3.5d,n_mag_vect(12),n_zones_vect(12),area_eff_vect(12),
area_tot_vect(12),Er_vect(12),Ec_vect(12));
B2.5: double_wire_horiz generic map(B,2,5,ZONE_H,ZONE_L)
      port map(B2.5d,B2.5d2,clkB,C1.5d,C3.6d,n_mag_vect(13),n_zones_vect(13),
area_eff_vect(13),area_tot_vect(13),Er_vect(13),Ec_vect(13));
144 D2.6: double_wire_horiz generic map(D,2,6,ZONE_H,ZONE_L)
      port map(D2.6d,D2.6d2,clkD,A1.6d,A9.6d,n_mag_vect(14),n_zones_vect(14),
area_eff_vect(14),area_tot_vect(14),Er_vect(14),Ec_vect(14));
B2.7: short_wire_horiz generic map(B,2,7,UP,ZONE_H,ZONE_L)
147 port map(B2.7d,clkB,C1.7d,n_mag_vect(15),n_zones_vect(15),area_eff_vect(15),
area_tot_vect(15),Er_vect(15),Ec_vect(15));
D2.8: short_wire_vert generic map(D,2,8,LX,ZONE_H,ZONE_L)
      port map(D2.8d,clkD,A1.8d,n_mag_vect(16),n_zones_vect(16),area_eff_vect(16),
area_tot_vect(16),Er_vect(16),Ec_vect(16));
150 A3.5: short_wire_horiz generic map(A,3,5,UP,ZONE_H,ZONE_L)
      port map(A3.5d,clkA,B2.5d2,n_mag_vect(21),n_zones_vect(21),area_eff_vect(21),
area_tot_vect(21),Er_vect(21),Ec_vect(21));
C3.6: short_wire_horiz generic map(C,3,6,UP,ZONE_H,ZONE_L)
153 port map(C3.6d,clkC,D2.6d2,n_mag_vect(22),n_zones_vect(22),area_eff_vect(22),
area_tot_vect(22),Er_vect(22),Ec_vect(22));
C3.8: long_wire generic map(C,3,8,RX,ZONE_H,ZONE_L)
      port map(C3.8d,clkC,D2.8d,n_mag_vect(24),n_zones_vect(24),area_eff_vect(24),
area_tot_vect(24),Er_vect(24),Ec_vect(24));
156 D4.6: inv_horiz generic map(D,4,6,DOWN,ZONE_H,ZONE_L)
      port map(D4.6d,clkD,A5.7d,n_mag_vect(30),n_zones_vect(30),area_eff_vect(30),
area_tot_vect(30),Er_vect(30),Ec_vect(30));
B4.7: crosswire generic map(B,4,7,ZONE_H,ZONE_L)
159 port map(B4.7d,B4.7d2,clkB,C3.8d,C5.8d,n_mag_vect(31),n_zones_vect(31),
area_eff_vect(31),area_tot_vect(31),Er_vect(31),Ec_vect(31));
A5.7: and_wire_lx generic map(A,5,7,UP,ZONE_H,ZONE_L)
      port map(A5.7d,A5.7d2,clkA,B4.7d2,n_mag_vect(39),n_zones_vect(39),area_eff_vect
(39),area_tot_vect(39),Er_vect(39),Ec_vect(39));
162 A9.6: and_wire_rx generic map(A,9,6,UP,ZONE_H,ZONE_L)
      port map(A9.6d,reset2,clkA,B8.5d2,n_mag_vect(70),n_zones_vect(70),area_eff_vect
(70),area_tot_vect(70),Er_vect(70),Ec_vect(70));
— Different in "First" PE
165 A1.1: short_wire_vert generic map(A,1,1,RX,ZONE_H,ZONE_L)
      port map(b_in,clkA,B2.1d,n_mag_vect(1),n_zones_vect(1),area_eff_vect(1),
area_tot_vect(1),Er_vect(1),Ec_vect(1));
A1.2: short_wire_horiz generic map(A,1,2,DOWN,ZONE_H,ZONE_L)
168 port map(A1.2d,clkA,B2.1d2,n_mag_vect(2),n_zones_vect(2),area_eff_vect(2),
area_tot_vect(2),Er_vect(2),Ec_vect(2));
B2.1: crosswire generic map(B,2,1,ZONE_H,ZONE_L)
      port map(B2.1d,B2.1d2,clkB,A_out,C3.2d,n_mag_vect(9),n_zones_vect(9),
area_eff_vect(9),area_tot_vect(9),Er_vect(9),Ec_vect(9));
171 D2.2: wire_2outputs generic map(D,2,2,RX,UP,ZONE_H,ZONE_L)
      port map(D2.2d,clkD,A1.2d,A3.3d,n_mag_vect(10),n_zones_vect(10),area_eff_vect
(10),area_tot_vect(10),Er_vect(10),Ec_vect(10));
D4.2: double_wire_horiz generic map(D,4,2,ZONE_H,ZONE_L)
174 port map(D4.2d,D4.2d2,clkD,A3.3d2,A5.3d,n_mag_vect(26),n_zones_vect(26),
area_eff_vect(26),area_tot_vect(26),Er_vect(26),Ec_vect(26));
B4.3: and_wire_lx generic map(B,4,3,DOWN,ZONE_H,ZONE_L)
      port map(B4.3d,B4.3d2,clkB,C5.4d,n_mag_vect(27),n_zones_vect(27),area_eff_vect
(27),area_tot_vect(27),Er_vect(27),Ec_vect(27));
177 C5.2: short_wire_horiz generic map(C,5,2,UP,ZONE_H,ZONE_L)
      port map(PE_in,clkC,D4.2d2,n_mag_vect(34),n_zones_vect(34),area_eff_vect(34),
area_tot_vect(34),Er_vect(34),Ec_vect(34));
A5.3: and_2outputs_lx generic map(A,5,3,ZONE_H,ZONE_L)

```

```

180     port map(A5_3d, A5_3d2, clkA, B4_3d2, B6_3d, n_mag_vect(35), n_zones_vect(35),
area_eff_vect(35), area_tot_vect(35), Er_vect(35), Ec_vect(35));
D6_2: long_wire generic map(D,6,2,RX,ZONE_H,ZONE_L)
    port map(D6_2d, clkD, A5_3d2, n_mag_vect(42), n_zones_vect(42), area_eff_vect(42),
area_tot_vect(42), Er_vect(42), Ec_vect(42));
183 B6_3: short_wire_horiz generic map(B,6,3,UP,ZONE_H,ZONE_L)
    port map(B6_3d, clkB, C5_4d2, n_mag_vect(43), n_zones_vect(43), area_eff_vect(43),
area_tot_vect(43), Er_vect(43), Ec_vect(43));
C7_2: long_wire generic map(D,7,2,RX,ZONE_H,ZONE_L)
186     port map(C7_2d, clkC, D6_2d, n_mag_vect(50), n_zones_vect(50), area_eff_vect(50),
area_tot_vect(50), Er_vect(50), Ec_vect(50));
C7_3: short_wire_horiz generic map(C,7,3,DOWN,ZONE_H,ZONE_L)
    port map(C7_3d, clkC, D8_2d, n_mag_vect(51), n_zones_vect(51), area_eff_vect(51),
area_tot_vect(51), Er_vect(51), Ec_vect(51));
189 B8_1: crosswire generic map(B,8,1,ZONE_H,ZONE_L)
    port map(B8_1d, B8_1d2, clkB, en_p_out, C7_2d, n_mag_vect(57), n_zones_vect(57),
area_eff_vect(57), area_tot_vect(57), Er_vect(57), Ec_vect(57));
D8_2: long_wire generic map(D,8,2,RX,ZONE_H,ZONE_L)
192     port map(D8_2d, clkD, A9_2d, n_mag_vect(58), n_zones_vect(58), area_eff_vect(58),
area_tot_vect(58), Er_vect(58), Ec_vect(58));
B8_3: crosswire generic map(B,8,3,ZONE_H,ZONE_L)
    port map(B8_3d, B8_3d2, clkB, C7_3d, C7_4d, n_mag_vect(59), n_zones_vect(59),
area_eff_vect(59), area_tot_vect(59), Er_vect(59), Ec_vect(59));
195 D8_4: wire_2outputs generic map(D,8,4,RX,DOWN,ZONE_H,ZONE_L)
    port map(D8_4d, clkD, A7_5d2, A9_4d, n_mag_vect(60), n_zones_vect(60), area_eff_vect
(60), area_tot_vect(60), Er_vect(60), Ec_vect(60));
A9_1: short_wire_vert generic map(A,9,1,RX,ZONE_H,ZONE_L)
198     port map(reset, clkA, B8_1d, n_mag_vect(65), n_zones_vect(65), area_eff_vect(65),
area_tot_vect(65), Er_vect(65), Ec_vect(65));
A9_2: short_wire_horiz generic map(A,9,2,UP,ZONE_H,ZONE_L)
    port map(A9_2d, clkA, B8_1d2, n_mag_vect(66), n_zones_vect(66), area_eff_vect(66),
area_tot_vect(66), Er_vect(66), Ec_vect(66));
201 A9_3: short_wire_vert generic map(A,9,3,RX,ZONE_H,ZONE_L)
    port map(p, clkA, B8_3d, n_mag_vect(67), n_zones_vect(67), area_eff_vect(67),
area_tot_vect(67), Er_vect(67), Ec_vect(67));
A9_4: short_wire_horiz generic map(A,9,4,UP,ZONE_H,ZONE_L)
204     port map(A9_4d, clkA, B8_3d2, n_mag_vect(68), n_zones_vect(68), area_eff_vect(68),
area_tot_vect(68), Er_vect(68), Ec_vect(68));
end generate;

207 Center: if (ELEMENT < N_BIT-1 and ELEMENT > 0) generate
begin
    A1_1: short_wire_vert generic map(A,1,1,RX,ZONE_H,ZONE_L)
210     port map(b_in, clkA, B2_1d, n_mag_vect(1), n_zones_vect(1), area_eff_vect(1),
area_tot_vect(1), Er_vect(1), Ec_vect(1));
    A1_2: short_wire_horiz generic map(A,1,2,DOWN,ZONE_H,ZONE_L)
    port map(A1_2d, clkA, B2_1d2, n_mag_vect(2), n_zones_vect(2), area_eff_vect(2),
area_tot_vect(2), Er_vect(2), Ec_vect(2));
213 A1_4: short_wire_horiz generic map(A,1,4,DOWN,ZONE_H,ZONE_L)
    port map(A1_4d, clkA, B2_3d, n_mag_vect(4), n_zones_vect(4), area_eff_vect(4),
area_tot_vect(4), Er_vect(4), Ec_vect(4));
    A1_6: short_wire_vert generic map(A,1,6,RX,ZONE_H,ZONE_L)
216     port map(A1_6d, clkA, res, n_mag_vect(6), n_zones_vect(6), area_eff_vect(6),
area_tot_vect(6), Er_vect(6), Ec_vect(6));
    B2_1: crosswire generic map(B,2,1,ZONE_H,ZONE_L)
    port map(B2_1d, B2_1d2, clkB, A_out, C3_2d, n_mag_vect(9), n_zones_vect(9),
area_eff_vect(9), area_tot_vect(9), Er_vect(9), Ec_vect(9));
219 D2_2: wire_2outputs generic map(D,2,2,RX,UP,ZONE_H,ZONE_L)

```

```

    port map(D2_2d, clkD, A1_2d, A3_3d, n_mag_vect(10), n_zones_vect(10), area_eff_vect
(10), area_tot_vect(10), Er_vect(10), Ec_vect(10));
222 D2_4: long_wire generic map(D,2,4,LX,ZONE_H,ZONE_L)
    port map(D2_4d, clkD, A1_4d, n_mag_vect(12), n_zones_vect(12), area_eff_vect(12),
area_tot_vect(12), Er_vect(12), Ec_vect(12));
B2_5: short_wire_horiz generic map(B,2,5,DOWN,ZONE_H,ZONE_L)
    port map(B2_5d, clkB, C3_5d, n_mag_vect(13), n_zones_vect(13), area_eff_vect(13),
area_tot_vect(13), Er_vect(13), Ec_vect(13));
225 D2_6: long_wire generic map(D,2,6,LX,ZONE_H,ZONE_L)
    port map(D2_6d, clkD, A1_6d, n_mag_vect(14), n_zones_vect(14), area_eff_vect(14),
area_tot_vect(14), Er_vect(14), Ec_vect(14));
C3_5: short_wire_horiz generic map(C,3,5,UP,ZONE_H,ZONE_L)
228    port map(C3_5d, clkC, D2_4d, n_mag_vect(21), n_zones_vect(21), area_eff_vect(21),
area_tot_vect(21), Er_vect(21), Ec_vect(21));
A3_6: long_wire generic map(A,3,6,LX,ZONE_H,ZONE_L)
    port map(A3_6d, clkA, B2_5d, n_mag_vect(22), n_zones_vect(22), area_eff_vect(22),
area_tot_vect(22), Er_vect(22), Ec_vect(22));
231 C3_7: crosswire generic map(C,3,7,ZONE_H,ZONE_L)
    port map(A_in, C3_7d2, clkC, D2_6d, D4_6d2, n_mag_vect(23), n_zones_vect(23),
area_eff_vect(23), area_tot_vect(23), Er_vect(23), Ec_vect(23));
D4_2: double_wire_horiz generic map(D,4,2,ZONE_H,ZONE_L)
234    port map(D4_2d, D4_2d2, clkD, A3_3d2, A5_3d, n_mag_vect(26), n_zones_vect(26),
area_eff_vect(26), area_tot_vect(26), Er_vect(26), Ec_vect(26));
B4_3: and_wire_lx generic map(B,4,3,DOWN,ZONE_H,ZONE_L)
    port map(B4_3d, B4_3d2, clkB, C5_4d, n_mag_vect(27), n_zones_vect(27), area_eff_vect
(27), area_tot_vect(27), Er_vect(27), Ec_vect(27));
237 D4_6: inv_with_wire_horiz generic map(D,4,6,DOWN,ZONE_H,ZONE_L)
    port map(D4_6d, D4_6d2, clkD, A5_7d, A3_6d, n_mag_vect(30), n_zones_vect(30),
area_eff_vect(30), area_tot_vect(30), Er_vect(30), Ec_vect(30));
B4_7: wire_2outputs generic map(B,4,7,LX,DOWN,ZONE_H,ZONE_L)
240    port map(B4_7d, clkB, C3_7d2, PE_out, n_mag_vect(31), n_zones_vect(31), area_eff_vect
(31), area_tot_vect(31), Er_vect(31), Ec_vect(31));
C5_2: short_wire_horiz generic map(C,5,2,UP,ZONE_H,ZONE_L)
    port map(PE_in, clkC, D4_2d2, n_mag_vect(34), n_zones_vect(34), area_eff_vect(34),
area_tot_vect(34), Er_vect(34), Ec_vect(34));
243 A5_3: and_2outputs_lx generic map(A,5,3,ZONE_H,ZONE_L)
    port map(A5_3d, A5_3d2, clkA, B4_3d2, B6_3d, n_mag_vect(35), n_zones_vect(35),
area_eff_vect(35), area_tot_vect(35), Er_vect(35), Ec_vect(35));
A5_7: and_wire_lx generic map(A,5,7,UP,ZONE_H,ZONE_L)
246    port map(A5_7d, A5_7d2, clkA, B4_7d, n_mag_vect(39), n_zones_vect(39), area_eff_vect(39),
area_tot_vect(39), Er_vect(39), Ec_vect(39));
D6_2: long_wire generic map(D,6,2,RX,ZONE_H,ZONE_L)
    port map(D6_2d, clkD, A5_3d2, n_mag_vect(42), n_zones_vect(42), area_eff_vect(42),
area_tot_vect(42), Er_vect(42), Ec_vect(42));
249 B6_3: short_wire_horiz generic map(B,6,3,UP,ZONE_H,ZONE_L)
    port map(B6_3d, clkB, C5_4d2, n_mag_vect(43), n_zones_vect(43), area_eff_vect(43),
area_tot_vect(43), Er_vect(43), Ec_vect(43));
C7_2: long_wire generic map(D,7,2,RX,ZONE_H,ZONE_L)
252    port map(C7_2d, clkC, D6_2d, n_mag_vect(50), n_zones_vect(50), area_eff_vect(50),
area_tot_vect(50), Er_vect(50), Ec_vect(50));
C7_3: short_wire_horiz generic map(C,7,3,DOWN,ZONE_H,ZONE_L)
    port map(C7_3d, clkC, D8_2d, n_mag_vect(51), n_zones_vect(51), area_eff_vect(51),
area_tot_vect(51), Er_vect(51), Ec_vect(51));
255 C7_7: short_wire_horiz generic map(C,7,7,DOWN,ZONE_H,ZONE_L)
    port map(en_p_in, clkC, D8_6d, n_mag_vect(55), n_zones_vect(55), area_eff_vect(55),
area_tot_vect(55), Er_vect(55), Ec_vect(55));
B8_1: crosswire generic map(B,8,1,ZONE_H,ZONE_L)
258    port map(B8_1d, B8_1d2, clkB, en_p_out, C7_2d, n_mag_vect(57), n_zones_vect(57),
area_eff_vect(57), area_tot_vect(57), Er_vect(57), Ec_vect(57));

```



```

D8_2: long_wire generic map(D,8,2,RX,ZONE_H,ZONE_L)
    port map(D8_2d,clkD,A9_2d,n_mag_vect(58),n_zones_vect(58),area_eff_vect(58),
area_tot_vect(58),Er_vect(58),Ec_vect(58));
261 B8_3: crosswire generic map(B,8,3,ZONE_H,ZONE_L)
    port map(B8_3d,B8_3d2,clkB,C7_3d,C7_4d,n_mag_vect(59),n_zones_vect(59),
area_eff_vect(59),area_tot_vect(59),Er_vect(59),Ec_vect(59));
D8_4: wire_2outputs generic map(D,8,4,RX_DOWN,ZONE_H,ZONE_L)
264 port map(D8_4d,clkD,A7_5d2,A9_4d,n_mag_vect(60),n_zones_vect(60),area_eff_vect
(60),area_tot_vect(60),Er_vect(60),Ec_vect(60));
A9_1: short_wire_vert generic map(A,9,1,RX,ZONE_H,ZONE_L)
    port map(reset,clkA,B8_1d,n_mag_vect(65),n_zones_vect(65),area_eff_vect(65),
area_tot_vect(65),Er_vect(65),Ec_vect(65));
267 A9_2: short_wire_horiz generic map(A,9,2,UP,ZONE_H,ZONE_L)
    port map(A9_2d,clkA,B8_1d2,n_mag_vect(66),n_zones_vect(66),area_eff_vect(66),
area_tot_vect(66),Er_vect(66),Ec_vect(66));
A9_3: short_wire_vert generic map(A,9,3,RX,ZONE_H,ZONE_L)
270 port map(p,clkA,B8_3d,n_mag_vect(67),n_zones_vect(67),area_eff_vect(67),
area_tot_vect(67),Er_vect(67),Ec_vect(67));
A9_4: short_wire_horiz generic map(A,9,4,UP,ZONE_H,ZONE_L)
    port map(A9_4d,clkA,B8_3d2,n_mag_vect(68),n_zones_vect(68),area_eff_vect(68),
area_tot_vect(68),Er_vect(68),Ec_vect(68));
273 D8_6: long_wire generic map(D,8,6,RX,ZONE_H,ZONE_L)
    port map(D8_6d,clkD,A9_6d,n_mag_vect(62),n_zones_vect(62),area_eff_vect(62),
area_tot_vect(62),Er_vect(62),Ec_vect(62));
A9_6: short_wire_horiz generic map(A,9,6,UP,ZONE_H,ZONE_L)
276 port map(A9_6d,clkA,B8_5d2,n_mag_vect(70),n_zones_vect(70),area_eff_vect(70)
,area_tot_vect(70),Er_vect(70),Ec_vect(70));
end generate;

279 First: if (ELEMENT = 0) generate
begin
    A1_2: short_wire_vert generic map(A,1,2,LX,ZONE_H,ZONE_L)
282 port map(b_in,clkA,B2_1d,n_mag_vect(2),n_zones_vect(2),area_eff_vect(2),
area_tot_vect(2),Er_vect(2),Ec_vect(2));
    B2_1: short_wire_vert generic map(B,2,1,RX,ZONE_H,ZONE_L)
    port map(B2_1d,clkB,C3_2d,n_mag_vect(9),n_zones_vect(9),area_eff_vect(9),
area_tot_vect(9),Er_vect(9),Ec_vect(9));
285 D2_2: short_wire_vert generic map(D,2,2,RX,ZONE_H,ZONE_L)
    port map(D2_2d,clkD,A3_3d,n_mag_vect(10),n_zones_vect(10),area_eff_vect(10),
area_tot_vect(10),Er_vect(10),Ec_vect(10));
    D4_2: short_wire_horiz generic map(D,4,2,UP,ZONE_H,ZONE_L)
288 port map(D4_2d,clkD,A3_3d2,n_mag_vect(26),n_zones_vect(26),area_eff_vect(26),
area_tot_vect(26),Er_vect(26),Ec_vect(26));
    B4_3: and_wire_lx generic map(B,4,3,DOWN,ZONE_H,ZONE_L)
    port map(B4_3d,zero_in,clkB,C5_4d,n_mag_vect(27),n_zones_vect(27),area_eff_vect
(27),area_tot_vect(27),Er_vect(27),Ec_vect(27));
291 B6_3: short_wire_horiz generic map(B,6,3,UP,ZONE_H,ZONE_L)
    port map(zero_in,clkB,C5_4d2,n_mag_vect(43),n_zones_vect(43),area_eff_vect(43),
area_tot_vect(43),Er_vect(43),Ec_vect(43));
    B8_3: short_wire_vert generic map(B,8,3,RX,ZONE_H,ZONE_L)
294 port map(B8_3d,clkB,C7_4d,n_mag_vect(59),n_zones_vect(59),area_eff_vect(59),
area_tot_vect(59),Er_vect(59),Ec_vect(59));
    D8_4: short_wire_vert generic map(D,8,4,RX,ZONE_H,ZONE_L)
    port map(D8_4d,clkD,A7_5d2,n_mag_vect(60),n_zones_vect(60),area_eff_vect(60),
area_tot_vect(60),Er_vect(60),Ec_vect(60));
297 A9_4: short_wire_vert generic map(A,9,4,LX,ZONE_H,ZONE_L)
    port map(p,clkA,B8_3d,n_mag_vect(68),n_zones_vect(68),area_eff_vect(68),
area_tot_vect(68),Er_vect(68),Ec_vect(68));

```

```

300  — Different in "Last" PE
A1_4: short_wire_horiz generic map(A,1,4,DOWN,ZONE_H,ZONE_L)
      port map(A1_4d,clkA,B2_3d,n_mag_vect(4),n_zones_vect(4),area_eff_vect(4),
303  area_tot_vect(4),Er_vect(4),Ec_vect(4));
A1_6: short_wire_vert generic map(A,1,6,RX,ZONE_H,ZONE_L)
      port map(A1_6d,clkA,res,n_mag_vect(6),n_zones_vect(6),area_eff_vect(6),
      area_tot_vect(6),Er_vect(6),Ec_vect(6));
D2_4: long_wire generic map(D,2,4,LX,ZONE_H,ZONE_L)
306  port map(D2_4d,clkD,A1_4d,n_mag_vect(12),n_zones_vect(12),area_eff_vect(12),
      area_tot_vect(12),Er_vect(12),Ec_vect(12));
B2_5: short_wire_horiz generic map(B,2,5,DOWN,ZONE_H,ZONE_L)
      port map(B2_5d,clkB,C3_5d,n_mag_vect(13),n_zones_vect(13),area_eff_vect(13),
      area_tot_vect(13),Er_vect(13),Ec_vect(13));
309  D2_6: long_wire generic map(D,2,6,LX,ZONE_H,ZONE_L)
      port map(D2_6d,clkD,A1_6d,n_mag_vect(14),n_zones_vect(14),area_eff_vect(14),
      area_tot_vect(14),Er_vect(14),Ec_vect(14));
C3_5: short_wire_horiz generic map(C,3,5,UP,ZONE_H,ZONE_L)
312  port map(C3_5d,clkC,D2_4d,n_mag_vect(21),n_zones_vect(21),area_eff_vect(21),
      area_tot_vect(21),Er_vect(21),Ec_vect(21));
A3_6: long_wire generic map(A,3,6,LX,ZONE_H,ZONE_L)
      port map(A3_6d,clkA,B2_5d,n_mag_vect(22),n_zones_vect(22),area_eff_vect(22),
      area_tot_vect(22),Er_vect(22),Ec_vect(22));
315  C3_7: crosswire generic map(C,3,7,ZONE_H,ZONE_L)
      port map(A_in,C3_7d2,clkC,D2_6d,D4_6d2,n_mag_vect(23),n_zones_vect(23),
      area_eff_vect(23),area_tot_vect(23),Er_vect(23),Ec_vect(23));
D4_6: inv_with_wire_horiz generic map(D,4,6,DOWN,ZONE_H,ZONE_L)
318  port map(D4_6d,D4_6d2,clkD,A5_7d,A3_6d,n_mag_vect(30),n_zones_vect(30),
      area_eff_vect(30),area_tot_vect(30),Er_vect(30),Ec_vect(30));
B4_7: wire_2outputs generic map(B,4,7,LX,DOWN,ZONE_H,ZONE_L)
      port map(B4_7d,clkB,C3_7d2,PE_out,n_mag_vect(31),n_zones_vect(31),area_eff_vect
321  (31),area_tot_vect(31),Er_vect(31),Ec_vect(31));
A5_7: and_wire_lx generic map(A,5,7,UP,ZONE_H,ZONE_L)
      port map(A5_7d,A5_7d2,clkA,B4_7d,n_mag_vect(39),n_zones_vect(39),area_eff_vect
      (39),area_tot_vect(39),Er_vect(39),Ec_vect(39));
C7_7: short_wire_horiz generic map(C,7,7,DOWN,ZONE_H,ZONE_L)
324  port map(en_p_in,clkC,D8_6d,n_mag_vect(55),n_zones_vect(55),area_eff_vect(55),
      area_tot_vect(55),Er_vect(55),Ec_vect(55));
D8_6: long_wire generic map(D,8,6,RX,ZONE_H,ZONE_L)
      port map(D8_6d,clkD,A9_6d,n_mag_vect(62),n_zones_vect(62),area_eff_vect(62),
      area_tot_vect(62),Er_vect(62),Ec_vect(62));
327  A9_6: short_wire_horiz generic map(A,9,6,UP,ZONE_H,ZONE_L)
      port map(A9_6d,clkA,B8_5d2,n_mag_vect(70),n_zones_vect(70),area_eff_vect(70),
      area_tot_vect(70),Er_vect(70),Ec_vect(70));
      end generate;
330  end behavior;

```

A.3 Multiply Accumulate unit (MAC)

This section reports the listings for the three implementations of the MAC unit. The basic blocks of the parallel and serial-parallel MAC are not included. The listings of the ME-NML

Galois Multiplier already give a clear example of how to describe a basic block starting from the drawing.

A.3.1 Parallel MAC

Here are only the top entity `MAC_N_bit`, and its two main components: the `Multiplier` and the `Adder/Accumulator ACC`. The interconnection components are not shown.

Listing A.21. Top entity of the Parallel MAC: `MAC_N_bit`.

```

library ieee;
2 use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
  use ieee.std_logic_arith.all;
5 use work.MENML_package.all;

entity MAC_N_BIT is
8   port(A,B:in std_logic_vector (N_BIT-1 downto 0);
      reset: in std_logic;
      MAC_result: out std_logic_vector (2*N_BIT-1 downto 0);
11   MAC_Co:out std_logic;
      clkA, clkB, clkC, clkD: in std_logic;
      n_mag: out natural := init_natural;
14   n_zones: out natural := init_natural;
      AREA_EFF: out natural;
      AREA_TOT: out natural;
17   Er: out natural;
      Ec: out natural);
end MAC_N_BIT;
20

architecture behavior of MAC_N_BIT is
  [...] --Components definition
23
  -- Vectors of natural for magnets and cell count, area and energy evaluation
  type natural_vector is array (natural range <>) of natural;
26   signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
      natural_vector ((N_BIT-1)*N_BIT downto 1) := (others => init_natural);
  -- Connections among macro-blocks
  signal result_from_ACC, result_from_Bconn: std_logic_vector (2*N_BIT-1 downto 0);
29   signal in_B_conn_from_ACC: std_logic_vector (N_BIT-4 downto 0);
  signal in_ACC_from_B_conn: std_logic_vector (N_BIT-4 downto 0);
  signal B_for_FA_ACC, B_for_mul, A_from_Aconn: std_logic_vector (N_BIT-1 downto 0);
32   signal Za_vect, Zb_vect: std_logic_vector (2*N_BIT-2 downto 0);
  signal Z_last, C_temp_in, C_temp_out, reset_from_ACC: std_logic;
  signal reset_lat: std_logic_vector (N_BIT-1 downto 0);
35   signal temp_from_ACC_to_Mul: std_logic_vector (N_BIT-3 downto 0);
  signal temp_from_Mul_to_ACC: std_logic_vector (N_BIT-3 downto 0);
begin
38   -- SUM OF ARRAYS OF NATURAL ELEMENTS -----
  -- This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec of every
  PE instantiated.
  -- Results are given as outputs of this "Galois_MAC_N_BIT" component.
41   N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
      Ec_vect)

```

```

variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
:= init_natural;
44 variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
sum_tot_Er, sum_tot_Ec: natural := init_natural;
begin
n_nat_mag := n_mag_vect;
47 n_nat_zones := n_zones_vect;
nat_area_eff := area_eff_vect;
nat_area_tot := area_tot_vect;
50 nat_Er := Er_vect;
nat_Ec := Ec_vect;

sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
sum_Ec:= 0;
for i in 0 to n_mag_vect'length -1 loop
56 sum_n_mag := sum_n_mag + n_nat_mag(i);
sum_n_zones := sum_n_zones + n_nat_zones(i);
sum_area_eff := sum_area_eff + nat_area_eff(i);
sum_area_tot := sum_area_tot + nat_area_tot(i);
59 sum_Er := sum_Er + nat_Er(i);
sum_Ec := sum_Ec + nat_Ec(i);
end loop;
62 sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
65 sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

68 n_mag <= sum_tot_n_mag;
n_zones <= sum_tot_n_zones;
71 area_eff <= sum_tot_area_eff;
area_tot <= sum_tot_area_tot;
Er <= sum_tot_Er;
74 Ec <= sum_tot_Ec;
end process;

77 B_input: B_connection
port map(
B, result_from_ACC, in_B_conn_from_ACC,
80 B_for_FA_ACC, result_from_Bconn, in_ACC_from_B_conn,
clk, clkA, clkB, clkC, clkD,
n_mag_vect(1), n_zones_vect(1), area_eff_vect(1), area_tot_vect(1), Er_vect(1),
Ec_vect(1));
83
Accumulator: ACC
port map(
86 Za_vect, Zb_vect, Z_last, B_for_FA_ACC, C_temp.in, reset, reset_lat, temp_from_Mul_to_ACC,
in_ACC_from_B_conn,
temp_from_ACC_to_Mul, in_B_conn_from_ACC, B_for_mul, result_from_ACC, MAC.Co, C_temp_out,
reset_from_ACC,
clk, clkA, clkB, clkC, clkD,
89 n_mag_vect(2), n_zones_vect(2), area_eff_vect(2), area_tot_vect(2), Er_vect(2),
Ec_vect(2));
Mult: Multiplier
port map(
92 A_from_Aconn, B_for_mul, temp_from_ACC_to_Mul, C_temp_out, reset_from_ACC,

```

```

    C_temp_in, Z_last, Za_vect, Zb_vect, temp_from_Mul_to_ACC, reset_lat ,
    clk, clkA, clkB, clkC, clkD,
95    n_mag_vect(3), n_zones_vect(3), area_eff_vect(3), area_tot_vect(3), Er_vect(3),
    Ec_vect(3));
    A_input: A_conn
    port map(
98    A, A_from_Aconn,
    clk, clkA, clkB, clkC, clkD,
    n_mag_vect(4), n_zones_vect(4), area_eff_vect(4), area_tot_vect(4), Er_vect(4),
    Ec_vect(4));
101    Connecti_out: MAC_conn
    port map(result_from_Bconn, MAC_result,
    clk, clkA, clkB, clkC, clkD,
104    n_mag_vect(5), n_zones_vect(5), area_eff_vect(5), area_tot_vect(5), Er_vect(5),
    Ec_vect(5));
end behavior;

```

Listing A.22. Array Multiplier: Multiplier.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML_package.all;
6
entity Multiplier is
    port(A_mul_in, B_mul_in: in std_logic_vector (N_bit-1 downto 0); -- input data vector
9    temp_ACC_in: in std_logic_vector (N_bit-3 downto 0);
    C0_in, res_acc: in std_logic;
    C1_out, Z_lat_sup: out std_logic;
12    Za, Zb: out std_logic_vector (2*N_bit-2 downto 0);
    temp_ACC_out: out std_logic_vector (N_bit-3 downto 0);
    res_lat: out std_logic_vector (N_bit-1 downto 0);
15    clkA, clkB, clkC, clkD: in std_logic;
    n_mag: out natural := init_natural;
    n_zones: out natural := init_natural;
18    AREA_EFF: out natural;
    AREA_TOT: out natural;
    Er: out natural;
21    Ec: out natural);
end Multiplier;

24 architecture behavior of Multiplier is
    [...] -- Components definition

27 -- Vectors of natural for magnets and cell count, area and energy evaluation
    type natural_vector is array (natural range <>) of natural;
    signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
        natural_vector ((N_bit-1)*N_bit downto 1) := (others => init_natural);
30    signal x_vect, y_vect, C_out: std_logic_vector (N_bit**2-N_bit-1 downto 0);
    signal temp_up, temp_down, S_prev: std_logic_vector (N_bit**2-3*N_bit+1 downto 0);
    signal Co_last: std_logic_vector (N_bit-3 downto 0);
33    signal tempZ0, tempSangle, GND: std_logic;
    signal res_vect: std_logic_vector (N_bit-2 downto 0);
begin
36 -- SUM OF ARRAYS OF NATURAL ELEMENTS -----
    -- This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec of every
    PE instantiated.

```

```

39  -- Results are given as outputs of this "Galois_Multiplier" component.
N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
Ec_vect)
    variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
    natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
    variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
    := init_natural;
42    variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
    sum_tot_Er, sum_tot_Ec: natural := init_natural;
begin
    n_nat_mag := n_mag_vect;
45    n_nat_zones := n_zones_vect;
    nat_area_eff := area_eff_vect;
    nat_area_tot := area_tot_vect;
48    nat_Er := Er_vect;
    nat_Ec := Ec_vect;

51    sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
    sum_Ec:= 0;
    for i in 0 to n_mag_vect'length -1 loop
54        sum_n_mag := sum_n_mag + n_nat_mag(i);
        sum_n_zones := sum_n_zones + n_nat_zones(i);
        sum_area_eff := sum_area_eff + nat_area_eff(i);
57        sum_area_tot := sum_area_tot + nat_area_tot(i);
        sum_Er := sum_Er + nat_Er(i);
        sum_Ec := sum_Ec + nat_Ec(i);
    end loop;
60    sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
    sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
    sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
63    sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
    sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
    sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;
66
    n_mag <= sum_tot_n_mag;
    n_zones <= sum_tot_n_zones;
69    area_eff <= sum_tot_area_eff;
    area_tot <= sum_tot_area_tot;
    Er <= sum_tot_Er;
72    Ec <= sum_tot_Ec;
end process;

75  -- The structure of this macro-block is like a matrix. In order to describe
-- the position of PE, signals and energy signal, vectors are used and the
-- indeces are computed in according with the following rules:
78  -- 1)the first row is the lowest one (in according to the drawing), index
--    'k' identifies the row starting from 0;
-- 2)moving from the left to the right in the first row (the used index is
81  --    'i'), the position starts from 0 (1 for the energy) up to the last
--    element of the row;
-- 3)for the other rows the same procedure is used, but the offset,
84  --    corresponding to the number of elements of the previous rows, is added.
Za(0)<=tempZ0;
Zb(0)<=tempZ0;
87  Za(N_BIT-1)<=tempSangle;
  Zb(N_BIT-1)<=tempSangle;
  Mul: for k in 0 to N_bit-1 generate
90      row_0: if (k=0) generate--row 0 from down
          row: for i in 0 to N_bit-2 generate

```



```

135         n_mag_vect(k*(N_bit-1)+i+1), n_zones_vect(k*(N_bit-1)+i+1),
area_eff_vect(k*(N_bit-1)+i+1), area_tot_vect(k*(N_bit-1)+i+1), Er_vect(k*(N_bit-1)+
i+1), Ec_vect(k*(N_bit-1)+i+1));
138     end generate;
last_el: if (i=N_bit-2) generate
141     el: FA_fin_cent
port map(
138         x_vect(k*(N_bit-1)+i), y_vect((i+1)*(N_BIT-1)+k-1), C_out(i*(N_bit-1)+k-1)
, S_prev(k*(N_BIT-2)+i-1), temp_down(k*(N_BIT-2)+i-1), res_vect(k-1),
y_vect((i+1)*(N_BIT-1)+k), Za(N_bit-1+k), Zb(N_bit-1+k), C_out(i*(N_bit-1)+
k), temp_up(k*(N_BIT-2)+i-1), res_vect(k), res_lat(k-1),
141         clk, clkA, clkB, clkC, clkD,
n_mag_vect(k*(N_bit-1)+i+1), n_zones_vect(k*(N_bit-1)+i+1),
area_eff_vect(k*(N_bit-1)+i+1), area_tot_vect(k*(N_bit-1)+i+1), Er_vect(k*(N_bit-1)+
i+1), Ec_vect(k*(N_bit-1)+i+1));
end generate;
end generate;
144 end generate;
last_row: if (k=N_bit-1) generate
147 row: for i in 0 to N_bit-2 generate
el_0: if (i=0) generate
147     el: AND_HA_sup
port map(
150         x_vect(k*(N_bit-1)), y_vect(k-1), y_vect(N_BIT+k-2), C_out(i*(N_bit-1)+k-1)
, temp_up((k-1)*(N_BIT-2)+i),
temp_down((k-1)*(N_BIT-2)+i), GND, x_vect(k*(N_bit-1)+i+1), S_prev((k-1)*(
N_BIT-2)+i), Co_last(i),
153         clk, clkA, clkB, clkC, clkD,
n_mag_vect(k*(N_bit-1)+i+1), n_zones_vect(k*(N_bit-1)+i+1),
area_eff_vect(k*(N_bit-1)+i+1), area_tot_vect(k*(N_bit-1)+i+1), Er_vect(k*(N_bit-1)+
i+1), Ec_vect(k*(N_bit-1)+i+1));
end generate;
other_el: if ((i>0)and(i<N_bit-2)) generate
156     el: FA_cent_sup
port map(
159         x_vect(k*(N_bit-1)+i), y_vect((i+1)*(N_BIT-1)+k-1), C_out(i*(N_bit-1)+k-1)
, temp_up((k-1)*(N_BIT-2)+i), Co_last(i-1),
S_prev((k-1)*(N_BIT-2)+i), Co_last(i), temp_down((k-1)*(N_BIT-2)+i), x_vect
(k*(N_bit-1)+i+1),
162         clk, clkA, clkB, clkC, clkD,
n_mag_vect(k*(N_bit-1)+i+1), n_zones_vect(k*(N_bit-1)+i+1),
area_eff_vect(k*(N_bit-1)+i+1), area_tot_vect(k*(N_bit-1)+i+1), Er_vect(k*(N_bit-1)+
i+1), Ec_vect(k*(N_bit-1)+i+1));
end generate;
last_el: if (i=N_bit-2) generate
165     el: FA_fin_sup
port map(
168         Co_last(i-1), x_vect(k*(N_bit-1)+i), y_vect((i+1)*(N_BIT-1)+k-1), C_out(i*(
N_bit-1)+k-1), res_vect(k-1),
Za(N_bit-1+k), Zb(N_bit-1+k), Z_lat_sup, res_lat(k), res_lat(k-1),
168         clk, clkA, clkB, clkC, clkD,
n_mag_vect(k*(N_bit-1)+i+1), n_zones_vect(k*(N_bit-1)+i+1),
area_eff_vect(k*(N_bit-1)+i+1), area_tot_vect(k*(N_bit-1)+i+1), Er_vect(k*(N_bit-1)+
i+1), Ec_vect(k*(N_bit-1)+i+1));
end generate;
end generate;
171 end generate;
end generate;
174 end behavior;

```

Listing A.23. Ripple Carry Adder: ACC.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML-package.all;
6
entity ACC is
  port(Za,Zb: in std_logic_vector (2*N_bit-2 downto 0);
9     Z_lat_sup: in std_logic;
    B_ACC_in: in std_logic_vector (N_bit-1 downto 0);
    Cl,res_in: in std_logic;
12    res_lat_in: in std_logic_vector (N_bit-1 downto 0);
    temp_MUL_in: in std_logic_vector (N_bit-3 downto 0);
    temp_Bconn_in: in std_logic_vector (N_bit-4 downto 0);
15    temp_MUL_out: out std_logic_vector (N_bit-3 downto 0);
    temp_Bconn_out: out std_logic_vector (N_bit-4 downto 0);
    B_ACC_out: out std_logic_vector (N_bit-1 downto 0);
18    S_ACC: out std_logic_vector (N_bit*2-1 downto 0);
    Co_ACC,C0,res_out: out std_logic;
    clk, clkA, clkB, clkC, clkD: in std_logic;
21    n_mag: out natural := init_natural;
    n_zones: out natural := init_natural;
    AREA_EFF: out natural;
24    AREA_TOT: out natural;
    Er: out natural;
    Ec: out natural);
27 end ACC;

architecture behavior of ACC is
30   [...] ---Components definition.

   signal mrbit_vect,en_p_vect, results_vect: std_logic_vector (N_BIT downto 0);
   --- Vectors of natural for magnets and cell count, area and energy evaluation
   type natural_vector is array (natural range <>) of natural;
   signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
       natural_vector (N_BIT*2 downto 1) := (others => init_natural);
36   signal C_vect_inf_a: std_logic_vector (N_bit-3 downto 0);
   signal C_vect_inf_b: std_logic_vector (N_bit-3 downto 0);
   signal C_vect_lat: std_logic_vector (N_bit+2 downto 0);
39   signal temp_in: std_logic_vector (N_bit-2 downto 1);
   signal temp_out: std_logic_vector (N_bit-2 downto 1);
   signal res_vect: std_logic_vector (N_bit-3 downto 0);
42   signal S_prev: std_logic_vector (N_bit-3 downto 0);
   signal S_prev_vect_out: std_logic_vector (N_bit-2 downto 0);
   signal B2,temp_first_in,temp_first_out,err1,err2: std_logic;
45   begin

   --- SUM OF ARRAYS OF NATURAL ELEMENTS ---
48   --- This process sums up the values of n_mag,n_zones,area_eff,area_tot,Er,Ec of every
       PE instantiated.
   --- Results are given as outputs of this "Galois_Multiplier" component.
   N_mag_sum: process (n_mag_vect,n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
       Ec_vect)
51     variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
       natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
     variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
       := init_natural;

```

```

variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
sum_tot_Er, sum_tot_Ec: natural := init_natural;
54 begin
    n_nat_mag := n_mag_vect;
    n_nat_zones := n_zones_vect;
57 nat_area_eff := area_eff_vect;
    nat_area_tot := area_tot_vect;
    nat_Er := Er_vect;
60 nat_Ec := Ec_vect;

    sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
    sum_Ec:= 0;
63 for i in 0 to n_mag_vect'length-1 loop
    sum_n_mag := sum_n_mag + n_nat_mag(i);
    sum_n_zones := sum_n_zones + n_nat_zones(i);
66 sum_area_eff := sum_area_eff + nat_area_eff(i);
    sum_area_tot := sum_area_tot + nat_area_tot(i);
    sum_Er := sum_Er + nat_Er(i);
69 sum_Ec := sum_Ec + nat_Ec(i);
end loop;
sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
72 sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
75 sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

78 n_mag <= sum_tot_n_mag;
    n_zones <= sum_tot_n_zones;
    area_eff <= sum_tot_area_eff;
81 area_tot <= sum_tot_area_tot;
    Er <= sum_tot_Er;
    Ec <= sum_tot_Ec;
84 end process;
-- The structure of this macro-block is like a vector. The indices for
-- signals and energy signal are assigned in according to their position
87 -- in the vector structure,
-- 'i' identifies the position.; in particular i=0 identifies the leftmost PE.
B_ACC_out(0)<=B_ACC_in(0);
90 ACC: for i in 0 to N_bit*2-2 generate
    first_element: if (i=0) generate
        HA_FA: FIRST_HA_FA_ACC
93         port map(
            Za(i), B_ACC_in(i+1), B_ACC_in(i+2), res_in, C1, Za(i+1), Zb(i+1), temp_MUL_in(i),
            temp_first_in,
            S_ACC(i), B_ACC_out(i+1), temp_MUL_out(i), temp_first_out, C0, C_vect_inf_a(i),
            C_vect_inf_b(i), S_prev(i), res_vect(i), B2,
96             clk, clkA, clkB, clkC, clkD,
            n_mag_vect(i+1), n_zones_vect(i+1), area_eff_vect(i+1), area_tot_vect(i+1),
            Er_vect(i+1), Ec_vect(i+1));
        end generate;
    second_element: if (i=1) generate
        FA: FA_ACC_inf
        port map(
102         temp_first_out, res_vect(i-1), Za(i+1), Zb(i+1), C_vect_inf_a(i-1), temp_MUL_in(i),
            temp_Bconn_in(i-1), temp_in(i), B2, S_prev(i-1),
            B_ACC_out(i+1), res_vect(i), S_ACC(i), S_prev(i), C_vect_inf_a(i), C_vect_inf_b(i),
            temp_MUL_out(i), temp_Bconn_out(i-1), temp_out(i), temp_first_in,
            clk, clkA, clkB, clkC, clkD,

```



```

105     n_mag_vect(i+1), n_zones_vect(i+1), area_eff_vect(i+1), area_tot_vect(i+1),
Er_vect(i+1), Ec_vect(i+1));
end generate;

108 other_inf_element: if ((i>1)and(i<N_bit-2)) generate
FA:FA_ACC_inf
port map(
111 B_ACC_in(i+1), res_vect(i-1), Za(i+1), Zb(i+1), C_vect_inf_a(i-1), temp_MUL_in(i),
temp_Bconn_in(i-1), temp_in(i), temp_out(i-1), S_prev(i-1),
B_ACC_out(i+1), res_vect(i), S_ACC(i), S_prev(i), C_vect_inf_a(i), C_vect_inf_b(i),
temp_MUL_out(i), temp_Bconn_out(i-1), temp_out(i), temp_in(i-1),
clk, clkA, clkB, clkC, clkD,
114 n_mag_vect(i+1), n_zones_vect(i+1), area_eff_vect(i+1), area_tot_vect(i+1),
Er_vect(i+1), Ec_vect(i+1));
end generate;
angle_element: if (i=N_bit-2) generate
117 FA:FA_ACC_angle_rx
port map(
B_ACC_in(i+1), res_vect(i-1), Za(i+1), C_vect_inf_a(i-1), temp_out(i-1), err1, S_prev(i-1),
120 B_ACC_out(i+1), res_out, S_ACC(i), S_ACC(i+1), C_vect_lat(i-N_bit+2), temp_in(i-1), err2
,
clk, clkA, clkB, clkC, clkD,
n_mag_vect(i+1), n_zones_vect(i+1), area_eff_vect(i+1), area_tot_vect(i+1),
Er_vect(i+1), Ec_vect(i+1));
123 end generate;
lat_element: if ((i>N_bit-2)and(i/=2*N_bit-2)) generate
FA:FA_ACC_lat_cent
126 port map(
Za(i+1), Zb(i+1), res_lat_in(i-N_bit+1), C_vect_lat(i-N_bit+1),
S_ACC(i+1), C_vect_lat(i-N_bit+2),
129 clk, clkA, clkB, clkC, clkD,
n_mag_vect(i+1), n_zones_vect(i+1), area_eff_vect(i+1), area_tot_vect(i+1),
Er_vect(i+1), Ec_vect(i+1));
end generate;
132 last_element: if (i=2*N_bit-2) generate
FA:FA_ACC_lat_sup
port map(
135 Z_lat_sup, res_lat_in(i-N_bit+1), C_vect_lat(i-N_bit+1),
S_ACC(i+1), Co_ACC,
clk, clkA, clkB, clkC, clkD,
138 n_mag_vect(i+1), n_zones_vect(i+1), area_eff_vect(i+1), area_tot_vect(i+1),
Er_vect(i+1), Ec_vect(i+1));
end generate;
end generate;
141 end behavior;

```

A.3.2 Serial-Parallel MAC

Here is reported the top entity **MAC_1D** of the Serial-Parallel MAC, together with its four components. This version of the MAC has been divided in four regions: body **MAC_1D.body**,

connections above MAC_1D_conn_above, connections below MAC_1D_conn_below, input B conditioning MAC_1D_input_cond.

Listing A.24. Top entity of the Serial-Parallel MAC: MAC_1D.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML_package.all;
6
entity MAC1D is
    port (DataA, DataB, Rst: in std_logic;
9         Result: out std_logic_vector (2*N_BIT-1 downto 0);

        clkA, clkB, clkC, clkD: in std_logic;
12

        n_mag: out natural := init_natural;
        n_zones: out natural := init_natural;
15        AREA_EFF: out natural;
        AREA_TOT: out natural;
        Er: out natural;
18        Ec: out natural);
end MAC1D;

21 architecture behavior of MAC1D is
    [...] — Components definitions

24    — Vectors of natural for magnets and cell count, area and energy evaluation
    type natural_vector is array (natural range <>) of natural;
    signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
        natural_vector (4 downto 1) := (others => init_natural);
27

    — in and out considered from the body
    signal DataA_prop_in: std_logic_vector ((2*N_BIT-2)-1 downto 0);
30    signal DataA_prop_out: std_logic_vector ((2*N_BIT-2)-1 downto 0);
    signal DataB_prop_in: std_logic_vector (3*(2*N_BIT-2)-1 downto 0);
    signal DataB_prop_out: std_logic_vector (2*(2*N_BIT-2)-1 downto 0);
33    signal Rst_prop_in, Rst_prop_out: std_logic_vector ((2*N_BIT-1)-1 downto 0);
    signal Res_prop_in: std_logic_vector (2*(2*N_BIT-1)-1 downto 0);
    signal Res_prop_out: std_logic_vector (3*(2*N_BIT-1)-1 downto 0);
36    signal InputCond2Body: std_logic;
    signal InputCond2ConnAbove: std_logic_vector (2*N_BIT-3 downto 0);

39 begin

    — SUM OF ARRAYS OF NATURAL ELEMENTS —————
42    — This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec
    — of every standard cell instantiated.
    — Results are given as outputs of this "PE_galois" component.
45    N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
        Ec_vect)
        variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
            natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
        variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
            := init_natural;

```

```

48   variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
sum_tot_Er, sum_tot_Ec: natural := init_natural;
begin
  n_nat_mag := n_mag_vect;
51   n_nat_zones := n_zones_vect;
  nat_area_eff := area_eff_vect;
  nat_area_tot := area_tot_vect;
54   nat_Er := Er_vect;
  nat_Ec := Ec_vect;

57   sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
sum_Ec:= 0;
  for i in 0 to n_mag_vect'length-1 loop
    sum_n_mag := sum_n_mag + n_nat_mag(i);
60    sum_n_zones := sum_n_zones + n_nat_zones(i);
    sum_area_eff := sum_area_eff + nat_area_eff(i);
    sum_area_tot := sum_area_tot + nat_area_tot(i);
63    sum_Er := sum_Er + nat_Er(i);
    sum_Ec := sum_Ec + nat_Ec(i);
  end loop;
66   sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
  sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
  sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
69   sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
  sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
  sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

72   n_mag <= sum_tot_n_mag;
  n_zones <= sum_tot_n_zones;
75   area_eff <= sum_tot_area_eff;
  area_tot <= sum_tot_area_tot;
  Er <= sum_tot_Er;
78   Ec <= sum_tot_Ec;
end process;

```

```

81   Body_block: MAC_1D_body port map(
  DataA => DataA,
84   DataB => InputCond2Body,
  Rst => Rst,
  Res_MSB => Result(2*N.BIT-1),

87   DataA_in_vect => DataA_prop_in((2*N.BIT-2)-2 downto 0),
  DataB_in_vect => DataB_prop_in,
90   DataA_out_vect => DataA_prop_out,
  DataB_out_vect => DataB_prop_out,

93   Rst_in_vect => Rst_prop_in,
  Rst_out_vect => Rst_prop_out,
  Res_in_vect => Res_prop_in,
96   Res_out_vect => Res_prop_out,

  clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
99   n_mag=>n_mag_vect(1), n_zones=>n_zones_vect(1), AREA_EFF=>AREA_EFF_vect(1), AREA_TOT=>
  AREA_TOT_vect(1), Er=>Er_vect(1), Ec=>Ec_vect(1));

Conn_above: MAC_1D_conn_above port map(
102  Inputs => InputCond2ConnAbove,
  DataA_in => DataA_prop_out,

```

```

105   DataA_out => DataA_prop_in ,
      DataB_in => DataB_prop_out ,
      DataB_out => DataB_prop_in ,
108   clkA=>clkA , clkB=>clkB , clkC=>clkC , clkD=>clkD ,
      n_mag=>n_mag_vect(2) , n_zones=>n_zones_vect(2) , AREA_EFF=>AREA_EFF_vect(2) , AREA_TOT=>
      AREA_TOT_vect(2) , Er=>Er_vect(2) , Ec=>Ec_vect(2) );

111 Conn_below: MAC_1D_conn_below port map(
      Outputs => Result(2*N_BIT-2 downto 0) ,
      Res_prop_in => Res_prop_out ,
114   Res_prop_out => Res_prop_in ,
      Rst_prop_in => Rst_prop_out ,
      Rst_prop_out => Rst_prop_in ,
117   clkA=>clkA , clkB=>clkB , clkC=>clkC , clkD=>clkD ,
      n_mag=>n_mag_vect(3) , n_zones=>n_zones_vect(3) , AREA_EFF=>AREA_EFF_vect(3) , AREA_TOT=>
      AREA_TOT_vect(3) , Er=>Er_vect(3) , Ec=>Ec_vect(3) );

120 Input_conditioning: MAC_1D_input_cond port map(
      Data_in => DataB ,
123   Data_out(2*N_BIT-3 downto 0) => InputCond2ConnAbove ,
      Data_out(2*N_BIT-2) => InputCond2Body ,
126   clkA=>clkA , clkB=>clkB , clkC=>clkC , clkD=>clkD ,
      n_mag=>n_mag_vect(4) , n_zones=>n_zones_vect(4) , AREA_EFF=>AREA_EFF_vect(4) , AREA_TOT=>
      AREA_TOT_vect(4) , Er=>Er_vect(4) , Ec=>Ec_vect(4) );

129 end behavior;

```

Listing A.25. Body of the Serial-Parallel MAC: MAC_1D_body.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML_package.all;
6
entity MAC_1D_body is
  port(DataA,DataB,Rst: in std_logic;
9     Res_MSB: out std_logic;

      DataA_in_vect: in std_logic_vector((2*N_BIT-2)-2 downto 0);
12   DataB_in_vect: in std_logic_vector(3*(2*N_BIT-2)-1 downto 0);
      DataA_out_vect: out std_logic_vector((2*N_BIT-2)-1 downto 0);
      DataB_out_vect: out std_logic_vector(2*(2*N_BIT-2)-1 downto 0);
15

      Rst_in_vect: in std_logic_vector((2*N_BIT-1)-1 downto 0);
      Rst_out_vect: out std_logic_vector((2*N_BIT-1)-1 downto 0);
18   Res_in_vect: in std_logic_vector(2*(2*N_BIT-1)-1 downto 0);
      Res_out_vect: out std_logic_vector(3*(2*N_BIT-1)-1 downto 0);
21

      clkA , clkB , clkC , clkD: in std_logic;

      n_mag: out natural := init_natural;
24   n_zones: out natural := init_natural;
      AREA_EFF: out natural;
      AREA_TOT: out natural;

```

```

27     Er: out natural;
    Ec: out natural);
end MAC_1D_body;

30
architecture behavior of MAC_1D_body is
    [...] — Components definitions
33
    — Vectors of natural for magnets and cell count, area and energy evaluation
    type natural_vector is array (natural range <>) of natural;
36    signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
        natural_vector (2*N.BIT downto 1) := (others => init_natural);
    type matrix_2Nx2 is array (2*N.BIT-1 downto 0) of std_logic_vector (1 downto 0);
    type matrix_2Nx3 is array (2*N.BIT-1 downto 0) of std_logic_vector (2 downto 0);
39    type matrix_2Nx4 is array (2*N.BIT downto 0) of std_logic_vector (3 downto 0);
    signal DataA_array, DataB_array, Carry_in_array, Carry_out_array, Result_out_array:
        std_logic_vector (2*N.BIT downto 0);
    signal DataA_v_in_array, Rst_v_in_array, Res_v_in_array, Rst_v_out_array, Res_v_out_array:
        std_logic_vector (2*N.BIT downto 0);
42    signal DataA_v_out_array: matrix_2Nx2;
    signal DataB_v_in_array, DataB_v_out_array: matrix_2Nx3;
    signal Res_prev_in_array, Res_prev_out_array: matrix_2Nx4;
45 begin
    — SUM OF ARRAYS OF NATURAL ELEMENTS —————
    — This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec
48    — of every standard cell instantiated.
    — Results are given as outputs of this "PE-galois" component.
    N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
        Ec_vect)
51        variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
            natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
        variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
            := init_natural;
        variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
            sum_tot_Er, sum_tot_Ec: natural := init_natural;
54        begin
            n_nat_mag := n_mag_vect;
            n_nat_zones := n_zones_vect;
57            nat_area_eff := area_eff_vect;
            nat_area_tot := area_tot_vect;
            nat_Er := Er_vect;
60            nat_Ec := Ec_vect;

            sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
            sum_Ec:= 0;
63            for i in 0 to n_mag_vect'length -1 loop
                sum_n_mag := sum_n_mag + n_nat_mag(i);
                sum_n_zones := sum_n_zones + n_nat_zones(i);
66                sum_area_eff := sum_area_eff + nat_area_eff(i);
                sum_area_tot := sum_area_tot + nat_area_tot(i);
                sum_Er := sum_Er + nat_Er(i);
69                sum_Ec := sum_Ec + nat_Ec(i);
            end loop;
            sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
72            sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
            sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
            sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
75            sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
            sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

```

```

78     n_mag <= sum_tot_n_mag;
    n_zones <= sum_tot_n_zones;
    area_eff <= sum_tot_area_eff;
81     area_tot <= sum_tot_area_tot;
    Er <= sum_tot_Er;
    Ec <= sum_tot_Ec;
84 end process;

DataA_array(0) <= DataA;
87 DataB_array(0) <= DataB;
Rst_v_in_array(0) <= Rst;
Res_MSB <= Result_out_array(2*N_BIT-1);

90 DataA_v_in_array(2*N_BIT-3 downto 1) <= DataA_in_vect;
Rst_v_in_array(2*N_BIT-1 downto 1) <= Rst_in_vect;
93 Rst_out_vect <= Rst_v_out_array(2*N_BIT-2 downto 0);

Prop_Above: for ind in 0 to 2*N_BIT-3 generate
96     DataA_out_vect(ind) <= DataA_v_out_array(ind)(0);
    DataB_out_vect(ind*2+1 downto ind*2) <= DataB_v_out_array(ind)(1 downto 0);
    DataB_v_in_array(ind) <= DataB_in_vect(ind*3+2 downto ind*3);
99 end generate;
Prop_Below: for ind2 in 0 to 2*N_BIT-2 generate
    Res_prev_in_array(ind2+1)(3 downto 2) <= Res_in_vect(ind2*2+1 downto ind2*2);
102     Res_out_vect(ind2*3+2 downto ind2*3) <= Res_prev_out_array(ind2+1)(3 downto 1);
end generate;

105 MAC_Body: for i in 0 to 2*N_BIT-1 generate
    MAC_1D_Body_map: MAC_1D_body_PE generic map(ELEMENT=>i)
    port map(
108         DataA=>DataA_array(i),
        DataB=>DataB_array(i),
        Carry_in=>Carry_in_array(i),
111         Carry_out=>Carry_out_array(i),
        clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
        Result_out=>Result_out_array(i),
114         -- other in/out
        DataA_v_in=>DataA_v_in_array(i),
        Rst_v_in=>Rst_v_in_array(i),
117         Res_v_in=>Res_v_in_array(i),
        DataB_v_in=>DataB_v_in_array(i),
        Res_prev_in=>Res_prev_in_array(i),
120
        Rst_v_out=>Rst_v_out_array(i),
        Res_v_out=>Res_v_out_array(i),
123         DataA_v_out=>DataA_v_out_array(i),
        DataB_v_out=>DataB_v_out_array(i),
        Res_prev_out=>Res_prev_out_array(i),
126
        n_mag=>n_mag_vect(i+1), n_zones=>n_zones_vect(i+1),
        Area_eff=>Area_eff_vect(i+1), Area_tot=>Area_tot_vect(i+1),
129         Er=>Er_vect(i+1), Ec=>Ec_vect(i+1));

Res_prev_in_array(i+1)(1) <= Result_out_array(i);
132 Res_prev_in_array(i+1)(0) <= Res_v_out_array(i);
Res_v_in_array(i) <= Res_prev_out_array(i+1)(0);
DataA_array(i+1) <= DataA_v_out_array(i)(1);
135 DataB_array(i+1) <= DataB_v_out_array(i)(2);
Carry_in_array(i+1) <= Carry_out_array(i);

```

```

138   end generate;
   end behavior;

```

Listing A.26. Preskew network of the Serial-Parallel MAC: MAC_1D_conn_above.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.MENML_package.all;
6
entity MAC_1D_conn_above is
  port (Inputs: in std_logic_vector (2*N_BIT-3 downto 0);
9    DataA_in: in std_logic_vector (2*N_BIT-3 downto 0);
    DataA_out: out std_logic_vector (2*N_BIT-3 downto 0);
    DataB_in: in std_logic_vector (2*(2*N_BIT-2)-1 downto 0);
12   DataB_out: out std_logic_vector (3*(2*N_BIT-2)-1 downto 0);
    clkA, clkB, clkC, clkD: in std_logic;

15   n_mag: out natural := init_natural;
    n_zones: out natural := init_natural;
    AREA_EFF: out natural;
18   AREA_TOT: out natural;
    Er: out natural;
    Ec: out natural);
21 end MAC_1D_conn_above;

architecture behavior of MAC_1D_conn_above is
24   [...] — Components definitions

   — Vectors of natural for magnets and cell count, area and energy evaluation
27   type natural_vector is array (natural range <>) of natural;
   signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
     natural_vector (N_CELLS_CONN_ABOVE+N_CELLS_CONN_TRIANGLE downto 0) := (others =>
       init_natural);

30   constant COLUMNS: integer := 2*N_BIT-2;
   type matrix_prop_vert is array ((COLUMNS/4+1)*(COLUMNS+(COLUMNS mod 4)/2) downto 0) of
     std_logic_vector (5 downto 0);
   signal sig_prop_vert: matrix_prop_vert;
33   signal sig_prop_horiz: std_logic_vector (((COLUMNS+1)*(COLUMNS+2)/2 -1) downto 0);

   signal sig_prop_triangle: std_logic_vector (((2*N_BIT-2)*(2*N_BIT-1)/2 -1) downto 0);
36 begin
   — SUM OF ARRAYS OF NATURAL ELEMENTS —————
39   — This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec
   — of every standard cell instantiated.
   — Results are given as outputs of this "PE_galois" component.
42   N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
     Ec_vect)
     variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
       natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
     variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
       := init_natural;
45     variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
       sum_tot_Er, sum_tot_Ec: natural := init_natural;
   begin

```

```

n_nat_mag := n_mag_vect;
48 n_nat_zones := n_zones_vect;
nat_area_eff := area_eff_vect;
nat_area_tot := area_tot_vect;
51 nat_Er := Er_vect;
nat_Ec := Ec_vect;

sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
sum_Ec:= 0;
for i in 0 to n_mag_vect'length -1 loop
    sum_n_mag := sum_n_mag + n_nat_mag(i);
57    sum_n_zones := sum_n_zones + n_nat_zones(i);
    sum_area_eff := sum_area_eff + nat_area_eff(i);
    sum_area_tot := sum_area_tot + nat_area_tot(i);
60    sum_Er := sum_Er + nat_Er(i);
    sum_Ec := sum_Ec + nat_Ec(i);
end loop;
63 sum_tot_n_mag := sum_n_mag * INTERCONNECT.OVERHEAD;
sum_tot_n_zones := sum_n_zones * INTERCONNECT.OVERHEAD;
sum_tot_area_eff := sum_area_eff * INTERCONNECT.OVERHEAD;
66 sum_tot_area_tot := sum_area_tot * INTERCONNECT.OVERHEAD;
sum_tot_Er := sum_Er * INTERCONNECT.OVERHEAD;
sum_tot_Ec := sum_Ec * INTERCONNECT.OVERHEAD;
69
n_mag <= sum_tot_n_mag;
n_zones <= sum_tot_n_zones;
72 area_eff <= sum_tot_area_eff;
area_tot <= sum_tot_area_tot;
Er <= sum_tot_Er;
75 Ec <= sum_tot_Ec;
end process;

```

```

78 Right_part: for col in 0 to COLUMNS-1 generate
    Select_Column0: if (col mod 4) = 0 generate
        Conn4_up: MAC_1D_c4_up port map(
81     sig_in => sig_prop_horiz((col+2)*(col+3)/2 -1),
        sig_out => sig_prop_horiz((col+1)*(col+2)/2 -1),
        sig_bottom_in=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(2 downto 0),
84     sig_bottom_out=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(5 downto 3),

        clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
87     n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
        n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
        AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
90     AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
        Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
        Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
93     end generate;
        Select_Column1: if (col mod 4) = 1 generate
            Conn8_up: MAC_1D_c8_up port map(
96     sig_in => sig_prop_horiz((col+2)*(col+3)/2 -1 downto (col+2)*(col+3)/2 -2),
        sig_out => sig_prop_horiz((col+1)*(col+2)/2 -1 downto (col+1)*(col+2)/2 -2),
        sig_bottom_in=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(2 downto 0),
99     sig_bottom_out=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(5 downto 3),

        clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
102     n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
        n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
        AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),

```



```

105 AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
108 end generate;
    Select_Column2:if (col mod 4) = 2 generate
Conn12_up: MAC_1D_c12_up port map(
111 sig_in => sig_prop_horiz((col+2)*(col+3)/2 -1 downto (col+2)*(col+3)/2 -3),
sig_out => sig_prop_horiz((col+1)*(col+2)/2 -1 downto (col+1)*(col+2)/2 -3),
sig_bottom_in=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(2 downto 0),
114 sig_bottom_out=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(5 downto 3),

clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
117 n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
120 AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
123 end generate;
    Select_Column3:if (col mod 4) = 3 generate
Conn16_up: MAC_1D_c16_up port map(
126 sig_in => sig_prop_horiz((col+2)*(col+3)/2 -1 downto (col+2)*(col+3)/2 -4),
sig_out => sig_prop_horiz((col+1)*(col+2)/2 -1 downto (col+1)*(col+2)/2 -4),
sig_bottom_in=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(2 downto 0),
129 sig_bottom_out=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(5 downto 3),

clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
132 n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
135 AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
138 end generate;
    Select_STD_el: if col/4 > 0 generate
For_smth: for N_cSTD_up in 0 to (col/4)-1 generate
141 ConnSTD_up: MAC_1D_cSTD_up port map(
sig_in => sig_prop_horiz(((col+1)*(col+2)/2 + 4*N_cSTD_up)+4 downto ((col+1)*(col+2)/2 + 4*N_cSTD_up)+1),
sig_out => sig_prop_horiz((col*(col+1)/2 + 4*N_cSTD_up)+3 downto (col*(col+1)/2 + 4*N_cSTD_up)),
144 sig_bottom_in => sig_prop_vert(col+(2*N.BIT-2)*N_cSTD_up)(2 downto 0),
sig_bottom_out=> sig_prop_vert(col+(2*N.BIT-2)*N_cSTD_up)(5 downto 3),
sig_top_in => sig_prop_vert(col+(2*N.BIT-2)*(N_cSTD_up+1))(5 downto 3),
147 sig_top_out => sig_prop_vert(col+(2*N.BIT-2)*(N_cSTD_up+1))(2 downto 0),

clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
150 n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_up),
n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_up),
AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_up),
153 AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_up),
Er=>Er_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_up),
Ec=>Ec_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_up));
156 end generate;
end generate;
sig_prop_vert(col)(2) <= DataA_in(2*N.BIT-3-col);
159 DataA_out(2*N.BIT-3-col) <= sig_prop_horiz(col*(col+1)/2);
sig_prop_vert(col)(1 downto 0) <= DataB_in((2*N.BIT-3-col)*2+1 downto (2*N.BIT-3-col)*2);

```

```

162   DataB_out((2*N.BIT-3-col)*3+2 downto (2*N.BIT-3-col)*3) <= sig_prop_vert(col)(5
      downto 3);
163   end generate;
164
165   Triangle: for col in 1 to 2*N.BIT-3 generate
166     Row_gen: for row in 1 to col generate
167       Green: if (((2*N.BIT-3)-(col-1)) mod 4) = 1 generate — green, phase D
168         Drow_col: long_wire generic map(D,row,(col+1)/2,LX,ZONE_H,ZONE_L)
169         port map(sig_prop_triangle(col*(col+1)/2+row),clkD,sig_prop_triangle(col*(col
170           -1)/2+row-1),n_mag_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),n_zones_vect(col*(
171             col-1)/2+row-1+N.CELLS.CONN_ABOVE),area_eff_vect(col*(col-1)/2+row-1+
172               N.CELLS.CONN_ABOVE),area_tot_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Er_vect(
173                 col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Ec_vect(col*(col-1)/2+row-1+
174                   N.CELLS.CONN_ABOVE));
175         end generate;
176       Yellow: if (((2*N.BIT-3)-(col-1)) mod 4) = 2 generate — yellow, phase A
177         Arow_col: long_wire generic map(A,row,(col+2)/2,LX,ZONE_H,ZONE_L)
178         port map(sig_prop_triangle(col*(col+1)/2+row),clkA,sig_prop_triangle(col*(col
179           -1)/2+row-1),n_mag_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),n_zones_vect(col*(
180             col-1)/2+row-1+N.CELLS.CONN_ABOVE),area_eff_vect(col*(col-1)/2+row-1+
181               N.CELLS.CONN_ABOVE),area_tot_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Er_vect(
182                 col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Ec_vect(col*(col-1)/2+row-1+
183                   N.CELLS.CONN_ABOVE));
184         end generate;
185       Pink: if (((2*N.BIT-3)-(col-1)) mod 4) = 3 generate — pink, phase B
186         Brow_col: long_wire generic map(B,row,(col+1)/2,LX,ZONE_H,ZONE_L)
187         port map(sig_prop_triangle(col*(col+1)/2+row),clkB,sig_prop_triangle(col*(col
188           -1)/2+row-1),n_mag_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),n_zones_vect(col*(
189             col-1)/2+row-1+N.CELLS.CONN_ABOVE),area_eff_vect(col*(col-1)/2+row-1+
190               N.CELLS.CONN_ABOVE),area_tot_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Er_vect(
191                 col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Ec_vect(col*(col-1)/2+row-1+
192                   N.CELLS.CONN_ABOVE));
193         end generate;
194       Cyan: if (((2*N.BIT-3)-(col-1)) mod 4) = 0 generate — cyan, phase C
195         Crow_col: long_wire generic map(C,row,(col+2)/2,LX,ZONE_H,ZONE_L)
196         port map(sig_prop_triangle(col*(col+1)/2+row),clkC,sig_prop_triangle(col*(col
197           -1)/2+row-1),n_mag_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),n_zones_vect(col*(
198             col-1)/2+row-1+N.CELLS.CONN_ABOVE),area_eff_vect(col*(col-1)/2+row-1+
199               N.CELLS.CONN_ABOVE),area_tot_vect(col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Er_vect(
200                 col*(col-1)/2+row-1+N.CELLS.CONN_ABOVE),Ec_vect(col*(col-1)/2+row-1+
201                   N.CELLS.CONN_ABOVE));
202         end generate;
203       First: if row = 1 generate
204         sig_prop_horiz((COLUMNS+1)*(COLUMNS+2)/2-col) <= sig_prop_triangle(col*(col-1)
205           /2+row-1);
206         end generate;
207       end generate;
208     end generate;
209
210     sig_prop_triangle(((2*N.BIT-2)*(2*N.BIT-1)/2-1) downto (2*N.BIT-3)*(2*N.BIT-2)/2+1)
211       <= Inputs(2*N.BIT-3 downto 1);
212     sig_prop_horiz((COLUMNS+1)*(COLUMNS+2)/2-COLUMNS)<= Inputs(0);
213   end behavior;

```

Listing A.27. Deskew network of the Serial-Parallel MAC: MAC_1D_conn_below.

```

1 library ieee;
2 use ieee.std_logic_1164.all;

```

```

use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
5 use work.MENML-package.all;

entity MAC_1D_conn_below is
8   port(Outputs: out std_logic_vector(2*N_BIT-2 downto 0);
        Res_prop_in: in std_logic_vector(3*(2*N_BIT-1)-1 downto 0);
        Res_prop_out: out std_logic_vector(2*(2*N_BIT-1)-1 downto 0);
11      Rst_prop_in: in std_logic_vector(2*N_BIT-2 downto 0);
        Rst_prop_out: out std_logic_vector(2*N_BIT-2 downto 0);
        clkA, clkB, clkC, clkD: in std_logic;

14      n_mag: out natural := init_natural;
        n_zones: out natural := init_natural;
17      AREA_EFF: out natural;
        AREA_TOT: out natural;
        Er: out natural;
20      Ec: out natural);
end MAC_1D_conn_below;

23 architecture behavior of MAC_1D_conn_below is
    [...] — Components definitions

26 — Vectors of natural for magnets and cell count, area and energy evaluation
    type natural_vector is array (natural range <>) of natural;
    signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
        natural_vector (N_CELLS.CONN_BELOW+N_CELLS.CONN_TAIL downto 0) := (others =>
            init_natural);
29    constant COLUMNS: integer := 2*N_BIT-2;
    type matrix_prop_vert is array ((COLUMNS/4+1)*(COLUMNS+(COLUMNS mod 4)/2) downto 0) of
        std_logic_vector (5 downto 0);
    signal sig_prop_vert: matrix_prop_vert;
32    signal sig_prop_horiz: std_logic_vector(((COLUMNS+1)*(COLUMNS+2)/2 -1) downto 0);
    type matrix_prop_tail is array ((2*N_BIT-1)-1 downto 0) of std_logic_vector ((4+N_BIT)
        *2-1 downto 0);
    signal sig_prop_tail: matrix_prop_tail;

35 begin
    — SUM OF ARRAYS OF NATURAL ELEMENTS —————
38    — This process sums up the values of n_mag,n_zones,area_eff,area_tot,Er,Ec
    — of every standard cell instantiated.
    — Results are given as outputs of this "PE-galois" component.
41    N_mag_sum: process (n_mag_vect,n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
        Ec_vect)
        variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
            natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
        variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
            := init_natural;
44        variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
            sum_tot_Er, sum_tot_Ec: natural := init_natural;
    begin
        n_nat_mag := n_mag_vect;
        n_nat_zones := n_zones_vect;
47        nat_area_eff := area_eff_vect;
        nat_area_tot := area_tot_vect;
50        nat_Er := Er_vect;
        nat_Ec := Ec_vect;

```

```

53 sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
sum_Ec:= 0;
for i in 0 to n_mag_vect'length -1 loop
56   sum_n_mag := sum_n_mag + n_nat_mag(i);
   sum_n_zones := sum_n_zones + n_nat_zones(i);
   sum_area_eff := sum_area_eff + nat_area_eff(i);
   sum_area_tot := sum_area_tot + nat_area_tot(i);
59   sum_Er := sum_Er + nat_Er(i);
   sum_Ec := sum_Ec + nat_Ec(i);
end loop;
62 sum_tot_n_mag := sum_n_mag * INTERCONNECT.OVERHEAD;
sum_tot_n_zones := sum_n_zones * INTERCONNECT.OVERHEAD;
sum_tot_area_eff := sum_area_eff * INTERCONNECT.OVERHEAD;
65 sum_tot_area_tot := sum_area_tot * INTERCONNECT.OVERHEAD;
sum_tot_Er := sum_Er * INTERCONNECT.OVERHEAD;
sum_tot_Ec := sum_Ec * INTERCONNECT.OVERHEAD;
68
n_mag <= sum_tot_n_mag;
n_zones <= sum_tot_n_zones;
71 area_eff <= sum_tot_area_eff;
area_tot <= sum_tot_area_tot;
Er <= sum_tot_Er;
74 Ec <= sum_tot_Ec;
end process;

```

```

77 Left_part: for col in 0 to COLUMNS-1 generate
  Select_c4: if (col mod 4) = 0 generate
    Conn4_down: MAC_1D_c4_down port map(
80       sig_in => sig_prop_horiz((col+1)*(col+2)/2 -1),
       sig_out => sig_prop_horiz((col+2)*(col+3)/2 -1),
       sig_top_in=> sig_prop_vert(col+(2*N_BIT-2)*(col/4))(2 downto 0),
83       sig_top_out=> sig_prop_vert(col+(2*N_BIT-2)*(col/4))(5 downto 3),

       clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
86       n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
89       AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
92   end generate;
  Select_c8: if (col mod 4) = 1 generate
    Conn8_down: MAC_1D_c8_down port map(
95       sig_in => sig_prop_horiz((col+2)*(col+1)/2 -1 downto (col+2)*(col+1)/2 -2),
       sig_out => sig_prop_horiz((col+3)*(col+2)/2 -1 downto (col+3)*(col+2)/2 -2),
       sig_top_in=> sig_prop_vert(col+(2*N_BIT-2)*(col/4))(2 downto 0),
98       sig_top_out=> sig_prop_vert(col+(2*N_BIT-2)*(col/4))(5 downto 3),

       clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
101      n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
104      AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
       Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
107   end generate;
  Select_c12: if (col mod 4) = 2 generate
    Conn12_down: MAC_1D_c12_down port map(
110      sig_in => sig_prop_horiz((col+2)*(col+1)/2 -1 downto (col+2)*(col+1)/2 -3),

```

```

113   sig_out => sig_prop_horiz((col+3)*(col+2)/2 -1 downto (col+3)*(col+2)/2 -3),
   sig_top_in=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(2 downto 0),
   sig_top_out=> sig_prop_vert(col+(2*N.BIT-2)*(col/4))(5 downto 3),

116   clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
   n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
119   AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
122 end generate;
   Select_c16: if (col mod 4) = 3 generate
   Conn16_down: MAC_1D_c16_down port map(
125   sig_in => sig_prop_horiz((col+2)*(col+1)/2 -1 downto (col+2)*(col+1)/2 -4),
   sig_out => sig_prop_horiz((col+3)*(col+2)/2 -1 downto (col+3)*(col+2)/2 -4),
   sig_top_in=> sig_prop_vert(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1)
   (2 downto 0),
128   sig_top_out=> sig_prop_vert(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1)
   -1)(5 downto 3),

   clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
131   n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
134   AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   Er=>Er_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1),
   Ec=>Ec_vect(2*(col/4)*(col/4+1) + (((col) mod 4)+1)*(col/4+1) -1));
137 end generate;
   Select_cSTD: if col/4 > 0 generate
   Select_N_cSTD: for N_cSTD_down in 0 to (col/4)-1 generate
140   ConnSTD_down: MAC_1D_cSTD_down port map(
   sig_in => sig_prop_horiz((col*(col+1)/2 + 4*N_cSTD_down)+3 downto (col*(col+1)
   /2 + 4*N_cSTD_down)),
   sig_out => sig_prop_horiz(((col+1)*(col+2)/2 + 4*N_cSTD_down)+4 downto ((col
   +1)*(col+2)/2 + 4*N_cSTD_down)+1),
143   sig_top_in => sig_prop_vert(col+(2*N.BIT-2)*N_cSTD_down)(2 downto 0),
   sig_top_out=> sig_prop_vert(col+(2*N.BIT-2)*N_cSTD_down)(5 downto 3),
   sig_bottom_in => sig_prop_vert(col+(2*N.BIT-2)*(N_cSTD_down+1))(5 downto 3),
146   sig_bottom_out => sig_prop_vert(col+(2*N.BIT-2)*(N_cSTD_down+1))(2 downto 0),

   clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
149   n_mag=>n_mag_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_down)
   ,
   n_zones=>n_zones_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) +
   N_cSTD_down),
   AREA_EFF=>area_eff_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) +
   N_cSTD_down),
152   AREA_TOT=>area_tot_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) +
   N_cSTD_down),
   Er=>Er_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_down),
   Ec=>Ec_vect(2*(col/4)*(col/4+1) + ((col) mod 4)*(col/4+1) + N_cSTD_down));
155   end generate;
end generate;

158 sig_prop_vert(col)(2 downto 0) <= Res_prop_in(col*3+2 downto col*3);
sig_prop_horiz(col*(col+1)/2) <= Rst_prop_in(col);

161 Rst_prop_out(col) <= sig_prop_vert(col)(3);

```

```

164   Res_prop_out(col*2+1 downto col*2) <= sig_prop_vert(col)(5 downto 4);
end generate;

167 Right_part: for row in 2*N_BIT-1 downto 1 generate
  Select_col: for col in 1 to 4+(row+1)/2 generate
173     Select_row: if row<2*N_BIT-1 generate
      Yellow: if (col mod 2)=1 and (row mod 2)=1 generate — yellow
        Arow_col: double_wire_horiz generic map(A,row,(4+N_BIT)-(col-1),ZONE_H,ZONE_L)
170         port map(sig_prop_tail(row-1)(col*2-1), sig_prop_tail(row)(col*2-1), clkA,
          sig_prop_tail(row-1)((col-1)*2), sig_prop_tail(row)((col-1)*2),
            n_mag_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), n_zones_vect((row-1)
              *(4+N_BIT)+col+N_CELLS.CONN_BELOW), area_eff_vect((row-1)*(4+N_BIT)+col+
                N_CELLS.CONN_BELOW), area_tot_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Er_vect
                  ((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Ec_vect((row-1)*(4+N_BIT)+col+
                    N_CELLS.CONN_BELOW));
        end generate;
173     Green: if (col mod 2)=1 and (row mod 2)=0 generate — green
      Drow_col: double_wire_horiz generic map(D,row,(4+N_BIT)-(col-1),ZONE_H,ZONE_L)
        port map(sig_prop_tail(row-1)(col*2), sig_prop_tail(row)(col*2), clkD,
176         sig_prop_tail(row-1)(col*2-1), sig_prop_tail(row)(col*2-1),
          n_mag_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), n_zones_vect((row-1)
            *(4+N_BIT)+col+N_CELLS.CONN_BELOW), area_eff_vect((row-1)*(4+N_BIT)+col+
              N_CELLS.CONN_BELOW), area_tot_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Er_vect
                ((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Ec_vect((row-1)*(4+N_BIT)+col+
                  N_CELLS.CONN_BELOW));
        end generate;
179     Cyan: if (col mod 2)=0 and (row mod 2)=1 generate — cyan
      Crow_col: double_wire_horiz generic map(C,row,(4+N_BIT)-(col-1),ZONE_H,ZONE_L)
        port map(sig_prop_tail(row-1)(col*2-1), sig_prop_tail(row)(col*2-1), clkC,
182         sig_prop_tail(row-1)((col-1)*2), sig_prop_tail(row)((col-1)*2),
          n_mag_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), n_zones_vect((row-1)
            *(4+N_BIT)+col+N_CELLS.CONN_BELOW), area_eff_vect((row-1)*(4+N_BIT)+col+
              N_CELLS.CONN_BELOW), area_tot_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Er_vect
                ((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Ec_vect((row-1)*(4+N_BIT)+col+
                  N_CELLS.CONN_BELOW));
        end generate;
185     Pink: if (col mod 2)=0 and (row mod 2)=0 generate — pink
      Brow_col: double_wire_horiz generic map(B,row,(4+N_BIT)-(col-1),ZONE_H,ZONE_L)
        port map(sig_prop_tail(row-1)(col*2), sig_prop_tail(row)(col*2), clkB,
188         sig_prop_tail(row-1)(col*2-1), sig_prop_tail(row)(col*2-1),
          n_mag_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), n_zones_vect((row-1)
            *(4+N_BIT)+col+N_CELLS.CONN_BELOW), area_eff_vect((row-1)*(4+N_BIT)+col+
              N_CELLS.CONN_BELOW), area_tot_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Er_vect
                ((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Ec_vect((row-1)*(4+N_BIT)+col+
                  N_CELLS.CONN_BELOW));
        end generate;
191     Last_row: if row=2*N_BIT-1 generate
      Yellow_el: if (col mod 2) = 1 generate — yellow
        Arow_col: short_wire_horiz generic map(A,row,(4+N_BIT)-(col-1),UP,ZONE_H,
          ZONE_L)
        port map(sig_prop_tail(row-1)(col*2-1), clkA, sig_prop_tail(row-1)((col-1)*2),
194         n_mag_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), n_zones_vect((row-1)*(4+N_BIT)+
          col+N_CELLS.CONN_BELOW), area_eff_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW),
          area_tot_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW), Er_vect((row-1)*(4+N_BIT)+
          col+N_CELLS.CONN_BELOW), Ec_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW));
        end generate;
      Cyan_el: if (col mod 2) = 0 generate — cyan

```

```

Crow_col: short_wire_horiz generic map(C,row,(4+N_BIT)-(col-1),UP,ZONE_H,
ZONE_L)
197   port map(sig_prop_tail(row-1)(col*2-1),clkC,sig_prop_tail(row-1)((col-1)*2),
n_mag_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW),n_zones_vect((row-1)*(4+N_BIT)+
col+N_CELLS.CONN_BELOW),area_eff_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW),
area_tot_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW),Er_vect((row-1)*(4+N_BIT)+
col+N_CELLS.CONN_BELOW),Ec_vect((row-1)*(4+N_BIT)+col+N_CELLS.CONN_BELOW));
   end generate;
   end generate;
200   end generate;
   Outputs(2*N_BIT-row-1) <= sig_prop_tail(row-1)(0);
end generate;
203
D0_col: short_wire_horiz generic map(D,0,4+N_BIT,DOWN,ZONE_H,ZONE_L)
port map(sig_prop_tail(0)(2),clkD,sig_prop_tail(0)(1),n_mag_vect(6+N_CELLS.CONN_BELOW)
,n_zones_vect(6+N_CELLS.CONN_BELOW),area_eff_vect(6+N_CELLS.CONN_BELOW),
area_tot_vect(6+N_CELLS.CONN_BELOW),Er_vect(6+N_CELLS.CONN_BELOW),Ec_vect(6+
N_CELLS.CONN_BELOW));
206
Sig_Propagation: for i in 1 to 2*N_BIT-2 generate
   sig_prop_tail(i)(9+i) <= sig_prop_horiz(COLUMNS*(COLUMNS+1)/2 +i);
209 end generate;

sig_prop_tail(0)(9) <= Rst_prop_in(2*N_BIT-2);
212 Rst_prop_out(2*N_BIT-2) <= sig_prop_tail(0)(8);
sig_prop_tail(0)(7) <= Res_prop_in(6*N_BIT-6);
sig_prop_tail(0)(5) <= Res_prop_in(6*N_BIT-5);
215 sig_prop_tail(0)(3) <= Res_prop_in(6*N_BIT-4);
Res_prop_out(4*N_BIT-4) <= sig_prop_tail(0)(6);
Res_prop_out(4*N_BIT-3) <= sig_prop_tail(0)(4);
218 end behavior;

```

Listing A.28. Circuit for distribution of the input B to the Serial-Parallel MAC:

MAC_1D_input_cond.

```

1 library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
4 use ieee.std_logic_arith.all;
use work.MENML_package.all;

7 entity MAC_1D_input_cond is
   port(Data_in: in std_logic;
10       Data_out: out std_logic_vector(2*N_BIT-2 downto 0);
       clkA, clkB, clkC, clkD: in std_logic;

       n_mag: out natural := init_natural;
13       n_zones: out natural := init_natural;
       AREA_EFF: out natural;
       AREA_TOT: out natural;
16       Er: out natural;
       Ec: out natural);
end entity MAC_1D_input_cond;
19
architecture struct of MAC_1D_input_cond is
   [...] — Components definitions
22
   — Vectors of natural for magnets and cell count, area and energy evaluation

```

```

25 type natural_vector is array (natural range <>) of natural;
signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
    natural_vector (N_BIT-1 downto 1) := (others => init_natural);
signal propagation: std_logic_vector(N_BIT*3-1 downto 0);

28 begin
    -- SUM OF ARRAYS OF NATURAL ELEMENTS -----
    -- This process sums up the values of n_mag,n_zones,area_eff,area_tot,Er,Ec of every
    -- standard cell instantiated.
31 -- Results are given as outputs of this "PE-galois" component.
N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
    Ec_vect)
    variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
        natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
34 variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
    := init_natural;
    variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
        sum_tot_Er, sum_tot_Ec: natural := init_natural;
begin
37     n_nat_mag := n_mag_vect;
        n_nat_zones := n_zones_vect;
        nat_area_eff := area_eff_vect;
40     nat_area_tot := area_tot_vect;
        nat_Er := Er_vect;
        nat_Ec := Ec_vect;

43     sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
        sum_Ec:= 0;
        for i in 0 to n_mag_vect'length -1 loop
46             sum_n_mag := sum_n_mag + n_nat_mag(i);
                sum_n_zones := sum_n_zones + n_nat_zones(i);
                sum_area_eff := sum_area_eff + nat_area_eff(i);
49             sum_area_tot := sum_area_tot + nat_area_tot(i);
                sum_Er := sum_Er + nat_Er(i);
                sum_Ec := sum_Ec + nat_Ec(i);
52         end loop;
        sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
        sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
55     sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
        sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
        sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
58     sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

        n_mag <= sum_tot_n_mag;
61     n_zones <= sum_tot_n_zones;
        area_eff <= sum_tot_area_eff;
        area_tot <= sum_tot_area_tot;
64     Er <= sum_tot_Er;
        Ec <= sum_tot_Ec;
    end process;

67 -----
propagation(1) <= Data_in;
Data_out(2*N_BIT-2) <= propagation(N_BIT*3-2);

70 Input_Conditioning: for i in 0 to N_BIT-2 generate -- 0 parte dal basso
    InCondElement: MAC_1D.input_cond_element generic map (i)
73     port map
        ( prop_vect_in_up => propagation(i*3+1 downto i*3),
          prop_vect_out_up => propagation(i*3+2),

```



```

76   prop_vect_in_down => propagation((i+1)*3+2),
   prop_vect_out_down => propagation((i+1)*3+1 downto (i+1)*3),
   Outputs(0) => Data_out(2*(N_BIT-2)+1-(2*i)),
79   Outputs(1) => Data_out(2*(N_BIT-2)+1-(2*i+1)),

   clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
82   n_mag=>n_mag_vect(i+1),
   n_zones=>n_zones_vect(i+1),
   AREA_EFF=>area_eff_vect(i+1),
85   AREA_TOT=>area_tot_vect(i+1),
   Er=>Er_vect(i+1),
   Ec=>Ec_vect(i+1));
88   end generate;
   end architecture;

```

A.3.3 Serial MAC

This section contains the top entity MAC_OD_8bit of the 8-bit Serial MAC, together with the listing of its shared Adder MAC_OD_Adder_8bit.

Listing A.29. Top entity of the 8-bit Serial MAC: MAC_OD_8bit.

```

1  library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
4  use ieee.std_logic_arith.all;
   use work.MENML_package.all;

7  entity MAC_OD_8bit is
   port(DataA,DataB: in std_logic_vector(N_BIT-1 downto 0);
        Feedback_ctrl,Carry_rst: in std_logic_vector(N_BIT-1 downto 0);
10   Ctrl_results,Acc_rst,Rst_acc_new,Carry_rst_shared_FA: in std_logic;
        Results: out std_logic_vector(N_BIT-1 downto 0);
        Results_serial: out std_logic;

13   clkA, clkB, clkC, clkD: in std_logic;
        n_mag: out natural := init_natural;
16   n_zones: out natural := init_natural;
        AREA_EFF: out natural;
        AREA_TOT: out natural;
19   Er: out natural;
        Ec: out natural);
22 end MAC_OD_8bit;

   architecture behavior of MAC_OD_8bit is
   [...] — Components definition

25   type natural_vector is array (natural range <>) of natural;
   signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
        natural_vector(N_BIT+1 downto 1) := (others => init_natural);
28   signal Acc_in,Acc_out,Partials_out: std_logic;
   type matrix is array(N_BIT downto 0) of std_logic_vector(13 downto 0);
   signal prop_matrix: matrix;
31

```

```

begin
  prop_matrix(0)(2) <= Ctrl_results;
34  prop_matrix(N_BIT)(8) <= Acc_in;
  prop_matrix(N_BIT)(9) <= Rst_acc_new;
  Acc_out <= prop_matrix(N_BIT)(0);
37  Partial_out <= prop_matrix(N_BIT)(1);

  El: for i in 0 to N_BIT-1 generate
40    MAC.Body: MAC_0D_body_8bit generic map(ELEMENT => i)
      port map(
        DataA=>DataA(i), DataB=>DataB(i),
43        Feedback_ctrl=>Feedback_ctrl(i), Carry_rst=>Carry_rst(i),
        Result=>Results(i),

46        prop_left_in => prop_matrix(i)(7 downto 0),
        prop_right_in => prop_matrix(i+1)(13 downto 8),
        prop_left_out => prop_matrix(i)(13 downto 8),
49        prop_right_out => prop_matrix(i+1)(7 downto 0),

        clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
52        n_mag=>n_mag_vect(i+1), n_zones=>n_zones_vect(i+1),
        Area_eff=>Area_eff_vect(i+1), Area_tot=>Area_tot_vect(i+1),
        Er=>Er_vect(i+1), Ec=>Ec_vect(i+1)
55      );
  end generate;

58  Shared_FA: MAC_0D_Adder_8bit
  port map(
    Partial_out=>Partial_out, Acc_out=>Acc_out,
61    Acc_rst=>Acc_rst, Carry_rst=>Carry_rst_shared_FA,
    Acc_in=>Acc_in, Results_serial=>Results_serial,
    clkA=>clkA, clkB=>clkB, clkC=>clkC, clkD=>clkD,
64    n_mag=>n_mag_vect(N_BIT+1), n_zones=>n_zones_vect(N_BIT+1),
    Area_eff=>Area_eff_vect(N_BIT+1), Area_tot=>Area_tot_vect(N_BIT+1),
    Er=>Er_vect(N_BIT+1), Ec=>Ec_vect(N_BIT+1)
67  );

  -- SUM OF ARRAYS OF NATURAL ELEMENTS --
70  -- This process sums up the values of n_mag, n_zones, area_eff, area_tot, Er, Ec
  -- of every standard cell instantiated.
  -- Results are given as outputs of this "PE_galois" component.
73  N_mag_sum: process (n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
    Ec_vect)
    variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
      natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
    variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
      := init_natural;
76    variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
      sum_tot_Er, sum_tot_Ec: natural := init_natural;
  begin
    n_nat_mag := n_mag_vect;
79    n_nat_zones := n_zones_vect;
    nat_area_eff := area_eff_vect;
    nat_area_tot := area_tot_vect;
82    nat_Er := Er_vect;
    nat_Ec := Ec_vect;

85    sum_n_mag := 0; sum_n_zones := 0; sum_area_eff := 0;
    sum_area_tot := 0; sum_Er := 0; sum_Ec := 0;

```

```

88   for i in 0 to n_mag_vect'length -1 loop
      sum_n_mag := sum_n_mag + n_nat_mag(i);
      sum_n_zones := sum_n_zones + n_nat_zones(i);
      sum_area_eff := sum_area_eff + nat_area_eff(i);
91     sum_area_tot := sum_area_tot + nat_area_tot(i);
      sum_Er := sum_Er + nat_Er(i);
      sum_Ec := sum_Ec + nat_Ec(i);
94   end loop;
      sum_tot_n_mag := sum_n_mag * INTERCONNECT_OVERHEAD;
      sum_tot_n_zones := sum_n_zones * INTERCONNECT_OVERHEAD;
97     sum_tot_area_eff := sum_area_eff * INTERCONNECT_OVERHEAD;
      sum_tot_area_tot := sum_area_tot * INTERCONNECT_OVERHEAD;
      sum_tot_Er := sum_Er * INTERCONNECT_OVERHEAD;
100    sum_tot_Ec := sum_Ec * INTERCONNECT_OVERHEAD;

      n_mag <= sum_tot_n_mag;
103    n_zones <= sum_tot_n_zones;
      area_eff <= sum_tot_area_eff;
      area_tot <= sum_tot_area_tot;
106    Er <= sum_tot_Er;
      Ec <= sum_tot_Ec;
      end process;
109 end architecture;

```

Listing A.30. Shared adder: MAC_0D_Adder_8bit.

```

library ieee;
2 use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
  use ieee.std_logic_arith.all;
5 use work.MENML_package.all;

entity MAC_0D_Adder_8bit is
8   port(Partials_out, Acc_out: in std_logic;
      Acc_rst, Carry_rst: in std_logic;
      Acc_in, Results_serial: out std_logic;
11
      clkA, clkB, clkC, clkD: in std_logic;
      n_mag: out natural := init_natural;
14     n_zones: out natural := init_natural;
      AREA_EFF: out natural;
      AREA_TOT: out natural;
17     Er: out natural;
      Ec: out natural);
end MAC_0D_Adder_8bit;
20

architecture behavior of MAC_0D_Adder_8bit is
  [...] -- Components definition
23
  -- Vectors of natural for magnets and cell count, area and energy evaluation
  type natural_vector is array (natural range <>) of natural;
26   signal n_mag_vect, n_zones_vect, area_eff_vect, area_tot_vect, Er_vect, Ec_vect:
      natural_vector (49 downto 1) := (others => init_natural);
  -- Connections among cells
  signal D1_1d, D1_1d2, B1_2d, D1_3d, B1_4d, A2_1d, C2_2d, C2_2d2, A2_3d, A2_3d2, C2_4d, C2_4d2,
      A2_5d: std_logic;
29   signal B3_1d, B3_1d2, D3_2d, D3_2d2, B3_3d, B3_3d2, D3_4d, D3_4d2, C4_1d, A4_2d, A4_2d2, C4_3d,
      C4_3d2, A4_4d, A4_4d2, C4_5d: std_logic;

```

```

32 signal D5_1d,D5_1d2,B5_2d,B5_2d2,D5_3d,D5_3d2,B5_4d,B5_4d2,C6_1d,C6_1d2,A6_2d,A6_2d2,
    C6_3d,A6_4d,A6_4d2: std_logic;
33 signal D7_1d,D7_1d2,B7_2d,B7_2d2,D7_3d,D7_3d2,B7_4d,B7_4d2,C8_1d,A8_2d,A8_2d2,C8_3d,
    C8_3d2,A8_4d,A8_4d2,C8_5d,C8_5d2: std_logic;
34 signal D9_1d,D9_1d2,B9_2d,B9_2d2,D9_3d,D9_3d2,B9_4d,B9_4d2,D9_5d,A10_1d,C10_2d,A10_3d,
    C10_4d,A10_5d,A10_5d2: std_logic;
35 signal D9_5q2,A10_1q,A10_5q2: std_logic;

36 begin
37   -- SUM OF ARRAYS OF NATURAL ELEMENTS -----
38   -- This process sums up the values of n_mag,n_zones,area_eff,area_tot,Er,Ec
39   -- of every standard cell instantiated.
40   -- Results are given as outputs of this "PE_galois" component.
41   N_mag_sum: process (n_mag_vect,n_zones_vect, area_eff_vect, area_tot_vect, Er_vect,
    Ec_vect)
42     variable n_nat_mag, n_nat_zones, nat_area_eff, nat_area_tot, nat_Er, nat_Ec:
    natural_vector (n_mag_vect'length-1 downto 0) := (others => init_natural);
43     variable sum_n_mag, sum_n_zones, sum_area_eff, sum_area_tot, sum_Er, sum_Ec: natural
    := init_natural;
44     variable sum_tot_n_mag, sum_tot_n_zones, sum_tot_area_eff, sum_tot_area_tot,
    sum_tot_Er, sum_tot_Ec: natural := init_natural;
45   begin
46     n_nat_mag := n_mag_vect;
47     n_nat_zones := n_zones_vect;
48     nat_area_eff := area_eff_vect;
49     nat_area_tot := area_tot_vect;
50     nat_Er := Er_vect;
51     nat_Ec := Ec_vect;

52     sum_n_mag := 0; sum_n_zones := 0; sum_area_eff:= 0; sum_area_tot:= 0; sum_Er:= 0;
    sum_Ec:= 0;
53     for i in 0 to n_mag_vect'length -1 loop
54       sum_n_mag := sum_n_mag + n_nat_mag(i);
55       sum_n_zones := sum_n_zones + n_nat_zones(i);
56       sum_area_eff := sum_area_eff + nat_area_eff(i);
57       sum_area_tot := sum_area_tot + nat_area_tot(i);
58       sum_Er := sum_Er + nat_Er(i);
59       sum_Ec := sum_Ec + nat_Ec(i);
60     end loop;
61     sum_tot_n_mag := sum_n_mag * INTERCONNECT.OVERHEAD;
62     sum_tot_n_zones := sum_n_zones * INTERCONNECT.OVERHEAD;
63     sum_tot_area_eff := sum_area_eff * INTERCONNECT.OVERHEAD;
64     sum_tot_area_tot := sum_area_tot * INTERCONNECT.OVERHEAD;
65     sum_tot_Er := sum_Er * INTERCONNECT.OVERHEAD;
66     sum_tot_Ec := sum_Ec * INTERCONNECT.OVERHEAD;

67     n_mag <= sum_tot_n_mag;
68     n_zones <= sum_tot_n_zones;
69     area_eff <= sum_tot_area_eff;
70     area_tot <= sum_tot_area_tot;
71     Er <= sum_tot_Er;
72     Ec <= sum_tot_Ec;
73   end process;

74   -----

77   C6_1d <= Partial_out;
78   C8_1d <= Acc_out;
79   A10_5d2 <= Acc_rst;
80   D1_1d <= Carry_rst;

```

```

Acc.in <= A10.1q;
Results_serial <= A10.5q2;

83  -- STANDARD CELLS INSTANTIATIONS -----
86  -- Labels have the following form: (letter)(number)_(number)
86  -- The letter is the clock phase, first number the relative row within the PE,
86  -- second number the relative column within the PE
86  -- In order to assign signals *d or *d2 to a cell I follow this sequence:
89  -- up_left, up_right, down_left, down_right

--Row1
92  D1.1: and_wire_rx generic map(D,1,1,DOWN,ZONE_H,ZONE_L)
      port map(D1.1d,D1.1d2,clkD,A2.1d,n_mag_vect(1),n_zones_vect(1),area_eff_vect(1),
      area_tot_vect(1),Er_vect(1),Ec_vect(1));
      B1.2: short_wire_horiz generic map(B,1,2,DOWN,ZONE_H,ZONE_L)
95      port map(B1.2d,clkB,C2.2d,n_mag_vect(2),n_zones_vect(2),area_eff_vect(2),
      area_tot_vect(2),Er_vect(2),Ec_vect(2));
      D1.3: short_wire_horiz generic map(D,1,3,DOWN,ZONE_H,ZONE_L)
      port map(D1.3d,clkD,A2.3d,n_mag_vect(3),n_zones_vect(3),area_eff_vect(3),
      area_tot_vect(3),Er_vect(3),Ec_vect(3));
98      B1.4: short_wire_horiz generic map(B,1,4,DOWN,ZONE_H,ZONE_L)
      port map(B1.4d,clkB,C2.4d,n_mag_vect(4),n_zones_vect(4),area_eff_vect(4),
      area_tot_vect(4),Er_vect(4),Ec_vect(4));
--Row2
101  A2.1: short_wire_vert generic map(A,2,1,RX,ZONE_H,ZONE_L)
      port map(A2.1d,clkA,B3.1d,n_mag_vect(6),n_zones_vect(6),area_eff_vect(6),
      area_tot_vect(6),Er_vect(6),Ec_vect(6));
      C2.2: double_wire_horiz generic map(C,2,2,ZONE_H,ZONE_L)
104      port map(C2.2d,C2.2d2,clkC,D1.1d2,D3.2d,n_mag_vect(7),n_zones_vect(7),area_eff_vect
      (7),area_tot_vect(7),Er_vect(7),Ec_vect(7));
      A2.3: double_wire_horiz generic map(A,2,3,ZONE_H,ZONE_L)
      port map(A2.3d,A2.3d2,clkA,B1.2d,B3.3d,n_mag_vect(8),n_zones_vect(8),area_eff_vect
      (8),area_tot_vect(8),Er_vect(8),Ec_vect(8));
107      C2.4: double_wire_horiz generic map(C,2,4,ZONE_H,ZONE_L)
      port map(C2.4d,C2.4d2,clkC,D1.3d,D3.4d,n_mag_vect(9),n_zones_vect(9),area_eff_vect
      (9),area_tot_vect(9),Er_vect(9),Ec_vect(9));
      A2.5: short_wire_vert generic map(A,2,5,LX,ZONE_H,ZONE_L)
110      port map(A2.5d,clkA,B1.4d,n_mag_vect(10),n_zones_vect(10),area_eff_vect(10),
      area_tot_vect(10),Er_vect(10),Ec_vect(10));
--Row3
      B3.1: double_wire_vert generic map(B,3,1,ZONE_H,ZONE_L)
113      port map(B3.1d,B3.1d2,clkB,C4.1d,C2.2d2,n_mag_vect(11),n_zones_vect(11),
      area_eff_vect(11),area_tot_vect(11),Er_vect(11),Ec_vect(11));
      D3.2: double_wire_horiz generic map(D,3,2,ZONE_H,ZONE_L)
      port map(D3.2d,D3.2d2,clkD,A2.3d2,A4.2d,n_mag_vect(12),n_zones_vect(12),
      area_eff_vect(12),area_tot_vect(12),Er_vect(12),Ec_vect(12));
116      B3.3: double_wire_horiz generic map(B,3,3,ZONE_H,ZONE_L)
      port map(B3.3d,B3.3d2,clkB,C2.4d2,C4.3d,n_mag_vect(13),n_zones_vect(13),
      area_eff_vect(13),area_tot_vect(13),Er_vect(13),Ec_vect(13));
      D3.4: double_wire_horiz generic map(D,3,4,ZONE_H,ZONE_L)
119      port map(D3.4d,D3.4d2,clkD,A2.5d,A4.4d,n_mag_vect(14),n_zones_vect(14),area_eff_vect
      (14),area_tot_vect(14),Er_vect(14),Ec_vect(14));
--Row4
      C4.1: short_wire_vert generic map(C,4,1,RX,ZONE_H,ZONE_L)
122      port map(C4.1d,clkC,D5.1d,n_mag_vect(15),n_zones_vect(15),area_eff_vect(15),
      area_tot_vect(15),Er_vect(15),Ec_vect(15));
      A4.2: double_wire_horiz generic map(A,4,2,ZONE_H,ZONE_L)
      port map(A4.2d,A4.2d2,clkA,B3.1d2,B5.2d,n_mag_vect(16),n_zones_vect(16),
      area_eff_vect(16),area_tot_vect(16),Er_vect(16),Ec_vect(16));

```

```

125 C4.3: double_wire_horiz generic map(C,4,3,ZONE_H,ZONE_L)
      port map(C4_3d,C4_3d2,clkC,D3_2d2,D5_3d,n_mag_vect(17),n_zones_vect(17),
      area_eff_vect(17),area_tot_vect(17),Er_vect(17),Ec_vect(17));
A4.4: double_wire_horiz generic map(A,4,4,ZONE_H,ZONE_L)
128 port map(A4_4d,A4_4d2,clkA,B3_3d2,B5_4d,n_mag_vect(18),n_zones_vect(18),
      area_eff_vect(18),area_tot_vect(18),Er_vect(18),Ec_vect(18));
C4.5: short_wire_vert generic map(C,4,5,LX,ZONE_H,ZONE_L)
      port map(C4_5d,clkC,D3_4d2,n_mag_vect(19),n_zones_vect(19),area_eff_vect(19),
      area_tot_vect(19),Er_vect(19),Ec_vect(19));
131 --Row5
D5.1: double_wire_horiz generic map(D,5,1,ZONE_H,ZONE_L)
      port map(D5_1d,D5_1d2,clkD,A4_2d2,A6_2d,n_mag_vect(20),n_zones_vect(20),
      area_eff_vect(20),area_tot_vect(20),Er_vect(20),Ec_vect(20));
134 B5.2: crosswire generic map(B,5,2,ZONE_H,ZONE_L)
      port map(B5_2d,B5_2d2,clkB,C4_3d2,C6_3d,n_mag_vect(21),n_zones_vect(21),
      area_eff_vect(21),area_tot_vect(21),Er_vect(21),Ec_vect(21));
D5.3: double_wire_horiz generic map(D,5,3,ZONE_H,ZONE_L)
137 port map(D5_3d,D5_3d2,clkD,A4_4d2,A6_4d,n_mag_vect(22),n_zones_vect(22),
      area_eff_vect(22),area_tot_vect(22),Er_vect(22),Ec_vect(22));
B5.4: or_wire_lx generic map(B,5,4,UP,ZONE_H,ZONE_L)
      port map(B5_4d,B5_4d2,clkB,C4_5d,n_mag_vect(23),n_zones_vect(23),area_eff_vect(23),
      area_tot_vect(23),Er_vect(23),Ec_vect(23));
140 --Row6
C6.1: wire_2outputs generic map(C,6,1,RX_DOWN,ZONE_H,ZONE_L)
      port map(C6_1d,clkC,D5_1d2,D7_1d,n_mag_vect(25),n_zones_vect(25),area_eff_vect(25),
      area_tot_vect(25),Er_vect(25),Ec_vect(25));
143 A6.2: crosswire generic map(A,6,2,ZONE_H,ZONE_L)
      port map(A6_2d,A6_2d2,clkA,B5_2d2,B7_2d,n_mag_vect(26),n_zones_vect(26),
      area_eff_vect(26),area_tot_vect(26),Er_vect(26),Ec_vect(26));
C6.3: wire_2outputs generic map(C,6,3,RX_UP,ZONE_H,ZONE_L)
146 port map(C6_3d,clkC,D5_3d2,D7_3d,n_mag_vect(27),n_zones_vect(27),area_eff_vect(27),
      area_tot_vect(27),Er_vect(27),Ec_vect(27));
A6.4: crosswire generic map(A,6,4,ZONE_H,ZONE_L)
      port map(A6_4d,A6_4d2,clkA,B5_4d2,B7_4d,n_mag_vect(28),n_zones_vect(28),
      area_eff_vect(28),area_tot_vect(28),Er_vect(28),Ec_vect(28));
149 --Row7
D7.1: and_2outputs_lx generic map(D,7,1,ZONE_H,ZONE_L)
      port map(D7_1d,D7_1d2,clkD,A6_2d2,A8_2d,n_mag_vect(30),n_zones_vect(30),
      area_eff_vect(30),area_tot_vect(30),Er_vect(30),Ec_vect(30));
152 B7.2: or_wire_lx generic map(B,7,2,DOWN,ZONE_H,ZONE_L)
      port map(B7_2d,B7_2d2,clkB,C8_3d,n_mag_vect(31),n_zones_vect(31),area_eff_vect(31),
      area_tot_vect(31),Er_vect(31),Ec_vect(31));
D7.3: and_2outputs_lx generic map(D,7,3,ZONE_H,ZONE_L)
155 port map(D7_3d,D7_3d2,clkD,A6_4d2,A8_4d,n_mag_vect(32),n_zones_vect(32),
      area_eff_vect(32),area_tot_vect(32),Er_vect(32),Ec_vect(32));
B7.4: or_wire_lx generic map(B,7,4,DOWN,ZONE_H,ZONE_L)
      port map(B7_4d,B7_4d2,clkB,C8_5d,n_mag_vect(33),n_zones_vect(33),area_eff_vect(33),
      area_tot_vect(33),Er_vect(33),Ec_vect(33));
158 --Row8
C8.1: wire_2outputs generic map(C,8,1,RX_DOWN,ZONE_H,ZONE_L)
      port map(C8_1d,clkC,D7_1d2,D9_1d,n_mag_vect(35),n_zones_vect(35),area_eff_vect(35),
      area_tot_vect(35),Er_vect(35),Ec_vect(35));
161 A8.2: crosswire generic map(A,8,2,ZONE_H,ZONE_L)
      port map(A8_2d,A8_2d2,clkA,B7_2d2,B9_2d,n_mag_vect(36),n_zones_vect(36),
      area_eff_vect(36),area_tot_vect(36),Er_vect(36),Ec_vect(36));
C8.3: and_2outputs_lx generic map(C,8,3,ZONE_H,ZONE_L)
164 port map(C8_3d,C8_3d2,clkC,D7_3d2,D9_3d,n_mag_vect(37),n_zones_vect(37),
      area_eff_vect(37),area_tot_vect(37),Er_vect(37),Ec_vect(37));
A8.4: crosswire generic map(A,8,4,ZONE_H,ZONE_L)

```

```

167     port map(A8_4d,A8_4d2,clkA,B7_4d2,B9_4d,n_mag_vect(38),n_zones_vect(38),
        area_eff_vect(38),area_tot_vect(38),Er_vect(38),Ec_vect(38));
167 C8_5: and_wire_lx generic map(C,8,5,DOWN,ZONE_H,ZONE_L)
        port map(C8_5d,C8_5d2,clkC,D9_5d,n_mag_vect(39),n_zones_vect(39),area_eff_vect(39)
        ,area_tot_vect(39),Er_vect(39),Ec_vect(39));
--Row9
170 D9_1: double_wire_horiz generic map(D,9,1,ZONE_H,ZONE_L)
        port map(D9_1d,D9_1d2,clkD,A8_2d2,A10_1d,n_mag_vect(40),n_zones_vect(40),
        area_eff_vect(40),area_tot_vect(40),Er_vect(40),Ec_vect(40));
173 B9_2: inv_with_wire generic map(B,9,2,UP,ZONE_H,ZONE_L)
        port map(B9_2d,B9_2d2,clkB,C8_3d2,C10_2d,n_mag_vect(41),n_zones_vect(41),
        area_eff_vect(41),area_tot_vect(41),Er_vect(41),Ec_vect(41));
176 D9_3: double_wire_horiz generic map(D,9,3,ZONE_H,ZONE_L)
        port map(D9_3d,D9_3d2,clkD,A8_4d2,A10_3d,n_mag_vect(42),n_zones_vect(42),
        area_eff_vect(42),area_tot_vect(42),Er_vect(42),Ec_vect(42));
179 B9_4: inv_with_wire generic map(B,9,4,UP,ZONE_H,ZONE_L)
        port map(B9_4d,B9_4d2,clkB,C8_5d2,C10_4d,n_mag_vect(43),n_zones_vect(43),
        area_eff_vect(43),area_tot_vect(43),Er_vect(43),Ec_vect(43));
179 D9_5: wire_2outputs generic map(D,9,5,RXDOWN,ZONE_H,ZONE_L)
        port map(D9_5d,clkD,A10_5d,D9_5q2,n_mag_vect(44),n_zones_vect(44),area_eff_vect(44),
        area_tot_vect(44),Er_vect(44),Ec_vect(44));
--Row10
182 A10_1: short_wire_horiz generic map(A,10,1,UP,ZONE_H,ZONE_L)
        port map(A10_1d,clkA,A10_1q,n_mag_vect(45),n_zones_vect(45),area_eff_vect(45),
        area_tot_vect(45),Er_vect(45),Ec_vect(45));
185 C10_2: short_wire_horiz generic map(C,10,2,UP,ZONE_H,ZONE_L)
        port map(C10_2d,clkC,D9_1d2,n_mag_vect(46),n_zones_vect(46),area_eff_vect(46),
        area_tot_vect(46),Er_vect(46),Ec_vect(46));
188 A10_3: short_wire_horiz generic map(A,10,3,UP,ZONE_H,ZONE_L)
        port map(A10_3d,clkA,B9_2d2,n_mag_vect(47),n_zones_vect(47),area_eff_vect(47),
        area_tot_vect(47),Er_vect(47),Ec_vect(47));
191 C10_4: short_wire_horiz generic map(C,10,4,UP,ZONE_H,ZONE_L)
        port map(C10_4d,clkC,D9_3d2,n_mag_vect(48),n_zones_vect(48),area_eff_vect(48),
        area_tot_vect(48),Er_vect(48),Ec_vect(48));
        A10_5: and_2outputs_rx generic map(A,10,5,ZONE_H,ZONE_L)
        port map(A10_5d,A10_5d2,clkA,B9_4d2,A10_5q2,n_mag_vect(49),n_zones_vect(49),
        area_eff_vect(49),area_tot_vect(49),Er_vect(49),Ec_vect(49));
191 end behavior;

```

A.4 Testbench template

This section contains the Testbench used for testing the Parallel MAC without interleaving. It has the same structure as all the other testbench used. After defining the clock and reset signals, the input signals are acquired from a text file containing 100 random numbers in the required range. They are fed to the circuit and after the right amount of time the circuit's

output, as well as the information on area and energy, are written into another text file. The output results will be compared afterwards to a file containing the expected results.

Listing A.31. Testbench template.

```

library ieee;
use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.math_real.all;
6 use work.MENML_package.all;
library std;
use std.textio.all; -- For txt file handling
9 use ieee.std_logic_textio.all;

entity tb_MAC_Nbit is
12 end tb_MAC_Nbit;

architecture behavior of tb_MAC_Nbit is
15 component MAC_Nbit is
port(A,B: in std_logic_vector (N_BIT-1 downto 0);
reset: in std_logic;
18 MAC_result: out std_logic_vector (2*N_BIT-1 downto 0);
MAC_Co: out std_logic;
clk, clkA, clkB, clkC, clkD: in std_logic;
21 n_mag: out natural := init_natural;
n_zones: out natural := init_natural;
AREA_EFF: out natural;
24 AREA_TOT: out natural;
Er: out natural;
Ec: out natural);
27 end component;

signal A,B: std_logic_vector (N_bit-1 downto 0);
30 signal ACC: std_logic_vector (2*N_BIT-1 downto 0);
signal clk, clkA, clkB, clkC, clkD, reset, Carry_out: std_logic;
signal n_mag, n_zones, AREA_EFF, AREA_TOT, Er, Ec: natural;
33
begin
-- CLOCK GENERATION
-- Clock zones have clock with DC of 35%, and overlap with the contiguous
36 PhaseA_process: process
begin
39 clkA <= '1';
wait for 3 ns;
clkA <= '0';
42 wait for 6.5 ns;
clkA <= '1';
wait for 0.5 ns;
45 end process;
PhaseB_process: process
begin
48 clkB <= '0';
wait for 2 ns;
clkB <= '1';
51 wait for 3.5 ns;

```



```

        clkB <= '0';
        wait for 4.5 ns;
54 end process;
PhaseC_process: process
begin
57     clkC <= '0';
        wait for 4.5 ns;
        clkC <= '1';
60     wait for 3.5 ns;
        clkC <= '0';
        wait for 2 ns;
63 end process;
PhaseD_process: process
begin
66     clkD <= '1';
        wait for 0.5 ns;
        clkD <= '0';
69     wait for 6.5 ns;
        clkD <= '1';
        wait for 3 ns;
72 end process;

-- RESET GENERATION --
75 data_process: process
begin
    reset <= '0';
78     wait for 5*CLK.PERIOD;
    loop
        reset <= '1';
81         wait for CLK.PERIOD;
    end loop;
end process;

84
-- The data generation process takes the data from the "test_vector_x" file , where x can
-- be 4, 8,
-- 16 or 32. The data, taken from the file , was generated by MatLab using the function "
-- rand",
87 -- followed by some manipulations in order to have integers in the correct range. The
-- name of the
-- file has to be changed in FILE_OPEN(,) according to the number of bits.
-- The number of test vector is always 1000.
90 Data_generate: process
    variable dataA,dataB: std_logic_vector (N_bit-1 downto 0);
    variable good : boolean;
93    variable InLine : line;
    variable line_content:std_logic_vector (2*N_bit-1 downto 0);
    file test_vectors : text;
96    variable LineNumber : integer :=0;
begin
    FILE_OPEN (test_vectors , "test_vectors/test_vect_4.txt" , READMODE);
99
    for i in 0 to 1000 loop
        dataA:= (others => '0');
102        dataB:= (others => '0');
        readline (test_vectors , InLine);
        read (InLine,line_content);
105        for k in 0 to N_BIT-1 loop
            dataA(k):=line_content(k);
            dataB(k):=line_content(N_bit+k);

```

```

108     end loop;
        A<=dataA;
        B<=dataB;
111     wait for 5*CLK.PERIOD;
    end loop;
    file_close(test_vectors);
114 end process;

----- INSTANTIATION OF THE MULTIPLIER -----
117 MAC: MAC_N_bit
    port map(A=>A,
        B=>B, reset=>reset , MAC_result=>ACC,MAC.Co=>Carry_out ,
120     clk=>clk , clkA=>clkA , clkB=>clkB , clkC=>clkC , clkD=>clkD ,
        n_mag=>n_mag , n_zones=>n_zones ,
        area_eff=>area_eff , area_tot=>area_tot ,
123     Er=>Er , Ec=>Ec);

----- RESULTS WRITING -----
126 -- The results are written in a file called "Results". At the end of the simulation the
    stored
    -- values have to be compared with the ones computed by MatLab. These were saved in the
    files named
    -- "result_x" in the same folder; also in this case x can be 4, 8, 16 or 32. Concerning
    the
129 -- simulation times, they are 50,28 us, 50.56 us, 51.12 us and 52.24 microseconds for 4,
    8, 16 and
    -- 32 case respectively.
    Writing_file_process : process
132     file      outfile : text; --declare output file
        variable line_content ,MsgLine : line; --line number declaration
        -- variable line_content : string(1 to 2*N_BIT);
135     variable result: std_logic_vector(2*N_bit-1 downto 0);
        variable start: natural := 0;
    begin
138     wait for 28*CLK.PERIOD+(N_BIT-4)*7*CLK.PERIOD; -- wait until the first bit of the
        result is available
        loop
            for ind in 2*N_BIT-1 downto 0 loop
141                 result(ind):=ACC(ind);
            end loop;
            file_open(outfile,"Results/results.txt",append_MODE);
144            write (MsgLine, result);
            writeline (outfile,MsgLine);
            file_close(outfile);
147            wait for 5*CLK.PERIOD;
            end loop;
        end process;
150 -- Write in a file Energy and Area -----
    Writing : process
        file      outfile : text; --declare output file
153        variable outline : line; --line number declaration
    begin
        wait for 50 ns;
156        file_open(outfile,"Results/area-energy.txt",APPEND_MODE);

        write(outline,n_mag); --write the line.
        writeline (outfile,outline); --write the contents into the file
159        write(outline,n_zones); --write the line.
        writeline (outfile,outline); --write the contents into the file

```

```
162     write(outline,real(AREA_EFF)/1.0e+06); --write the line.
      writeline (outfile,outline); --write the contents into the file
      write(outline,real(AREA_TOT)/1.0e+06); --write the line.
165     writeline (outfile,outline); --write the contents into the file
      write(outline,real(Er)/1.0e+09); --write the line.
      writeline (outfile,outline); --write the contents into the file
168     write(outline,real(Ec)/1.0e+06); --write the line.
      writeline (outfile,outline); --write the contents into the file
      file_close(outfile);
171   end process;
end behavior;
```

CITED LITERATURE

1. Vacca, M., Graziano, M., Chiolerio, A., Lamberti, A., Laurenti, M., Balma, D., Enrico, E., Celegato, F., Tiberto, P., Boarino, L., and Zamboni, M.: Electric Clock for NanoMagnet Logic Circuits. In Field-Coupled Nanocomputing, eds. N. G. Anderson and S. Bhanja, Lecture Notes in Computer Science, pages 73–110. Springer Berlin Heidelberg, 2014.
2. International technology roadmap for semiconductors (ITRS). 2013, <http://www.itrs.com>.
3. Haron, N. and Hamdioui, S.: Why is cmos scalig coming to an end? Design and Test Workshop, 2008. IDT 2008. 3rd International, pages 98–103, 20-22 Dec 2008.
4. Kim, Y.-B.: Review paper: Challenges for nanoscale mosfets and emerging nanoelectronics. Trans. Electr. Electron. Mater. 10(1) 21, 2009.
5. Deleonibus, S., Salvo, B. D., Clavelier, L., Ernst, T., Faynot, O., Poiroux, T., and Vinet, M.: Cmos devices architectures for the end of the roadmap and beyond. Solid-State and Integrated Circuit Technology, 8th International Conference on, pages 51–54, 23-26 Oct 2006.
6. Lent, C., Tougaw, P., Porod, W., and Bernstein, G.: Quantum cellular automata. Nanotechnology, 4:49–57, 1993.
7. Tougaw, P. and Lent, C.: Dynamic behavior of quantum cellular automata. Journal Of Applied Physics, (80):4722–4736, 1996.
8. Kummamuru, R., Orlov, A., Ramasubramaniam, R., Lent, C., Bernstein, G., and Snider, G.: Operation of a quantum-dot cellular automata (qca) shift register and analysis of errors. IEEE Trans. On Electron Devices, 50:1906, September 2003.
9. Csurgay, A., Porod, W., and Lent, C.: Signal processing with near neighborcoupled time-varying quantum-dot arrays. IEEE Transaction On Circuits and Systems, 47(8):12121223, 2000.
10. Khitun, A. and Wang, K.: Multi-functional edge driven nano-scale cellular automata based on semiconductor tunneling nano-structure with a self-assembled quantum dot layer. Superlattices and Microstructures, 37(1):55–76, January 2005.

CITED LITERATURE (Continued)

11. Smith, C., Gardelis, S., Rushforth, A., Crook, R., Cooper, J., Ritchie, D., Linfield, E., Jin, Y., and Pepper, M.: Realization of quantum-dot cellular automata using semiconductor quantum dots. Superlattices and Microstructures, 34(3-6):195–203, 2003.
12. Niemier, M. and al.: Nanomagnet logic: progress toward system-level integration. J. Phys.: Condens. Matter, 23:34, November 2011.
13. Cowburn, R. and Welland, M.: Room temperature magnetic quantum cellular automata. Science, 287:1466–1468, 2000.
14. Vacca, M., Graziano, M., Di Crescenzo, L., Chiolerio, A., Lamberti, A., Balma, D., Canavese, G., Celegato, F., Enrico, E., Tiberto, P., Boarino, L., and Zamboni, M.: Magnetoelastic Clock System for Nanomagnet Logic. IEEE Trans. Nanotechnol., 13(5):963–973, Sep. 2014.
15. Augustine, C., Fong, X., Behin-Aein, B., and Roy, K.: Ultra-Low Power Nano-Magnet Based Computing: A System-Level Perspective. IEEE Trans. Nanotechnol., 10(4):778–788, 2011.
16. Lent, C. and Isaksen, B.: Clocked Molecular Quantum-Dot Cellular Automata. IEEE Transactions on Electron Devices, 50(9):1890–1896, September 2003.
17. Y. Lu, M. L. C. L.: Molecular electronics - from structure to circuit dynamics. In Sixth IEEE Conference on Nanotechnology, pages 62–65, Cincinnati-Ohio, USA, 2006. IEEE.
18. Pulimeno, A., Graziano, M., Demarchi, D., and Piccinini, G.: Towards a molecular QCA wire: Simulation of write-in and read-out systems. Solid-State Electronics, Elsevier, 1:7, 2012.
19. Pulimeno, A., Graziano, M., V.Cauda, Sanginario, A., Demarchi, D., and Piccinini, G.: Bis-ferrocene molecular qca wire: ab-initio simulations of fabrication driven fault tolerance. IEEE Trans. Nanotechnol., 12(3), 2013.
20. Wolkow, R., Livadaru, L., Pitters, J., Taucer, M., Piva, P., Salomons, M., Cloutier, M., and Martins, B.: Silicon Atomic Quantum Dots Enable Beyond-CMOS Electronics. In Field-Coupled Nanocomputing, eds. N. G. Anderson and S. Bhanja, pages 33–58. Springer Berlin Heidelberg, 2014.
21. Haider, M. B. and al.: Controlled coupling and occupation of silicon atomic quantum dots at room temperature. Phys. Rev. Lett., 102, January 2009.

CITED LITERATURE (Continued)

22. Vacca, M., Graziano, M., Wang, J., Cairo, F., Causapruno, G., Urgese, G., Biroli, A., and Zamboni, M.: NanoMagnet Logic: An Architectural Level Overview. In Field-Coupled Nanocomputing, eds. N. G. Anderson and S. Bhanja, Lecture Notes in Computer Science, pages 223–256. Springer Berlin Heidelberg, 2014.
23. Alam, M., Siddiq, M., Bernstein, G., Niemier, M., Porod, W., and Hu, X.: On-chip Clocking for Nanomagnet Logic Devices. IEEE Trans. Nanotechnol., 2009.
24. Csaba, G. and Porod, W.: Behavior of Nanomagnet Logic in the Presence of Thermal Noise. In IEEE Int. Workshop Computational Electronics, pages 1–4, Pisa, Italy, 2010.
25. Imre, A.: Experimental study of nanomagnets for Quantum-dot cellular automata(MQCA)logic applications. Doctoral dissertation, University of Notre Dame, Notre Dame, Indiana, December 2005.
26. Niemier, M., Varga, E., Bernstein, G., Porod, W., Alam, M., Dingler, A., Orlov, A., and Hu, X.: Shape Engineering for Controlled Switching With Nanomagnet Logic. IEEE Trans. Nanotechnol., 11(2):220–230, Mar. 2012.
27. Vacca, M., Graziano, M., and Zamboni, M.: Majority Voter Full Characterization for NanoMagnet Logic Circuits. IEEE Trans. Nanotechnol., 11(5):940–947, 2012.
28. Graziano, M., Vacca, M., Chiolerio, A., and M.Zamboni: An NCL-HDL Snake-Clock-Based Magnetic QCA Architecture. IEEE Trans. Nanotechnol., 10(5):1141–1149, Sep. 2011.
29. Vacca, M.: Nanoarchitectures based on magnetic QCA. Doctoral dissertation, Politecnico di Torino, Turin, Italy, 2008.
30. Graziano, M., Chiolerio, A., and Zamboni, M.: A technology aware magnetic qca ncl-hdl architecture. In IEEE Int. Conf. Nanotechnol., pages 763 – 766, Genova, Italy, 2009.
31. Das, J., Alam, S., and Bhanja, S.: Low power magnetic quantum cellular automata realization using magnetic multi-layer structures. J. on Emerging and Selected Topics in Circuits and Systems, 1(3):267–276, Sep 2011.
32. Rizos, N., Omar, M., Lugli, P., Csaba, G., Becherer, M., and Landsiedel, D. S.: Clocking schemes for field coupled devices from magnetic multilayers. In IEEE International Workshop on Computational Electronics, pages 1–4, Beijing, China, 2009.

CITED LITERATURE (Continued)

33. Atulasimha, J. and Bandyopadhyay, S.: Hybrid spintronic/straintronics: A super energy efficient computing scheme based on interacting multiferroic nanomagnets. In 12th IEEE International Conference on Nanotechnology, pages 1–4, 2012.
34. Giri, D., Vacca, M., Causapruno, G., Rao, W., Graziano, M., and Zamboni, M.: A standard cell approach for MagnetoElastic NML circuits. In EEE/ACM Int. Symp. Nanoscale Architectures (NANOARCH), pages 65–70, Jul. 2014.
35. Vacca, M., Graziano, M., Chiolerio, A., Lamberti, A., Laurenti, M., Balma, D., Enrico, E., Celegato, F., Tiberto, P., Boarino, L., and Zamboni, M.: Electric Clock for NanoMagnet Logic Circuits. In Field-Coupled Nanocomputing, eds. N. G. Anderson and S. Bhanja, Lecture Notes in Computer Science, pages 73–110. Springer Berlin Heidelberg, 2014.
36. Fashami, M. S., Atulasimha, J., and Bandyopadhyay, S.: Magnetization dynamics, throughput and energy dissipation in a universal multiferroic nanomagnetic logic gate with fan-in and fan-out. Nanotechnology, 23(10), February 2012.
37. Vacca, M., Graziano, M., and Zamboni, M.: Nanomagnetic Logic Microprocessor: Hierarchical Power Model. IEEE Trans. on VLSI Systems, 21(8):1410–1420, August 2012.
38. Johnson, E., Janulis, J., Henderson, S., and Tourgaw, P.: Incorporating Standard CMOS Design Process Methodologies into the QCA Logic Design Process. IEEE Transaction on Nanotechnology, 3(1):29, 2004.
39. Imre, A., Ji, L., Csaba, G., Orlov, A., Bernstein, G., and Porod, W.: Magnetic logic devices based on field-coupled nanomagnets. In Int. Symp. Semiconductor Device Research, Dec. 2005.
40. Vacca, M., Frache, S., Graziano, M., Riente, F., Turvani, G., Roch, M., and Zamboni, M.: ToPoliNano: NanoMagnet Logic Circuits Design and Simulation. In Field-Coupled Nanocomputing, eds. N. G. Anderson and S. Bhanja, Lecture Notes in Comput. Science, pages 274–306. Springer Berlin Heidelberg, 2014.
41. Kung, H., Leiserson, C., and of Comput. Science, C.-M. U. D.: Systolic Arrays for VLSI. CMU-CS. Carnegie-Mellon University, Department of Comput. Science, 1978.
42. Lu, L., Liu, W., O’Neill, M., and Swartzlander, E.: Qca systolic array design. IEEE Trans. Comput., 62(3):548–560, Mar. 2013.

CITED LITERATURE (Continued)

- 43. Awais, M., Vacca, M., Graziano, M., Roch, M. R., and Masera, G.: Quantum dot Cellular Automata Check Node Implementation for LDPC Decoders. IEEE Trans. Nanotechnol., 12(3), 2013.
- 44. MacKay, D.: Information Theory, Inference, and Learning Algorithms. CMU-CS. Hardback, 2003.
- 45. Grossschadl, J.: A Low Power Bit-Serial Multiplier For Finite Fields $GF(2^m)$. In The 2001 IEEE International Symposium on Circuits and Systems, volume 4, pages 37–40, Sydney, NSW, May 2001.
- 46. Vacca, M.: Emerging Technologies - NanoMagnets Logic (NML). Doctoral dissertation, Politecnico di Torino, Turin, Italy, April 2013.

VITA

NAME	Davide Giri
EDUCATION	B.S., IT Engineering, Politecnico di Torino, Italy, 2012 M.S., Electronic Engineering, Politecnico di Torino, Italy, 2014 M.S., Electrical and Computer Engineering, UIC, 2015
HONORS	Politong mobility scholarship for Special Program, 2010-2011 TOP-UIC mobility scholarship for Special program, 2013
PUBLICATIONS	D.Giri, M.Vacca, G.Causapruno, W.Rao, M.Graziano, M.Zamboni, “ <i>A Standard Cell Approach for MagnetoElastic NML Circuits</i> ”, 10th ACM/IEEE International Symposium on Nanoscale Architectures (NANOARCH 2014), 8-10 July 2014, Paris, France