Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Architectural exploration of Logic-in-Memory systems with DExIMA-CAD

Supervisors
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Prof. Giovanna Turvani

Candidate
Cristian Neagu
student ID: 284734

April 2023

Summary

In the past years, the "memory wall problem" has questioned the feasibility of the traditional von Neumann architecture for new electronic systems. Due to several technological concerns, a performance bottleneck is found in the memory subsystem. As regards power consumption, energy is wasted in the processor-memory data traffic. Many beyond von Neumann alternatives are found in the scientific literature. For instance, in Logic-in-Memory (LiM) arrays, memory cells are endowed with computational capabilities, creating a naturally-parallel processing element. Thus, such structures may support Single-Instruction, Multiple-Data (SIMD) operations and may successfully accelerate a large variety of algorithms. In addition, as the processing is carried out within the memory array, less energy is wasted in the processor-memory data traffic.

At the VLSI laboratory of the Politecnico di Torino, researchers have been studying LiM solutions, proving their effectiveness in mitigating the memory wall problem. The research work highlighted the need of new software tools to support the design flow for LiM architectures and to characterize their main figures of merit. Thus, a LiM development toolchain was born, which includes several components. Octantis is a High-Level Synthesis (HLS) tool, which compiles an input algorithm to produce a LiM array. DExIMA (Design Explorer for In-Memory Architectures) is a C++ figures of merit estimator, which has been conceived to evaluate the critical path, the area occupation and the power consumption of a LiM array. DExIMA-CAD is the Python Graphical User Interface (GUI) front-end for DExIMA and it provides an environment for the structural description of a LiM array.

The aim of this thesis is twofold. Firstly, the functionalities of DExIMA CAD are expanded, improving its structural description and the architectural exploration capabilities, in order to provide a more robust support for the LiM design flow. Secondly, the revisited front-end is employed to implement a variety of LiM structures and algorithms, possibly fostering the SIMD paradigm.

The structural description capabilities of DExIMA CAD are largely enriched by the introduction of array interconnections. Horizontal interconnections provide a

communication route for all cells belonging to a memory row, enabling the in-row implementation of such operators as ripple-carry adders and bit-wise shift registers. Vertical interconnections may be deployed to connect any pair of array elements, creating arbitrary data movement patterns within the LiM array. A new dedicated module offers a set of interconnections-related analysis and synthesis functionalities, which are used to expand the processing abilities of a LiM array.

In DExIMA CAD, the simulation-time behaviour of a LiM array is defined by a micro-programmed control unit based on an algorithm Read-Only Memory (ROM). The procedure for defining the content of this ROM was particularly cumbersome and inefficient: a set of improvements is thus introduced in the algorithm description module. In the revisited DExIMA CAD, the operating modes of the LiM array elements are defined by nano-instructions, which assign a mnemonic to a set of values for the required control signals. Nano-instructions may be combined together to form a scalar control, which is extended to a set of array elements by means of an activation pattern, enabling a true in-memory Single-Instruction, Multiple-Data (SIMD) processing. The new Graphical User Interface (GUI) features of the algorithm description module largely ease the ROM programming procedure, making it easier for the designer to specify the simulation-time behaviour of a LiM array.

As regards the architectural exploration capabilities of DExIMA CAD, the support for multi-array scenarios is introduced in the uppermost architectural level of the complete LiM system. In the revisited DExIMA CAD, the top level can host multiple instances of different LiM array types, and near-memory sections may be created by the allocation of library components and arbitrarily-complex subsystems. The large variability of the uppermost architectural level is supported by a dedicated module, the top-level analyzer, which offers several analysis and synthesis tasks, which include, but are not limited to, the automatic configuration of a UVM testbench and the generation of the HDL code for the complete system. A top-level simulation dashboard helps define the simulation-time behaviour of the complete LiM system, easing the simulation process and embedding it in the environment of DExIMA CAD.

The effectiveness of the revisited DExIMA CAD is initially proved by the implementation of existing LiM structures, including the components of an in-memory Binary Neural Network (BNN) and the Hybrid-SIMD architecture, a general-purpose LiM co-processor. The new front-end features are then used to implement *ex novo* in-memory solutions for the Secure Hash Algorithm (SHA) and the Advanced Encryption Standard (AES).

All presented LiM structures would have been unattainable by the previous version of DExIMA CAD. Their implementation highlights the increased breadth of the

structural description functionalities and the architectural exploration capabilities of DExIMA CAD, which has become more versatile and efficient in supporting the implementation of different LiM systems and algorithms. For all proposed implementations, the main figures of merit, e.g. critical path, static power and dynamic power, are estimated both by Synopsys Design Compiler and by DExIMA CAD. The two different estimation processes yield compatible results, further proving the effectiveness of the tool in handling a variety of LiM structures.

The revisited DExIMA CAD is thus a valid support to the LiM design flow, helping the exploration of increasingly complex LiM systems.

To all the people who love me.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction to Logic-in-Memory (LiM) processing

For many years, the von Neumann model has been extensively used to lay out the structure of electronic systems. In a von Neumann architecture, the structure of which is presented in Figure 1.1, the computational task are demanded to a Central Processing Unit (CPU), while storage tasks are fulfilled by a memory sub-system.

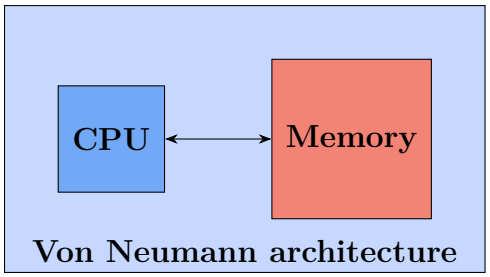


Figure 1.1: General representation of a von Neumann architecture.

The structure of the two components of a von Neumann architecture mirrors their functional separation. In fact, micro-processors are designed to offer the required hardware support for a predefined instruction set, and they typically integrate low-capacity, fast-access memory elements for the temporary storage. On the other hand, memory arrays are specifically conceived to act as high-capacity storage elements with the main drawback of being slower than CPUs.

The performance gap between micro-processors and memory arrays has rapidly increased over the years. The reason for this behaviour lies in the fact that microprocessor have a larger performance increase rate with respect to memories. CPUs become increasingly faster, implying that their throughput is constantly increasing, but memories cannot cope with very large processing rates because of their latency.

Such a performance bottleneck, which is evidently found in the memory sub-system, is typically referred to as "memory wall problem".

A further motivation questions the feasibility of the von Neumann architecture for modern electronic system. Since a micro-processor has very limited storage capabilities, for most applications data is continuously fetched from the memory sub-subsystem, transferred to the CPU, elaborated and then brought back to the memory. In extremely data-intensive applications, a significant data traffic is observed between the processor and the memory, which worsens not only the actual throughput of the system, but also its power budget, as energy is wasted in transferring data back and forth.

The scientific literature contains many different examples of "beyond von Neumann" alternatives. Typically, the rationale behind this solutions is to involve a memory array in the computational tasks to some degree. A comprehensive taxonomy of this alternatives is presented in [9], which reports four main approaches:

- Computation-near-Memory (CnM), where the computational facilities are kept as close as possible to the memory sub-system;
- Computation-in-Memory (CiM), where part of the processing takes place in the peripheral circuits of the storage element;
- Computation-with-Memory (CwM), where the memory array provides results useful to the computation, e.g. in look-up tables;
- Logic-in-Memory (LiM), where logic elements are integrated in the memory cells, endowing them with computational capabilities, as hinted for instance in [2].

In a LiM structure, the array elements are able to implement specific processing tasks and to elaborate the data they store. As part of the processing is moved in the LiM array, the processor-memory data traffic is reduced and the power budget of the system is improved. Moreover, since multiple array elements can execute the same task, a LiM array is a naturally-parallel processing element, which may easily used to foster a Single-Instruction, Multiple-Data (SIMD) elaboration. As a consequence, specific algorithms may benefit from an in-memory implementation, as their parallel sections may be accelerated, leading to a performance speedup.

LiM solutions have been widely studied by researchers from the "VLSI laboratory" of the Politecnico di Torino. In-memory alternatives for implementing a Convolutional Neural Network (CNN) are detailed in [9], which presents the Configurable Logic-in-Memory Architecture (CLiMA), and in [11]. A LiM implementation of the bitmap indexing algorithm is presented in [10]. Hybrid-SIMD, a general-purpose and modular LiM co-processor, is presented in [16]. A RISC-V LiM framework is introduced in [17]. This work follows the architectural paradigm presented in Figure 1.2 and it shows remarkable improvements to the energy budget

of the system if a LiM approach is used, in spite of a larger area occupation and of a slightly worse critical path.

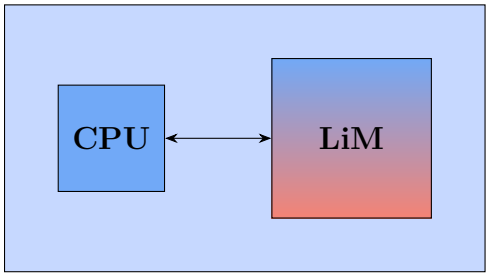


Figure 1.2: Architectural paradigm employed by the work presented in [17]. With respect to the von Neumann architecture depicted in Figure 1.1, a modified memory is endowed with computational capabilities, so part of the processing may take place within the LiM array.

All these works prove that specific algorithms may benefit from an in-memory implementation and, in terms of performances and power consumption, a LiM solution may be more favourable than the traditional von Neumann approach. In addition, they all suggest the need of specific tools to support the design flow for such systems. As a consequence, researchers from the "VLSI laboratory" of the Politecnico di Torino started the development of the LiM Development Toolchain.

1.2 Structure of LiM Development Toolchain

Figure 1.3 presents a general representation of the LiM Development Toolchain developed at the Politecnico di Torino, which currently consists of Octantis, DExIMA Backend and DExIMA CAD.

Octantis is a High-Level Synthesis (HLS) tool able to compile an input algorithm into a LiM array. Its development was started by [13] and was continued by [18].

DExIMA Backend is a C++ figures of merit estimator, which has been specifically conceived to evaluate the critical path, the area occupation and the power consumption of a LiM array. The development of DExIMA Backend was started by [8] and was continued by [15]. The estimation process is made possible by detailing the structure of a LiM array in a custom-format textual description, namely a `.dex` file, which reports the content of all LiM cells, all additional logic elements and their mutual connections.

The supported DExIMA Backend syntax is thoroughly detailed in [15] and is versatile enough to enable the description of a large variety of LiM arrays. Nevertheless, as DExIMA Backend is merely an estimator, there was no support for easing the generation of the `.dex` file, so the structural description process was rather burdensome and time-consuming.

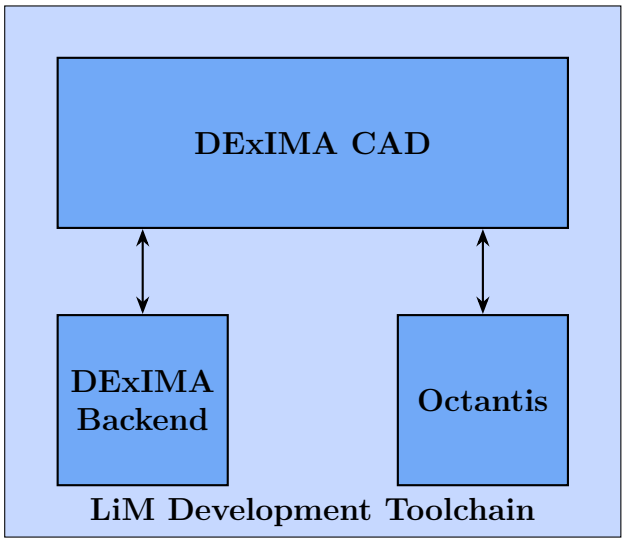


Figure 1.3: Representation of the LiM Development Toolchain developed at the Politecnico di Torino.

To help speed up the structural description process and, in turn, the time required for a complete DExIMA Backend estimation, a Graphical User Interface (GUI) front-end was developed, leading to DExIMA CAD, which will be introduced in Section 1.3.

1.3 Introduction to DExIMA CAD

DExIMA CAD has been conceived as a vital front-end support to DExIMA Backend. A set of Graphical User Interface (GUI) functionalities reduces the burden of describing the LiM array structure in the back-end input `.dex` file, speeding up the estimation process and, in turn, the architectural exploration. Its main features will be detailed in the following paragraphs.

Structural paradigm The design flow offered by DExIMA CAD is based on a well-defined structural paradigm for LiM arrays, which is based on LiM cells and on Intra-Row Logic (IRL) blocks.

In DExIMA CAD, all memory cells are LiM cells, meaning that they may integrate additional components, e.g. logic gates or basic arithmetic circuits, so that specific computational tasks may be carried out within a memory row: examples of such tasks include, but are not limited to, bit-wise logic operations and elementary arithmetic operations, e.g. addition and subtraction. Should a more complex computational task be required, a memory row may host a further logic module,

i.e. a IRL block, which may be demanded such operations as multiplication, division, additional storage and so forth. Memory rows are thus interleaved with IRL blocks, creating the structure depicted in Figure 1.4. Two different allocation patterns are observed, the LiM cells pattern, which specifies what types of LiM cells are allocated in the cells matrix, and the IRL blocks pattern.

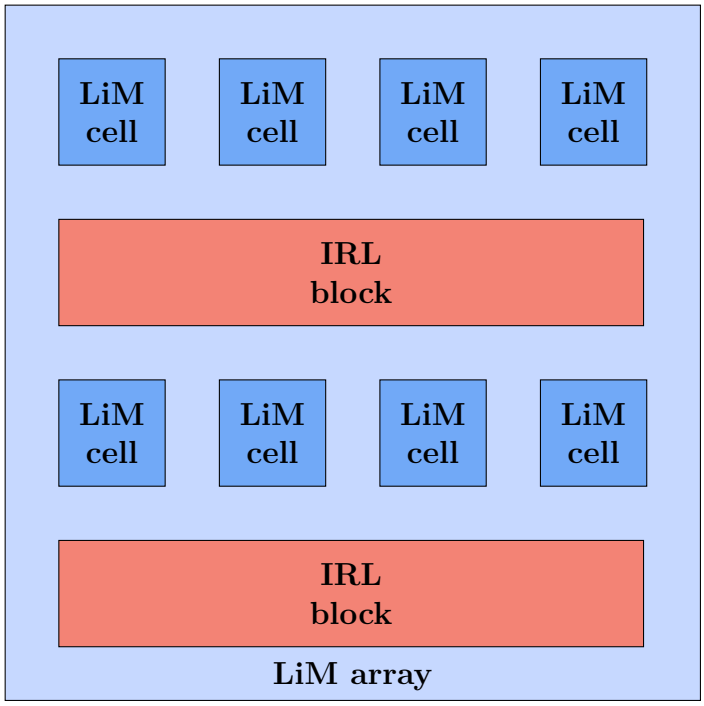


Figure 1.4: Structure of a LiM array, as intended by DExIMA CAD.

LiM array template Given that a LiM cell is typically enhanced with additional arithmetic, logic and routing components, it is expected that its I/O interface should include a set of input control signals, to properly coordinate all in-cell operations, and a set of output data signals, to provide multiple outputs to the complete array. Similar considerations also apply to the I/O interface of a IRL block.

To quickly adapt the description capabilities to new LiM arrays, DExIMA CAD offers the possibility of selecting a LiM array template, which defines the number of outputs and control signals for each LiM cell and for each IRL block, as briefly explained in the following:

- `LiM0`, `LiM1` and so on are the LiM cell outputs;
- `IRL0`, `IRL1` and so forth are the IRL block outputs;

- `S0`, `S1` and so on are the LiM cell control signals;
- `SI0`, `SI1` and so forth are the IRL block control signals.

LiM cells and IRL blocks design With reference to the structural paradigm presented in Figure 1.4, DExIMA CAD offers the possibility of quickly describing the structure of all constitutive elements of a LiM array, i.e. its LiM cells and IRL blocks, by means of a schematic editor. An example of this functionality is reported in Figure 1.5.

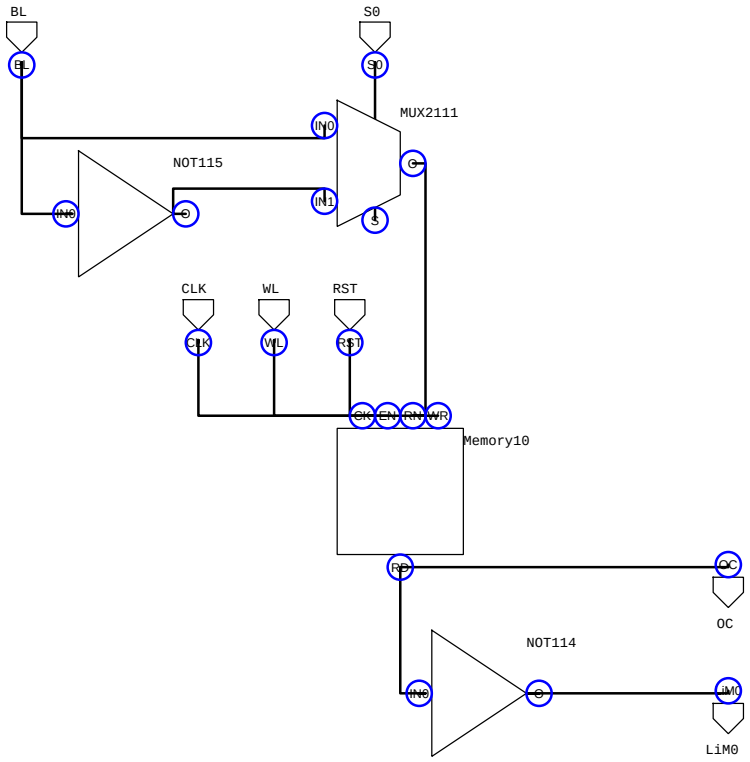


Figure 1.5: Example of LiM cell described by means of the DExIMA CAD schematic editor. The presented structure has been conceived for mere representation purposes.

The schematic editor generates a set of files, which may be referred to as "DExIMA design files", which contain the structure of an array element. The `.lim` extension is used to identify a LiM cell, while the `.irl` extension is associated to an IRL block. Despite this difference, the file format is the same in both cases, as Chapter 2 will thoroughly explain.

LiM array structure With its schematic editor, DExIMA CAD allows to specify the local structure of a LiM array, i.e. the structure of its constitutive elements.

With reference to the structural paradigm, the LiM cells and the IRL blocks patterns must be specified. In DExIMA CAD, this task is fulfilled by two different Comma-Separated Values (CSV) files, which may be generated by a GUI functionality and which represent the cells and the blocks patterns. An example of such files is reported in the following.

```
lim_array.csv                                   | lim_array_intrarow.csv

cell1,cell1,cell1,cell1,cell1,cell1,cell1,cell1 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
cell2,cell3,cell3,cell3,cell3,cell3,cell3,cell3 | irl_block
```

It is immediate to understand that the textual description presented in the above example may be used by DExIMA CAD to infer the global structure of the LiM array, as presented for instance in Figure 1.4. Moreover, as the generation of the previously mentioned CSV files is aided by a GUI functionality, it is extremely easy to handle the structural description process of a large variety of LiM structures.

Simulation-time behaviour of a LiM array In DExIMA CAD, the simulation-time behaviour of a LiM array is defined by a micro-programmed control unit based on an algorithm Read-Only Memory (ROM). While the interaction between a LiM array and its control unit will be thoroughly detailed in Chapter 5, it is important to point out that a DExIMA CAD GUI functionality may be employed to describe the algorithm to be implemented by a designed LiM array.

An algorithm comprises a sequence of operations, which are referred to as ”instructions“. Each instruction requires a set of values for all control signals in the LiM array, i.e. the LiM and the IRL control signals. In practice, an instruction is a ”scalar control“, which specifies the behaviour of an active array element. In an SIMD-like processing, the same operation is likely to be executed by multiple array elements at a time: for this reason, each instruction requires an activation

pattern, which is used to vectorize the scalar control and extend it to a set of array elements.

All these concepts are exemplified by Figure 1.6, which shows the previously mentioned DExIMA CAD GUI functionality. Each column is an instruction and it contains all the required control signals values. For each instruction, an activation pattern may be selected to extend the scalar control to multiple array elements.

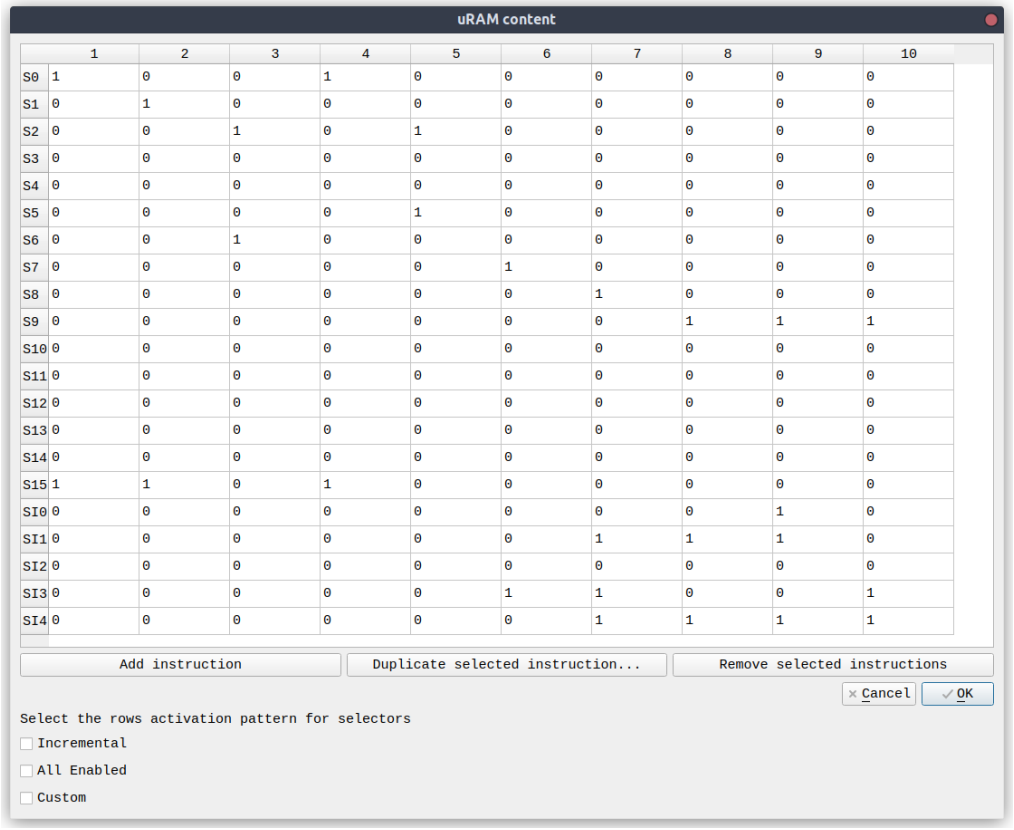


Figure 1.6: DExIMA CAD GUI functionality for defining the simulation-time behaviour of a LiM array.

Synthesis tasks When both the structure of the LiM array and its simulation-time behaviour are defined, DExIMA CAD may carry out its primary synthesis tasks, which pave the way to the simulation of the designed LiM architecture and to the estimation of its main figures of merit. For the former activity, the front-end generates the VHDL code of the complete LiM architecture, which may be then simulated with a Universal Verification Methodology (UVM) testbench; for the latter, it creates a `.dex` description of the LiM array, preparing the estimation with DExIMA Backend.

Comparison to a traditional von Neumann solution An interesting functionality is integrated by DExIMA CAD to show the effectiveness of the LiM paradigm for a selected algorithm. To fulfill this analysis task, DExIMA CAD relies on *gem5* [12], an open-source computer architectures simulator, and on *Cacti* [1].

The C programming language is used to describe the target algorithm, the execution of which on a traditional von Neumann architecture may be simulated with *gem5*, assuming a RISC-V CPU. This simulation, which requires a prior specification of the L1 and L2 cache sizes, is used to extract such quantities as the CPU execution time, the number of cache accesses and the hit rates. Then, a further simulation with *gem5* is carried out, taking as input the following algorithm.

```
#include <stdio.h>

int main(){
        volatile int memory_content[128] = {0};
        volatile int data;

        for(int i = 0; i < 128; i++){
                data = memory_content[i];
        }

        return 0;
}
```

The above algorithm strives to model the architectural paradigm presented in Figure 1.2, where the memory is actually endowed with computational capabilities. As a consequence, the CPU is not requested to perform any actual processing, but is only required to move the to-be-elaborated data to the memory array. This simulation, which requires a prior specification of the number of rows in the LiM array (128 in the above example), extracts the same quantities as the previous one, preparing a comparison process.

With the above simulations, DExIMA CAD is able to evaluate the effect of the LiM paradigm on the number of memory accesses, on the required processor instructions and on the execution time. The comparison process is completed by an interaction with *Cacti*, leading to an energy-wise evaluation of the LiM processing for the target algorithm.

Design flow in DExIMA CAD All previous paragraphs have described the main functionalities of DExIMA CAD, which add up to its design flow. For the sake of completeness, the design flow is summarized in the following.

- A working directory, or project path, is specified, in which all files generated

and required by DExIMA CAD will be found.

- A LiM array template is selected, specifying the number of LiM and IRL control signals and outputs.
- The schematic editor is used to implement the required array elements. All implemented design units are saved to `.lim` and `.irl` files, so that they may be used at a later stage.
- The LiM array structure is specified in the two previously mentioned CSV files.
- The functionality presented in Figure 1.6 is used to define the simulation-time behaviour of the LiM array.
- The VHDL code of the LiM architecture is generated and the design is simulated with the DExIMA CAD UVM testbench.
- The `.dex` description of the LiM array is generated and is passed to DExIMA Backend, which estimates its main figures of merit.

1.4 Aim of the thesis

Aim of the thesis The main features of DExIMA CAD and the most significant details on the design flow it offers have been discussed in Section 1.3. It is noteworthy to stress that it certainly is an interesting tool, which could be useful in aiding the design process of a LiM system, although some room for improvement was noticed.

The aim of this thesis is twofold. Firstly, the functionalities of DExIMA CAD are expanded, improving its structural description and architectural exploration capabilities, in order to provide a more robust support for the LiM design flow. Secondly, the revisited front-end is employed to implement a variety of LiM structures and algorithms, possibly fostering the SIMD paradigm.

Rationale behind this work Besides the aim of thesis, it is important to stress the rationale behind all presented work.

Initially, a set of existing LiM architectures was considered. For these structures (which include [9], [11], [16] and [17]), the feasibility of a complete DExIMA CAD implementation was analyzed, showing specific flaws which would have made their description unattainable by the tool. Thus, the required functionalities and improvements were inferred from these case studies and were then integrated in DExIMA CAD. The implementation of some of the mentioned case studies, including [11] and [16], proves the effectiveness of the new functionalities.

During the development of the present versions of DExIMA Backend and DExIMA CAD, the front-end has grown to become not only a mere Graphical User Interface (GUI) overlay, but also a vital support to the back-end. Evidently, the front-end consists of GUI sections and of computational components. In this context, a "computational component" is a module that does not require the full GUI overlay to carry out its tasks.

The integration of a new functionality in DExIMA CAD requires a certain degree of modifications to the existing source code: special attention must be paid to how the new DExIMA CAD code is introduced. In fact, the new modules should strive to be well-separated from all existing DExIMA CAD modules, with as little mutual interaction as possible, whilst still guaranteeing the intended behaviour. This is indeed necessary to avoid interference with already existing functionalities, to increase the maintainability of the code and to foster future expansions and improvements.

Within a new module, it would be advisable to provide a clear separation between its GUI components and its computational sections. From a general point of view, these types of elements are not expected to directly communicate with each other. Any new DExIMA CAD module introduced in this thesis prepares an internal file-based interface, which is typically fed by a set of files generated by the GUI components; the computational sections do not act on any GUI elements, but rather on existing files, meaning that they do not require the full DExIMA CAD GUI overlay. This approach has been specifically followed to foster future scripting functionalities in DExIMA CAD.

Structure of the thesis The second part of this thesis addresses all changes to DExIMA CAD, leading to what will be referred to as "revisited DExIMA CAD", which is used in the third part of this thesis to provide some implementations.

The structural description capabilities are enriched by the array interconnections module presented in Chapter 3, while the architectural exploration capabilities are enhanced by the top-level analyzer module discussed in Chapter 5: these changes are supported by underlying graph representations, which are introduced in Chapter 2. The algorithm description facilities, as seen for instance in Figure 1.6, are improved by the changes detailed in Chapter 4.

Variable-complexity case studies are then implemented in the revisited DExIMA CAD to show its effectiveness. Chapter 6 presents the components of the in-memory architecture discussed in [11], while Chapter 7 discusses the full DExIMA CAD implementation of the Hybrid-SIMD architecture, the general-purpose co-processor presented in [16].

The new front-end features are then used to implement *ex novo* in-memory solutions for the Secure Hash Algorithm (SHA) and the Advanced Encryption Standard (AES), in Chapter 8 and Chapter 9, respectively.

Results, concluding remarks and possible improvements are finally presented in

Chapter 10 and in Chapter 11.

Part II

Additional functionalities in DExIMA CAD

Chapter 2

Graph representations in DExIMA CAD

The purpose of Chapter 2 is to provide the most significant details on the DExIMA CAD graph representations, which has been introduced to support specific connectivity-related analysis and synthesis task in DExIMA CAD.

Section 2.1 introduces the structure of a DExIMA CAD design file. Section 2.2 and Section 2.3 present the models used in the graph representation, which is described in Section 2.4.

2.1 DExIMA CAD design files

To help speed up the design process of a LiM array, DExIMA CAD embeds a schematic editor, which offers two primary design contexts, the "LiM cells" and the "Intra-Row Logic" contexts. In the former, the designer can allocate an arbitrary number of single-bit components, which include logic gates, half-adders, full-adders, storage and routing elements; in the latter, it is possible to instantiate multi-bit components, e.g. adders, multipliers, registers, shifters, look-up tables, multiplexers and so forth. Regardless of the design context, all allocated components may be connected to one another by means of local connections and, besides, to the external I/O pins which will be used to interface the designed unit with other external components.

DExIMA CAD uses a textual description to represent the complete structure of an array element, i.e. a LiM cell or a IRL block, creating `.lim` or `.irl` files. Hereinafter, these types of files will be referred to as "DExIMA CAD design files".

DExIMA CAD design files are generated every time the designer decides to save the current content of the schematic editor. Despite a different file extension, both types of files use the same format to track the position and the nature of each component in the schematic editor and, in addition, the local connections. A

simple example of `.lim` or `.irl` file format is reported in the following.

```
BL 720.0 500.0 Ext 9 Input
 BL[1] :
CLK 540.0 520.0 Ext 9 Input
 CLK[1] :
RST 660.0 520.0 Ext 9 Input
 RST[1] :
WL 600.0 520.0 Ext 9 Input
 WL[1] :
OC 680.0 840.0 Ext 9 Output
 OC[1] :
Memory_19 680.0 660.0 Memory 1
 CK[1] : CLK.CLK
 EN[1] : WL.WL
 RN[1] : RST.RST
 WR[1] : BL.BL
 RD[1] : OC.OC
```

In a DExIMA CAD design file, every item in the schematic editor is represented by an header line, which primarily reports its unique identifier (i.e. a label or a name), its type and its position in the schematic editor. Two primary types of items may be identified:

- external I/O pins (`BL`, `CLK`, `RST`, `WL` and `OC` in the above example);
- actual components (`Memory_19` in the above example), which could either be single-bit or multi-bit components.

After each item header line, a DExIMA CAD design file reports a set of connection lines, which are used to specify the necessary local connections. For instance, with reference to the above example, connection line `CK[1] : CLK.CLK` indicates that pin `CK` belonging to the actual component `Memory_19` is connected to the external input pin `CLK`.

An equivalent graph representation may be associated to a DExIMA CAD design file, leading to what will be referred to as "DExIMA CAD graphs". Since each description of an array element may contain either external I/O pins or actual components, it seems reasonable to model these two different contributions, laying the groundwork for a complete DExIMA CAD graph representation.

2.2 I/O pins and bit-widths

Section 2.1 has mentioned that a DExIMA CAD design file contains a set of external I/O pins, for which a proper model should be defined. This model is inspired by the

traditional port representation in VHDL and in Verilog. Each pin is represented by a name, a direction, i.e. input or output, and by a bit-width. Several situations may be encountered, as shown by the following examples.

```
entity Adder is
        generic (
                nbit: integer := 8
        );
        port (
                A   : in  std_logic_vector(nbit-1 downto 0);
                B   : in  std_logic_vector(nbit-1 downto 0);
                AS  : in  std_logic;
                SUM : out std_logic_vector(nbit-1 downto 0)
        );
end Adder;
```

```
entity AES_GalMult2 is
        port (
                INPUT_BYTE  : in  std_logic_vector(7 downto 0);
                OUTPUT_BYTE : out std_logic_vector(7 downto 0)
        );
end entity;
```

In the first of the above examples, the component I/O interface consists of single-bit and multi-bit pins. For the latter type, the actual width of the bit vector is parametric, meaning that it is controlled by a parameter, i.e. a `generic`.

In the second of the above examples, the component I/O interface consists of multi-bit pins, but the actual width of the bit vector is no longer parametric, as in the first example, but rather is fixed.

Besides the previously described situations, a further need arises when managing the width of multi-bit pins, which is shown by the following example.

```
entity top is
        port (
                CLK         : in  std_logic;
                RST         : in  std_logic;
                BL          : in  std_logic_vector(15 downto 0);
                EN          : in  std_logic;
                WL          : in  std_logic_vector(0 to 15);
                queueIN     : in  std_logic_vector(15 downto 0);
                queueWen    : in  std_logic;
                LiMActivate : in  std_logic;
                MEM         : out std_logic_vector(0 to 255)
```

```
        );
end entity;
```

For multi-bit pins, the bit vector direction must be specified, so to cover two possible situations:

- `std_logic_vector(15 downto 0);`
- `std_logic_vector(0 to 15).`

All the above considerations lead to `class PinIO`, which may be used to model any I/O pin in DExIMA CAD and whose structure is summarized in Figure 2.1.

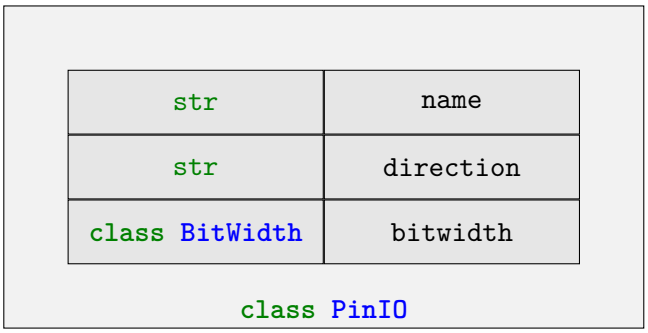


Figure 2.1: Main attributes of `class PinIO`.

To model a pin bit-width, a dedicated class, i.e. `class BitWidth`, is introduced. In truth, this class represents the root of an inheritance line, which is used to cover the previously identified situations:

- `class BitWidth` is the base class, which may be used to model the bit-width of single-bit pins;
- `class BitWidthVector` is derived from the base class and it may be used to represent multi-bit pins;
- `class BitWidthVectorFixed` is derived from `class BitWidthVector` and it is used to model such bit-widths as `std_logic_vector(7 downto 0);`
- `class BitWidthVectorParametric` is derived from `class BitWidthVector` and it is used to model such bit-widths as `std_logic_vector(nbit-1 downto 0)`, i.e. parametric bit-widths.

2.3 Components library

As specified in the introduction to Section 2.1, a large variety of components can be allocated in the DExIMA CAD schematic editor. For the purposes of building a graph representation, DExIMA CAD must know the main features of these components, including for instance their I/O interface. Thus, each library component is associated an equivalent model, and a collection of models is stored by DExIMA CAD in what will be referred to as "components library". In fact, as Section 2.4 will better clarify, components play a major role in the DExIMA CAD graph representation and, in this context, their role is twofold;

1. they are part of the DExIMA CAD graph itself, being the graph nodes;

2. they aid the graph construction process, by readily reporting its I/O interface to the module devoted to this specific task.

The components library is extremely important in constructing a DExIMA CAD graph, as it provides the blueprint for creating each graph node, which must include a clear indication of its I/O interface.

Component models A reasonable model of the main component features may be obtained by mimicking a VHDL `entity`, which is always represented by a name, i.e. an unique identifier, and by a description of its I/O interface, indicating what ports are used to communicate with other external elements. Each port may be easily represented by a properly set instance of `class PinIO`, while the entity name is simply a string. However, one point is still to be addressed. A design may contain several instances of the same library component, hence the need of a further attribute, acting as an unique identifier. These considerations add up to the structure presented in Figure 2.2, where `type` indicates the component type, `name` is the unique identifier, while `pins_input` and `pins_output` store the references to instances of `class PinIO`.

In truth, `class Component` is the root for a set of derived classes, including:

- `class SingleBitComponent`, which models single-bit components;

- `class MultiBitComponent`, which represents multi-bit component.

A multi-bit component may accept a set of parameters, which mimic the VHDL `generic` mechanism, so that the bit-width of its ports may be managed in a parametric manner. Additional parameters may be required by specific multi-bit components, e.g. the shift amount of a shifter.

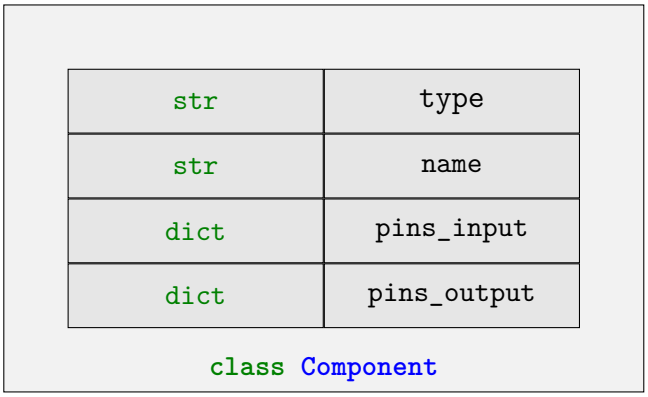


Figure 2.2: Main attributes of `class Component`.

Components library All required component models are stored by DExIMA CAD in a components library, which is defined in one of its internal modules. Since DExIMA CAD currently supports the NanGate 45 nm library only, the definition is actually static, meaning the features of all its components are statically written in the dedicated files.

The following example shows how single-bit components may be handled in the internal DExIMA CAD library.

```
component = SingleBitComponent('AND2')
component.appendPin('Input', 'IN0')
component.appendPin('Input', 'IN1')
component.appendPin('Output', 'O')
library_components['AND2'] = component
```

In the above example, an instance of `class SingleBitComponent` is created by specifying the component type, i.e. `AND2`. Then, the I/O interface is listed with the `appendPin()` method which, for single-bit components, only requires the pin direction and the pin name. Lastly, the created instance is appended to the internal components library.

The following example shows how multi-bit components may be handled in the internal DExIMA CAD library.

```
component = MultiBitComponent('Adder')
component.appendParameter('nbit')
component.appendPin(
        'Input',
        'A',
        BitWidthVectorParametric(
                BITWIDTH_DIRECTION_DOWNTO,
```

```
                'nbit',
                ONE2ONE
        )
)
component.appendPin(
        'Input',
        'B',
        BitWidthVectorParametric(
                BITWIDTH_DIRECTION_DOWNTO,
                'nbit',
                ONE2ONE
        )
)
component.appendPin(
        'Input',
        'AS',
        BitWidth()
)
component.appendPin(
        'Output',
        'SUM',
        BitWidthVectorParametric(
                BITWIDTH_DIRECTION_DOWNTO,
                'nbit',
                ONE2ONE
        )
)
library_components['Adder'] = component
```

With respect to the previous example, the declaration of a multi-bit component is evidently more complex. Similarly to the previous case, an instance of `class MultiBitComponent` is created by specifying the component type, i.e. `Adder`. Prior to the I/O interface declaration, all required parameters must be defined by means of the `appendParameter()` method. Then, all component pins may be added to the model, by carefully specifying the characteristics of the bit-width. Lastly, the created instance is appended to the internal components library.

2.4 DExIMA CAD graphs

A DExIMA CAD graph thoroughly models the content of a DExIMA CAD design file, taking into account not only the actual components and the external I/O pins, but also their mutual connections. This graph representation has the primary

purpose of aiding specific connectivity-related analysis and synthesis tasks implemented in the revisited DExIMA CAD. They thus offer a systematic approach for tackling connectivity issues. For instance, these graphs may be used by the array interconnections module (Chapter 3) to quickly identify all components involved in an array interconnection.

The DExIMA CAD graph representation is supported by the data structures and the algorithms offered by [5].

2.4.1 Structure of a DExIMA CAD graph

Each graph node is either an actual component or an external I/O pin, while each graph edge derives from the set of connection lines. Regardless of the actual implementation, a graph node is characterized by a name, i.e. an unique identifier, a type, which reports the component type, and a set of input and output pins. An abstract representation of these concepts is presented in Figure 2.3.

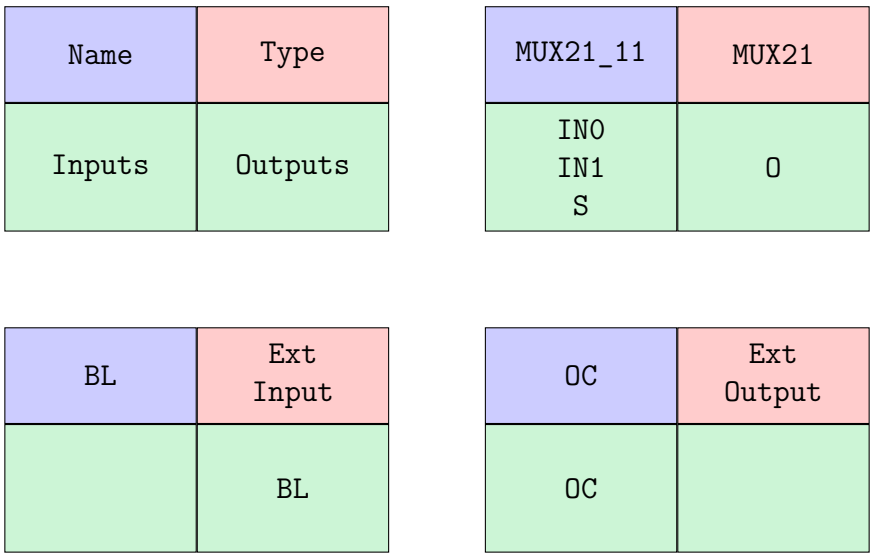


Figure 2.3: Abstract representation of nodes in DExIMA CAD graphs. In the top-left corner, a general representation of the class attributes. In the top-right corner, an example of actual component (a 2-to-1 single-bit multiplexer). In the bottom-left corner, an example of external input pin. In the bottom-right corner, an example of external output pin.

All edges in a DExIMA CAD graph represent the local connectivity of the associated array element. In this context, an edge must not only indicate what nodes should be connected, but also what pins are involved in the local connection. As a consequence, each edge is characterized by its source node, its destination node, the source pin and the destination pin, as depicted in Figure 2.4.

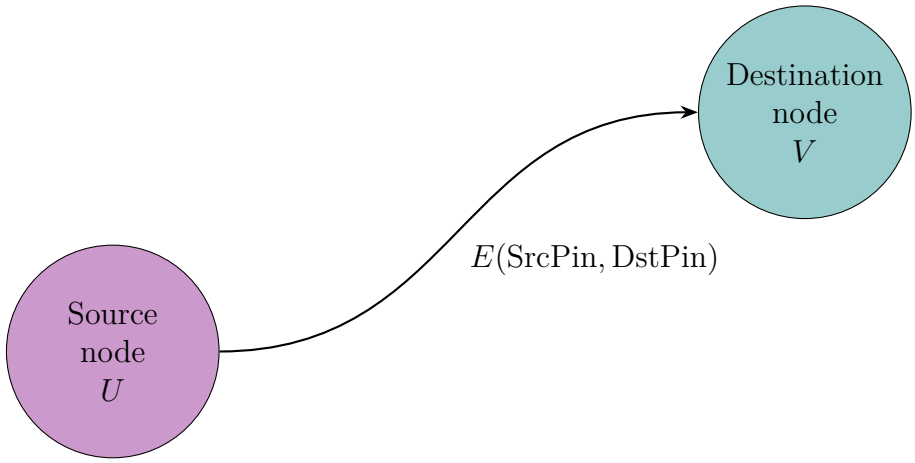


Figure 2.4: Abstract representation of edges in DExIMA CAD graph. A source node U and a destination node V are connected by an edge E, which is parametrized by the source pin SrcPin and by the destination pin DstPin.

All the above considerations eventually lead to the current implementation of a DExIMA CAD graph, which is represented by an instance of `class DExIMAGraph`.

2.4.2 Graph construction procedure

The structure of a DExIMA CAD graph is directly inferred from its associated DExIMA CAD design file, which is parsed and analyzed to return the corresponding graph representation. The parsing procedure is demanded to a parser, i.e. an instance of `class ParserFileDExIMA`, and it consists of two subsequent sub-phases, which will be detailed in the following.

First sub-phase In the first sub-phase, all connection lines are temporarily discarded and only the item header lines are taken into account. With some format-specific manipulation, the header line is primarily used to determine the type of component and its name, so that a proper graph node may be allocated. If an external I/O pin is found, the parser allocates an instance of `class PinIO`. On the other hand, an actual component will be represented by an instance of `class Component`.

The allocation of an actual component must be properly handled by the parser, since in the present format of a DExIMA CAD design file it is not possible to distinguish input and output pins in a connection line. For this reason, each graph node of this kind must also report its I/O interface, as depicted in Figure 2.3, which may not be inferred directly from the DExIMA CAD design file. To fulfill this task, the parser scans the components library to retrieve a copy of that component, which contains the pieces of information reported in Figure 2.3.

Second sub-phase At the end of the first sub-phase, the graph contains all required nodes, i.e. actual components and external I/O pins, but edges are still to be appended. This task is fulfilled by the second parsing sub-phase, in which all item header lines are discarded and only the connection lines are considered.

For any item in the DExIMA CAD design file, each connection line refers to one of its pins, which naturally is one end of the edge, and it has a LHS and a RHS: the former is the item pin, while the latter indicates the other end of the edge, by specifying the other item involved in the connection and its pin. In some cases, the RHS may be empty; from a more general point of view, it may contain one or more items.

The edge direction is actually determined by the considered pin and, more specifically, by its direction: if the pin belongs to the input interface of the component, then the edge will lead to the component, otherwise it will start from it.

The following example may be considered to clarify the above statements.

```
FA_28 1080.0 1080.0 FA 6
 A[1] : XOR2_22.O
 B[1] : XOR2_21.O
 C[1] : MUX21_35.O
 S[1] :
 CO[1] : TO_LEFT.TO_LEFT MUX21_29.IN1
```

All the above elements are found in the above example. Five connection lines are associated to component `FA_28`, which is in fact a full-adder. The first three connection lines refer to its input pins and will create three graph edges, sharing the same destination component and originating from components `XOR2_22`, `XOR2_21` and `MUX21_35`. The last two connection lines refer to the output pins. Two edges departing from the component will be created, one leading to item `TO_LEFT` and one to item `MUX21_29`.

2.4.3 Connectivity analysis

A DExIMA CAD graph may be useful in carrying out specific connectivity-related analysis tasks. In fact, the graph representations may help determine whether the local connectivity is consistent.

In this context, a ”consistent“ graph meets the following conditions:

- no edge should have the same source and destination nodes;
- no edge should have an external input pin as its destination node;
- no edge should start from and external input pin and lead to the output pin of an actual component;

- all edges leading to an external output pin must start from the output pin of a component.

A violation of any of the above rules leads to an ”inconsistent“ graph. A flawed connectivity must be properly identified in the structural description process, as it may be responsible for unwanted effects which lead to an incorrect functionality of the whole LiM array. It should be pointed out that the set of constraints can be modified at will, so to implement a stricter and more refined consistency check policy.

An instance of `class DExIMAGraph` offers the `checkConnectivity()` method, which verifies if the above conditions are met and returns the consistency status of the graph.

Chapter 3

LiM array interconnections in DExIMA CAD

The purpose of Chapter 3 is to detail a new functionality in DExIMA CAD, which is meant to support the integration of interconnections between the elements of a LiM array.

The global structure of DExIMA CAD fosters a specific LiM processing paradigm. To endow a memory array with computational capabilities, additional logic elements may be integrated in the array itself, for instance within or near a memory row. Nevertheless, the mere allocation of further components is not sufficient to guarantee a specific set of functionalities in the LiM array. In fact, in virtually any processing element, e.g. the datapath of a microprocessor, all processing capabilities are determined not only by the allocated sub-systems, but also by their mutual connections. This considerations also applies to LiM arrays, as the analysis of several LiM architectures may show. To prove this statement, some prime examples may be considered.

- In the Configurable Logic-in-Memory Architecture (CLiMA) presented in [9], the possibility of integrating additions in the modified memory cells derives from the interconnections of the in-cell full-adders, creating a link for the propagation of the carry signal and, consequently, a ripple-carry adder architecture.
- In the Hybrid-SIMD architecture presented in [16], the communication between smart rows, composed of arithmetic memory rows and their respective row interfaces, and standard memory rows is enabled by specific interconnections, leading to the implementation of specific data movement patterns.

Connections between the elements of a LiM array are thus essential to provide specific processing capabilities. Nevertheless, the past version of DExIMA CAD provided a rather limited support for array interconnections. Figure 3.1 provides a

representation of the available interconnections, which are explained in the following:

- a LiM cell can be connected to the very next IRL block;
- a IRL block can be forwarded to the input of the very next IRL block.

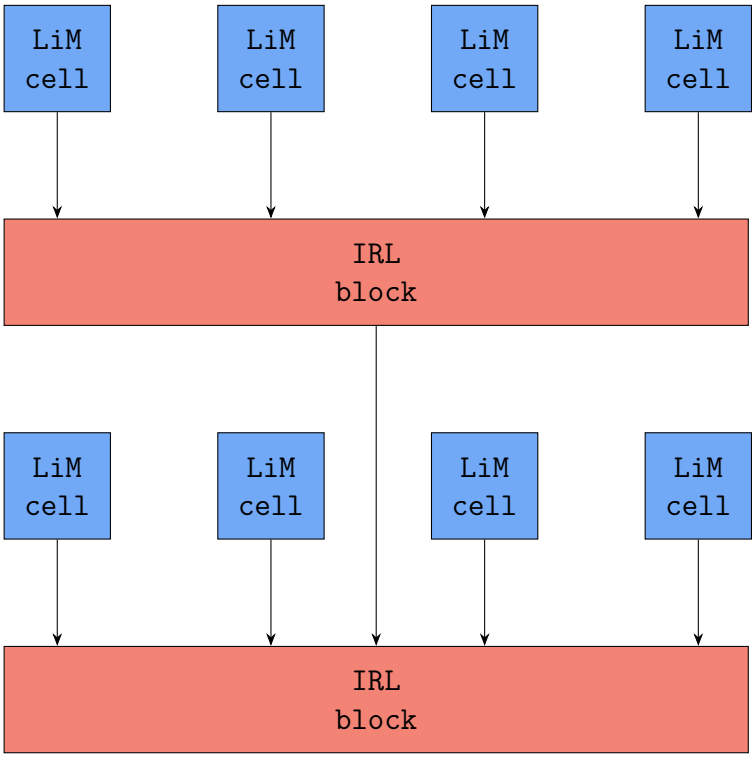


Figure 3.1: Existing array interconnections in the past version of DExIMA CAD.

Because of this very limited support for array interconnections, DExIMA CAD would struggle in the description of rather complex LiM array architectures, especially if refined data movement patterns are to be implemented. As a consequence, new structural description capabilities must be provided to DExIMA CAD.

Section 3.1 introduces a taxonomy of array interconnections. Given the state of DExIMA CAD and DExIMA Backend, some issues may arise when trying to integrate array interconnections, as detailed in Section 3.2. The new DExIMA CAD array interconnections module is then described in Section 3.3.

3.1 Taxonomy of LiM array interconnections in DExIMA CAD

3.1.1 Vertical interconnections

The taxonomical process is started by the analysis of the existing array interconnections in DExIMA CAD, as detailed by Figure 3.1. Apparently, the supported interconnections only move in the vertical direction, i.e. along the memory rows and the IRL blocks. Moreover, the available interconnections may only move from a memory row to its associated IRL block or from a IRL block to its very next IRL block. These considerations may be already of use in identifying the constitutive elements of a vertical interconnection, namely its source, its destination, its direction and its displacement.

If the Row-to-IRL block interconnection of Figure 3.1 is considered, then a further constitutive element may be identified. In fact, this type of interconnection may involve any LiM cell output pin, including the `OC` (Output Cell) pin and the template output pins, e.g. `LiM0`, `LiM1` and so forth. Besides the above elements, a vertical interconnection may be also characterized by its source pin. Evidently, the source pin is template-dependent: should the number of LiM control signals be increased in the template selection procedure, the designer may select the source pin from a broader set.

The main limitation of the existing array interconnections in DExIMA CAD is that their parameters, i.e. their constitutive elements, are rather constrained and not truly selectable by the designer. In fact, the extent of a vertical interconnection is always limited to at most one array element, the only allowed destination is a IRL block and their direction is always "downwards". It follows that the flexibility of DExIMA CAD may be largely increased if the constitutive elements are made truly parametric.

The taxonomical process thus leads to the vertical interconnection parameters presented in Table 3.1.

Name	Identifier	Possible values
Source	`Source`	`{Row, IRL}`
Destination	`Destination`	`{Row, IRL}`
Direction	`Direction`	`Prev, Next, Fixed`
Source pin	`SourcePin`	Template- and source-dependent
Displacement	`Displacement`	Positive value

Table 3.1: Constitutive elements of a vertical interconnection in DExIMA CAD.

With respect to Figure 3.1, memory rows and IRL blocks can both be the source

and the destination of a vertical interconnection, implying that all LiM array elements may be mutually connected. The interconnection pattern that may be implemented by DExIMA CAD is truly arbitrary, thanks to the parametric management of both the direction and the displacement of a vertical interconnection. The parameters of a vertical interconnection are summed up in the interconnection name, which is derived by concatenating the values of its constitutive elements. Examples of possible vertical interconnections in DExIMA CAD are presented in Figure 3.2.

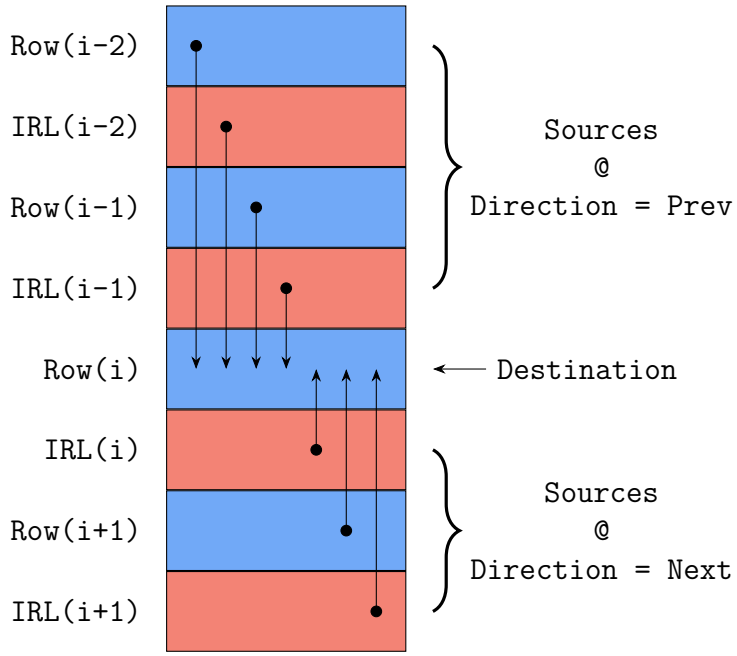


Figure 3.2: Examples of vertical interconnections in DExIMA CAD.

In truth, the representation provided by Figure 3.2 is slightly imprecise, as each arrow actually identifies a set of vertical interconnections, which are based on a different template-dependent pin. It is important to point out that the coupling between the LiM array template selection and the parameters of a vertical interconnection is beneficial for the flexibility of DExIMA CAD. Should the designer need multiple connections between two arbitrary array elements, the output signals may be increased in the template selection, so that multiple links may be integrated in the LiM array.

3.1.2 Horizontal interconnections

With respect to the examples presented in the introduction to Chapter 3, an important matter is still to be addressed. Without any further change to DExIMA CAD, it would be impossible to tackle the description of the CLiMA architecture

presented in [9], as a proper mechanism is still to be devised to provide an interconnection between the in-cells full-adders, creating an in-row ripple-carry adder.

To address this issue, horizontal interconnection are introduced in DExIMA CAD. These interconnections, which are depicted in Figure 3.3, are aimed at managing the in-memory implementation of bit-wise shift operations and simple arithmetic operations, e.g. addition and subtraction, and consist of the following sub-types:

- LSB-to-MSB, propagating a signal from the Least-Significant Bit (LSB) of a row to its Most-Significant Bit (MSB);
- MSB-to-LSB, propagating a signal from the MSB of a row to its LSB.

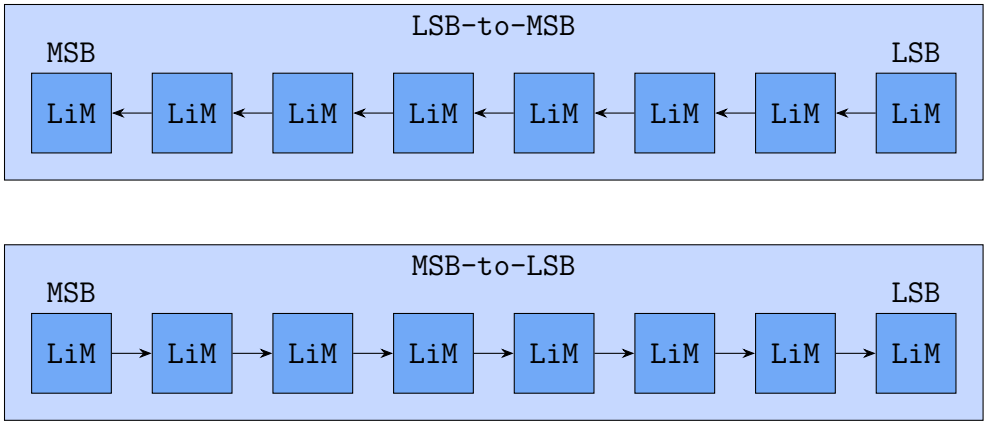


Figure 3.3: Examples of horizontal interconnections in DExIMA CAD.

3.1.3 Additional interconnections

In the LiM neural network implementation of [11], two different, yet interfaced, LiM arrays are employed, namely:

- a XNOR array, which is meant to compute the convolution between an input feature map and a kernel;
- a ones counter array, which is aimed at calculating how many ones are obtained in a convolution window.

From the point of view of DExIMA CAD capabilities, the previously mentioned arrays are rather interesting architectures. In fact, their descriptions would be possible in DExIMA CAD, but they would rely on the definition of additional LiM selectors, used as elements of a further data bus, to propagate all data to the memory array. As a consequence, this solution would have two flaws:

- the description would become cumbersome, especially for large memory arrays, as it would require the implementation of more memory cells than actually required;

- the implementation would inevitably mix data and control signals, as the values for the data signals would be specified in the content of the micro-ROM.

The extent of the above flaws can be reduced if proper interconnection mechanisms are devised, proactively changing the input interface of the memory array itself. In this context, an "additional" interconnection consists of two sub-types, namely a horizontal bus and a vertical bus, which behave as described in the following:

- the bit width of a horizontal bus is equal to the number of columns in the LiM array, and each the j-th bit of the horizontal bus may be propagated to all the rows belonging to column j in the LiM array;

- a vertical bus consists of as many bits as the number of rows in the LiM array, and the i-th bit can be propagated to all columns in row i-th of the array.

As a matter of fact, an horizontal bus is rather similar to a bitline and, alongside with the vertical bus, it may be used to create a rather simplified interconnection matrix, as shown in Figure 3.4, which can be effectively employed to speed up the DExIMA CAD description of the previously mentioned XNOR and ones counter arrays. The DExIMA CAD implementation of these architectures is deferred to Chapter 6.

3.1.4 Interconnection input and output pins

A mechanism should be devised to support the allocation of an array interconnection in the schematic editor of DExIMA CAD. When describing the structure of a LiM array, the designer must follow a well-defined steps sequence:

- the structure of all LiM cells is defined in the "LiM cells" design context of the DExIMA CAD schematic editor;

- the structure of all IRL blocks is defined in the "Intra-Row Logic" design context of the DExIMA CAD schematic editor;

- the structure of the LiM array is defined by specifying the LiM cells pattern and the IRL blocks pattern in two separate CSV files.

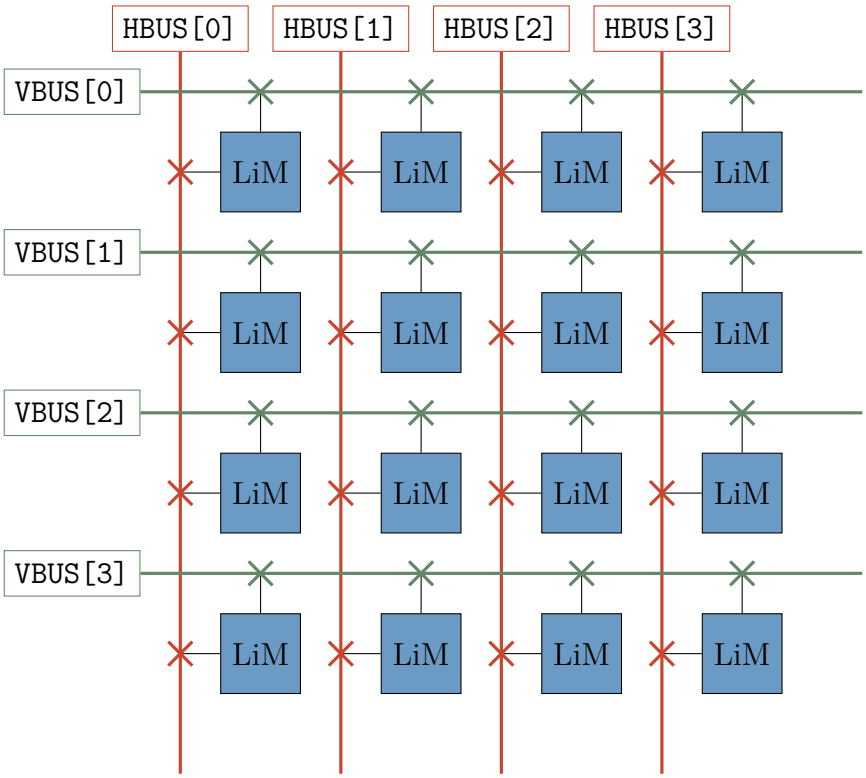


Figure 3.4: Representation of the horizontal and vertical busses in DExIMA CAD.

Given the nature of the presently supported structural description process, the designer may request an array interconnection only when defining the structure of a LiM cell or of a IRL block. The most straightforward solution is to allocate a special pin, i.e. an interconnection input pin, which is then connected to the components allocated in the element under design. Evidently, to enable the description of complex LiM arrays, multiple interconnections can be activated in the same design element, including additional, horizontal and vertical interconnections.

In the context of DExIMA CAD, an interconnection input pin is physically introduced by the designer in the schematic editor, it is a placeholder in the destination array element and it indicates to which components (e.g. full-adders, memory cells, multiplexers and so forth) the interconnection source must be routed. A pin of this kind is expected to be connected to the input ports of the destination components. On the other hand, an interconnection output pin is always a template output signal (e.g. `OC`, `LiM1` or `IRL0`) connected to the output port of a source component. These definitions are better visualized in Figure 3.5.

In a vertical interconnection, the designer may create an interconnection input pin, whose name is derived from the actual parameters of the interconnection: some examples are presented in Table 3.2.

The implementation of a horizontal interconnection relies on the definition of

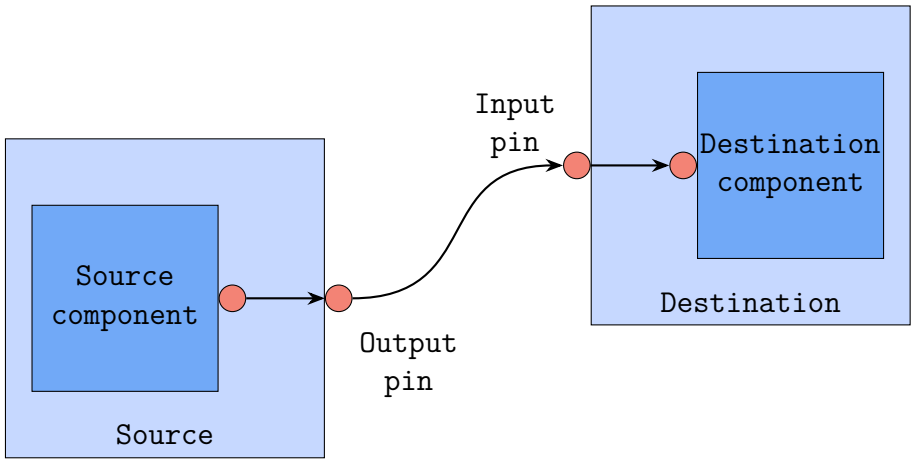


Figure 3.5: Interconnection input and output pins, source and destination components, source and destination array elements in DExIMA CAD. "Output pin" and "Input pin" are the output and input pins of the considered array interconnection, respectively.

Interconnection	Output pin	Input pin
Next-1-IRL2-to-Row	IRL2	NEXT_1_IRL2_TO_ROW
Prev-1-OC-to-Row	OC	PREV_1_OC_TO_ROW
Prev-1-LiM0-to-IRL	LiM0	PREV_1_LIM0_TO_IRL
Fixed-0-OC-IRL	OC	FIXED_0_OC_TO_IRL

Table 3.2: Examples of input and output pins for vertical array interconnections.

a pair of pins, which can indicate the propagation of a specific signal, as reported in Figure 3.3. For instance, in a LSB-to-MSB interconnection, the input pin `FROM_RIGHT` and the output pin `TO_LEFT` are created in the schematic editor and can be arbitrarily connected to the components in the LiM cell; a similar consideration holds for a MSB-to-LSB interconnection, with its `FROM_LEFT` input pin and its `TO_RIGHT` output pin.

When a horizontal bus is requested by the designer, DExIMA CAD creates the interconnection input pins `HBUS_int` or `HBUS_int_IRL`, in the "LiM cells" and in the "Intra-Row Logic" design contexts, respectively. Similarly, when a vertical bus is requested, DExIMA CAD creates the interconnection input pin `VBUS_int`.

3.2 Issues and needs

The elements presented in Section 3.1 lay the groundwork for integrating array interconnections in DExIMA CAD, by providing a clear taxonomical system and

by highlighting the intended behaviour of the new functionalities. The following paragraphs provide an overview of some of the points to be addressed if array interconnections are to be effectively integrated in DExIMA CAD.

Design context Subsection 3.1.4 has already introduced a possible solution to give the designer the possibility of allocating array interconnections in the DExIMA CAD schematic editor. One of the main consequences of this solution is that the design context is enriched with the interconnection status, i.e. the set of all active interconnections in a design context, which must be properly tracked, saved and restored, respectively during the design phase, when saving the design and when opening it at a later stage. A possible way of tracking the active interconnections is by means of a separate interconnection file, which is easily recognized in the project path thanks to its special extension `.intr`.

A possible structure for an interconnection file is presented in the following.

```
HorizontalBus:16
Fixed-85-OC-to-IRL:33
Fixed-86-OC-to-IRL:34
Fixed-87-OC-to-IRL:35
Fixed-88-OC-to-IRL:36
Fixed-84-IRL0-to-IRL:37
Fixed-81-IRL0-to-IRL:38
Fixed-82-IRL0-to-IRL:39
Fixed-83-IRL0-to-IRL:40
Prev-1-OC-to-IRL:18
```

Each line in a `.intr` file indicates an active array interconnection, which is easily recognized by its name. Besides the identifier, each line includes a separator, i.e. the colon character, and a numerical quantity. This quantity is an index, indicating the position of the associated interconnection input pin in the DExIMA CAD vector which stores all items in the schematic editor. In this way, all interconnection input pins, which are somewhat special pins in DExIMA CAD, may be easily tracked and identified, so that their dynamic creation and removal may be simplified.

Consistency The allocation of an array interconnection in any array element is evidently expected to change the global structure of a LiM array. As the process of requesting an array interconnection is directly started from the designer, a set of analysis tools must be prepared to check the consistency of the resulting structural description.

Modularity of the new computational facilities For the purposes of this description, the complete structure of a LiM array can be conceived as two-layered.

The first layer derives from the allocation of all array elements, i.e. LiM cells and IRL blocks, in the array matrix; the first layer, which will be referred to as "structural layer", only includes the local connectivity, i.e. the set of connections within an array element. The global connectivity is determined by the second layer, which encompasses all mutual interconnections between the constitutive array elements and which will be referred to as "interconnections layer". These definitions are better visualized in Figure 3.6.

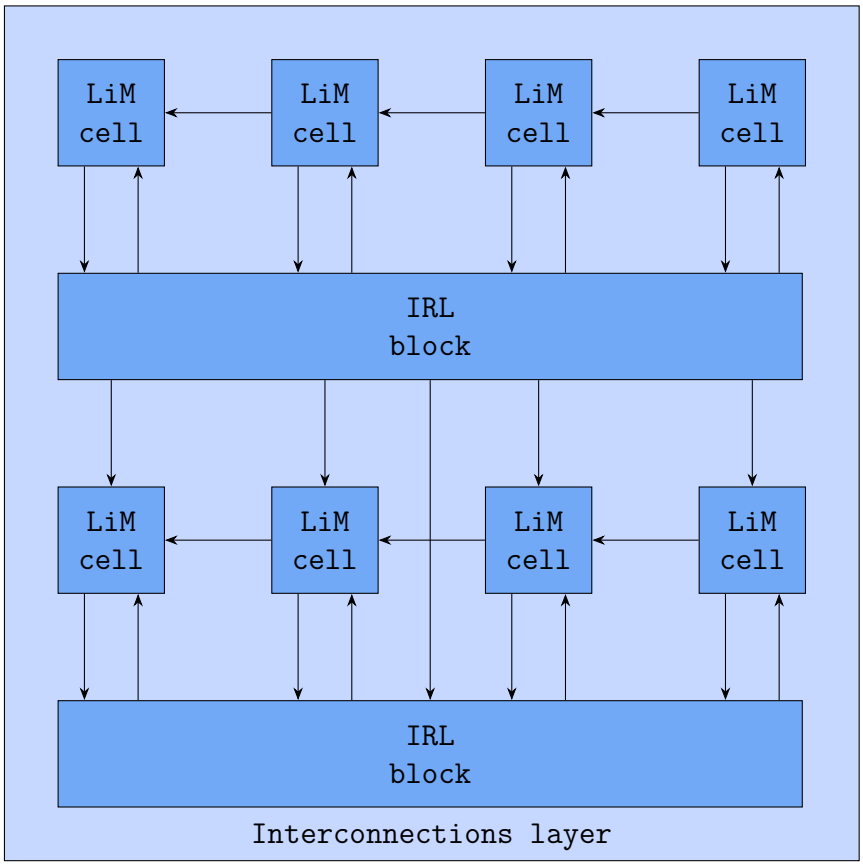


Figure 3.6: Structural and interconnections layers in a DExIMA CAD LiM array. All elements in dark blue belong to the structural layer, while all edges, which on the light blue container, belong to the interconnections layer.

The two layers are not well-separated, as no interconnection may exist if no underlying array matrix is created. Nevertheless, it would make sense to demand the analysis of these layers to separate modules. The new module in DExIMA CAD should thus strive to comply with this specific observation.

Interconnections description The primary outputs of the DExIMA CAD design process are the VHDL and the DExIMA Backend descriptions of the designed

LiM array. Evidently, only the interconnections presented in the introduction to Chapter 3, and more specifically in Figure 3.1, are currently supported. DExIMA CAD should thus be able to extend the generation capabilities by integrating the all possible types of interconnections.

As regards the generation of the two different descriptions, it is expected that the related algorithms will be characterized by different complexities. In fact, the generation of the VHDL description is supposed to be simpler, as it can rely on the syntax of a structured, consolidated and powerful Hardware Description Language (HDL), while the custom syntax of DExIMA Backend has fewer description capabilities, leading to a more complex generation policy.

Array interconnections do not lead to further components in the structural description of the LiM array, so no changes are expected in the `logic` and the `cells` sub-sections in the array `.dex` file: in fact, array interconnections are indeed supposed to be found only in the `map` sub-section of the array.

DExIMA CAD must compensate a limitation in the present version of DExIMA Backend, which derives from how timing analysis is carried out. In fact, DExIMA Backend only computes the timing for the paths specified in the `instructions` sub-section of a LiM array. With respect to the timing paths, the structural layer is rather straightforward to analyze, as the local connectivity is easily identified by means of `class DExIMAGraph`. On the other hand, a large complexity may derive from the interconnections layer: in fact, the large flexibility of vertical interconnections requires a larger scope for the timing analysis, which must involve the whole LiM array. DExIMA CAD must thus be able to enumerate all possible timing paths in the array, so that DExIMA Backend may carry out its timing analysis.

3.3 Array interconnections module

To integrate array interconnections (as presented in Section 3.1) and tackle the issues detailed in Section 3.2, DExIMA CAD is expanded by a dedicated array interconnections module, which complies with the principles presented in Section 1.4 and primarily consists of two main elements:

- a set of GUI components, which support the dynamic creation and removal of interconnection pins in the DExIMA CAD schematic editor;
- a set of computational components, which are demanded interconnections-related analysis and synthesis tasks.

An instance of `class ArrayInterconnections`, whose structure is depicted in Figure 3.7, fixes the interface for the communication with the schematic editor, which is enabled by special GUI classes, namely `class InterconnectsPanelRow` and `class InterconnectsPanelIRL`, aiding the dynamic creation and removal of interconnection pins. The schematic editor notifies the change in the design context

to a further class, namely `class StatusArrayInterconnections`, which tracks the currently active interconnections.

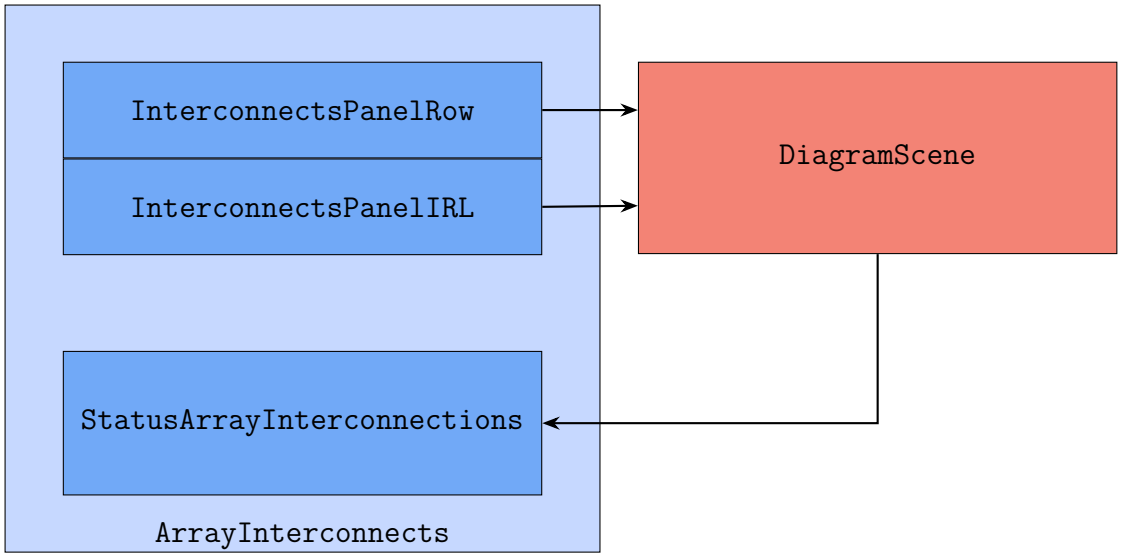


Figure 3.7: Structure of `class ArrayInterconnections` and its interaction with the schematic editor, i.e. an instance of `DiagramScene`.

An example of vertical interconnection in the DExIMA CAD schematic editor is presented in Figure 3.8. Having requested a `Next-1-IRL0-to-Row` vertical interconnection, the corresponding interconnection input pin is created in the schematic editor: this pin may then be used to indicate where the `IRL0` pin of the very next IRL block should be routed in the current design.

An example of LSB-to-MSB horizontal interconnection in the DExIMA CAD schematic editor is presented in Figure 3.9. Having requested this type of interconnection, two different interconnection pins are created in the schematic editor, namely `FROM_RIGHT` (input pin) and `TO_LEFT` (output pin). These pins are connected to the carry in and the carry out signals of a full-adder, respectively.

When the designer decides to save the content of a design context, a `.intr` file may be created by `class StatusArrayInterconnections`. In fact, if a LiM cell or a IRL block does not require any interconnection, such file is not created. At a later stage, the designer may decide the open a DExIMA design file, restoring a prior design. In such event, `class StatusArrayInterconnections` checks if a corresponding `.intr` file exists in the current project path: the absence of such file indicates that the restored array element does not require any interconnection, so no interconnection pin should be allocated and tracked.

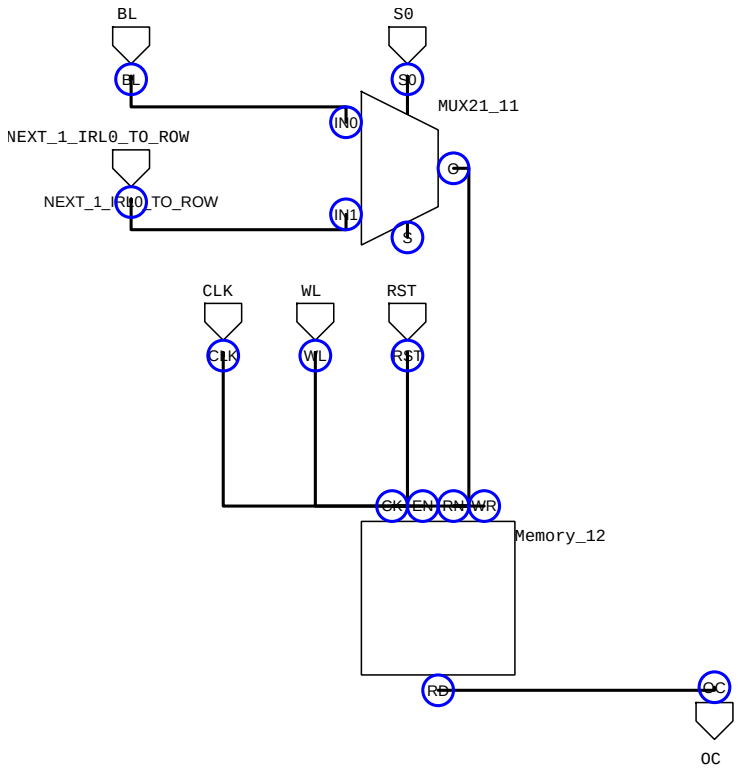


Figure 3.8: Example of a vertical interconnection in the DExIMA CAD schematic editor.

3.3.1 Array manager

The actual computational capabilities of the array interconnections module derive from an instance of `class ArrayManager`, which will be hereinafter referred to as "array manager". An array manager is expected to integrate all analysis and synthesis tasks related to the structure of a LiM array, which is represented not only from the structure of its constitutive elements, but also from their mutual interconnections. Besides answering the needs presented in Section 3.2, an array manager is also endowed with all VHDL and DExIMA generation tasks, which were previously demanded to existing DExIMA CAD modules, so to provide a complete environment for handling a LiM array.

Before any synthesis task may be offered by an array manager, a prior call to its `build()` method must be made. The build phase of an array manager consists of a set of sequential steps, which are demanded to different DExIMA CAD modules. Should any of these steps fail, the build procedure is immediately interrupted. A summary of the build phase is reported in the following.

- The array manager infers the global structure of a target LiM array from two

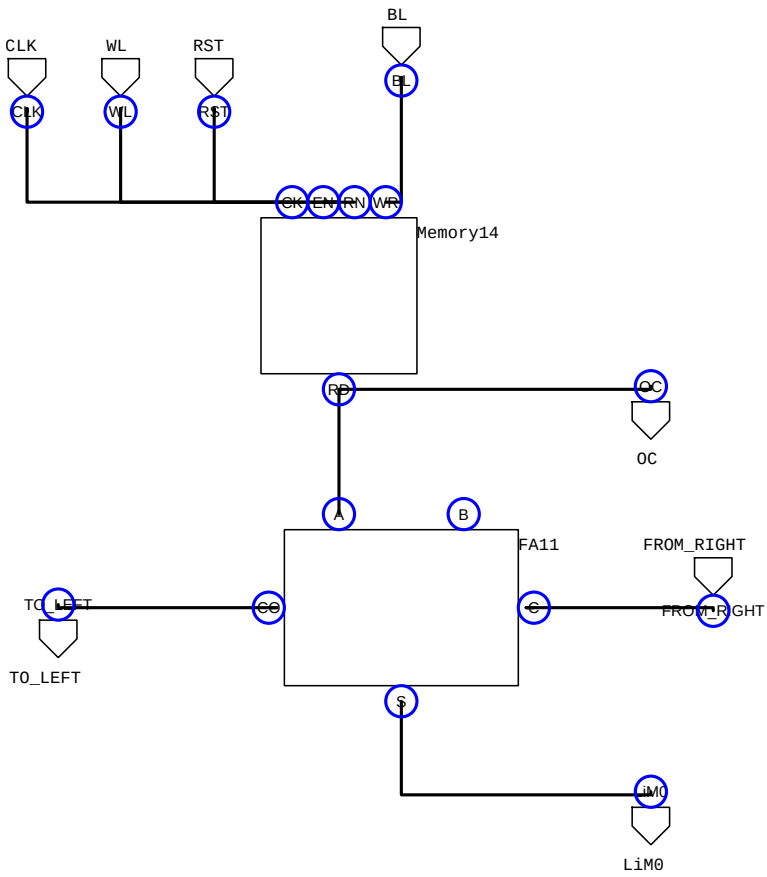


Figure 3.9: Example of a horizontal interconnection in the DExIMA CAD schematic editor.

CSV files, which describe the LiM cells pattern and the IRL blocks pattern. This procedure is demanded to an instance of `class ArrayDescriptionParser`. Further details are presented in Subsection 3.3.2.

- From the results of the previous step and from all the available DExIMA design files, the array manager infers the local structure of a target LiM array, i.e. the precise structure of all LiM cells and IRL blocks in the memory array. This procedure is demanded to an instance of `class ArrayContentParser`. Further details are presented in Subsection 3.3.3.

When the build phase is completed, the array manager may generate its primary outputs, namely the VHDL and the DExIMA Backend descriptions of the LiM array. To fulfill such synthesis tasks, the array manager coordinates the action of further DExIMA CAD modules, including:

- `class ArrayStructureAnalyzer`;

- `class ArrayInterconnectionsAnalyzer`;
- `class PathEnumerator`.

With respect to the definitions presented in Section 3.2, the structural layer is handled by a structure analyzer, i.e. an instance of `class ArrayStructureAnalyzer`, while the interconnection layer is handled by an interconnections analyzer, i.e. an instance of `class ArrayInterconnectionsAnalyzer`, and by a path enumerator, which is nothing but an instance of `class PathEnumerator`.

An array manager thus represents the uppermost level of a well-defined hierarchical structure, in which different DExIMA CAD modules perform separate and specialized tasks. This hierarchical structure is summarized in Figure 3.10, which also shows the most meaningful interactions between the array manager and other elements in DExIMA CAD.

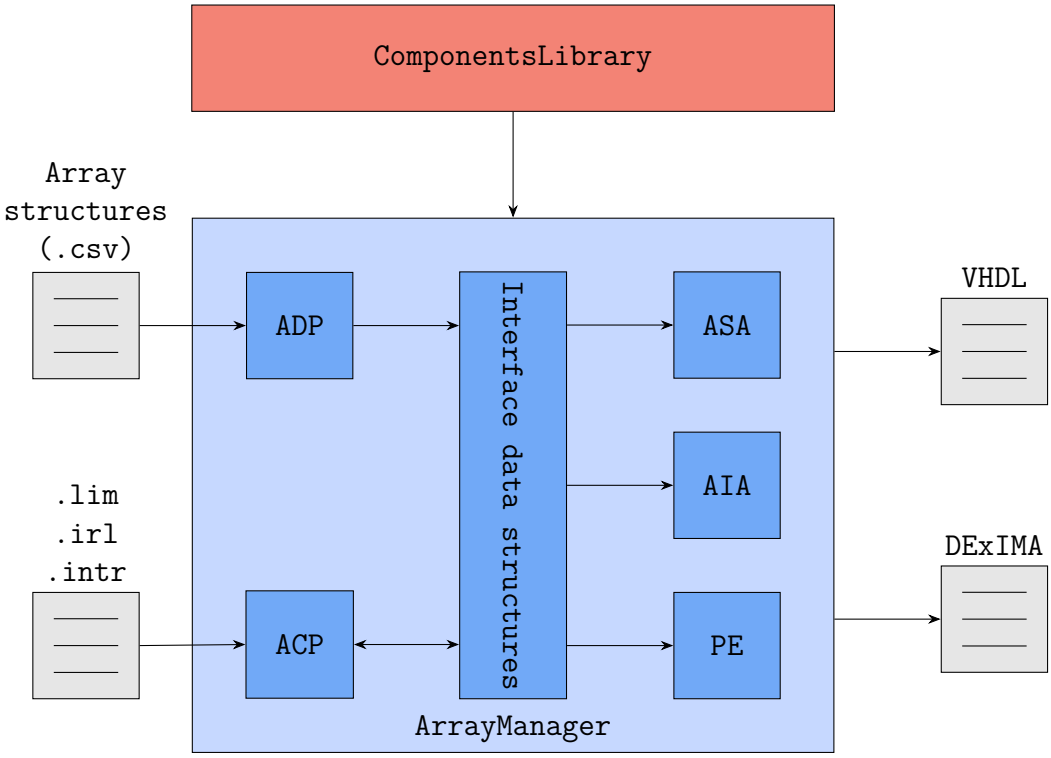


Figure 3.10: Hierarchical structure of `class ArrayManager` and its interaction with the DExIMA design files. `ADP` is an instance of `class ArrayDescriptionParser`, `ACP` is an instance of `class ArrayContentParser`, `ASA` is an instance of `class ArrayStructureAnalyzer`, `AIA` is an instance of `class ArrayInterconnectionsAnalyzer`, while `PE` is an instance of `class PathEnumerator`.

3.3.2 Array description parser

As already stated in the introduction to DExIMA CAD, the types of cells in the memory is specified in a CSV file, containing as many rows as the number of rows in the array and as many columns as the columns in the array; similarly, the required IRL blocks are specified in a further CSV file, containing as many rows as the number of rows in the array and one column only. It follows that the complete structure of the memory can be inferred by parsing these two files, and the array manager demands this kind of operation to an instance of `class ArrayDescriptionParser`, which will be hereinafter referred to as "description parser".

The primary purpose of the description parser is to transfer the content of the two aforementioned CSV files into dedicated data structures, namely a two-dimensional array for the cells pattern and a one-dimensional array for the IRL blocks pattern, which are stored within the object and can communicate with the array manager by means of the interface represented in Figure 3.10. In the following, two simplified examples of CSV files are reported.

```
/* lim_array.csv */
cell1,cell1,cell1,cell1
cell2,cell3,cell3,cell3
cell2,cell3,cell3,cell3
cell2,cell3,cell3,cell3
```

```
/* lim_array_intrarow.csv */
irl_block
irl_block
irl_block
irl_block
```

For the above files, the description parser would produce the following data structures.

```
pattern_array = [
        ['cell1', 'cell1', 'cell1', 'cell1'],
        ['cell2', 'cell3', 'cell3', 'cell3'],
        ['cell2', 'cell3', 'cell3', 'cell3'],
        ['cell2', 'cell3', 'cell3', 'cell3']
]

pattern_IRL = [
        'irl_block',
        'irl_block',
        'irl_block',
```

```
        'irl_block'
]
```

Besides gathering the content of the memory array, the description parser checks the consistency of the CSV files by verifying if the following conditions are met:

- both CSV files must have the same number of rows;
- all rows in the CSV files must have the same number of columns;
- the CSV file containing the IRL blocks must have one column only.

As a matter of fact, the consistency check is not crucial, as the structure of the CSV files is enforced by DExIMA CAD and is supposed to be correct by construction. Nevertheless, this operation is deemed useful, especially for future DExIMA CAD developments, where scripting capabilities may be integrated.

3.3.3 Array content parser

To infer the complete composition of the memory array, the analysis of the previously described CSV files is not actually sufficient. In fact, the array manager must first gain insight on the local structure of the LiM cells and the IRL blocks. To achieve this goal, the array manager triggers the intervention of an instance of class `ArrayContentParser`, which will be hereinafter referred to as "content parser".

The initial task of the content parser is to scan the two arrays produced by the description parser, in order to gather the types of LiM cells and IRL blocks required by the memory. Two one-dimensional lists are generated in this phase: with respect to the CSV files presented in Subsection 3.3.2, the outcome is reported in the following.

```
cell_names          = ['cell1', 'cell2', 'cell3']
block_names         = ['irl_block']
```

After this step, the content parser is aware of what DExIMA design files in the project path are actually useful to represent the local structure of a LiM array. The `.intr` files can thus be parsed to determine which array interconnections are used in each array element, as shown in the following example.

```
cell_interconnections  = {
        'cell1'      : ['Next-1-IRL0-to-Row'],
        'cell2'      : ['Next-1-IRL0-to-Row',
                        'Prev-1-OC-to-Row',
                        'Prev-1-IRL0-to-Row',
```

```
                        'Least-to-Most'
                       ],
        'cell3'     : ['Next-1-IRL0-to-Row',
                        'Prev-1-OC-to-Row',
                        'Prev-1-IRL0-to-Row',
                        'Least-to-Most'
                       ],
}
block_interconnections = {
        'irl_block' : ['Prev-1-OC-to-IRL']
}
```

Lastly, multiple instances of `class DExIMAGraph` parse all required `.lim` and `.irl` files, to build their equivalent DExIMA graph representations. Before yielding control of the generation flow back to the array manager, the content parser checks the local connectivity of all constructed graphs.

3.3.4 Array structure analyzer

As already mentioned in Subsection 3.3.1, a structure analyzer has the primary purpose of managing the structural layer of a LiM array, i.e. its interconnection-free structure. This layer must be handled for both the VHDL and the DExIMA description of the LiM array. In either case, the structure analyzer acts on all the data gathered by the description parser and by the content parser when explicitly invoked by the array manager; specific contributions are generated, which are then passed back to the array manager.

VHDL description The contribution of a structure analyzer to the VHDL description of a LiM array derives from three of its methods, as described in the following.

- With the `entity()` method, the structure analyzer specifies the contribution of the structural layer to the VHDL I/O interface declaration, i.e. the `entity` section of the complete `.vhd` file. The primary elements of this contribution are some general signals, e.g. the clock and the asynchronous reset, the template signals and the actual data/address I/O signals, e.g. bitlines and wordlines. An example is reported in the following.

```
CLK : in  std_logic;
RST : in  std_logic;
BL  : in  std_logic_vector(MEM_COLUMNS-1 downto 0);
WL  : in  std_logic_vector(0 to MEM_ROWS-1);
SO  : in  std_logic_vector(0 to MEM_ROWS-1);
```

```
S1  : in  std_logic_vector(0 to MEM_ROWS-1);
S2  : in  std_logic_vector(0 to MEM_ROWS-1);
SI0 : in  std_logic_vector(0 to MEM_ROWS-1);
SI1 : in  std_logic_vector(0 to MEM_ROWS-1);
SI2 : in  std_logic_vector(0 to MEM_ROWS-1);
MEM : out std_logic_vector(0 to MEM_ROWS*MEM_COLUMNS-1)
```

- With the `architectureDeclarative()` method, the structure analyzer specifies the contribution of the structural layer to the declarative part of the `architecture`. The elements of this contribution are the internal template output signals, e.g. `OC`, `LiM0`, `IRL1` and so forth. An example is reported in the following.

```
signal OC   : Matrix_TypeDef;
signal LiM0 : Matrix_TypeDef;
signal IRL0 : Matrix_TypeDef;
signal IRL1 : Matrix_TypeDef;
```

- With the `architectureExecutive()` method, the structure analyzer specifies the contribution of the structural layer to the executive part of the `architecture`. This contribution consists of multiple `port map` statements, which allocate all array elements and connect them to the external I/O interface or to the previously declared internal signals. Two examples are reported in the following: the first refers to the allocation of a LiM cell, the second represents the allocation of a IRL block.

```
CELL_69_22 : cell_16to79 port map (
        BL                 => BL(22),
        WL                 => WL(69),
        CLK                => CLK,
        RST                => RST,
        S0                 => S0(69),
        S1                 => S1(69),
        NEXT_1_IRL0_TO_ROW => NEXT_1_IRL0_TO_ROW(69)(22),
        PREV_3_OC_TO_ROW   => PREV_3_OC_TO_ROW(69)(22),
        PREV_8_OC_TO_ROW   => PREV_8_OC_TO_ROW(69)(22),
        PREV_14_OC_TO_ROW  => PREV_14_OC_TO_ROW(69)(22),
        PREV_16_OC_TO_ROW  => PREV_16_OC_TO_ROW(69)(22),
        SH0                => SH0(22),
        OC                 => OC(69)(22),
```

```
        LiM0                => LiM0(69)(22)
);
```

```
IRL_79 : irl_block_16to79 port map (
        CLK                 => CLK,
        RST                 => RST,
        BL                  => BL,
        SHO                 => SHO,
        WL                  => WL(79),
        TOP                 => TOP(79),
        SI0                 => SI0(79),
        SI1                 => SI1(79),
        SI2                 => SI2(79),
        SI3                 => SI3(79),
        SI4                 => SI4(79),
        SI5                 => SI5(79),
        SI6                 => SI6(79),
        SI7                 => SI7(79),
        SI8                 => SI8(79),
        PREV_1_LIM0_TO_IRL  => PREV_1_LIM0_TO_IRL(79),
        BTM                 => BTM(79),
        IRL0                => IRL0(79),
        IRL1                => IRL1(79)
);
```

In the above examples, the `port map` statements also involve specific interconnection pins, e.g. `NEXT_1_IRL0_TO_ROW` or `PREV_1_LIM0_TO_IRL`. Nevertheless, the interconnections layer is not yet involved: these statement are only meant to transfer the internal I/O pins of an array element to the internal matrix-like signals.

DExIMA description The contribution of a structure analyzer to the DExIMA description of a LiM array derives from three of its methods, as described in the following.

- With the `sectionLogic()` method, the structure analyzer declares all IRL blocks in the `logic` sub-section of a LiM array `.dex` file.
- With the `sectionCells()` method, the structure analyzer declares all LiM cells in the `cells` sub-section of a LiM array `.dex` file.
- With the `sectionMap()` method, the structure analyzer specifies the local connectivity of each array element. Two examples are reported in the following:

the first refers to the local connectivity of a LiM cell, the second represents the local connectivity of a IRL block.

```
MUX21_11(1,0).O -> Memory(1,0).WR
Memory(1,0).RD  -> FA_12(1,0).B
FA_12(1,0).S    -> MUX21_17(1,0).IN0
MUX21_16(1,0).O -> MUX21_11(1,0).IN0
MUX21_17(1,0).O -> MUX21_11(1,0).IN1
MUX21_11(1,1).O -> Memory(1,1).WR
```

```
for j in range(0,1,15) {
        Reg_13_1.Q[$j] -> Multiplier_14_1.B[$j]
}
for j in range(0,1,15) {
        Multiplier_14_1.O[$j] -> Muxnbit_15_1.IN1[$j]
}
for j in range(0,1,15) {
        Muxnbit_15_1.O[$j] -> Reg_16_1.D[$j]
}
```

In all the above cases, the structure analyzer relies on the underlying DExIMA graph representations, i.e. instances of `class DExIMAGraph`, to determine what components are declared in each array element and their mutual connections, i.e. the local connectivity.

3.3.5 Array interconnections analyzer

As already mentioned in Subsection 3.3.1, an interconnections analyzer handles the interconnections layer of a LiM array. Its purpose its twofold: firstly, it is supposed to analyze the provided description of array interconnections and verify their consistency; secondly, it can generate a description of the required interconnections, either in VHDL or in DExIMA format.

As regards the consistency of array interconnections, the following conditions must be met:

- all cells within a row must integrate precisely the same types of array interconnections;
- the array interconnections required by a row must be self-consistent and not ill-defined with respect to the geometrical characteristics of the memory array.

In practice, the former condition tries to enforce as regular a structure as possible in the memory array, while the latter is primarily meant to exclude two situations:

- vertical interconnections with non-existing sources, e.g. unallocated IRL blocks;
- vertical interconnections with absolute or relative displacements exceeding the boundary of the memory.

An interconnections analyzer is endowed with specific methods to integrate the required array interconnections in the VHDL and in the DExIMA Backend descriptions of the LiM array. As a further proof of this statement, it should be pointed out that the intervention of `DExIMAGraph` is actually required only while generating the DExIMA Backend description.

Three primary outputs can be provided by the analyser:

- a declaration string, containing the declaration of the required interconnection signals, to be embedded in the declarative part of the VHDL `architecture`;
- an assignment string, containing the mutual mappings of all interconnection signals, to be integrated in the executive part of the VHDL `architecture`;
- a mapping string, describing the array interconnections in DExIMA format.

The interconnections analyzer acts on all the data gathered by the description parser and by the content parser when explicitly invoked by the array manager; specific contributions are generated, which are then passed back to the array manager.

VHDL description The contribution of an interconnections analyzer to the VHDL description of a LiM array derives from three of its methods, as described in the following.

- With `generateSignalStringForVHDL()` method, the interconnections analyzer provides the declaration of a matrix-like signal for each interconnection type in the LiM array. An example of such declaration is presented in the following.

```
signal NEXT_1_IRL0_TO_ROW : Matrix_TypeDef;
signal PREV_3_OC_TO_ROW   : Matrix_TypeDef;
signal PREV_8_OC_TO_ROW   : Matrix_TypeDef;
signal PREV_14_OC_TO_ROW  : Matrix_TypeDef;
signal PREV_16_OC_TO_ROW  : Matrix_TypeDef;
signal FROM_RIGHT         : Matrix_TypeDef;
signal TO_LEFT            : Matrix_TypeDef;
```

- With the `generateMappingStringForVHDL()` method, the interconnections analyzer links all interconnection output pins to their corresponding input pins,

by means of a traditional VHDL assignment. For each row in the LiM array, the interconnections analyzer gathers the set of active interconnections and maps them accordingly. In the following, two examples are reported: the first shows how multiple towards-row interconnections are handled, while the second shows the management of a horizontal LSB-to-MSB interconnection.

```
-------------- Towards-Row interconnects @ Row 18 --------------
NEXT_1_IRL0_TO_ROW(18) <= IRL0(18);
PREV_3_OC_TO_ROW(18)   <= OC(15);
PREV_8_OC_TO_ROW(18)   <= OC(10);
PREV_14_OC_TO_ROW(18)  <= OC(4);
PREV_16_OC_TO_ROW(18)  <= OC(2);
```

```
process(TO_LEFT(80)) is
        variable FROM_RIGHT_v : std_logic_vector(31 downto 0);
begin
        for j in 0 to (32-1-1) loop
                FROM_RIGHT_v(j+1) := TO_LEFT(80)(j);
        end loop;
        FROM_RIGHT(80) <= FROM_RIGHT_v;
end process;
```

DExIMA description The `generateMappingStringForDExIMA()` method is the contribution of an interconnections analyzer to the DExIMA description of a LiM array. Since no HDL-like net declaration statement is available in DExIMA Backend, all components must be explicitly mapped to one another by specifying the pins involved in the local connection. In addition, the current DExIMA Backend does not allow multi-bit assignments, thus each bit from a multi-bit bus must be managed individually in the description. Furthermore, as LiM cells are naturally regarded as single-bit components and IRL blocks as multi-bit components, the syntax with which they described is slightly different, increasing the complexity of the management algorithm.

The cells pattern and the blocks pattern gathered by the description parser are scanned by the interconnections analyzer one row at a time, to manage array interconnections leading to memory rows and to IRL blocks, respectively. In this scan operation, the current element is the destination of any possible array interconnection and is readily associated to the set of required interconnections and its equivalent graph representation, i.e. an instance of `class DExIMAGraph`. Depending on the type of array interconnection, the source component is fetched and its graph is made available to the elaboration. When these data structures are ready, the

interconnections analyser implements an algorithm roughly based on the following steps:

1. identify the interconnection input pin;
2. identify the interconnection output pin;
3. in the destination graph, seek the components whose inputs are connected to the interconnection input pin (multiple components can be found), effectively preparing the Right-Hand Side (RHS) of the DExIMA mapping operation;
4. in the source graph, search the component whose output is connected to the interconnection output pin, effectively preparing the Left-Hand Side (LHS) of the DEXIMA mapping operation;
5. for all the components found in the destination graph, build a string by concatenating the LHS, the mapping operator `->` and the RHS.

An example of a `Next-1-IRL0-to-Row` vertical interconnection is reported in the following.

```
#        ---- Next-1-IRL0-to-Row @ ROW 0 ----
        Reg_16_0.Q[0] -> MUX21_11(0,0).IN1
        Reg_16_0.Q[1] -> MUX21_11(0,1).IN1
        Reg_16_0.Q[2] -> MUX21_11(0,2).IN1
        Reg_16_0.Q[3] -> MUX21_11(0,3).IN1
        Reg_16_0.Q[4] -> MUX21_11(0,4).IN1
        Reg_16_0.Q[5] -> MUX21_11(0,5).IN1
        Reg_16_0.Q[6] -> MUX21_11(0,6).IN1
        Reg_16_0.Q[7] -> MUX21_11(0,7).IN1
        Reg_16_0.Q[8] -> MUX21_11(0,8).IN1
        Reg_16_0.Q[9] -> MUX21_11(0,9).IN1
        Reg_16_0.Q[10] -> MUX21_11(0,10).IN1
        Reg_16_0.Q[11] -> MUX21_11(0,11).IN1
        Reg_16_0.Q[12] -> MUX21_11(0,12).IN1
        Reg_16_0.Q[13] -> MUX21_11(0,13).IN1
        Reg_16_0.Q[14] -> MUX21_11(0,14).IN1
        Reg_16_0.Q[15] -> MUX21_11(0,15).IN1
```

An example of a horizontal LSB-to-MSB interconnection is reported in the following.

```
#   ---- Least-to-Most @ ROW 1 ----
        FA_12(1,0).CO -> FA_12(1,1).C
        FA_12(1,1).CO -> FA_12(1,2).C
        FA_12(1,2).CO -> FA_12(1,3).C
```

```
FA_12(1,3).CO -> FA_12(1,4).C
FA_12(1,4).CO -> FA_12(1,5).C
FA_12(1,5).CO -> FA_12(1,6).C
FA_12(1,6).CO -> FA_12(1,7).C
FA_12(1,7).CO -> FA_12(1,8).C
FA_12(1,8).CO -> FA_12(1,9).C
FA_12(1,9).CO -> FA_12(1,10).C
FA_12(1,10).CO -> FA_12(1,11).C
FA_12(1,11).CO -> FA_12(1,12).C
FA_12(1,12).CO -> FA_12(1,13).C
FA_12(1,13).CO -> FA_12(1,14).C
FA_12(1,14).CO -> FA_12(1,15).C
```

3.3.6 Path enumeration in the DExIMA description

Section 3.3 has provided a thorough description of the new DExIMA CAD array interconnections module, but details on the behaviour of one of its components, i.e. the path enumerator, have been deliberately omitted.

Since DExIMA Backend may only compute the timing of a specified path, DExIMA CAD should provide a set of timing paths which must include the critical path at all costs, so that DExIMA Backend may quantify its length. To fulfill this task, the path enumerators builds a global graph representation of the LiM array, taking into account not only its structural layer, but also its interconnections layer.

The content parser creates a limited amount of instances of `class DExIMAGraph` (one for each unique array element in the LiM array). The most straightforward solution would be to replicate these instances for the entire LiM array, creating the structural layer, and then superimposing the interconnections layer. Unfortunately, using instances of `class DExIMAGraph` to model every LiM cell and every IRL block in the array is too burdensome from a computational perspective, even for limited-size LiM array.

A possible solution is offered by simplifying the structure of each graph node, i.e. each library component. In fact, a complete knowledge of the bit-widths and the parameters of a component are not of interest in this context, so a node may simply be a string, and no longer an instance of `class Component` or of `class PinIO`. Moreover, for the purposes of identifying timing paths in the LiM array, only the interconnection output and input pins are of interest, so all other internal pins may be removed from the global graph representation. An instance of `class DExIMAGraph` offers a `shallowCopy()` method, which returns a simplified graph, in which each node is a string that represents either an actual component or an interconnection output/input pin.

The path enumerator analyzes the LiM cells and the IRL blocks patterns, requesting a shallow graph copy for each array element, which is accumulated in a global graph representation. When all array elements have been analyzed, this

global graph representation provides a model, albeit simplified, of the structural layer of a LiM array.

The path enumerator then updates the global graph representation by integrating all necessary array interconnections. For each interconnection, the procedure is roughly represented by the following steps:

- all components connected to the interconnection output pin, i.e. the source components, are identified;
- all components connected to the interconnection input pin, i.e. the destination components, are identified;
- edges are appended to the global graph representation between source and destination components;
- interconnection output and input pins are eliminated from the global graph.

When the interconnections layer has been handled, the global graph may be used to identify all timing paths, which start from and end in a synchronous element: to fulfill this task, a procedure based on the Depth-First Search (DFS) is applied.

The timing paths found by the path enumerator are then specified in the `.dex` description, enabling the critical path identification by DExIMA Backend.

Chapter 4

Algorithm description in DExIMA CAD

The purpose of Chapter 4 is to describe a second extension to DExIMA CAD, which is aimed at enabling a flexible description of the algorithm to be simulated and implemented by the LiM architecture.

Section 4.1 introduces the existing algorithm description features and discusses their main limitations. Section 4.2 addresses the new functionalities in the algorithm description module, the structure of which is described in Section 4.3.

4.1 Existing algorithm description features

As already mentioned in Section 1.3, DExIMA CAD integrates an algorithm description functionality, based on the association of a scalar control and of an activation pattern, i.e. the set of active array elements, which is ultimately used to program the content of the algorithm ROM. Nevertheless, the existing algorithm description module presents some limitations, which will be briefly addressed in the remainder of this section.

A first limitation comes from the fact a scalar control word only defines the values of the LiM and IRL selectors and, as a consequence, it is not possible to activate a memory row during the simulation. As a matter of fact, the UVM testbench is directly asked to drive all wordline signals, activating them to initialize the content of the memory to a set of random values and then deactivating them for the entire simulation. As a consequence, within the environment of DExIMA CAD, it would not be possible to simulate such algorithms as those implemented by the Hybrid-SIMD architecture presented in [16], which require to overwrite the content of the memory rows whilst the algorithm is executed.

A further limitation may be inferred by taking into account Figure 1.6. In fact, the existing GUI functionalities for the algorithm description are denoted by a poor

readability, which is significantly worse when the LiM array has lots of LiM and IRL control signals. For this reason, the designer may find it difficult to specify the intended behaviour of an active array element.

Furthermore, few activation pattern types are supported by the algorithm description functionality, as reported in Figure 4.1. The designer can extend the scalar control word either to all array elements or to a set of list-specified elements; on top of that, a loop-like activation pattern enables the incremental execution of an instruction.

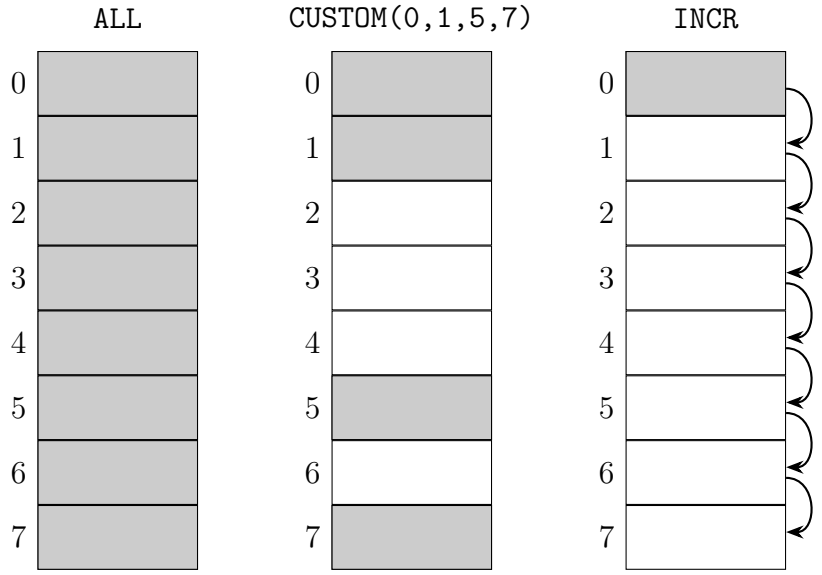


Figure 4.1: Examples of existing activation patterns in DExIMA CAD. Each row is a general representation of an array element, i.e. a memory row or a IRL block, and a shaded row is an active array element. The example of `CUSTOM` activation pattern assumes that the list of active elements consists of the values 0, 1, 5 and 7. As for the example of `INCREMENTAL` activation pattern, the array elements will be activated in a sequential fashion.

Many different algorithms consist of a start-up phase and of a kernel phase. This consideration may especially apply in SIMD processing, where the start-up phase is used to initialize the storage facilities, possibly ensuring a certain degree of data replication to allow a truly-parallel processing, while the kernel phase is responsible for producing the actual results. In most applications, the kernel phase primarily consists of loops, iterating several algorithm steps a given number of times. Nevertheless, the existing algorithm description module does not allow to replicate selected instructions groups, reducing the capabilities of the tool.

4.2 New algorithm description features

The revisited DExIMA CAD should strive to remove the limitations of the existing algorithm description module, which have been detailed in Section 4.1.

The rationale behind the new algorithm description functionality is to formally separate the definition of the scalar control from its vectorisation and extension to the set of LiM array elements. To comply to this principle, the algorithm description module offers two separate contexts, namely the "instructions" and the "algorithm" contexts.

Instructions context In the instructions context, the designer can define nano-instructions, which represent the intended behaviour of an active array element, by associating a scalar control word to a mnemonic, e.g. an opcode, which is expected to increase the readability of the produced description. For each nano-instruction, in addition to the values of the LiM and of the IRL control signals, the designer can specify if the memory row should be activated for a write operation.

The instructions context produces an output CSV file containing the nano-instructions defined by the designer. Besides an header line (which is extremely important, as it provides some pieces of information related to the employed array template), the file contains as many rows as the number of nano-instructions, while the number of columns is determined by the number of LiM and `IRL` control signals. An example of such file is reported in Figure 4.2.

```
OPCODE,EN,S0,S1,S2,SI0,SI1,SI2
NOP,0,0,0,0,0,0,0
LOAD,1,0,0,0,0,0,0
SHH,0,0,0,0,1,0,0
MULT,0,0,0,0,0,1,1
ROT,1,1,0,0,0,0,1
SUM,1,0,1,0,0,0,0
SHV,1,1,1,0,0,0,0
```

Figure 4.2: Example of CSV file produced by the instructions context, assuming to have three LiM control signals and tree IRL control signals.

While working in the instructions context, the designer should bear in mind that a nano-instruction is not necessarily expected to fully describe the behaviour of an active array element. As a matter of fact, some architectures, including the Hybrid-SIMD array presented in [16], are more easily managed if the operation to be executed, the source of its input operands and the destination of the computation are separated and not interdependent. The designer can thus define simpler nano-instructions, which partially describe the behaviour of an active array element, and

combine them together in the algorithm context. An example of such combination mechanism is reported in Figure 4.3.

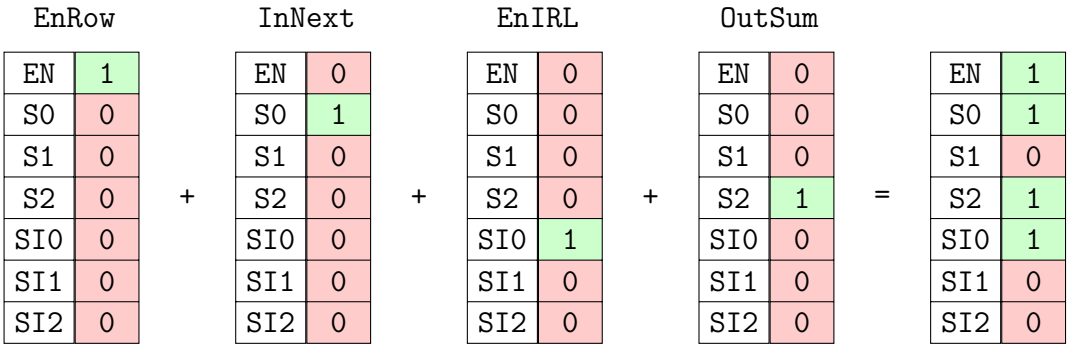


Figure 4.3: Example of nano-instructions composition in the algorithm description module: the "simpler" nano-instructions `EnRow`, `InNext`, `EnIRL` and `OutSum` are combined together in the resulting scalar control.

Algorithm context In the algorithm context, the designer can define the computational steps required by the target algorithm. With no difference with respect to the previous structure of the algorithm description facilities, each computational step consists of a scalar control and of an activation pattern.

As previously mentioned, the scalar control can be derived from one single nano-instruction, which completely describes the intended behaviour of the active array elements, or it can result from a combination of multiple simpler nano-instructions.

The set of supported activation patterns has been extended to include some new elements, ultimately leading to the comprehensive list reported in Table 4.1; examples of the new activation patterns are depicted in Figure 4.4. In addition, to increase the flexibility of the description, a composite activation pattern can be created by attaching a `CUSTOM` activation pattern to a `SINGLE`, a `RANGE`, a `INCREMENTAL` or a `DECREMENTAL` activation pattern, as reported in Figure 4.5.

It is important to point out that the above definition of single and composite activation patterns introduces some redundancy in the description capabilities, since the same results can be obtained in multiple equivalent ways: in such a early phase of the development process, this redundancy is regarded as beneficial and not detrimental to the design of DExIMA CAD.

Besides the new scalar control definition and activation patterns, the algorithm context offers the designer the possibility of grouping together multiple algorithm steps, so to iterate their execution a given number of times.

The algorithm context produces an output CSV file containing the algorithm steps defined by the designer. Besides an header line, the file contains as many rows as the number of algorithm steps; as for the columns, the first reports the

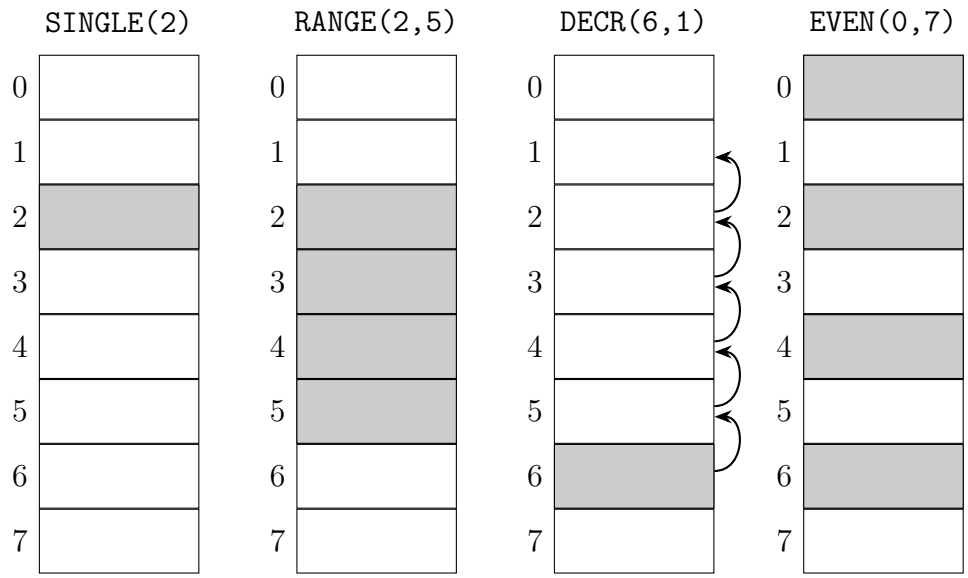


Figure 4.4: Examples of new activation patterns in DExIMA CAD. Each row is a general representation of an array element, i.e. a memory row or a IRL block, and a shaded row is an active array element. The example of `SINGLE` activation pattern assumes an index of 2, while that of `RANGE` activation pattern a start index of 2 and a stop index of 5. As for the example of `DECREMENTAL` activation pattern, the array elements will be activated in a sequential fashion, assuming a start index of 7 and a stop index of 0.

Pattern	Parameters	Behaviour
`NONE`	-	Do not activate any element
`SINGLE`	`P1`	Activate element `P1`
`RANGE`	`P1`, `P2`	Activate elements in range `P1:P2`
`INCR`	`P1`, `P2`	Activate elements from `P1` up to `P2`, sequentially
`DECR`	`P1`, `P2`	Activate elements from `P1` down to `P2`, sequentially
`ALL`	-	Activate all elements
`CUSTOM`	`L`	Activate elements specified in list `L`
`EVEN`	`P1`, `P2`	Activate even-index elements in range `P1:P2`
`ODD`	`P1`, `P2`	Activate odd-index elements in range `P1:P2`

Table 4.1: Comprehensive description of the activation patterns in DExIMA CAD. The parameters `P1` and `P2` and the elements belonging to list `L` are indexes, therefore they should comply to the geometrical characteristics of the memory array.

opcode of the desired nano-instruction, the second indicates the associated activation pattern and the third defines the possible starting points of algorithm groups.

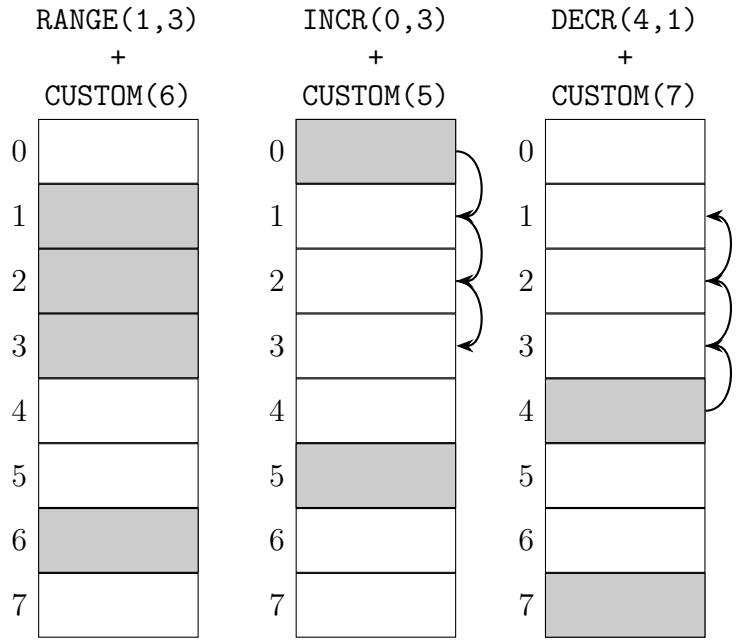


Figure 4.5: Examples of composite activation patterns in DExIMA CAD.

The algorithm context offers the possibility of grouping together multiple algorithm steps, so to iterate their execution a given number of times. Figure 4.6 reports an example of algorithm CSV which contains an algorithm group.

Expected improvements The new features of the algorithm description module are expected to remove the limitations presented in Section 4.1.

The introduction of nano-instructions with their associated mnemonics, alongside with the possibility of combining them, increases the readability of the produced description, making it easier for the designer to specify the intended simulation-time behaviour of the LiM array in a given algorithm step.

Since nano-instructions control the wordline signals, the content of the rows in a LiM array can change during the simulation and, more specifically, whilst executing the desired algorithm. As a consequence, such algorithms as those implemented by the Hybrid-SIMD architecture [16] may be simulated in DExIMA CAD.

The new activation pattern types significantly extend the number of ways in which a scalar control can be extended to the complete LiM array, increasing the flexibility of the algorithm description module, while the introduction of algorithm groups offers a very simple feature to iterate specific algorithm steps.

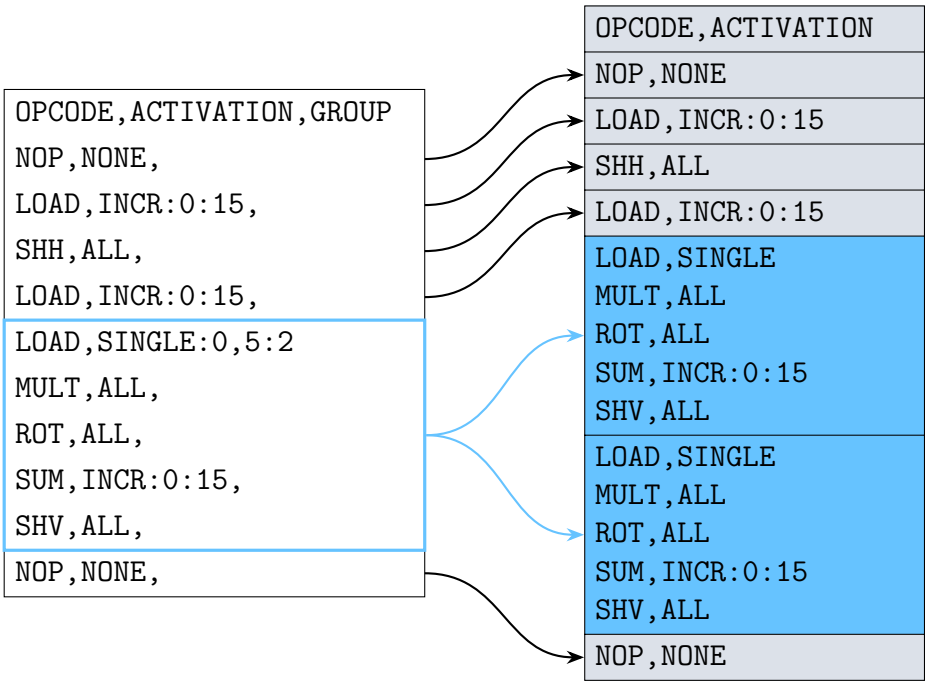


Figure 4.6: On the left-hand side, example of CSV file produced by the algorithm context, assuming to have the same pool of nano-instructions of Figure 4.2 and an algorithm group consisting of five nano-instructions and two iterations. On the right-hand side, expansion of the algorithm group.

4.3 Algorithm description module

Taking into account the guidelines presented in Section 1.4 for the injection of new source code in DExIMA CAD, the scheme depicted in Figure 4.7 has been used to lay out the structure of the algorithm description module.

The algorithm description module consists of a set of GUI components and of computational elements, which may only communicate by means of a file-based interface. The GUI components aid the generation of the two CSV files mentioned in Section 4.2, i.e. the instructions CSV and the algorithm CSV, which are then analyzed by the computational section of module, i.e. an instance of `class ControlGenerator`, which will be hereinafter referred to as ”control generator“.

To ease the work of the control generator and reduce the complexity of the algorithm it implements, additional support classes have been defined, with the aim of modeling all the elements described in Section 4.2: these classes will be briefly discussed in Subsection 4.3.1. It is however important to notice that the hierarchical structure offered by the support classes increases the maintainability of the source code and can foster future expansions and improvements.

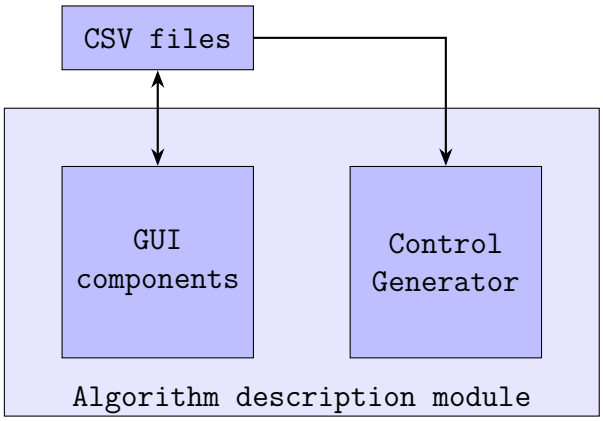


Figure 4.7: Top-level view on the algorithm description module

4.3.1 Support classes

Figure 4.8 reports an abstract representation of the support classes mentioned in Section 4.2.

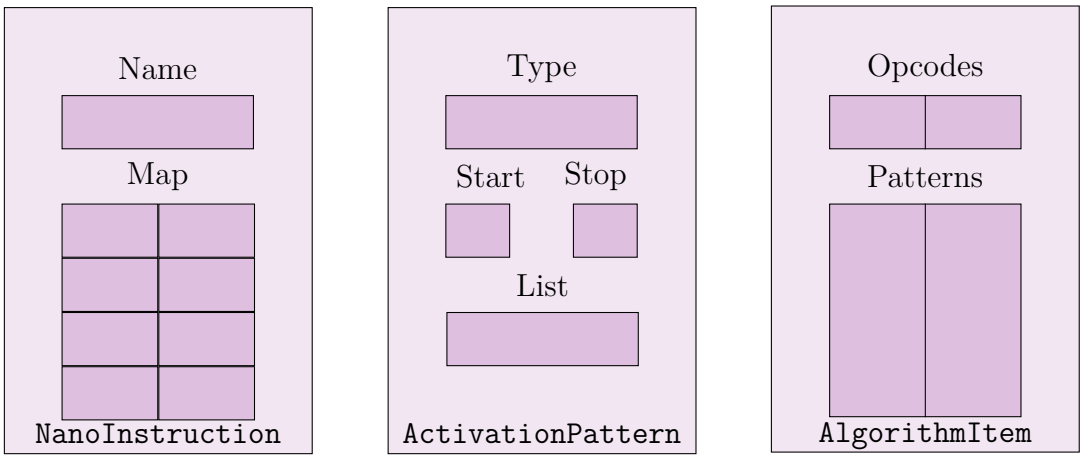


Figure 4.8: Abstract representation of the modelling classes employed by the algorithm description module.

An instance of `class NanoInstruction` is used to model a nano-instruction, i.e. a row in the CSV file resulting from the instructions context. It consists of a string to represent the opcode and of a map, which defines the values of the worldline and of the LiM and IRL control signals. When queried, a `NanoInstruction` object can return the map values in a structured form that may be directly be used by the top-level control generator. It is noteworthy to mention that this class supports the `+` operator, which has been properly overloaded, to ease the combination of multiple nano-instructions in the algorithm context, as presented in Section 4.2.

An instance of `class ActivationPattern` represents an activation pattern. Every instance of this class contains a string, which identifies the type of activation pattern, and additional data structures to model the optional parameters that define the pattern, as reported in Table 4.1. It is important to stress that, at this level, there is no distinction between a simple and a composite activation pattern. In fact, the latter is nothing but a "concatenation" of two simple activation patterns, the first one belonging to the set `{SINGLE, RANGE, INCR, DECR}`, the second being a `CUSTOM` pattern.

An instance of `class AlgorithmItem` can be used to model an algorithm step, i.e. a row in the CSV file resulting from the algorithm context. The representation of the scalar control is demanded to a list of strings which identify the nano-instructions defined in the instructions context, while one or two `ActivationPattern` objects are used to indicate the simple or the composite activation pattern.

In addition to the elements of Figure 4.8, the description of algorithm groups is enabled by a further class, `AlgorithmItem`, which is primarily meant to encompass those algorithm steps that are expected to be iterated a given number of times. As a matter of fact, isolated instructions, i.e. algorithm steps which do not belong to any algorithm group, are regarded as a special case of algorithm group, thus a `AlgorithmGroup` instance is created for this type of algorithm step as well.

4.3.2 Control generator

To parse the instructions and the algorithm contexts in a systematic way, the control generator, the internal structure of which is reported in Figure 4.9, coordinates the actions of two sub-modules, namely a `ParserInstructions` object and a `ParserAlgorithm` object, which will be hereinafter referred to as "instructions parser" and "algorithm parser", respectively.

The role of the instructions parser is to analyse the content of the CSV file containing the nano-instructions defined by the designer. The very first row in the file is used to retrieve the name of the required control signals, which include the actual wordline and the LiM and IRL selectors; all other rows in the file are instead used to properly allocate instances of `NanoInstruction` class, which store the necessary nano-instructions and allow possible combination of simple nano-instructions in the algorithm context.

The purpose of the algorithm parser is to scan the content of the CSV file containing the algorithm context, in order to gather not only the actual steps, but also the required iteration groups. As it reads the file line by line, the parser retrieves the opcode and the activation pattern, which could be either of simple or of composite type, allocates one or two `ActivationPattern` objects and builds instances of `AlgorithmItem` class; any algorithm group declaration is temporarily pushed into a local queue. When all lines in the file have been processed, the parser will have created a one-dimensional array of `AlgorithmItem` objects, without an

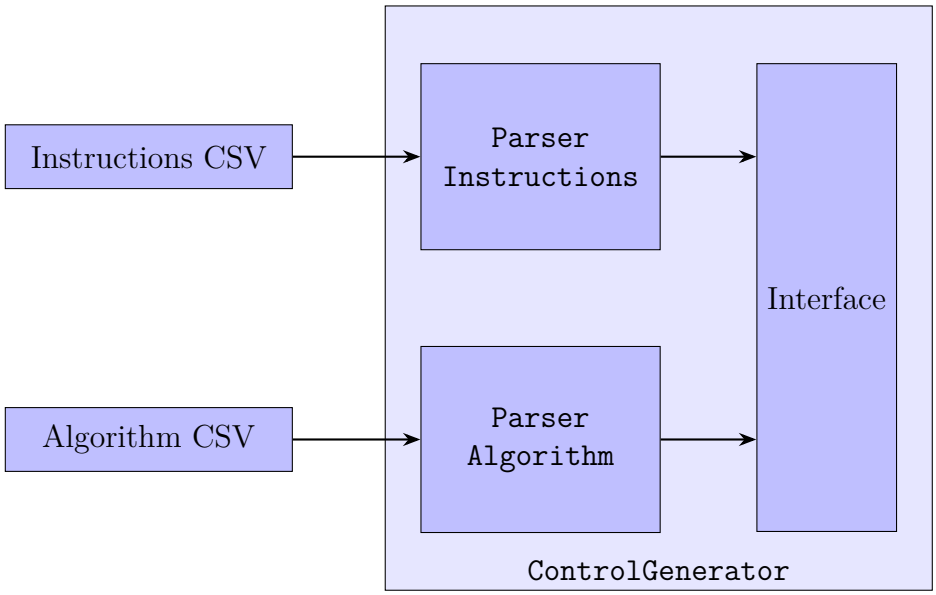


Figure 4.9: Internal structure of class `ControlGenerator` and its interaction with the CSV-based interface.

explicit reference to the possible algorithm groups. Therefore, the parser still needs to iterate over the elements of the queue, so to organise the gathered algorithm items in proper instances of `AlgorithmGroup` class.

When the instructions and the algorithm parsers have completed their tasks, the control generator can create a testbench-aimed description of the required algorithm. As a matter of fact, the object is endowed with a method that can program the content of the micro-ROM which supports the simulation of the algorithm in the testbench environment.

Chapter 5

System-level exploration in DExIMA CAD

The purpose of Chapter 5 is to describe a set of changes in the way DExIMA CAD manages the uppermost architectural level of a LiM system, which are aimed at largely augmenting the system-level exploration capabilities of the tool itself. With these new functionalities, DExIMA CAD is expected to be able to tackle increasingly complex LiM systems, paving the way for more refined and thorough analysis and synthesis tasks.

Chapter 3 and Chapter 4 presented a set of tools whose ultimate scope is the LiM array itself. In fact, array interconnections increase the set of LiM array structures which may be attained by DExIMA CAD: the combined use of horizontal, vertical and additional interconnections allows the implementation of finer data movement patterns within the array itself, which add up to an enhancement of its functionalities. On the other hand, the algorithm description facilities allow a user-friendly and rather straightforward definition of the simulation-time behaviour of a LiM array, by means of nano-instructions and activation patterns.

If the architectural exploration capabilities of DExIMA CAD are to be augmented, a new set of tools must be introduced and, evidently, its scope should not be limited to a LiM array, but it should encompass a larger architectural level. DExIMA CAD should thus be able to treat all designed LiM arrays as basic building blocks, which may be arbitrarily allocated and interconnected, so that multiple system-level solutions may be explored by the designer, should

As for the structure of Chapter 5, Section 5.1 outlines the changes to the structure of the uppermost architectural level of LiM system, as managed by the revisited version of DExIMA CAD. Since both DExIMA Backend and DExIMA CAD present some minor limitations, the new elements in the management of the uppermost architectural level lead to specific issues, which are detailed in Section 5.2, where suggestions on how to solve these problems are discussed. All the required

modifications to DExIMA CAD are then detailed in Section 5.3.

5.1 Changes to the uppermost architectural level

As regards the uppermost architectural level of a designed LiM system, the past version of DExIMA CAD presented some major limitations, related both the actual structure of the system and to its simulation.

The structure of the uppermost architectural level in the past version of DExIMA CAD is depicted in Figure 5.1.

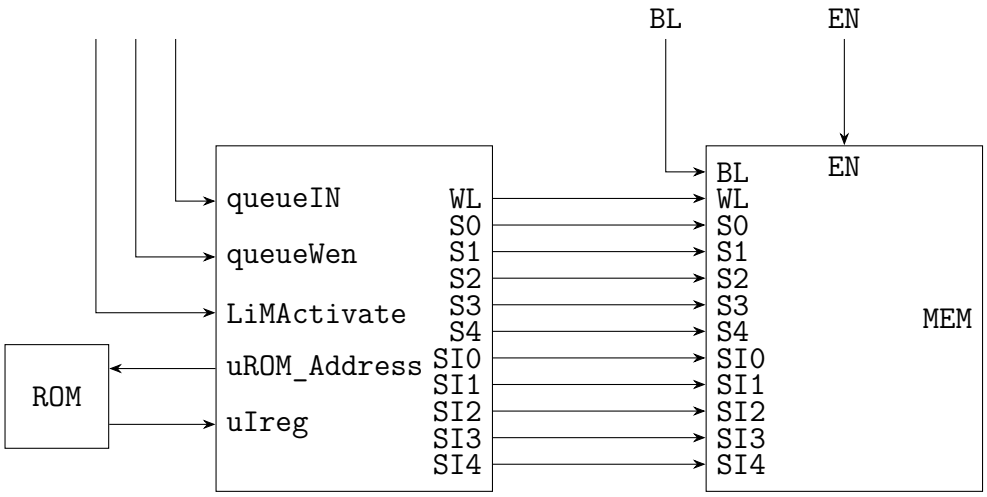


Figure 5.1: Structure of the uppermost architectural level of a LiM system in the past version of DExIMA CAD. Although not explicitly indicated, the LiM array and the memory interface are driven by the same clock and reset signals.

The past version of DExIMA CAD only allowed two types of components in the uppermost architectural level, namely a LiM array and its micro-programmed control unit with a dedicated instruction queue: the latter will be hereinafter referred to as "memory interface". Besides a set of internal connections, which hardwired the memory interface and the LiM array, a standard set of input pins provided the required I/O interface, which was controlled uniquely by the UVM testbench.

From Figure 5.1, it is straightforward to notice that the array wordline signal was controlled directly by the UVM testbench. In fact, whilst simulating the design, this signal was driven by the UVM testbench to initialise the content of the memory rows to random and unspecified values, before the micro-programmed control unit could start fetching the algorithm from its associated ROM; when this random initialisation phase was over, the wordline signal was deactivated by the UVM testbench for the entire remainder of the simulation.

Evidently, the above approach provided as set of random data for the simulation

of the LiM array. Nevertheless, a major shortcoming is readily found. In fact, as the worldline is deactivated, it is impossible to deliberately change the content of any memory row during the simulation, and this makes it impossible to simulate such architectures as the Hybrid-SIMD array presented in [16].

Internal and external control To overcome this limitation, the memory interface should be able to generate an enable signal, which may allow a write operation to one or more memory rows during the algorithm itself. If such signal is directly hardwired to the LiM array, the UVM testbench (or rather any other master communicating with the LiM system) would not be able to load random or specific values in the memory rows of the LiM array. For this reason, a switching mechanism must be devised so that both the UVM testbench and the memory interface may actually drive the wordline signal, in the initialisation and in the algorithm execution phases, respectively. This leads to two control sources: the internal control, deriving from the memory interface, and the external control, deriving from the UVM testbench.

From an architectural perspective, the most straightforward solution is to allocate a two-way multiplexer in the uppermost level of the complete LiM system. The bit-width of this component is equal to the number of rows in the LiM array, while its I/O pins are connected as reported in Figure 5.2.

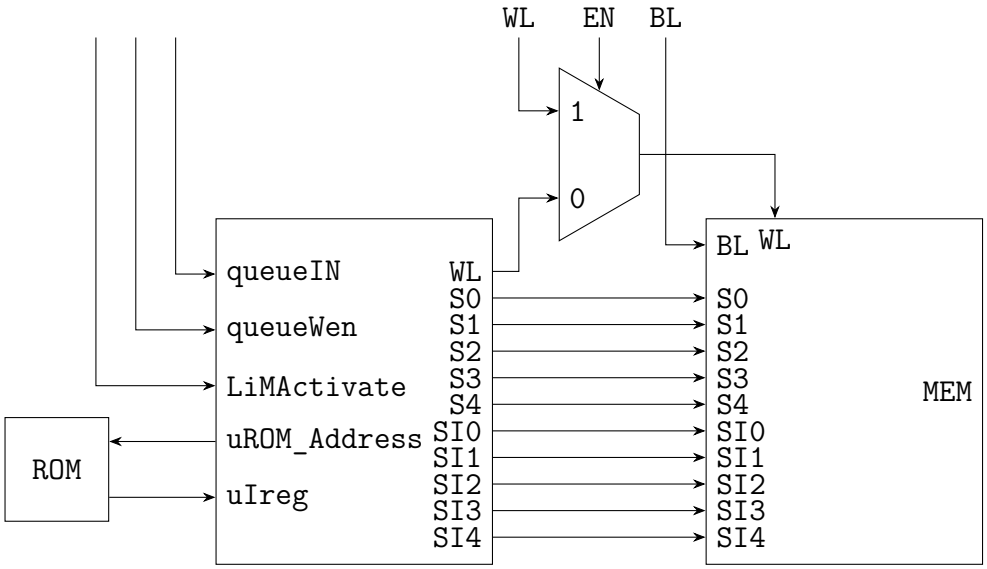


Figure 5.2: Introduction of the wordline multiplexer in the uppermost level of a DExIMA CAD LiM system.

Multiple LiM arrays In the structure of Figure 5.2, only one LiM array may be effectively instantiated in the uppermost architectural level of the complete LiM

system. Despite being a clear limitation, this assumption simplified the structure of the past version of DExIMA CADand it was associated to a perfectly feasible design flow for most case studies.

Nevertheless, as the capabilities of DExIMA CAD are ultimately expected to increase, it would seem reasonable to remove the above limitation, making it possible for the designer to allocate multiple LiM arrays, possibly of different types, in the uppermost architectural level of the complete LiM system. For instance, Chapter 9 explores the implementation of the AES-128 algorithm in different architectural solutions, in which the actual LiM processing may occur either in the same array or in separate arrays. Furthermore, multiple replicas of the same array type may be allocated in the top-level, for instance to introduce a larger degree of parallelism in the processing. To answer these specific needs, the uppermost level in DExIMA CAD requires a proper support for multiple array types and multiple array instances.

To simplify the management of the uppermost architectural level in DExIMA CAD, it has been decided to introduce the following simplifications:

- the uppermost architectural level may support an arbitrary number of array types;
- for each array type, multiple array instances may be allocated;
- for each array type, each unique instance must have the same behaviour as all the others.

Given the above assumptions, it immediately follows that all instances of the same array type must share both the internal and the external controls: to achieve this, they will be connected to the same memory interface and to the same wordline multiplexer, respectively. As a consequence, only one memory interface, with its associated algorithm ROM, is allowed for each array type.

Library components in the top-level It has been mentioned previously that the past version of DExIMA CAD only allowed one LiM array and its memory interface in the uppermost architectural level. Nevertheless, the introduction of a wordline multiplexer is a clear violation of this rule, as a further component must indeed be instantiated in the top-level. As a consequence, this constraint must be necessarily removed, making it possible for the designer to allocate a set of library components in the uppermost architectural level of the complete LiM system.

This change largely increases the architectural exploration capabilities of DExIMA CAD. In fact, if both LiM arrays and other building blocks (including logic gates, storage elements, arithmetic circuits and more complex sub-systems) may be allocated in the uppermost architectural level, then the design may explore a larger set of architectural solutions, in which the required computational tasks are

demanded to different parts of the system. For instance, all processing may occur in the same LiM array or in separate LiM array of different kinds; alternatively, part of the processing may involve a LiM, while the remained may be demanded to a set of near-memory components.

The structure of the uppermost architectural level in the revisited DExIMA CAD is summarised in Figure 5.3.

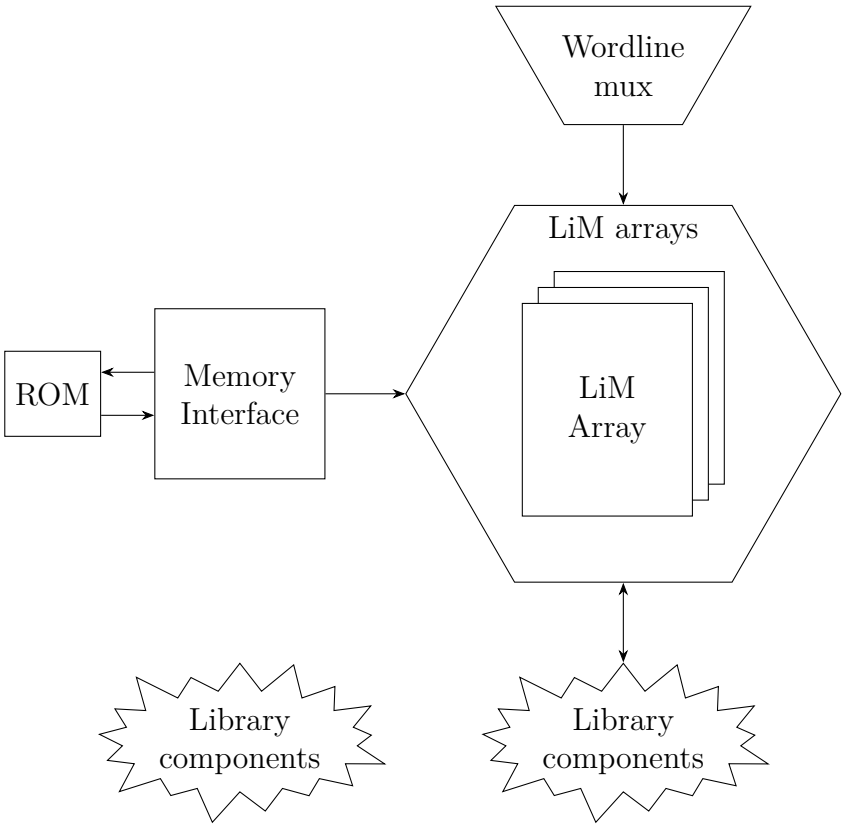


Figure 5.3: Structure of the uppermost architectural level of a LiM system in the revisited version of DExIMA CAD.

5.2 Issues related to the uppermost architectural level

With the changes detailed in Section 5.1, the structure of the uppermost architectural level may change greatly depending on the design. DExIMA CAD must integrate a new set of analysis and synthesis tools, which should be able to tackle the complexity deriving from the large variability of the complete LiM system. More specifically, a set of issues arises from the present state of both the front-end

and the back-end. Thus, if the changes to the uppermost architectural level are to be integrated, DExIMA CAD must be able to overcome these issues, which are detailed in the following.

Front-end issues In Section 5.1, it has been mentioned that multiple LiM array types may be allocated in the uppermost level of the complete LiM system, but a set of specific rules must be enforced. To fulfill this task, DExIMA CAD should thus be able to identify all crucial elements in the design, including LiM arrays, memory interfaces, algorithm ROMs and wordline multiplexers. Moreover, as the complete LiM system may contain different LiM array types, DExIMA CAD should be properly interacting with the algorithm description facilities, so that the behaviour of all memory types may be easily defined by the designer.

Besides the multi-array scenario, Section 5.1 introduced the possibility of allocating library components in the uppermost architectural level of the LiM system. Evidently, some components, e.g. registers, flip-flops, multiplexers, add/subtract units or bit-wise shift registers, require specific control signals, which are not necessarily derived from any memory interface in the design and which must be driven, for the purposes of the simulation, by the UVM testbench. DExIMA CAD should thus recognize these particular pins, alongside with the external control of all LiM arrays, so that a proper simulation-time behaviour may be defined by the designer. In this context, let an active pins be a non-floating input pin belonging to the global I/O interface of the complete LiM system.

Back-end issues With the new elements presented in Section 5.1, DExIMA CAD should compensate for some limitations of the current version of DExIMA Backend. More specifically, these limitations derive from the impossibility of describing any local connection involving a LiM array. In fact, in the present version of DExIMA Backend, the internal model of a LiM array does not offer any I/O interface signal, implying that an explicit connection between a near-memory component and a memory array may not be provided in the intended section of the DExIMA Backend `.dex` input file, i.e. the `map` section. Despite this, it is important to point out that the syntax would allow such connections, as shown in the following.

```
begin constants
        BUILT_IN CLOCK 10.0
        BUILT_IN SF 0
        BUILT_IN AR 3.092783
        BUILT_IN VDD 1.1
        BUILT_IN BA 0
        BUILT_IN PROB 0.5
end constants
begin init
        LIM state_key_array(5,8)
```

```
        ShiftRegister ShiftRegister_5(8)
end init

...

begin map
        ShiftRegister_5.POUT[0] -> state_key_array.HBUS[0]
        ShiftRegister_5.POUT[1] -> state_key_array.HBUS[1]
        ShiftRegister_5.POUT[2] -> state_key_array.HBUS[2]
        ShiftRegister_5.POUT[3] -> state_key_array.HBUS[3]
        ShiftRegister_5.POUT[4] -> state_key_array.HBUS[4]
        ShiftRegister_5.POUT[5] -> state_key_array.HBUS[5]
        ShiftRegister_5.POUT[6] -> state_key_array.HBUS[6]
        ShiftRegister_5.POUT[7] -> state_key_array.HBUS[7]
end map
```

In the previous example, the `init` section declares a LiM array and a near-memory bit-wise shift register; the parallel output of the shift register is later connected to the `HBUS` pin of the memory array in the `map` section.

To compensate for the previously described issue, DExIMA CAD must be able to scan the uppermost architectural level of the complete LiM system and identify the connections involving the actual LiM arrays. In this way, all near-memory components interacting with a specific array instance may be declared in the `logic` sub-section of the array itself, and their connections to the LiM cells and the IRL blocks may be embedded in the `map` sub-section. An example of the intended outcome of such procedure is shown in the following.

```
begin state_key_array
        begin logic
                ...
                ShiftRegister ShiftRegister_5(8)
        end logic
        begin cells
                ...
        end cells
        begin map
                ...
                ShiftRegister_5.POUT[0] -> ...
                ShiftRegister_5.POUT[1] -> ...
                ShiftRegister_5.POUT[2] -> ...
                ShiftRegister_5.POUT[3] -> ...
                ShiftRegister_5.POUT[4] -> ...
                ShiftRegister_5.POUT[5] -> ...
                ShiftRegister_5.POUT[6] -> ...
                ShiftRegister_5.POUT[7] -> ...
```

```
        end map
end state_key_array
```

The extent of the previously described limitation is limited to the presence of any connection involving a LiM array and a library component. Tackling this issue is essential in the revisited version of DExIMA CAD, as the new changes presented in Section 5.1 demand the allocation of a wordline multiplexer, i.e. a multi-bit library component, for each LiM array type in the uppermost architectural level.

A further limitation arises when the structure presented in Figure 5.3 should be handled by DExIMA Backend. The current syntax of DExIMA Backend would allow the declaration of multiple LiM arrays in the `init` section of the `.dex` file, while their description would be demanded to specific sections, one for each LiM array, which should precede the `map` section of the global `.dex` file. An example is shown in the following.

```
begin constants
        BUILT_IN CLOCK 10.0
        BUILT_IN SF 0
        BUILT_IN AR 3.092783
        BUILT_IN VDD 1.1
        BUILT_IN BA 0
        BUILT_IN PROB 0.5
end constants
begin init
        LIM state_array(4,8)
        LIM key_array(4,8)
end init
begin state_array
        ...
end state_array
begin key_array
        ...
end key_array
begin map
        ...
end map
```

In spite of the syntax, the current version of DExIMA Backend encounters a segmentation fault if multiple LiM sections are found by the parser in the same `.dex` file. Because of this issue, in case of a multi-array scenario, DExIMA CAD must split the description of the complete LiM system in multiple `.dex` files, one for each array. In truth, a workaround may be easily found to compress the description of the complete LiM system in a single `.dex` file. Nevertheless, the sections in the resulting `.dex` description would have an unspecified scope and would lack a proper

hierarchical organization. For this reason, regardless of the possible workaround, multiple `.dex` files are generated, and existing array-to-array connections in the uppermost architectural level are temporarily discarded.

5.3 Top-level analyzer

To answer the needs of the changes presented in Section 5.1 and tackle the issues detailed in Section 5.2, new computational facilities are introduced in DExIMA CAD: more specifically, `class TopLevelAnalyzer`, an instance of which will be hereinafter referred to as "top-level analyzer", is placed at the very top of a well-structured hierarchical organization, which encompasses both the structural and the algorithm description facilities in DExIMA CAD.

The top-level analyzer is a purely computational module and is conceived to primarily interact with existing files in the DExIMA CAD project path and with other computational modules in DExIMA CAD, including:

- instances of `class ArrayManager`;
- instances of `class ControlGenerator`;
- the components library, which is an instance of `class ComponentsLibrary`.

Thanks to these interactions, the top-level analyzer is able to fulfill different analysis tasks, which are carried out when its `build()` method is called and pave the way for its main synthesis tasks:

- generation of the HDL source code of the complete LiM system;
- configuration of the UVM testbench for the simulation of the complete LiM system;
- generation of a DExIMA Backend `.dex` description of the complete LiM system.

All analysis tasks implemented by the top-level analyzer are meant to support the changes presented in Section 5.1 and to simultaneously tackle the issues detailed in Section 5.2. Since the content of the uppermost architectural level is defined by a `.lim` DExIMA design file, the top-level analyzer can infer an equivalent DExIMA graph representation, which may be used to identify and sort all top-level components and their mutual connections. For this reason, `class TopLevelAnalyzer` inherits all attributes and methods from `class DExIMAGraphArchitecture`. Nevertheless, the actual behaviour of the main class method `build()` is overloaded, so that the additional top-level tasks can be carried out.

Figure 5.4 reports a summary of all interactions between the top-level analyzer and previously mentioned DExIMA CAD components. The top-level analyzer is always linked to the library of components, represented by an instance of `class ComponentsLibrary`. All files related to the description of a LiM array structure are handled by one or more instances of `class ArrayManager`, while all algorithm-related files are managed by an instance of `class ControlGenerator`. The top-level analyzer directly handles the `.csv` file with the values of the active pins and the `.lim` file representing the structure of the uppermost architectural level. After its build phase, the top-level analyzer may carry out its synthesis tasks.

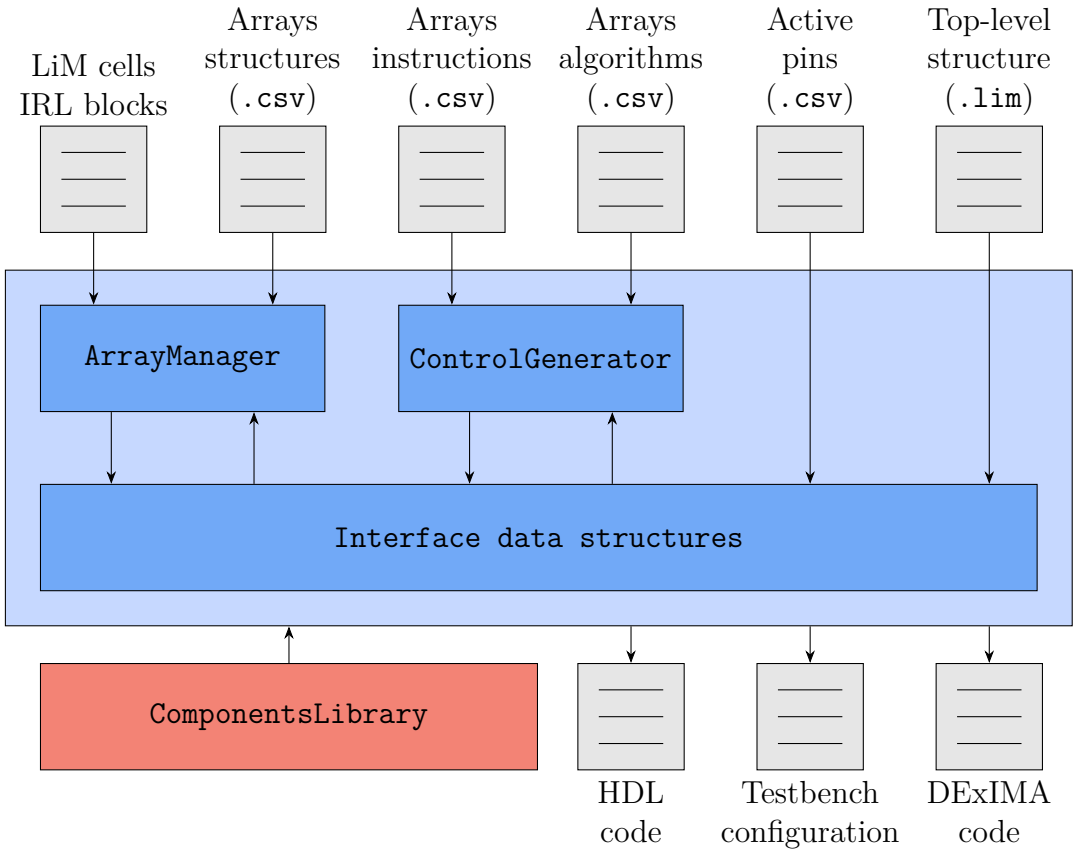


Figure 5.4: Interactions between the top-level analyzer and other components in DExIMA CAD.

Despite it being a purely computational module, the top-level analyzer must interact with other GUI elements in DExIMA CAD, which are supposed to aid the generation of part of the required design files. In fact, with respect to Figure 5.4, the active pins and all algorithm-related files are not available prior to the build phase, but they ar generated at a later stage of the design flow. More details on the design flow are presented in Subsection 5.3.1.

5.3.1 Revisited design flow

The introduction of `class TopLevelAnalyzer` in DExIMA CAD leads to slight modifications to the overall design flow, especially for the phases related to the uppermost architectural level of the complete LiM system.

With respect to the design flow offered by the past version of DExIMA CAD, no difference is found in the structural description of a LiM array. The structure of all LiM cells and IRL blocks is fixed by means of the schematic editor, which produces a set of `.lim`, `.irl` and `.intr` files. DExIMA CAD then infers the actual layout of the LiM array from two CSV files, which specify the LiM cells pattern and the IRL blocks pattern.

When all LiM arrays have been designed, the designer may move to the largest-scope design context, which supports the structure depicted in Figure 5.3. Multiple LiM arrays may be instantiated, but the designer must comply with the rules presented in Section 5.1: a wordline multiplexer, a memory interface and an algorithm ROM must be manually allocated for each array type, and all LiM arrays of the same kind must be connected in the same manner to their external and internal control elements. Lastly, should it be necessary, additional library components may be instantiated in the uppermost architectural level.

At this stage of the design process, besides the previously mentioned files, the project path should contain the `.lim` file describing the structure of the uppermost architectural level of the complete LiM system. The designer may thus invoke the intervention of the top-level analyzer, which carries out a set of tasks and interacts with some GUI elements in DExIMA CAD. Figure 5.5 offers a representation of the remaining design flow phases, which are supported by the top-level analyzer and will be described in the following.

1. **BUILD** The designer triggers a top-level analyzer build, which invokes its `build()` method. The top-level analyzer scans the content of the uppermost architectural level to identify all LiM arrays and their associated internal and external control elements. Further details on this phase are presented in Subsection 5.3.2.

2. **NOTIFY** The designer triggers an intervention of the algorithm description facilities and of the simulation dashboard. These components require a knowledge of the LiM array types and of the active pins in the uppermost architectural level. The top-level analyzer provides these pieces of information in this phase.

3. **PROGRAM** The designer interacts with the algorithm description module and with the simulation dashboard to specify the nano-instructions and the algorithm of each LiM array type. Moreover, for each algorithm step, the designer may force specific values for the available active pins. This phase generates the following files:

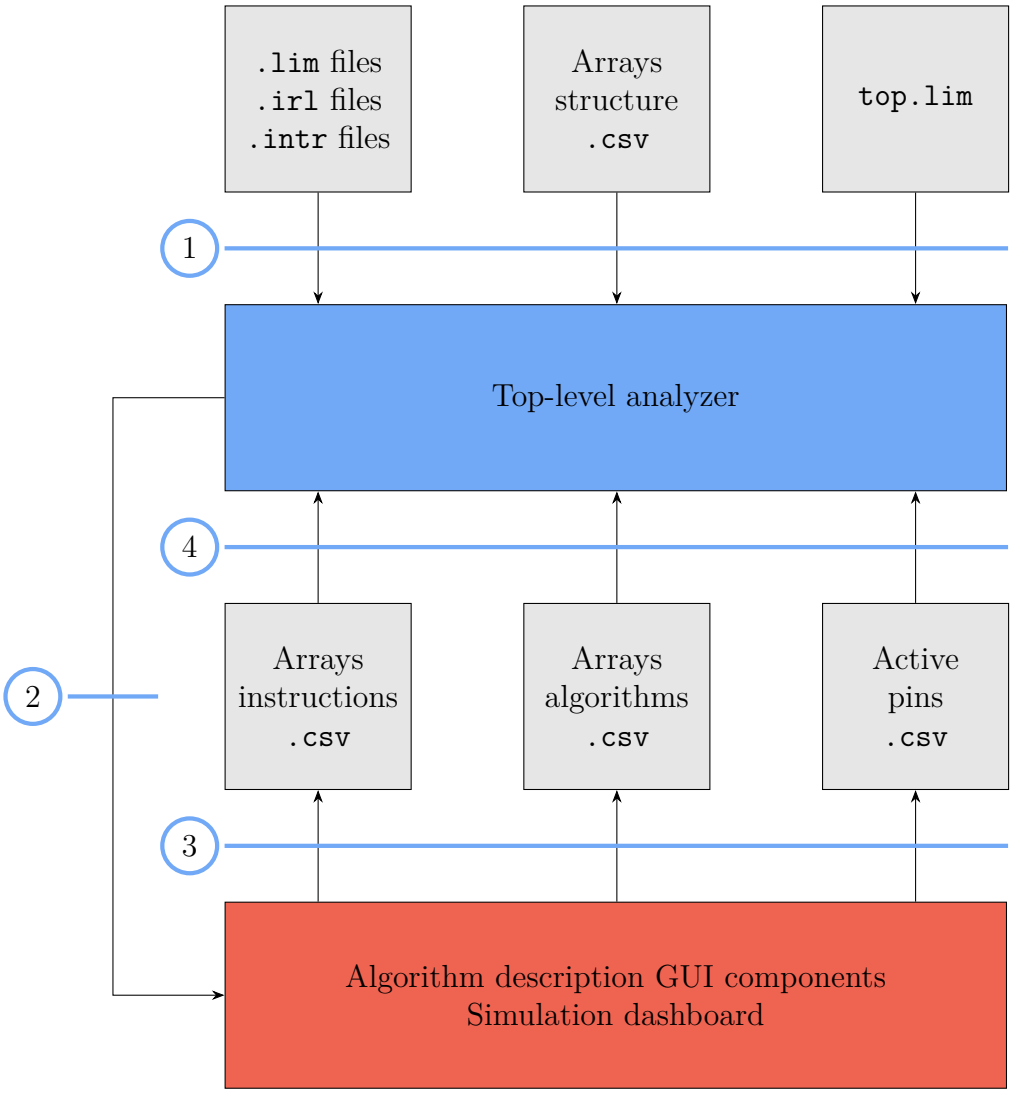


Figure 5.5: Representation of the last DExIMA CAD design flow phases, as detailed in Subsection 5.3.1. The first step is the **BUILD** phase, the second step is the **NOTIFY** phase, the third step is the **PROGRAM** phase, the fourth step is the **GENERATE** phase.

- for each LiM array type, a nano-instructions `.csv` file;
- for each LiM array type, an algorithm `.csv` file;
- a `.csv` file with the values of the active pins.

Further details on the simulation dashboard are presented in Subsection 5.3.3.

4. **GENERATE** At this stage of the design process, the top-level analyzer is aware of the structure of the complete LiM system at every hierarchical level

and of its intended simulation-time behaviour. The designer may thus activate the main synthesis tasks of the top-level analyzer. Further details are presented in Subsection 5.3.4, Subsection 5.3.5 and Subsection 5.3.6.

5.3.2 Build phase of the top-level analyzer

The build phase of the top-level analyzer is represented by its `build()` method, which consists of a set of sequential tasks. Should any of these tasks fail, the build procedure is immediately interrupted and the outcome is notified to the designer.

As the top-level analyzer is a complex computational module, its internal modularity is guaranteed by multiple private methods, which provide a separate implementation of the previously mentioned tasks.

- The local connectivity in the uppermost architectural level is verified. This task is actually carried out by the inherited method `checkConnectivity()`.
- With the `__seekMemoryArrayNodes()` method, the top-level analyzer looks for all memory array instances in the uppermost architectural level. Then, an association between all array instances and their corresponding types is created by the `__sortMemoryArrays()` method. The top-level analyzer is thus aware of the different LiM array types in the uppermost architectural level. For each array type, the top-level analyzer creates a dedicated `class ArrayManager` instance, which is used to check the consistency of its structural description.
- With the `__seekMemoryInterfaceNodes()` method, the top-level analyzer looks for all memory interfaces in the uppermost architectural level. A sorting procedure, represented by the `__sortMemoryInterfaces()` method, verifies if only one memory interface per array type is allocated in the top level.
- The above procedure is then repeated for all algorithm ROMs, more specifically by the methods `__seekAlgorithmROMNodes()` and `__sortAlgorithmROMNodes()`.
- The top-level analyzer looks for all active pins in the uppermost architectural level. The `__seekActivePins()` method uses the underlying DExIMA graph representation to identify all non-floating external input pins; specific pins, including `CLK` and `RST`, are discarded, as they should not be directly controlled by the designer.
- Some active pins have a special purpose in the uppermost architectural level. The top-level analyzer proceeds to the identification of the following classes of active pins:
  - enable-type active pins are expected to be connected to the selection signal `S` of a wordline multiplexer;

  - wordline-type active pins are supposed to be connected to the input signal `IN1` of a wordline multiplexer;
  - bitline-type active pins are expected to be connected to the `BL` port of a LiM array.

  These special types of active pins are identified by name. For instance, the name of a bitline-type active pin must start with `BL` and may be followed by a numerical identifier. As an example, the set `{BL, BL1, BL2}` identifies three different bitline-type active pins.

- With the `__seekWordlineMultiplexers()` method, the top-level analyzer looks for all wordline multiplexers in the uppermost architectural level. For each array type, the corresponding wordline multiplexer is found if the following conditions are met:

  - a wordline multiplexer is an instance of `class MultiBitComponent` with type `Muxnbit`;
  - the selection signal `S` of a wordline multiplexer must derive from the I/O interface of the complete LiM system, and it must be an enable-type active pin;
  - the input signal `IN1` of a wordline multiplexer must derive from the I/O interface of the complete LiM system, and it must be an wordline-type active pin;
  - the input signal `IN0` of a wordline multiplexer must derive from the memory interface related to the analyzed array type;
  - the output signal `O` of a wordline multiplexer must be connected to the `WL` pin the analyzed array type.

- With the `__seekMemoryClusters()` method, the top-level analyzer looks for memory clusters, i.e. the set of components which are connected, in either direction, to a LiM array. The identification of all memory clusters is essential for the purposes of generating the DExIMA Backend description of the complete LiM system, as explained in Subsection 5.3.6.

Figure 5.6 reports an abstract representation of the most meaningful data structures involved in the build phase of the top-level analyzer. After a call to its `build()` method, the top-level analyzer may use its data structures to quickly reference specific components or connections in the uppermost architectural level, supporting all remaining phase of the revisited design flow, as specified in Subsection 5.3.1.

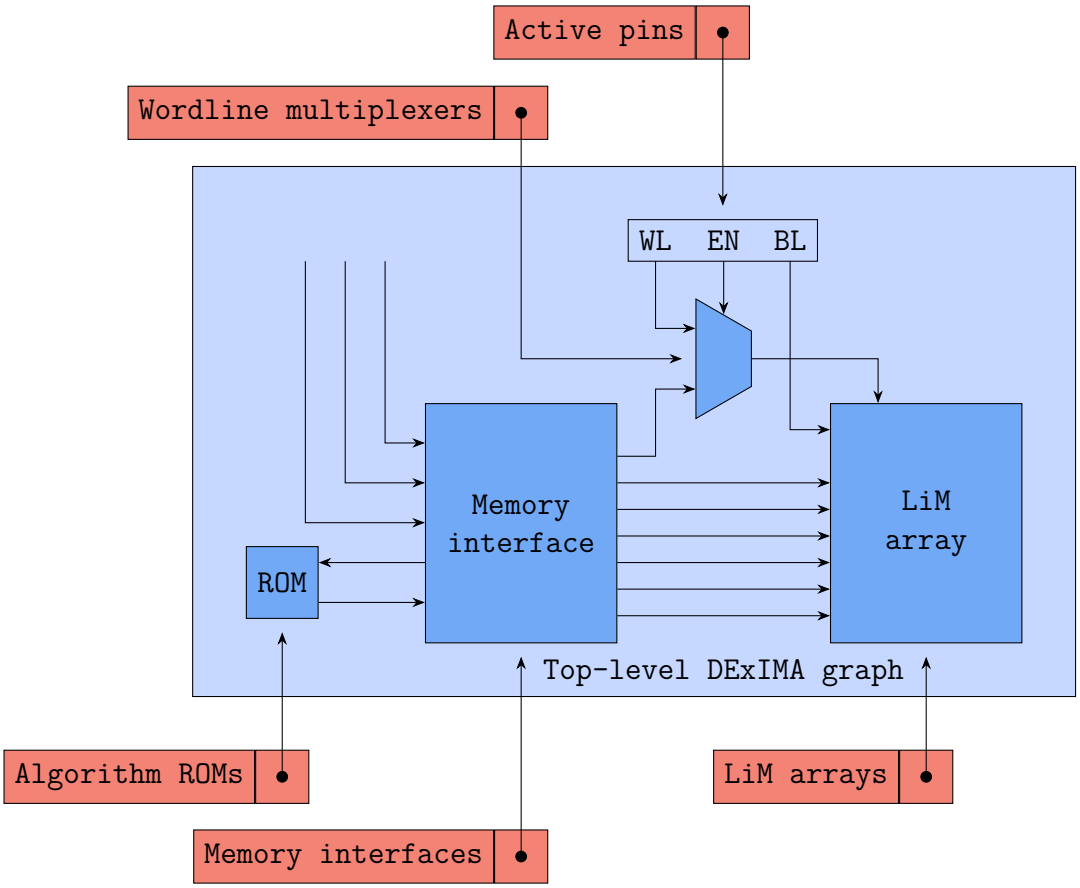


Figure 5.6: Simplified representation of the meaningful data structures involved in the build phase of the top-level analyzer.

5.3.3 Interaction with the simulation dashboard

After its build phase, the top-level analyzer is aware of the different LiM array types and of the active pins in the uppermost architectural level. This pieces of information can be notified to the GUI components of the algorithm description module.

The DExIMA CAD simulation dashboard is a GUI component that allows to specify the simulation-time behaviour of the complete LiM system. When the designer requests its intervention, the simulation dashboard creates as many algorithms context as the number of different LiM array types in the uppermost architectural level; furthermore, it shows all active pins, as gathered by the top-level analyzer. The designer may thus decide to add an algorithm step, for which

- in each algorithm context, the designer describes the behaviour of its related LiM array by combining a set of nano-instructions in a scalar control and by associating an activation pattern;

- the designer may force specific values on the available active pins.

At any time, the designer may decide to save the current content of the simulation dashboard. This operation is essential for any further synthesis task and it generates the following files:

- for each LiM array type, an algorithm `.csv` file;
- a `.csv` file with the values of the active pins.

Moreover, if all the above files are already available in the project path, e.g. because of a prior design, the designer may decide to restore the content of the simulation dashboard.

It is important to point out that the simulation dashboard does not create the nano-instructions `.csv` files. As a matter of fact, when created, it must already know all nano-instructions of all LiM arrays. As a consequence, these files are generated by a separate GUI component of the algorithm description module.

An example of DExIMA CAD simulation dashboard is reported in Figure 5.7.

DExIMA CAD simulation dashboard

	Control	Pattern	BL	WL	EN	ADDR
1	Init	SINGLE:80	0	0	0	0
2	Acc1	SINGLE:80	0	0	0	0
3	Acc2	SINGLE:80	0	0	0	0
4	Acc3	SINGLE:80	0	0	0	1
5	Acc4	SINGLE:80	0	0	0	0

Add step | Remove step | Save | Restore

Figure 5.7: Representation of the simulation dashboard in DExIMA CAD.

5.3.4 Source code generation

The generation of the synthesizable HDL source code of the complete LiM system is a hierarchical process which involves several modules in DExIMA CAD.

For each array type in the uppermost architectural level, the top-level analyzer must generate:

- the source code of its LiM cells and IRL blocks;
- the source code of the array itself, where its LiM cells and IRL blocks are properly allocated and interconnected;
- the source code of its associated algorithm ROM;
- the source code of its associated memory interface, i.e. its internal control.

An instance of `class ArrayManager`, allocated during the build phase of the top-level analyzer, handles the first two items of the above list by means of its `generateVHDL()` method. As for the remainder of the above items, the third is managed by a `class ControlGenerator` instance and its `generateMicroROM()` method, while the fourth is demanded to slightly modified versions of existing DExIMA CAD modules, namely `class MemoryInterfaceGenerator` and `class nCUGenerator`.

The source code generation process is completed by the creation of a HDL file for the uppermost architectural level itself. This task is rather easy to carry out, as an instance of `class GraphToHDL` may be used to convert the DExIMA graph representation of the top-level in a corresponding HDL description, allocating all LiM arrays and library components and creating their mutual connections.

5.3.5 Configuration of the UVM testbench

In Section 5.2, it has been mentioned that DExIMA CAD must be able to adapt its UVM testbench to the actual needs of the complete LiM system under test. The current implementation of the Universal Verification Methodology (UVM) offers a large variety of tools to manage multiple configurations of the same testbench, e.g. resources and configurations databases. However, it has been decided to deliberately avoid the use of such tools, as different simulation approaches may be explored in future developments of DExIMA CAD. For this reason, the revisited version of DExIMA CAD employs a macro-based approach to support the simulation-time configuration of the UVM testbench, the structure of which is reported in Figure 5.8.

The structure of a DExIMA-controlled testbench contains all the typical elements of a UVM testbench. An interface acts as a communication link between the UVM components and the actual hardware, i.e. the complete LiM system. A set of transactions is generated and transferred by the sequencer to the driver, which effectively drives the interface and, in turn, the Device Under Test (DUT). Several containers, including an agent, an environment and a test, are deployed to provide a proper hierarchical separation between all components.

In the present version of DExIMA CAD, the testbench consists of the following files:

- `Driver.sv`;

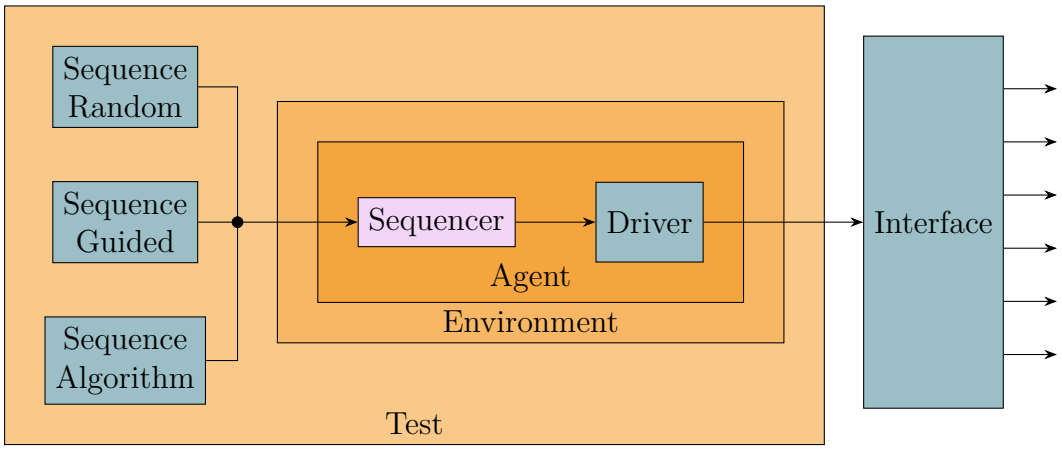


Figure 5.8: Structure of the DExIMA-controlled UVM testbench. All the elements in light blue are configured by DExIMA CAD and thus depend on the actual design.

- `Interface.sv`;
- `Packet.sv`;
- `Sequences.sv`;
- `Testbench.sv`.

All the above files invoke specific macros, which are generated by the top-level analyzer and defined in a separate file called `Definitions.sv`. Regardless of the design, DExIMA CAD does not directly modify the content of the above testbench files, which are thus static entities in DExIMA CAD, but rather changes the macro definition in `Definitions.sv`. This approach provides a simple, yet effective, testbench configuration mechanism.

In the following paragraphs, the meaningful pieces of information related to these configuration macro will be presented.

Interface The source code of the testbench-to-hardware interface is reported in the following.

```
interface InterfaceTop (input logic CLK, input logic RST);

/*
        Declaration of the active pins, i.e.
        non-floating input pins in the top-level.
*/
        `DExIMA_ACTIVE_PINS
```

```
endinterface : InterfaceTop
```

The interface only contains the `DExIMA_ACTIVE_PINS, which is used to specify the active pins involved in the simulation.

Packet The source code of `class Packet`, i.e. a UVM sequence item, is reported in the following.

```
class Packet extends uvm_sequence_item;

/*
        Declaration of the active pins, i.e.
        non-floating input pins in the top-level.
*/
        `DExIMA_ACTIVE_PINS

/*
        Factory registration
*/
        `uvm_object_utils_begin(Packet)
        `uvm_component_utils_end

/*
        Initialization function
*/
        function void initialize();
                `DExIMA_PACKET_INITIALIZATION
        endfunction : initialize

endclass : Packet
```

Each packet must integrate the `DExIMA_ACTIVE_PINS macro to specify its content. Moreover, this class is endowed with the `function void initialize()` method, which is used to zero-out its content: the body of this method is actually defined by the `DExIMA_PACKET_INITIALIZATION, as the initialization procedure must be aware of the active pins in the complete LiM system.

Driver The following listing reports the `task run_phase()` of the driver, i.e. an instance of `class Driver`.

```
task run_phase(uvm_phase phase);
        Packet pkt;
```

```
/*
        Zero-out all non-floating signals
        in the interface
*/
        `DExIMA_DRIVER_RUN_PHASE_BEGIN

/*
        Wait for the reset to stabilize
*/
        @ (posedge vif_top.RST);
        @ (posedge vif_top.RST);

        forever begin
        /*
                Get the next item from the sequencer
        */
                seq_item_port.get_next_item(pkt);

                @ (posedge vif_top.CLK);

        /*
                Packet-to-Interface transfer
        */
                `DExIMA_DRIVER_RUN_PHASE_MAIN

        /*
                Signal the sequencer that the
                item has been processed
        */
                seq_item_port.item_done();
        end
endtask : run_phase
```

The `DExIMA_DRIVER_RUN_PHASE_BEGIN macro is invoked at the very beginning of the run phase to initialize the testbench-to-hardware interface, zeroing-out all enable-type and wordline-type signals, in order to prevent any spurious operation in any LiM array. The `DExIMA_DRIVER_RUN_PHASE_BEGIN macro is then used to transfer the content of each new packet to the interface. Evidently, both operations must be aware of the active pins in the complete LiM system.

Sequences Three different sequences are used to apply the required stimuli to the LiM system under test:

- an instance of `class SequenceInitializationRandom` serves the purposes of

randomly initializing the content of all LiM arrays;

- an instance of `class SequenceInitializationGuided` manages the guided initialization of specific LiM arrays;
- an instance of `class SequenceAlgorithm` supports the execution of the algorithm, as specified by the designer in the simulation dashboard.

All the above sequences must be aware of the LiM arrays in the design and of the active pins, thus three different macros are generated by the top-level analyzer:

- `` `DExIMA_RANDOM_INITIALIZATION ``;
- `` `DExIMA_RANDOM_GUIDED ``;
- `` `DExIMA_ALGORITHM_SEQUENCE ``.

The `task body()` of `class SequenceInitializationRandom` is reported in the following.

```
task body();
        int i;
        Packet pkt;

        // Send a first 0-initialized packet to disable all arrays
        pkt = Packet::type_id::create("Packet");
        pkt.initialize();
        start_item(pkt);
        finish_item(pkt);

        // Random initialization for all arrays
        `DExIMA_RANDOM_INITIALIZATION

        // Send a last 0-initialized packet to disable all arrays
        pkt = Packet::type_id::create("Packet");
        pkt.initialize();
        start_item(pkt);
        finish_item(pkt);
endtask : body
```

With the `` `DExIMA_RANDOM_INITIALIZATION `` macro, for each array type, the top-level analyzer generates as many packets as the number of rows in the memory array, which only differ by the wordline signal, as it is sequentially increased to activate the whole array; each packet asserts the enable-type signal and contains a randomized bitline value.

The `task body()` of `class SequenceInitializationGuided` is reported in the following.

```
task body();
    int fd;
    string line;
    int addr, val;
    Packet pkt;

    // Send a first 0-initialized packet to disable all arrays
        pkt = Packet::type_id::create("Packet");
        pkt.initialize();
        start_item(pkt);
        finish_item(pkt);

    `DExIMA_GUIDED_INITIALIZATION

    // Send a last 0-initialized packet to disable all arrays
    pkt = Packet::type_id::create("Packet");
    pkt.initialize();
    start_item(pkt);
    finish_item(pkt);
endtask : body
```

With the `` `DExIMA_GUIDED_INITIALIZATION `` macro, for each LiM array requiring a specific initialization, the top-level analyzer generates as many packets as the number of lines in its corresponding initialization file; each packet asserts the associated enable-type signal and forces the required values on the associated bitline-type and wordline-type signals.

The `task body()` of `class SequenceAlgorithm` is reported in the following.

```
task body();
        Packet pkt;
        int i;

/*
        Write address 1 to the instruction queue
*/
        pkt = Packet::type_id::create("Packet");
        pkt.initialize();
        pkt.queueIN = 1;
        pkt.queueWen = 1;
        start_item(pkt);
        finish_item(pkt);
```

```
/*
        Activate the LiM array for 4 clock cycles
        to load address 1 to the uPC
*/
        for (i = 0; i < 4; i++) begin
                pkt = Packet::type_id::create("Packet");
                pkt.initialize();
                pkt.LiMActivate = 1;
                start_item(pkt);
                finish_item(pkt);
        end

        `DExIMA_ALGORITHM_SEQUENCE

/*
        Deactivate the LiM array
*/
        pkt = Packet::type_id::create("Packet");
        pkt.initialize();
        start_item(pkt);
        finish_item(pkt);
endtask : body
```

An initial transaction writes the address of the first instruction to the instructions queue of all LiM arrays in the design. Then, irrespective of the algorithm to be simulated, the `LiMActivate` signal is asserted for four consecutive clock cycles, so to allow the previously written address to exit the instructions queue and reach the program counter. Finally, with `` `DExIMA_ALGORITHM_SEQUENCE `` macro, the top-level analyzer generates as many packets as the number of actual algorithm steps, asserting the `LiMActivate` signal and thus enabling any LiM computation; each packet contains an indication of the active pins, should they be controlled during the LiM processing.

Testbench The last DExIMA-controlled UVM element is the testbench itself, i.e. `module Testbench`.

The testbench instantiates the general simulation signals, namely the asynchronous reset and the clock, the testbench-to-hardware interface and the complete LiM system, which is actually declared and connected to its interface by means of the `` `DExIMA_TOP_ARCHITECTURE_INSTANCE `` macro.

Furthermore, as the clock period may be changed by the designer, the behaviour of the aforementioned general simulation signals is actually controlled by the `` `DExIMA_CLOCK_RESET_TASKS `` macro, which adapts the body of the following tasks:

- `task generateClock();`
- `task generateReset().`

5.3.6 Generation of the DExIMA Backend description

To support an estimation of the main figures of merit of the complete LiM system, the top-level analyzer must generate a DExIMA Backend `.dex` file, which contains a description of its structure and of the mutual connections between all building blocks and components. In fact, at the very end of the design flow, the `generateDescriptionDExIMA()` method of the top-level analyzer may be invoked to trigger the `.dex` file generation process.

Section 5.2 introduced a specific scope-related limitation of DExIMA Backend in supporting the description of such structures as that of Figure 5.3, which results in the impossibility of describing a connection involving a LiM array and a library component in the `map` section of the global DExIMA Backend `.dex` file. A workaround is provided by the following procedure.

- With its `build()` method, the top-level analyzer is able to identify all memory clusters in the uppermost architectural level. Evidently, the top-level analyzer is also aware of all top-level components which do not belong to any memory cluster: in this context, these components will be referred to as ”out-of-memory components“.
- The top-level analyzer prepares a set of `.dex` files, which will collectively represent the thorough description of the complete LiM system: besides a `.dex` file for each LiM array type, a further file, namely `nma.dex`, is prepared.
- The elements belonging to a memory cluster are passed to the corresponding instance of `class ArrayManager`, so that they can be properly declared and mapped by its `generateDExIMA()` method.
- All out-of-memory components are declared and connected in `nma.dex` by the top-level analyzer, without a direct intervention of any other DExIMA CAD module.

The above steps are summarized in Figure 5.9.

This solution is perfectly feasible, as the output DExIMA Backend description is expected to mimic the actual structure of the complete LiM system, but it deliberately violates the scope of all `.dex` sections and sub-sections. The main shortcoming of such violation is the introduction of a circular dependency between the top-level analyzer and an instance of `class ArrayManager`, as the latter requires the former to generate the `.dex` file of a LiM array. Future expansions of DExIMA Backend may

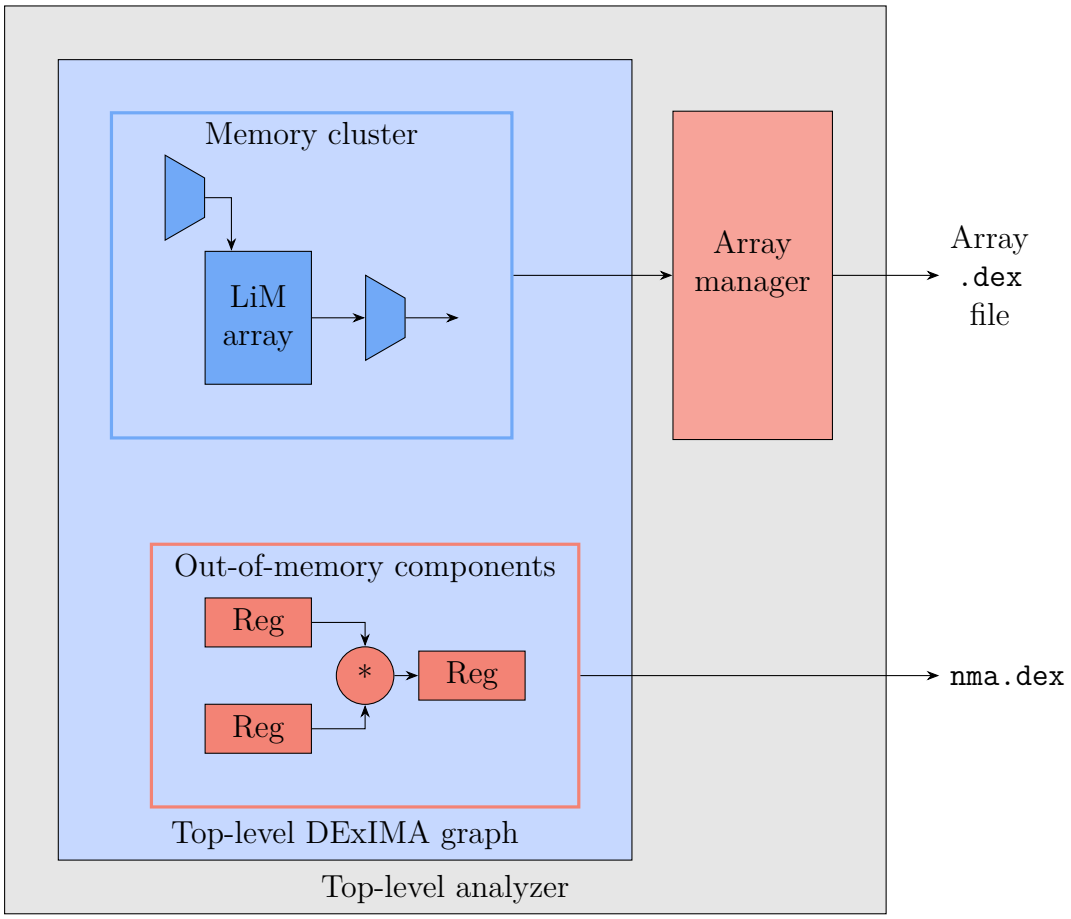


Figure 5.9: Representation of the `.dex` file generation process, as implemented by the top-level analyzer. All memory clusters are handled by specific instances of `class ArrayManager`, while out-of-memory components are directly managed by the top-level analyzer.

remove the limitations presented in Section 5.2, leading to a more straightforward and linear `.dex` generation process.

Some components belonging to a memory cluster have a special meaning in the context of DExIMA CAD. In the present version of DExIMA CAD, these components are:

- wordline multiplexers;
- memory rows multiplexers.

An instance of `class ArrayManager`, when its method `generateDExIMA()` is called, must be able to recognize these components and handle them in a specific manner.

Wordline multiplexer The array manager must ensure a proper connection between a wordline multiplexer and its associated LiM array. More specifically, the i-th bit of the multiplexer output signal must be connected to the `EN` pin of all memory cells belonging to row i. To fulfill this task, the `map` section in `.dex` file of the LiM array must contain a mapping procedure, as shown in the following example, which applies to a 32-row, 8-column memory array.

```
for i in range(0,1,31) {
        for j in range(0,1,7) {
                WLMux.O[$i] -> Memory($i,$j).EN
        }
}
```

It is important to stress that the syntax in the previous example would not be accepted by the DExIMA Backend parser, as it only allows line-based statement: the indentation has been manually added for representation purposes only.

Memory rows multiplexer The array manager must ensure a proper connection between a memory rows multiplexer and its associated LiM array. More specifically, all memory cells must be collected by the `packed` input, as shown by the following examples, which applies to a 128-row, 32-column memory array.

```
for i in range(0,1,127) {
        for j in range(0,1,31) {
                Memory($i,$j).RD ->
                        MRMux.packed[$( 32 * $i + 32 - 1 - $j )$]
        }
}
```

Moreover, the memory rows multiplexer may be connected back to the memory array, if required. In the present version of DExIMA CAD, this kind of connection is supported by the horizontal bus, it involves the `multiplexed` output of the memory rows multiplexer and the dummy component `HandlerHBUS`.

```
for j in range(0,1,31) {
        MRMux.multiplexed[$j] -> HandlerHBUS.HandlerIN[$j]
}
```

Part III

Implementations in DExIMA CAD

Chapter 6

Case studies implementation in DExIMA CAD

The purpose of Chapter 6 is to provide an initial proof of the effectiveness of the new DExIMA CAD structural description capabilities, which are proved by the DExIMA CAD implementation of some LiM arrays, which will be hereinafter referred to as ”case studies“.

Section 6.1 deals with the DExIMA CAD implementation of the two LiM arrays presented in [11]. These case studies serve the purpose of showing how additional array interconnections, as defined in Subsection 3.1.3, may ease the description of a LiM array.

In Section 6.2, a LiM array for a Finite Impulse Response (FIR) digital filter algorithm is implemented, which is expected to highlight the effectiveness of horizontal and vertical array interconnections.

For the LiM ones counter and the LiM FIR arrays, the use of the new algorithm description facilities, as presented in Chapter 4, will be described.

6.1 LiM XNOR and LiM ones counter arrays

In Chapter 3, and more specifically in Subsection 3.1.3, it has been mentioned that the DExIMA CAD additional interconnections, namely the horizontal and the vertical busses, have been inspired by the two LiM arrays presented in [11]. The primary purpose of Section 6.1 is to show the implementation of such architectures in the revisited DExIMA CAD.

6.1.1 LiM XNOR array

The structure of the LiM XNOR array is extremely straightforward: each memory cell integrates a XNOR gate, which is fed by the cell itself and by a weight dispatch

mechanism. In the DExIMA CAD implementation of this array, the weight dispatch mechanism is easily integrated by means of a horizontal bus, as shown in Figure 6.1.

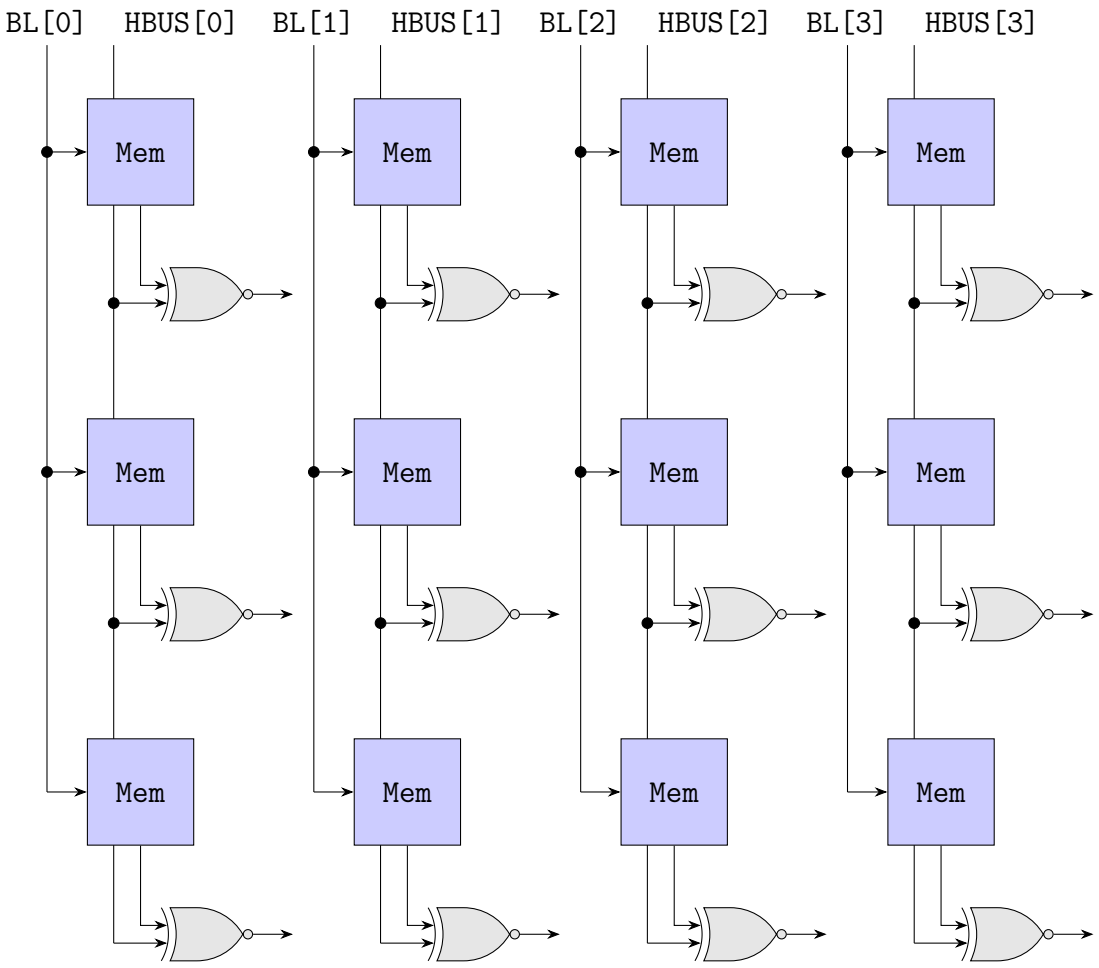


Figure 6.1: Horizontal bus in the LiM XNOR array [11]: the bits from the horizontal bus are used to dispatch the weights word, i.e. the convolution kernel, to the memory array which stores the input feature map.

In Subsection 3.1.3 it has been mentioned that the description of such structure as that of Figure 6.1 was indeed possible in the past version of DExIMA CAD. Nevertheless, because of the lack of such mechanism as the horizontal bus, the description would have needed a separate LiM cell for each column in the array, causing the design process to be more cumbersome. Moreover, in the past version of DExIMA CAD, the template LiM control signals would have been used to provide the weights to the LiM cells.

In the revisited DExIMA CAD, the description of the complete LiM XNOR array is made possible by a single type of LiM cell, which integrates a XNOR gate

and a horizontal bus interconnection, as shown in Figure 6.2. It is immediate to notice that replicating the LiM cell of Figure 6.2 for a certain amount of columns and rows yields precisely the overall structure presented in Figure 6.1.

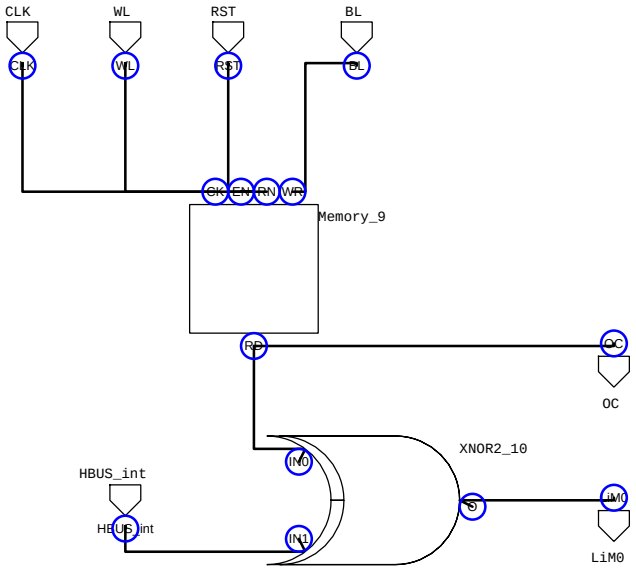


Figure 6.2: DExIMA CAD implementation of the LiM XNOR cells. The `HBUS_int` pin is the input pin of the horizontal bus interconnection.

From both Figure 6.1 and Figure 6.2, it is immediate to notice that no LiM control signal is actually required by the array. As a consequence, its simulation in DExIMA CAD is extremely straightforward. The following set of nano-instructions may be defined.

- `NOP`: Do not perform any operation.
- `LOAD`: Prepare a write operation to a selected memory row.

The random initialization sequence presented in Chapter 5 takes care of loading a set of random values in the LiM array. The algorithm sequence may simply consist of `NOP` nano-instructions with a `NONE` activation pattern.

6.1.2 LiM ones counter array

The structural description of the LiM ones counter array is attainable by the combined use of horizontal interconnections and of a vertical bus, as shown in Figure 6.3.

It is important to point out that the past version of DExIMA CAD would not have allowed the implementation of such structure as that of Figure 6.3, because

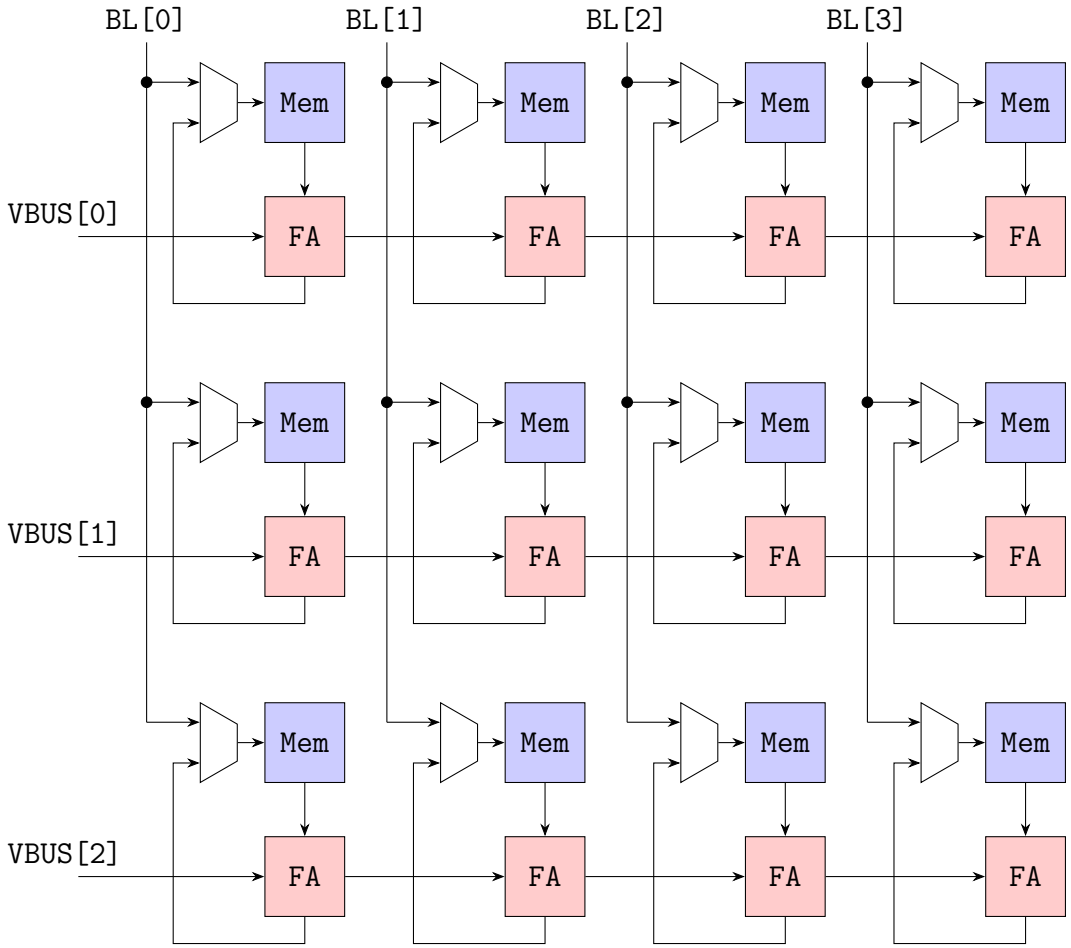
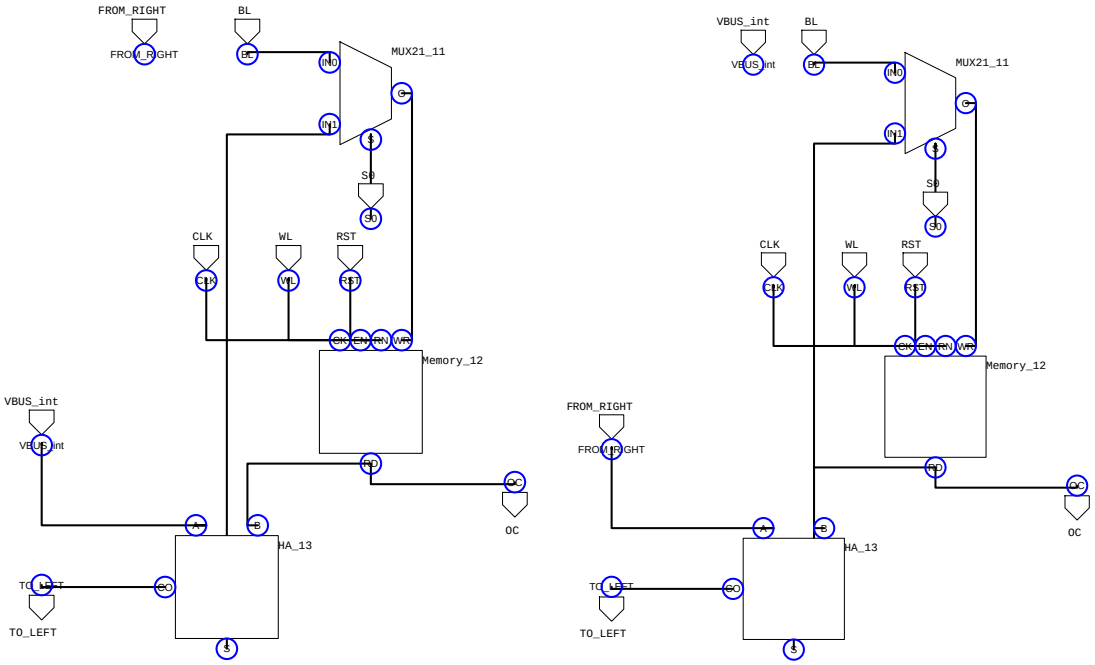


Figure 6.3: Vertical bus in the LiM ones counter array [11]: the bits from the vertical bus are used to feed the arithmetic counters, the structure of which is made possible by horizontal LSB-to-LSB interconnections.

horizontal array interconnections were not supported. In the revisited DExIMA CAD, the LiM ones counter array requires two different types of LiM cells:

- the first type of LiM cell, `cell_lim_0`, is allocated on the LSB column, it uses the vertical bus to initialize the carry propagation, which is continued by a horizontal LSB-to-MSB interconnection;
- the second type of LiM cell, `cell_lim_j`, is allocated on every other column, it does no longer require an explicit connection from the vertical bus and it continues to carry propagation up to the MSB.

The DExIMA CAD implementation of the above LiM cells is presented in Figure 6.4.



(a) LSB cell.

(b) All other cells.

Figure 6.4: DExIMA CAD implementation of the LiM ones counter cells. The left-hand side reports the structure of `cell_lim_0`, with the required connection to the vertical bus. The right-hand side shows the structure of `cell_lim_j`, which integrates the horizontal interconnection pins `TO_LEFT` and `FROM_RIGHT`.

The uppermost architectural level only contains the LiM ones counter array, its internal control (i.e. its memory interface and its algorithm ROM) and its external control (i.e. its wordline multiplexer). Since the vertical bus is a non-floating input pin, it is recognized by the top-level analyzer as an active pin, and it can thus be controlled by the simulation dashboard.

Nano-instructions From both Figure 6.3 and Figure 6.4, it is straightforward to notice that a single LiM control signal is required to define the behaviour of the cell multiplexer. The following set of nano-instructions may thus be defined.

- `NOP`: Do not perform any operation.
- `LOAD`: Enable a write operation to a selected memory row.
- `ACC`: Select the output of the half-adder as the write input of a memory cell.

Simulation dashboard Prior to the actual algorithm, to make sure that the counting procedure is properly executed, all memory rows must be initialized to zero. This task can be fulfilled by extending a `LOAD` scalar control to the whole LiM array, by means of a `ALL` activation pattern, and by enforcing the value `0` on the bitline.

Each algorithm step consists of a `ACC` scalar control with an associated `ALL` activation pattern. To simplify the functional verification phase, the `VBUS` sends a set of ones either to the even-indexed rows or to the odd-indexed rows. Assuming an 8-row LiM array, this can be easily done by continuously switching between values 0x55 (85 in decimal notation) and 0xAA (170 in decimal notation).

All the above considerations may be used in the DExIMA CAD simulation dashboard, as shown in Figure 6.5.

DExIMA CAD simulation dashboard

	Control	Pattern	BL	WL	EN	VBUS
1	LOAD	ALL	0	0	0	0
2	ACC	ALL	0	0	0	85
3	ACC	ALL	0	0	0	170
4	ACC	ALL	0	0	0	85
5	ACC	ALL	0	0	0	170

Figure 6.5: Representation of the DExIMA CAD simulation dashboard for the LiM ones counter array.

6.2 Logic-in-Memory implementation of a Finite Impulse Response digital filter

Section 6.1 has already shown how horizontal interconnections are able to expand the structural description capabilities. Before considering more complex LiM architectures, a preliminary study is carried out, aimed at developing a LiM implementation of a Finite Impulse Response (FIR) digital filter. The target architecture of such study is expected to be a rather application-specific solution which can implement the required algorithm following the paradigm of Logic-in-Memory computing and, more specifically, the design philosophy presented in [9] and [16], which can be summed up as follows:

- bit-wise logic operations and simple arithmetic operations, including additions

and subtractions, are directly integrated in the LiM cells;

- complex arithmetic operations, e.g. multiplications, are demanded to IRL blocks.

Given the above constraints and the actual operations involved in the FIR filter, it is expected that the output architecture will require some degree of data movement at array level, which is necessary to achieve the required functionality. In other words, the revisited DExIMA CAD should be able to tackle the description of the designed LiM array.

6.2.1 Derivation of the LiM architecture

A Finite Impulse Response digital filter of order N provides the hardware implementation of Equation 6.1.

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + ... + b_N \cdot x[n-N] = \sum_{k=0}^{N} b_k \cdot x[n-k] \qquad (6.1)$$

In Equation 6.1, $x[n]$ and $y[n]$ are the input and output samples, respectively, and b_k are the filter coefficients. Hereinafter, the set of past samples $x[n-1], ..., x[n-N]$ will be referred to as the "state buffer" of the filter.

To highlight the operators required by a hardware implementation, Equation 6.1 can be used to derive the Data Flow Graph of the algorithm, an example of which is reported in Figure 6.6.

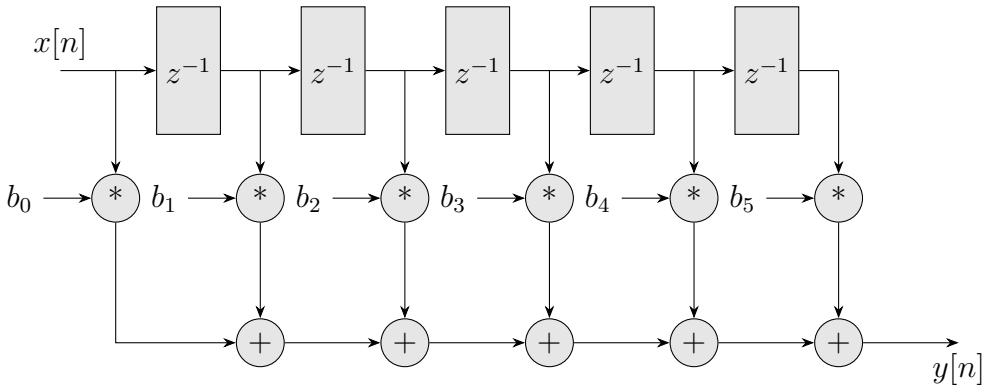


Figure 6.6: Example of Data Flow Graph for a Finite Impulse Response filter of order $N = 5$.

An accurate analysis of Equation 6.1 shows that:

- N storage operators are needed to store N past samples of the input signal;
- $(N+1)$ multiplication operations are required to compute the $(N+1)$ partial products $b_k \cdot x[n-k]$;
- N addition operators are required to accumulate all partial products to produce the final value of the output sample.

Evidently, the memory rows of the LiM array can act as the storage operators, provided that the array consists of at least $(N+1)$ rows, to store both the current sample and past values of the input signal.

As regards the multiplications, each row in the memory array has an associated IRL block which contains a multiplier and an output buffer to store the partial product. Moreover, a input buffer will be used to store the values of the coefficients b_k, leading to the structure of Figure 6.7.

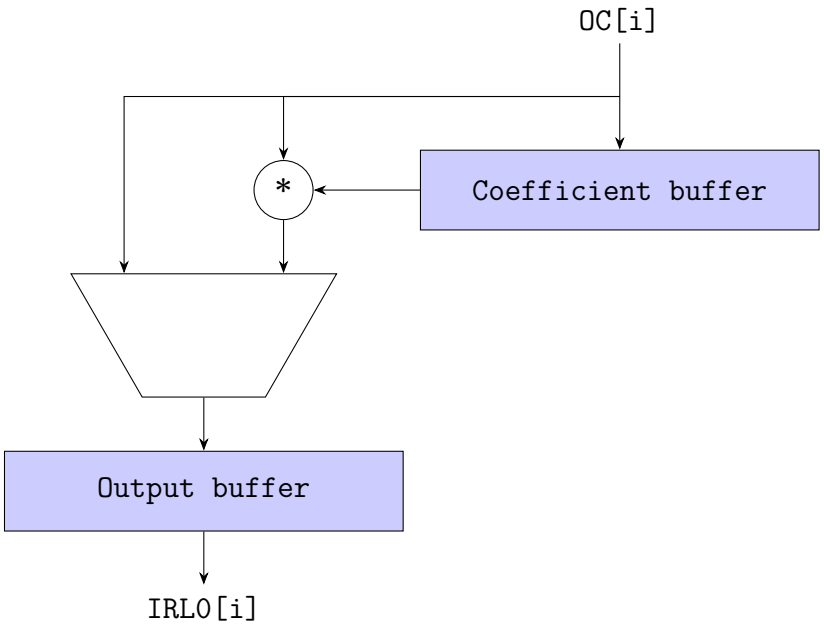


Figure 6.7: LiM FIR architecture: structure of the Intra-Row Logic (IRL) blocks. `OC[i]` is a placeholder for the input pin of a `Prev-1-OC-to-IRL` vertical interconnection.

To implement the additions, all memory rows but the first one integrate an arithmetic circuit, namely a Full-Adder (FA), which can lead to a in-row Ripple-Carry Adder (RCA) architecture.

The first row of the memory array consists of the slightly modified cells depicted in Figure 6.8, integrating an interconnection from the very next IRL block, which is required to bring the partial product $b_0 \cdot x[n]$ to the memory cells.

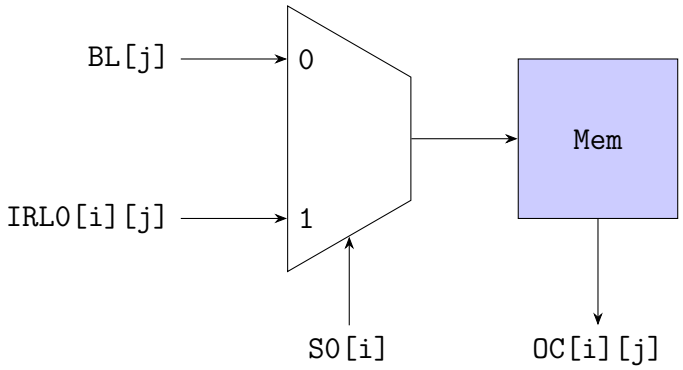


Figure 6.8: LiM FIR architecture: structure of modified memory cells. `IRL0[i][j]` is a placeholder for the input pin of a `Next-1-IRL0-to-Row` vertical interconnection.

The remaining rows of the memory array consist of the arithmetic cells depicted in Figure 6.9, which, in addition to the interconnection element of Figure 6.8, integrate:

- an interconnection from the very previous memory row, to enable the accumulation of the partial products $b_k \cdot x[n-k]$;
- an interconnection from the very previous IRL block, to manage the update of the state buffer;
- an interconnection from the very previous column, to propagate the carry signal, as required by a RCA architecture.

6.2.2 Algorithm description in DExIMA CAD

LiM array behaviour The algorithm which will be supported by the designed architecture consists of two consecutive phases, namely:

1. the start-up phase, which is required to load all filter coefficients b_k and to zero-out the state buffer;
2. the kernel phase, which elaborates each input sample $x[n]$ to produce the output sample $y[n]$.

In the start-up phase, the filter coefficients b_k are initially stored in the memory rows. Then, the coefficients are moved to the input buffers in the IRL blocks in a fully-parallel manner. Lastly, the filter state buffer is initialized in parallel, by loading the value 0 on the bitline and by enabling all the rows in the memory.

In the kernel phase, each input sample $x[n]$ is initially loaded in the first memory row. Then, a parallel multiplication computes all required partial products $b_k \cdot$

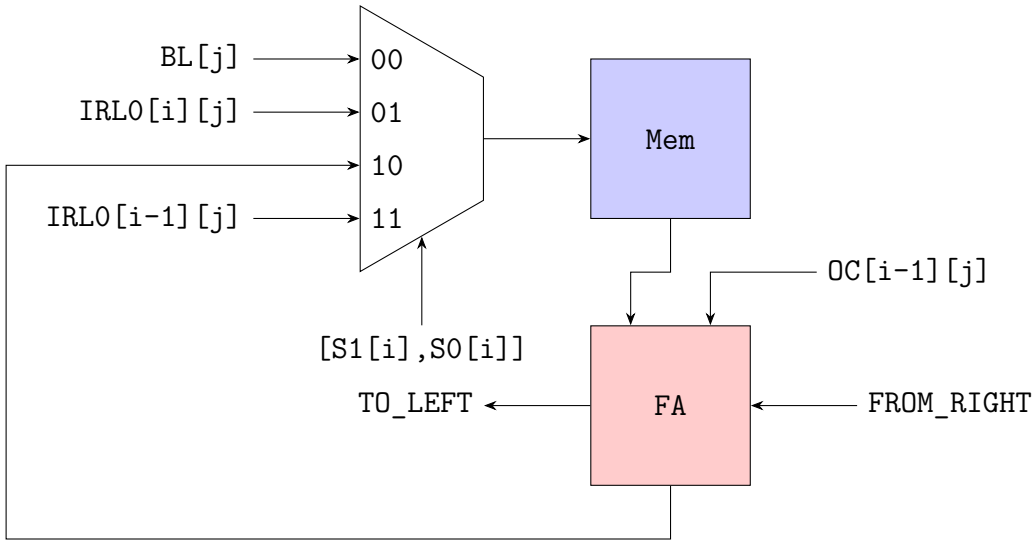


Figure 6.9: LiM FIR architecture: structure of the arithmetic memory cells. Several interconnection placeholders are found: `IRL0[i][j]` indicates `Next-1-IRL0-to-Row` vertical interconnection, `IRL0[i-1][j]` represents `Prev-1-IRL0-to-Row` vertical interconnection, `OC[i-1][j]` indicates a `Prev-1-OC-to-Row` vertical interconnection, while `FROM_RIGHT` and `TO_LEFT` are associated to a LSB-to-MSB horizontal interconnection.

$x[n-k]$ and stores them in the output buffers. Because the accumulation may only involve the actual memory rows, as the structure of the arithmetic cells clearly suggests, all partial products must be transferred from the output buffers to the memory rows; nevertheless, to prevent losing the state buffer, the content of the memory rows is temporarily moved to the output buffers, effectively resulting in an exchange operation which can take place in the same computational cycle. When the data movement is complete, partial products may be accumulated. Lastly, to update the state buffer, each memory row but the first one is loaded with the content of the previous output buffer, effectively completing the shift operation.

Nano-instructions To implement the behaviour detailed by the above considerations, the following set of nano-instructions may be defined.

- `NOP`: Do not perform any operation.
- `LOAD`: Prepare a write operation to a selected memory row.
- `SHH`: Enable the coefficient buffers.
- `MULT`: Multiply memory row and coefficient buffer, save result to output buffer.

- `ROT`: Exchange the content of the memory row and and its associated output buffer.
- `SUM`: Sum two adjacent memory rows, save result to the larger-index memory row.
- `SHV`: Shift the content of an output buffer to the very next memory row.

Simulation dashboard The simulation of the LiM FIR may be thoroughly simplified by taking into account the way the DExIMA CAD UVM testbench works. After the random initialization sequence, each memory row contains a random value, which can be used both as a filter coefficient and as an element in the state buffer, provided that a `SHH` nano-instruction is issued with a `ALL` activation pattern. All partial products are computed by means of a `MULT` nano-instruction, extended to the whole LiM array by means of a `ALL` activation pattern; a subsequent `ROT` nano-instruction, with an associated `ALL` activation pattern, moves all partial products to the memory rows. The accumulation procedure is carried out by a `SUM` nano-instruction with an associated `INCR:1:15` activation pattern. Lastly, the state buffer is upated by means of a `SHV` nano-instruction.

All the above considerations may be used in the DExIMA CAD simulation dashboard, as shown in Figure 6.10.

DExIMA CAD simulation dashboard

	Control	Pattern	BL	WL	EN
1	SHH	ALL	0	0	0
2	MULT	ALL	0	0	0
3	ROT	ALL	0	0	0
4	SUM	INCR:1:15	0	0	0
5	SHV	ALL	0	0	0

Figure 6.10: Representation of the DExIMA CAD simulation dashboard for the LiM FIR array.

Chapter 7

Hybrid-SIMD in DExIMA CAD

The purpose of Chapter 7 is to describe the DExIMA CAD implementation of the Hybrid-SIMD memory array presented in [16], with the primary concern of proving the effectiveness of the new structural description capabilities of DExIMA CAD.

Section 7.1 reports a general overview of the Hybrid-SIMD architecture, while Section 7.2 deals with some simplifications and assumptions which ease the DExIMA CAD implementation. Details on the structural description of the architecture, as carried out in DExIMA CAD, as discussed in Section 7.3, while Section 7.4 presents how the algorithm description facilities are employed to describe the simulation-time behaviour of the LiM array.

7.1 Overview of the Hybrid-SIMD architecture

The Hybrid-SIMD architecture [16] is a versatile general-purpose Logic-in-Memory (LiM) solution, which has been devised with the primary purpose of reducing the performance and energy communication overhead in a traditional von Neumann system. It is meant to be embedded in a processor-memory environment, so to act as an accelerator for specific algorithms.

The core of the complete LiM system is a memory array, referred to as the "SIMD array", which is partitioned in two main sections, the standard and the smart sections, which may communicate with each other via the cache and the routing layers, as depicted in Figure 7.1. In the standard section, standard-type memory cells are allocated to create a "traditional" storage facility; in the smart section, standard memory rows are interleaved with smart rows, which are the computational and processing elements of the architecture. Figure 7.2 reports the interleaved pattern which characterises the smart section of the array and the structure of a smart row, which consists of:

- an arithmetic memory row;
- a set of row interfaces;
- a I/O buffer.

To provide the required functionalities, the above elements are tightly linked with one another, implying that each smart row must integrate a well-structured local interconnection pattern.

In the smart section, an important data movement mechanism is deployed to enable the communication between different smart rows. In fact, being encompassed by two standard rows, each smart row has an up-row and a down-row, and the down-row of a smart row is the up-row of the following smart row. In this way, data can be passed on from a smart row to the next one.

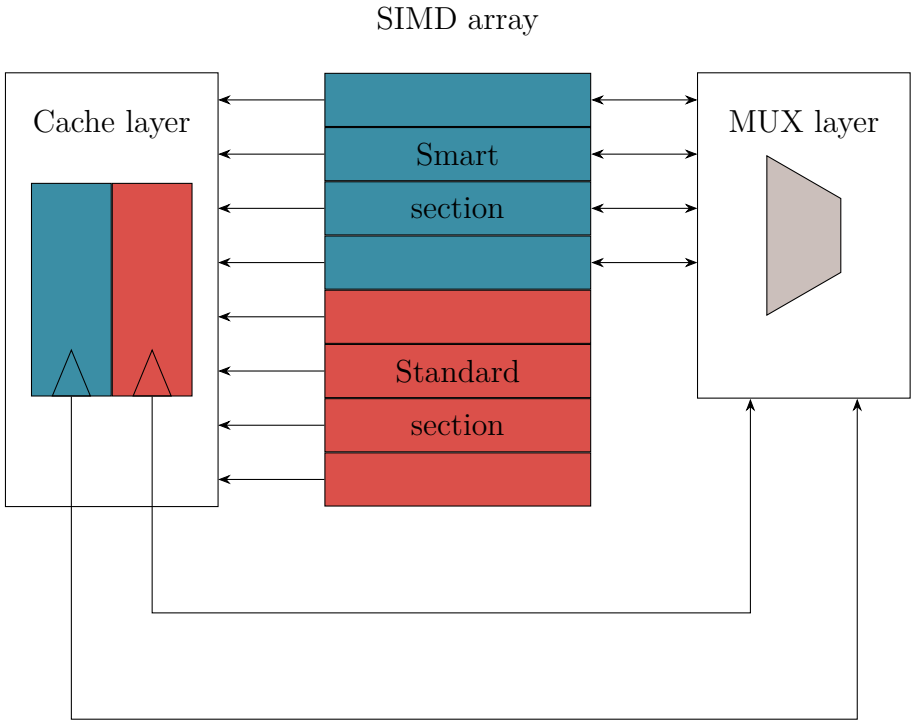


Figure 7.1: Top-level view of the structure of the memory array in the Hybrid-SIMD architecture. The SIMD array is partitioned in two sections, the standard and the smart sections.

Arithmetic memory rows are made of properly interconnected Hybrid-SIMD arithmetic/logic cells, whose structure is reported in Figure 7.3, which, besides a storage element (Mem), integrate the following components:

- a three-way multiplexer, driving the write input of the memory component;

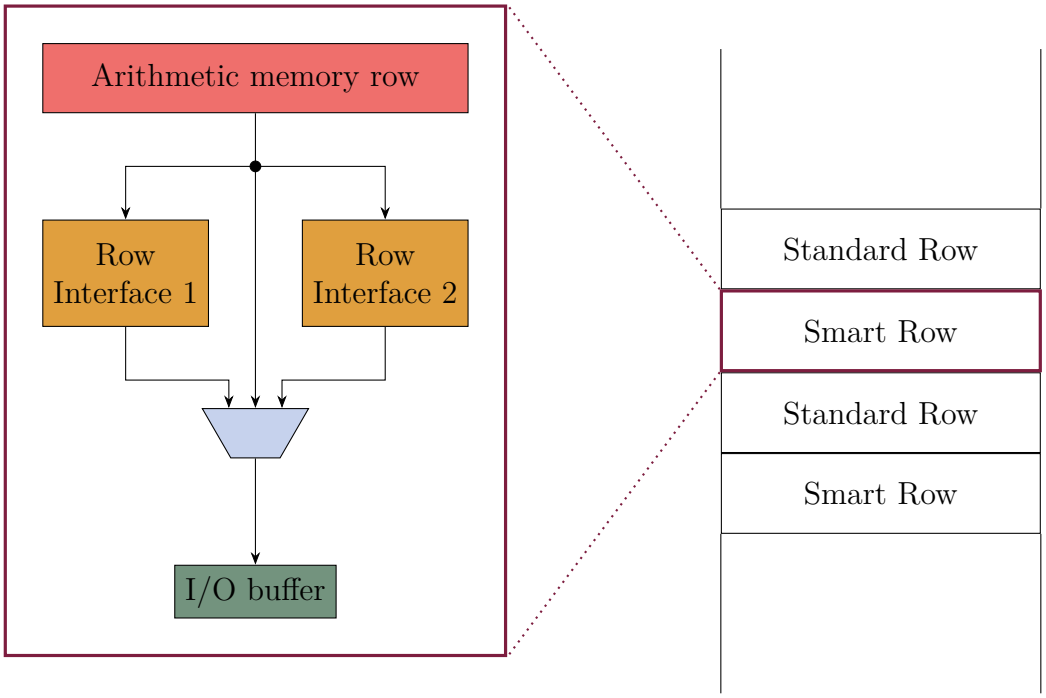


Figure 7.2: Hierarchical structure of the memory array in the Hybrid-SIMD architecture. In the smart section, standard memory rows are interleaved with smart rows, consisting of an arithmetic memory row, a set of row interfaces and a I/O buffer: in this example, two row interfaces are allocated.

- a configurable logic element, namely a Full-Adder (FA), enabling the in-row execution of simple arithmetic operations, e.g. addition and subtraction, and bit-wise logic operations;
- two XOR gates, driving the inputs of the FA and expanding the set of support bit-wise logic operations;
- a two-way and a four-way multiplexer, driving the operands used by the arithmetic/logic element.

In a smart row, row interfaces may be used for more complex tasks, e.g. multiplication, absolute value, shift and so forth, while a set of I/O buffers is allocated for temporary data storage: output buffers are typically used to save the result of a computation, deriving either from an arithmetic memory row or from a row interface, while input buffers can retain data to be used at a later stage of an algorithm.

It is important to stress that, in order to boost the modularity and the reconfigurability of the architecture, row interfaces have a predefined I/O interface, which

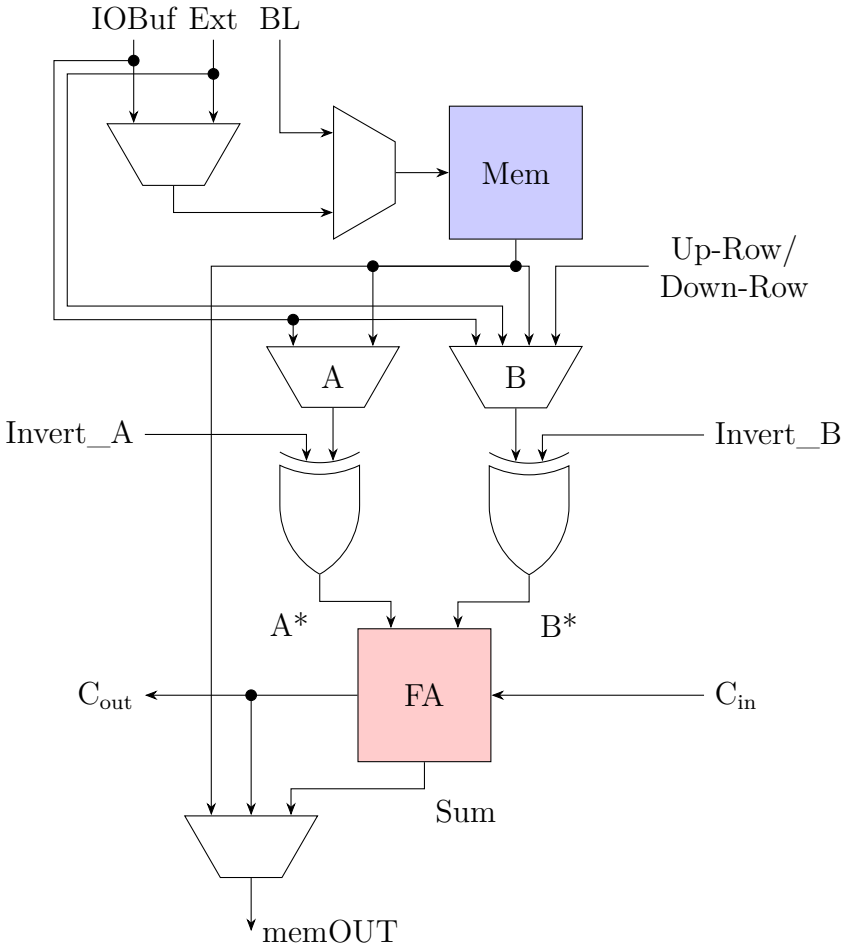


Figure 7.3: Structure of a Hybrid-SIMD arithmetic/logic cell.

is required to fetch the input operands, thus they should differ only by the hardware operator they host. An example of such I/O interface is presented in Figure 7.4.

7.2 Simplifications and assumptions

Given the details presented in Section 7.1, it is noteworthy to stress that the Hybrid-SIMD architecture represents a rather important case study, as, from the point of view of DExIMA CAD, it is quite a complex system. As a consequence, implementing this memory array in DExIMA CAD is expected to show not only the current capabilities of the tool itself, which have been boosted by the expansions presented in Chapter 3 and Chapter 4, but also some of its present limitations. As a matter of fact, attaining a complete description and simulation of the Hybrid-SIMD memory array completely within DExIMA CAD should prove the new improvements in the

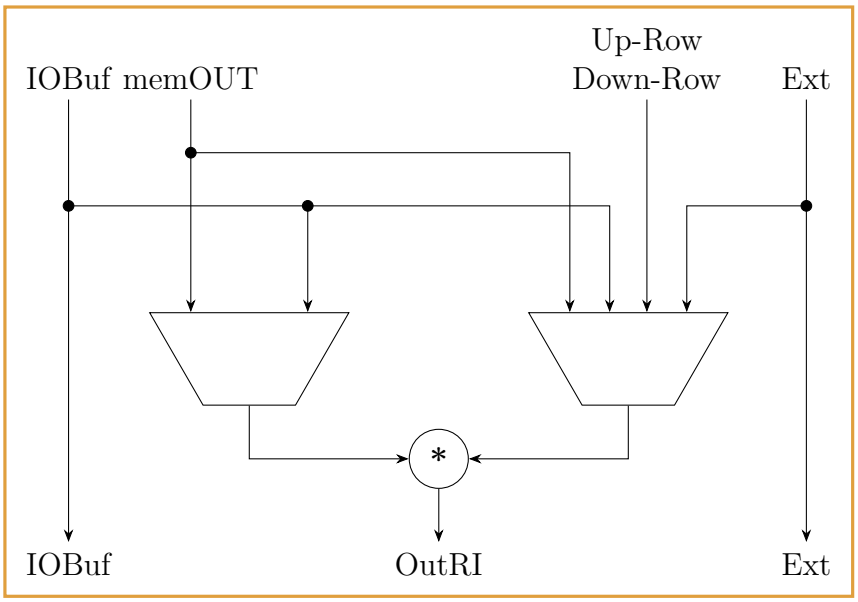


Figure 7.4: Example of a multiplier row interface in the Hybrid-SIMD architecture.

tool.

The structural description of the SIMD array in DExIMA CAD would not be possible without the intra-array interconnections feature presented in Chapter 3. Firstly, the lack of horizontal interconnections would prevent any connection between adjacent Hybrid-SIMD cells, making it impossible to integrate simple arithmetic operations, e.g. addition and subtraction, in the arithmetic memory rows. Secondly, vertical interconnections provide a communication link between the smart rows and their adjacent up-rows and down-rows, creating the data movement patterns required by the architecture. Lastly, the horizontal bus additional interconnection provides the designer with a rather simplified (and simplistic) way to emulate the "Ext" bus.

The Hybrid-SIMD architecture presents lots of operating modes, which derive from the large amount of input operands-operations-output operands combinations, thus the prior algorithm description facilities would have made the definition of the run-time behaviour of the array excruciatingly burdensome. In fact, as already mentioned in Section 4.2, the possibility of declaring simpler nano-instructions, only partially describing the behaviour of an active array element, and later combining them in the algorithm context simplifies this simulation-related task by a great amount, improving the designer experience and reducing the design time. In truth, the combination of simple nano-instructions, as described in Section 4.2, has been inspired by how nano-instructions are defined in [16].

Despite the improvements brought forth by array interconnections and by the algorithm description module, in the present state of DExIMA CAD some limitations

are found, which prevent a thorough description of the complete LiM system. As a consequence, to be able to develop the architecture completely within DExIMA CAD, some simplifications, which will be discussed in the following, are required.

A first limitation derives from the current scope of DExIMA CAD, which primarily covers the structure of the LiM array and its interaction with the micro-programmed control unit, thus making it impossible to introduce, in a rather straightforward and user-friendly manner, more complex elements in the complete LiM architecture: for instance, the present state of DExIMA CAD does not allow to introduce the cache and the routing layers presented in Figure 7.1. Thus, to carry out the structural description completely within DExIMA CAD, the aforementioned layers must be temporarily removed. To preserve the functionality of the LiM array, any routing and data dispatch mechanism is moved from a dedicated sub-system, i.e. the routing layer, external to the actual memory array, to within the array itself. This assumption leads to the following considerations:

- the Hybrid-SIMD cells, core of the arithmetic memory rows, as additional routing components are required to multiplex the data signals deriving from their related up-rows and down-rows;

- in the smart section, the standard-type memory cells should be slightly modified to account for the required routing operations.

Despite these slight differences, in this DExIMA CAD case study, the structure of all memory cells is expected to be rather close to the original implementation presented in [16].

Furthermore, as the cache layer partially compensates the complexity of the routing mechanisms, its removal may lead to a quite severe performance degradation. Nevertheless, such shortcoming is acceptable by all means, as the aim of this case study is not to developed an optimised architecture, but rather to show the new capabilities of DExIMA CAD: with further developments and expansions, the required performance may be easily recovered.

It is of utmost importance to stress that the above simplifications do not alter the functionality of the LiM array whatsoever and do not introduce any side effect which could be detrimental to the modularity and the reconfigurability of the architecture. Indeed, all required data movement patterns are quite well-handled by the DExIMA CAD description, while the functionality of the Hybrid-SIMD array is preserved and can be boosted at will by properly deploying row interfaces in the smart row structure of Figure 7.2. As a consequence, the behaviour of the LiM array may be effectively simulated by only instantiating a multiplier row interface, as the one presented in Figure 7.4, with the purpose of simulating the execution of the Matrix-Vector Multiplication (MVM) algorithm, precisely as described in [16].

7.3 Architectural description in DExIMA CAD

7.3.1 Structure of the LiM cells

The details regarding the structure of the Hybrid-SIMD architecture, presented in Section 7.1, suggest the need of at least two types of memory cells, one for the arithmetic/logic cells and one for the standard-type memory cells. Nevertheless, because of the simplifications introduced in Section 7.2, a further cell type is required to counterbalance the removal of the routing layer. To sum up, the following types of memory cells are needed:

- `cell_std`, a standard-type memory cell, deployed in the standard section of the SIMD array;
- `cell_lim`, an arithmetic/logic memory cell, the basic building block of an arithmetic memory row;
- `cell_semistd`, a semi-standard-type memory cell, used in the smart section of the SIMD array to implement the communication link between a smart row and its related up-row.

In the standard section of the SIMD array, the memory rows may be implemented by means of the standard-type memory cells presented in Figure 7.5, which only include a storage element (Mem).

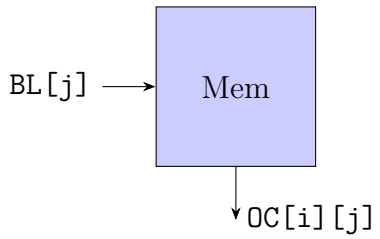


Figure 7.5: Structure of the standard-type memory cells in the DExIMA CAD description of the Hybrid-SIMD memory array.

In the smart section of the SIMD array, the up-rows and the down-rows may be implemented by means of the semi-standard-type memory cells depicted in Figure 7.6, which include a storage element (Mem), a three-way multiplexer, used to drive the write input of the memory component and the following array interconnections:

- a vertical interconnection, with parameters `Source = Row`, `SourcePin = OC`, `Direction = Previous` and `Displacement = 1`, to fetch the output of the previous arithmetic memory row;

- a vertical interconnection, with parameters `Source = IRL`, `SourcePin = IRL1`, `Direction = Previous` and `Displacement = 1`, to fetch the content of the output buffer of the previous smart row.

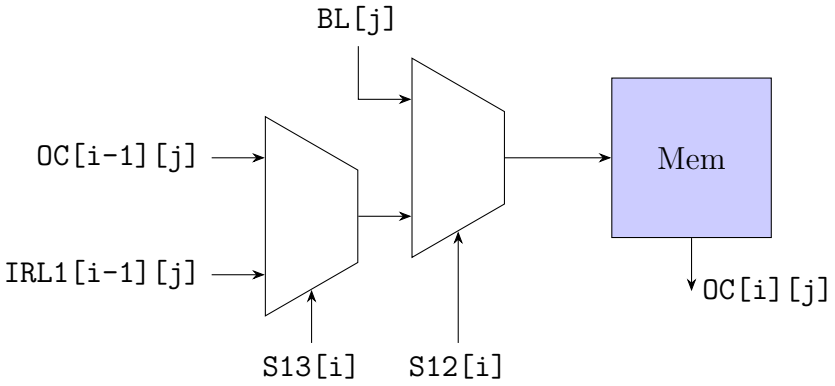


Figure 7.6: Structure of the semi-standard-type memory cells in the DExIMA CAD description of the Hybrid-SIMD memory array.

The arithmetic memory rows, as seen for instance in Figure 7.2, may be implemented by means of the arithmetic/logic memory cells reported in Figure 7.7, which, with respect to the scheme presented in Figure 7.3, include the following additional components:

- a two-way multiplexer, controlled by the LiM template signal `S3`, which feeds the second operand multiplexer with the content of either the up-row or the down-row;
- a two-way multiplexer, controlled by the LiM template signal `S7`, setting the cell either for an arithmetic or for a bit-wise logic operation.

Besides the above elements, the cells integrate the following array interconnections:

- a vertical interconnection, with parameters `Source = Row`, `SourcePin = OC`, `Direction = Previous` and `Displacement = 1`, to fetch the output of the up-row;
- a vertical interconnection, with parameters `Source = Row`, `SourcePin = OC`, `Direction = Next` and `Displacement = 1`, to fetch the output of the down-row;
- a vertical interconnection, with parameters `Source = IRL`, `SourcePin = IRL0`, `Direction = Next` and `Displacement = 1`, to fetch the content of the I/O buffers;

- a horizontal interconnection, of type LSB-to-MSB, to propagate the carry signal and thus create an in-row Ripple-Carry Adder (RCA), enabling arithmetic operations;
- a horizontal bus additional interconnection, emulating the ”Ext“ bus, as seen for instance in Figure 7.3.

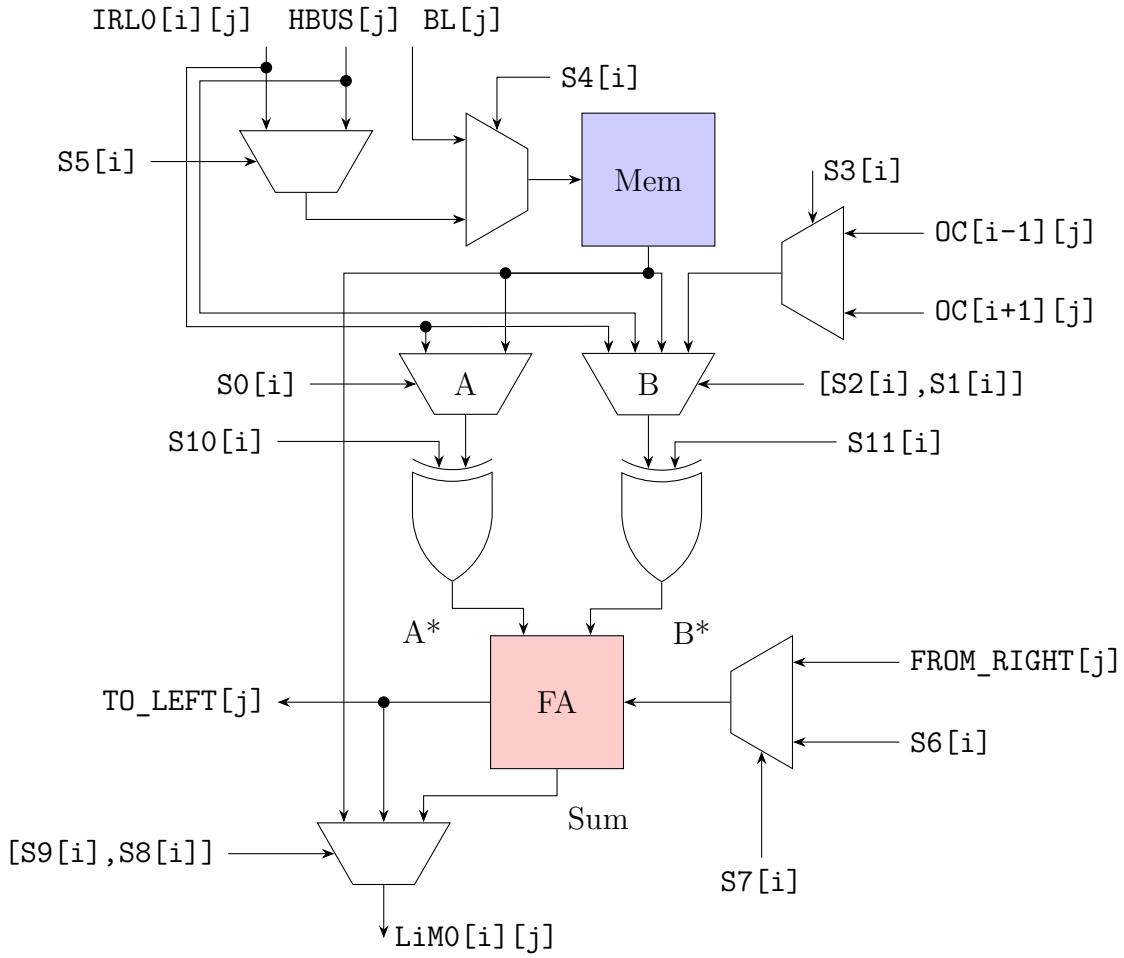


Figure 7.7: Structure of the arithmetic/logic memory cells in the DExIMA CAD description of Hybrid-SIMD memory array.

7.3.2 Structure of the IRL blocks

To complete the structure of a smart row, it is necessary to allocate, somewhere in the design, a multiplier row interface and some I/O buffers. In the current state of

DExIMA CAD, such components may be easily embedded in a IRL block, which will be associated to an arithmetic memory row in the complete architecture of the LiM array. This DExIMA CAD implementation only requires one type of IRL block, which will be referred to as `irl_block` and whose structure is reported in Figure 7.8. Besides the elements of the multiplier row interface, as presented in Figure 7.4, and two registers, acting as input and output buffers, this type of IRL block includes the following array interconnections:

- a vertical interconnection, with parameters `Source = Row`, `SourcePin = OC`, `Direction = Previous` and `Displacement = 2`, to fetch the output of the up-row;
- a vertical interconnection, with parameters `Source = Row`, `SourcePin = OC`, `Direction = Next` and `Displacement = 1`, to fetch the output of the down-row;
- a horizontal bus additional interconnection, emulating the "Ext" bus.

7.3.3 Structure and geometry of the LiM array

Once the structure of all required LiM cells and IRL blocks is determined, the number of control signals and outputs they required is known, thus a DExIMA CAD memory template can be selected, leading to the design choices reported in Table 7.1.

Template item	Value
LiM control signals	14
IRL control signals	8
LiM outputs	1
IRL outputs	2

Table 7.1: Template selection for the DExIMA CAD implementation of the Hybrid-SIMD architecture.

Having fixed the LiM array template, the DExIMA CAD schematic editor may be employed to implement all required LiM cells and IRL blocks: the results of the structural description process are reported in Appendix A.

The actual geometry of the LiM array may be inferred from the algorithm to be implemented by the architecture. As the primary concern of this DExIMA CAD implementation is to show the capabilities of the tool, to avoid unnecessary complexities in the design verification phase, it has been decided to consider a 4-row, 4-column matrix $\underline{\underline{A}}$ and, consequently, 4-row vectors $\underline{b}$ and $\underline{x}$, which are related to one another as shown in Equation 7.1.

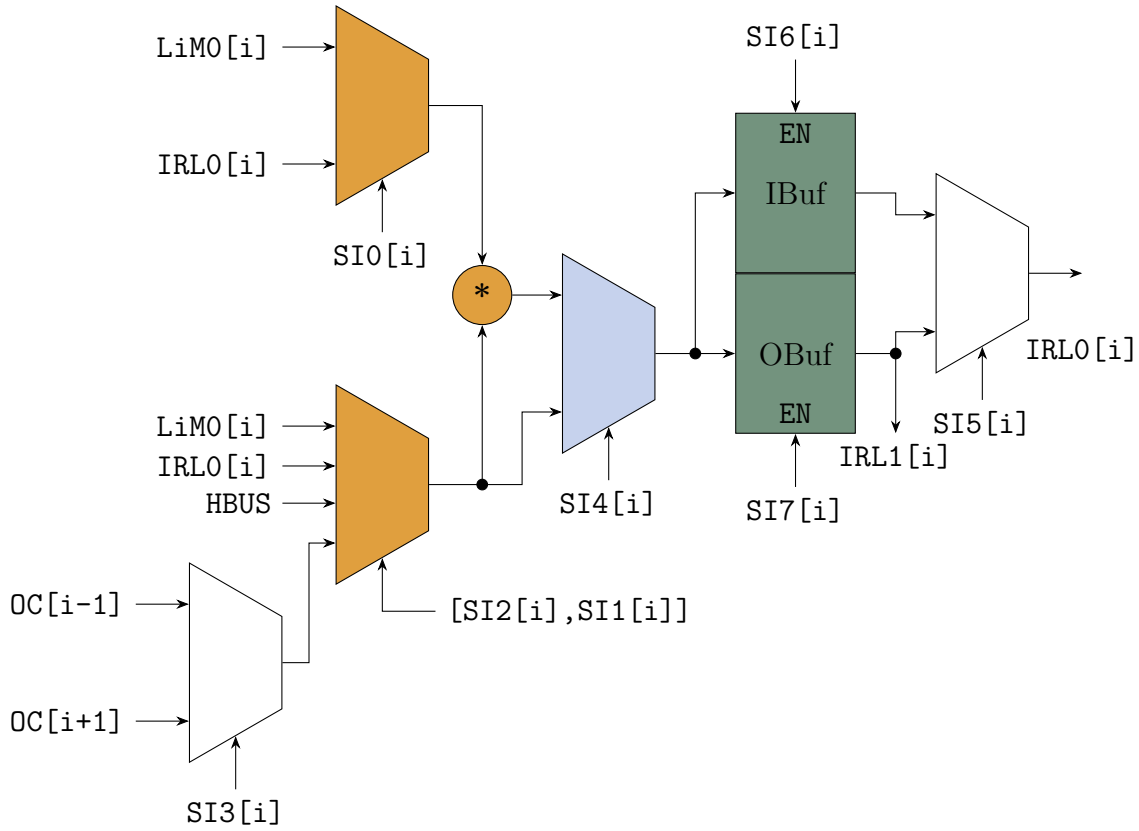


Figure 7.8: Structure of the Intra-Row Logic (IRL) blocks in the DExIMA CAD description of the Hybrid-SIMD memory array. The same colours of Figure 7.2 are used to highlight the row interface, the I/O buffers and the routing element.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{7.1}$$

Given the algorithm mapping details presented in [16] and given the structure of Equation 7.1, in order to enable a fully-parallel computation of all partial products, a total of sixteen multiplier row interfaces must be allocated in the smart section of the SIMD array, which must thus consist of sixteen smart rows and, consequently, sixteen standard rows. Hence, both the smart and the standard sections are composed of thirty-two memory rows each.

In the smart section, to mimic the interleaved pattern depicted in Figure 7.2, cells of type `cell_lim` are allocated in the odd-indexed memory rows, while cells of type `cell_semistd` are allocated in the even-indexed memory rows, with one

exception only: the first row (index 0) may employ cells of type `cell_std`, as there is no actual need to integrate further routing mechanisms. On the other hand, the standard section only contains cells of type `cell_std`.

The smart row structure is completed by allocating IRL blocks of type `irl_block` in the odd-indexed slots, thus pairing them with their related arithmetic memory rows.

The above considerations eventually lead to the scheme of Figure 7.9, which reports the structure of the Hybrid-SIMD memory array, as described in DExIMA CAD, for the first eight memory rows, i.e. for the first four smart rows in the smart section.

7.4 Algorithm description in DExIMA CAD

When the structure of the complete memory array is defined, the new functionalities of the algorithm description module, as presented in Chapter 4, may be employed to specify its simulation-time behaviour. As the Hybrid-SIMD architecture is conceived to be a general-purpose co-processor, this DExIMA CAD implementation is expected to support a set of general-purpose nano-instructions, which will then be combined to produce the required Matrix-Vector Multiplication (MVM) algorithm.

Nano-instructions Because of its complexity, the behaviour of a Hybrid-SIMD array is described by combinations of simpler nano-instructions, which control specific sections of each LiM cells and IRL blocks.

The following set of nano-instructions may be applied to any kind of memory row in the Hybrid-SIMD array.

- `NOP`: Do not perform any operation.
- `LOAD`: Prepare a write operation to a selected memory row.

To deal with the up-row/down-row routing mechanism, the behaviour of the semi-standard type memory cells depicted in Figure 7.6 is defined by means of the following set of nano-instructions.

- `UDRow_BL`: Select the bitline as the write input of an enabled down-row.
- `UDRow_Smart`: Select the smart row as the write input of an enabled down-row.
- `UDRow_OBuf`: Select the output buffer as the write input of an enabled down-row.

As for the arithmetic/logic Hybrid-SIMD cells of Figure 7.7, the following nano-instructions are introduced to control the input multiplexers, feeding the write input of the storage element, and the output multiplexer, driving the template output signal `LiMO`.

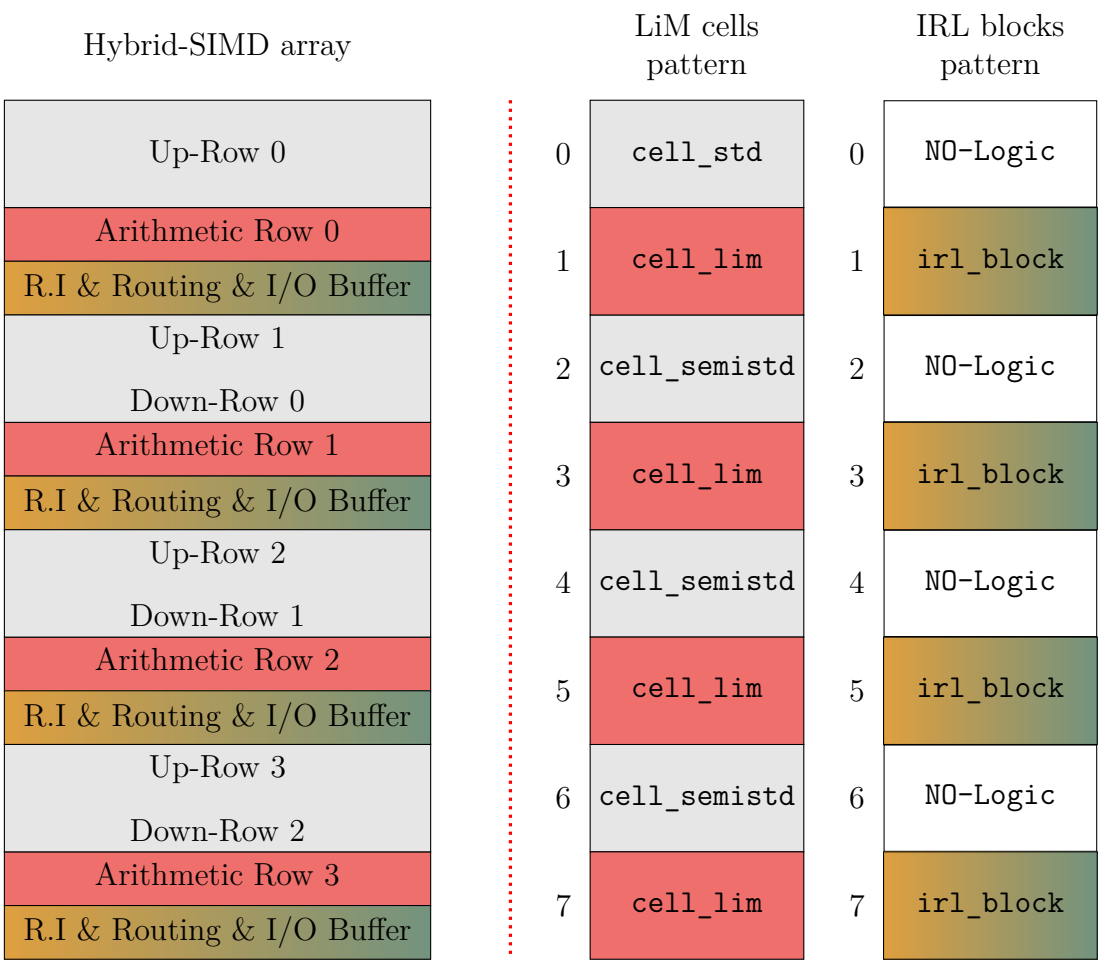


Figure 7.9: DExIMA CAD description of the complete Hybrid-SIMD memory array. The leftmost part of the picture reports the intended structure of the Hybrid-SIMD array, with same colours of Figure 7.2 to highlight the different elements in a smart row, i.e. arithmetic memory rows, row interfaces (R. I), routing elements and I/O buffers. The rightmost part of the picture depicts the LiM cells and IRL blocks patterns, as used in DExIMA CAD. For representation purposes, only the first eight memory rows are shown.

- `Input_BL`: Select the bitline as the write input of an arithmetic memory row.
- `Input_IOBuf`: Select the I/O buffer as the write input of an arithmetic memory row.
- `Input_Ext`: Select the `Ext` line as the write input of an arithmetic memory row.
- `Output_OC`: Select the memory cell as the `LiMO` output of an arithmetic memory

row.

- `Output_S`: Select the sum signal as the `LiMO` output of an arithmetic memory row.
- `Output_C`: Select the carry signal as the `LiMO` output of an arithmetic memory row.

Besides the I/O sections, the following set of nano-instructions controls the input operands of the arithmetic rows and of the IRL blocks.

- `OPR_A_memOUT`, `OPR_A_IRL_memOUT`: Select the output of an arithmetic memory row as the first operand.
- `OPR_A_IOBuf`, `OPR_A_IRL_IOBuf`: Select the I/O buffer as the first operand.
- `OPR_B_memOUT`, `OPR_B_IRL_memOUT`: Select the output of an arithmetic memory row as the second operand.
- `OPR_B_IOBuf`, `OPR_B_IRL_IOBuf`: Select the I/O buffer as the second operand.
- `OPR_B_Ext`, `OPR_B_IRL_Ext`: Select the `Ext` line as the second operand.
- `OPR_B_URow`, `OPR_B_IRL_URow`: Select the up-row as the second operand.
- `OPR_B_DRow`, `OPR_B_IRL_DRow`: Select the down-row as the second operand.

The I/O buffers in the IRL blocks may be controlled by the following set of nano-instructions.

- `IOBuf_IBuf`: Select the input buffer.
- `IOBuf_OBuf`: Select the output buffer.
- `Enable_IBuf`: Enable a write operation to the input buffer.
- `Enable_OBuf`: Enable a write operation to the output buffer.
- `Mux_ArithRow`: Select the output of the four-way multiplexer as the write input of the I/O buffers.
- `Mux_Operator`: Select the output of the multiplier as the write input of the I/O buffers.

With all above nano-instructions, the simulation-time behaviour of the Hybrid-SIMD array may be easily controlled in any algorithm step.

Simulation dashboard The implementation of the MVM algorithm in a Hybrid-SIMD system is made possible by the following steps.

Prior to the beginning of the algorithm, the smart section of the SIMD array must be initialized, thus matrix elements a_{ij} are stored in the arithmetic memory rows, while vector components x_i are stored in the up-rows, ensuring a proper data replication so to parallelize the computation of partial products. In quantitative terms, as a row-major order scheme is employed for matrix $\underline{\underline{A}}$, its elements will be stored in the memory rows with indexes $1, 3, 5, ..., 31$, i.e. in the arithmetic memory rows; as for the vector $\underline{x}$:

- the first component x_0 is stored in the memory rows with indexes 0, 8, 16 and 24;
- the second component x_1 is stored in the memory rows with indexes 2, 10, 18 and 26;
- the third component x_2 is stored in the memory rows with indexes 4, 12, 20 and 28;
- the fourth component x_3 is stored in the memory rows with indexes 6, 14, 22 and 30.

The actual processing, which may start when the smart section has been properly initialized, consists of a sequence of steps which are easily modelled with the facilities presented in Section 4.2. In the following, the required steps are presented, alongside with the scalar control and the activation pattern.

- Elements x_i are copied from the up-rows to the input buffers.
  - Scalar control
    `Enable_IBuf`
    `Mux_ArithRow`
    `OPR_B_IRL_URow`
  - Activation pattern
    `ODD:0:31`
- Partial products are computed and stored in the output buffers.
  - Scalar control
    `Enable_OBuf`
    `Mux_Operator`
    `OPR_A_IRL_memOUT`
    `OUTPUT_OC`
    `OPR_B_IRL_IOBuf`
    `IOBuf_IBuf`

  - Activation pattern
    `ODD:0:31`

- Partial products are copied from the output buffers to the arithmetic rows.
  - Scalar control
    `LOAD`
    `INPUT_IOBuf`
    `IOBuf_OBuf`
  - Activation pattern
    `ODD:0:31`

- Partial products are copied from the output buffers to the down-rows.
  - Scalar control
    `LOAD`
    `UDRow_Smart`
  - Activation pattern
    `EVEN:2:31`

- Partial products are accumulated along the memory array.
  - Arithmetic rows and up-rows are summed, the result is stored in the output buffers.
    * Scalar control
      `Enable_OBuf`
      `Mux_ArithRow`
      `OPR_B_IRL_memOUT`
      `OPR_A_memOUT`
      `OPR_B_URow`
      `OUTPUT_S`
      `SUM`
    * Activation pattern
      First sum `CUSTOM:3:11:19:27`
      Second sum `CUSTOM:5:13:21:29`
      Third sum `CUSTOM:7:15:23:31`
  - The content of the output buffers is transferred to the down-rows.
    * Scalar control
      `LOAD`
      `UDRow_OBuf`

∗ Activation pattern
  First sum `CUSTOM:4:12:20:28`
  Second sum `CUSTOM:6:14:22:30`
  Third sum `CUSTOM:8:16:24:32`

Chapter 8

SHA-1 in DExIMA CAD

The purpose of this chapter is to describe the DExIMA CAD implementation of a LiM architecture which supports the execution of the SHA-1 algorithm. The primary concern of this description is to prove that DExIMA CAD may be used to effectively support the design flow of a *ex novo* LiM system, thanks to its new architectural exploration capabilities and to its simulation and estimation facilities.

Section 8.1 presents the main details of the SHA-1 algorithm, which are effectively used in Section 8.2 to support the architectural derivation process. All design choices are then transferred to the DExIMA CAD implementation, which is described in Section 8.3.

8.1 Description of the SHA-1 algorithm

The SHA-1 algorithm [6] operates on a 512-bit input message to produce a 160-bit hash value. The elaboration progressively updates five 32-bit words H_0, H_1, H_2, H_3 and H_4, which will then form the message digest, i.e. the output hash value, according to Equation 8.1.

$$\text{Message digest} = [H_0, H_1, H_2, H_3, H_4] \tag{8.1}$$

Prior to the elaboration phase, the 512-bit input block is initially organised as a set of sixteen 32-bit words $W[i], i = 0, 1, ..., 15$. Furthermore, the 32-bit message digest components H_0, H_1, H_2, H_3 and H_4 are initialised as indicated in the following:

- $H_0 \leftarrow$ `0x67452301`;
- $H_1 \leftarrow$ `0xEFCDAB89`;
- $H_2 \leftarrow$ `0x98BADCFE`;

- $H_3 \leftarrow$ `0x10325476`;
- $H_4 \leftarrow$ `0xC3D2E1F0`.

The computational section of the algorithm consists of the sequence of operations presented in Algorithm 1, which will be detailed in the following.

Algorithm 1 Pseudo-code of the SHA-1 algorithm.

```
1:  i ← 16
2:  while i < 80 do
3:      W[i] = rotateLeft(W[i − 3] ⊕ W[i − 8] ⊕ W[i − 14] ⊕ W[i − 16], 1)
4:      i ← i + 1
5:  end while
6:
7:  A = H0, B = H1, C = H2, D = H3, E = H4
8:
9:  i ← 0
10: while i < 80 do
11:     TEMP ← rotateLeft(A, 5) + roundFunction(B, C, D, i) + E + K(i) + W[i]
12:     E ← D
13:     D ← C
14:     C ← rotateLeft(B, 30)
15:     B ← A
16:     A ← TEMP
17:     i ← i + 1
18: end while
19:
20: H0 ← H0 + A, H1 ← H1 + B, H2 ← H2 + C, H3 ← H3 + D, H4 ← H4 + E
```

Initially, the algorithm generates an additional set of 64 32-bit words $W[i], i = 16, 17, ..., 79$, which are derived from the input message by means of simple bit-wise XOR and circular left shift operations (lines 1 to 5 in Algorithm 1). As a consequence, the subsequent elaboration phases will operate on a set of 80 32-bit words $W[i], i = 0, 1, ..., 79$.

When the 80-element array is successfully initialised, the current content of the 32-bit words H_0, H_1, H_2, H_3 and H_4 is transferred to the temporary buffers A, B, C, D and E (line 7 in Algorithm 1). At this point, the content of the temporary buffers is progressively updated for a total of 80 computational rounds (lines 10 to 18 in Algorithm 1). Eventually, the temporary buffers are used to update the message digest (line 20 in Algorithm 1).

During the main 80 computational rounds, a temporary buffer TEMP is used to accumulate several input quantities. Besides the currently processed 32-bit word

$W[i]$, the contributions to this accumulation derive from a round-dependent constant $K(i)$ and from the temporary buffers A, B, C, D and E. More specifically, E is accumulated as is, A undergoes a prior 5-position circular left shift, while B, C and D are applied a round-dependent function roundFunction(B, C, D, i). The round-dependent quantities are detailed in Table 8.1.

Round	$K(i)$	roundFunction(B, C, D, i)
$0 \leq i \leq 19$	`0x5A827999`	$(\mathrm{B} \cdot \mathrm{C}) \oplus (\bar{\mathrm{B}} \cdot \mathrm{D})$
$20 \leq i \leq 39$	`0x6ED9EBA1`	$\mathrm{B} \oplus \mathrm{C} \oplus \mathrm{D}$
$40 \leq i \leq 59$	`0x8F1BBCDC`	$(\mathrm{B} \cdot \mathrm{C}) \oplus (\mathrm{B} \cdot \mathrm{D}) \oplus (\mathrm{C} \cdot \mathrm{D})$
$60 \leq i \leq 79$	`0xCA62C1D6`	$\mathrm{B} \oplus \mathrm{C} \oplus \mathrm{D}$

Table 8.1: Round-dependent quantities in the SHA-1 algorithm.

8.2 Derivation of the LiM architecture

The details of the SHA-1 algorithm, thoroughly presented in Section 8.1, may be used to support the derivation of a LiM architecture that strives to reduce as much as possible the execution time by allocating a proper number of hardware resources.

Given the overall structure of Algorithm 1, the to-be-designed LiM array should consist of 32-bit columns and at least 80 rows, one for each 32-bit word $W[i], i = 0, 1, ..., 79$. As the first sixteen words are not actually modified during the execution of algorithm, it is expected that the first sixteen rows in the LiM array are standard memory rows, i.e. with storage capabilities only. On the other hand, the remaining sixty-four rows are supposed to be endowed with computational capabilities, as line 3 in Algorithm 1 clearly shows.

Five modified memory rows must be dedicated to the message digest components H_0, H_1, H_2, H_3 and H_4. To support the accumulation operation (line 20 in Algorithm 1), the memory cells in these rows are expected to integrate some processing capabilities. For these rows, the IRL blocks allocate a register to store the corresponding temporary buffer (A for H_0, B for H_1, C for H_2, D for H_3 and E for H_4).

Lastly, four standard memory rows are required by the LiM array to store the found round constants K_{0-19}, K_{20-39}, K_{40-59} and K_{60-79}. The resulting number of rows would be 89, which is not a power-of-2 quantity, hence the LiM array is extended to 128 rows. The geometry of the LiM array is thus 128 rows by 32 columns, leading to an array size of 512 bytes. The resulting structure and its memory map is summarised in Figure 8.1.

For each of the 64 words $W[i], i = 16, 17, ..., 79$, the corresponding memory cells allocate three XOR gates and four vertical interconnections to combine the

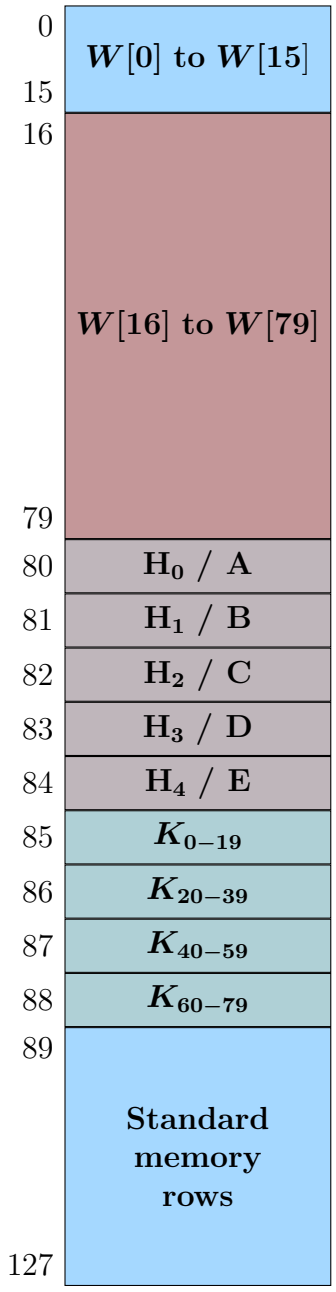


Figure 8.1: LiM SHA-1 architecture: structure of the LiM array.

quantities $W[i-3]$, $W[i-8]$, $W[i-14]$ and $W[i-16]$. The result of this combination is transferred to the associated IRL block, which computes the circular left shift and feed its output back to its above memory row. For rows 16 to 79, the structure of the LiM cells and of the IRL blocks is reported in Figure 8.2 and in Figure 8.3, respectively.

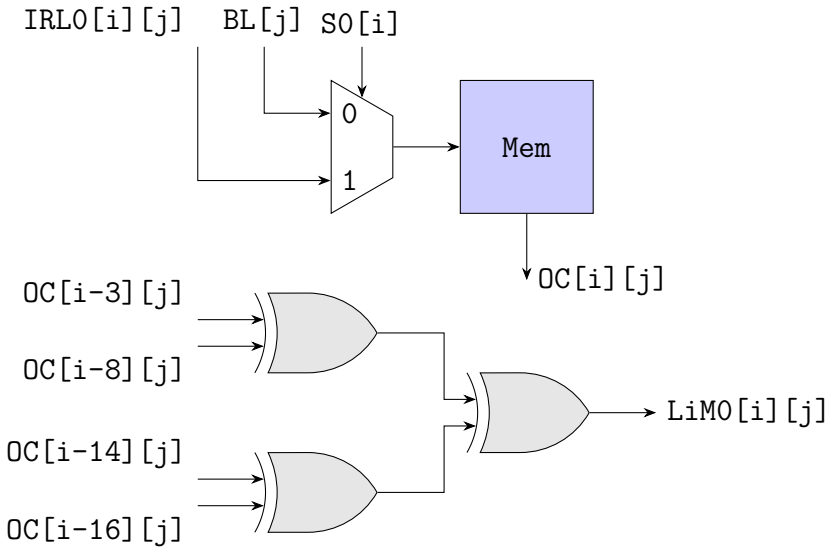


Figure 8.2: LiM SHA-1 architecture: LiM cells for rows 16 to 79.

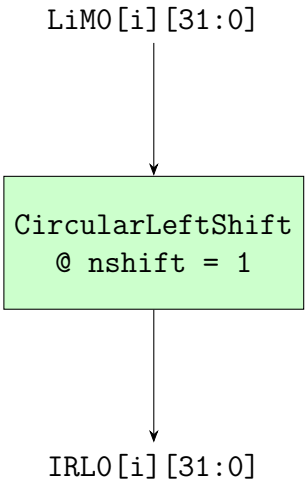


Figure 8.3: LiM SHA-1 architecture: IRL blocks for rows 16 to 79.

The content of the temporary buffers A, B, C, D and E is made available to the LiM array by means of the `IRL0` template output. It follows that, to update the message digest components (line 20 in Algorithm 1), their corresponding memory cells must integrate:

- a Full-Adder (FA) and a LSB-toMSB horizontal interconnection, creating an in-row adder;
- a vertical interconnection to fetch the related temporary buffer.

The resulting structure of this cells is reported in Figure 8.4.

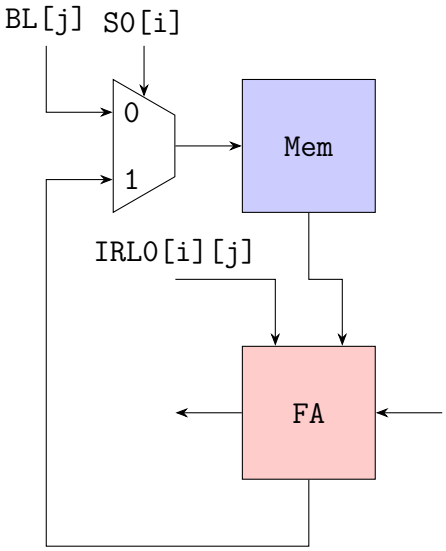


Figure 8.4: LiM SHA-1 architecture: LiM cells for rows 80 to 84.

Lines 12 to 15 in Algorithm 1 create a data movement pattern which shifts the content of the B, C, D and E temporary buffers. In DExIMA CAD, this pattern is readily implemented by allocating a proper vertical interconnection in the associated IRL blocks. The content of temporary buffer B is routed to a circular left shift component, whose output is then forwarded to temporary buffer C by means of the `IRL1` template output and a vertical interconnection. The resulting structure of these IRL blocks is reported in Figure 8.5.

Figure 8.6 reports the structure of IRL block which implements lines 11 and 16 in Algorithm 1. An accumulation unit, i.e. an adder and a register, is deployed to progressively update the content of the TEMP buffer. Given the data layout enforced by the LiM array structure presented in Figure 8.1, this IRL block must be fed by the following set of vertical interconnections:

- `OC[85]`, `OC[86]`, `OC[87]` and `OC[88]` are the input interconnection pins of Row-to-IRL interconnections, fetching the round constants K_{0-19}, K_{20-39}, K_{40-59} and K_{60-79}, respectively;
- `IRL0[81]`, `IRL0[82]`, `IRL0[83]` and `IRL0[84]` are the input interconnection pins of IRL-to-IRL interconnections, retrieving the content of temporary buffers B, C, D and E, respectively.

In each computational round i, the required round constant is initially loaded in the TEMP buffer. Then, the 32-bit word $W[i]$ is routed to the accumulation unit by means of an Out-Of-Memory component, namely a memory rows multiplexer,

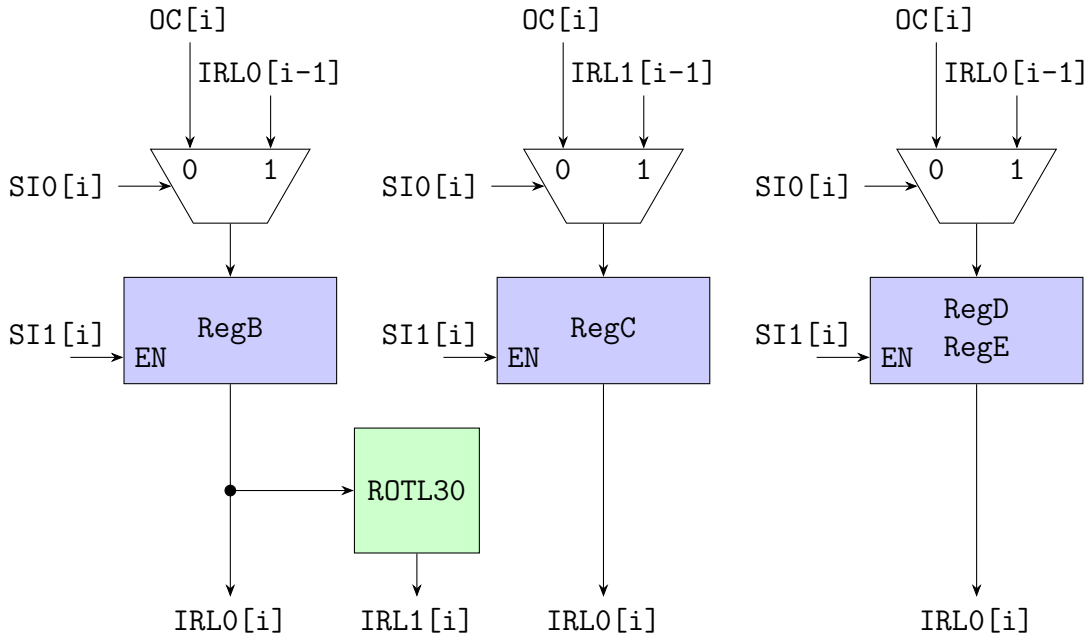


Figure 8.5: LiM SHA-1 architecture: IRL blocks for rows 81 to 84.

whose output is paired to the horizontal bus according to the scheme depicted in Figure 8.7. In the subsequent computational steps, the TEMP buffer is updated by accumulating properly transformed versions of the temporary buffers A, B, C, D and E:

- E is summed as is;
- B, C and D are first combined in the round function unit, which consists of the three blocks `SHA1_f1`, `SHA1_f2` and `SHA1_f3`;
- A undergoes a prior 30-position left shift, which is represented by the `ROTL5` block in Figure 8.6.

The three blocks `SHA1_f1`, `SHA1_f2` and `SHA1_f3` implement Equation 8.2, Equation 8.3 and Equation 8.4.

$$F = (B \cdot C) \oplus (\bar{B} \cdot D) \tag{8.2}$$

$$F = B \oplus C \oplus D \tag{8.3}$$

$$F = (B \cdot C) \oplus (B \cdot D) \oplus (C \cdot D) \tag{8.4}$$

When the accumulation is over, the content of the TEMP buffer is eventually transferred to temporary buffer A, completing computational round i.

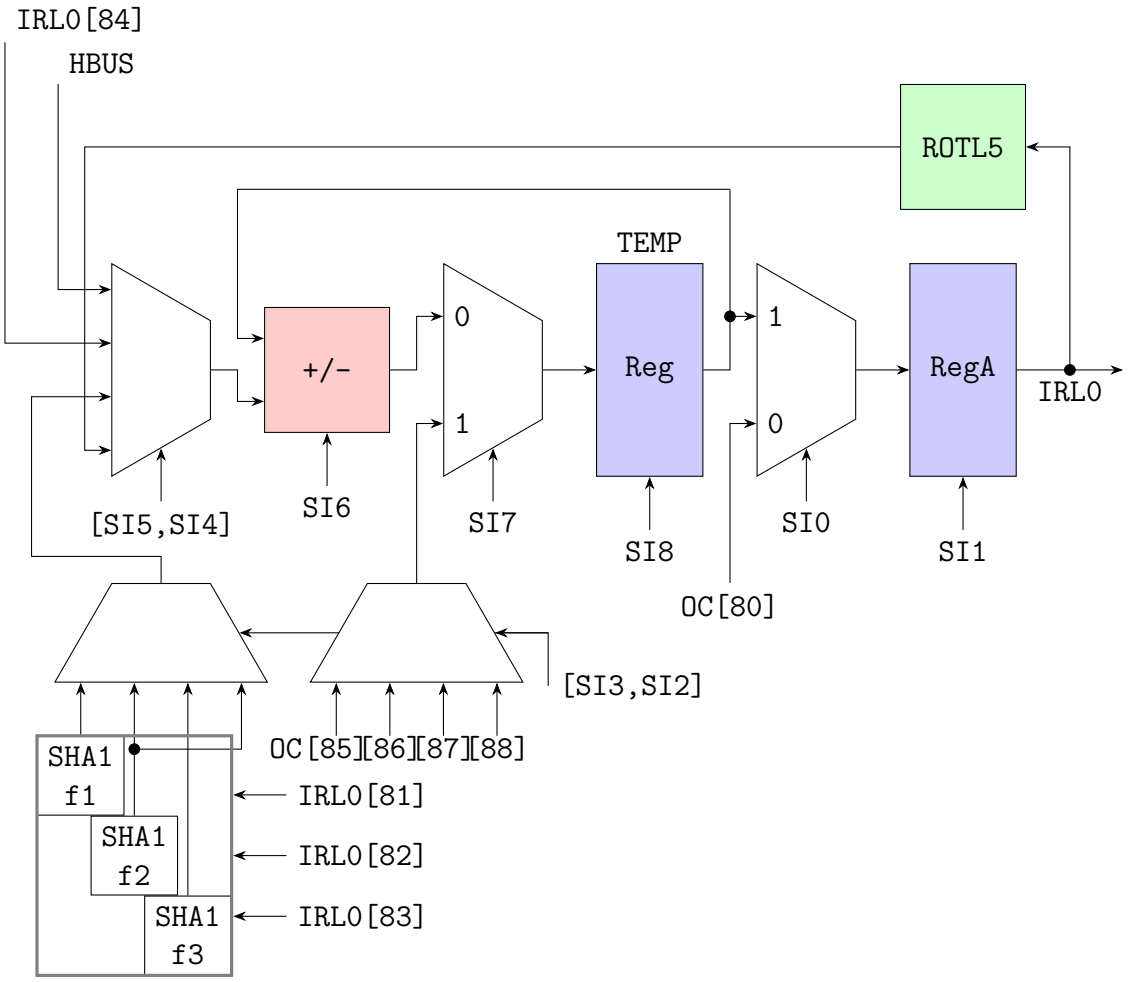


Figure 8.6: LiM SHA-1 architecture: IRL block for row 80.

8.3 Architectural description in DExIMA CAD

All considerations presented in Section 8.2 fix the structure of all elements of the LiM array and their interconnections. The description capabilities of DExIMA CAD can be effectively used to implement the required LiM cells and IRL blocks and, in turn, the structure of the complete LiM array. The results of the structural description process are presented in Appendix B.

Uppermost architectural level Before any other task can be carried out within DExIMA CAD, a definition of the uppermost architectural level must be provided. In this implementation, only one LiM array is required to implement the algorithm: to support this unique array instance, the allocation of a single micro-programmed

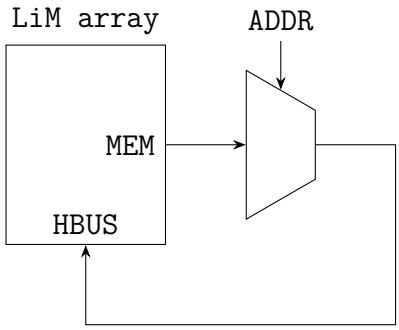


Figure 8.7: LiM SHA-1 architecture: use of an Out-Of-Memory memory rows multiplexer to route $W[i]$ to the accumulation unit.

control unit and a single micro-ROM may suffice. Furthermore, a wordline multiplexer must be allocated to manage the internal control provided by the micro-programmed control unit and the external control provided by the UVM testbench. Moreover, a memory rows multiplexer must be instantiated in the uppermost level, so that the required 32-bit word $W[i]$ can be fetched in computational round i and dispatched to the accumulation unit. Ultimately, the uppermost architectural level is reported in Figure 8.8.

Nano-instructions When the structure of the uppermost architectural level is defined, the nano-instructions required by the only LiM array type may be provided.

The following set of nano-instructions may successfully describe the behaviour of all LiM cells in the design, regardless of their actual type.

- `NOP`: Deactivate the LiM array.
- `STORE`: Prepare a write operation in the selected memory row.
- `InputMem_BL`: Select the bitline as the write input of a selected memory row.
- `InputMem_Other`: Select the output of the very next IRL block as the write input of a selected memory row.

Most of the processing in this DExIMA CAD implementation takes place in the A, B, C, D and E buffers, which are allocated in IRL blocks 80, 81, 82, 83 and 84, respectively. To control this section of the LiM array, the following set of nano-instructions is defined.

- `Buffers_Enable`: Write to the selected temporary buffer.
- `Buffers_InputTransfer`: Select the message digest component as the write input of the selected temporary buffer.

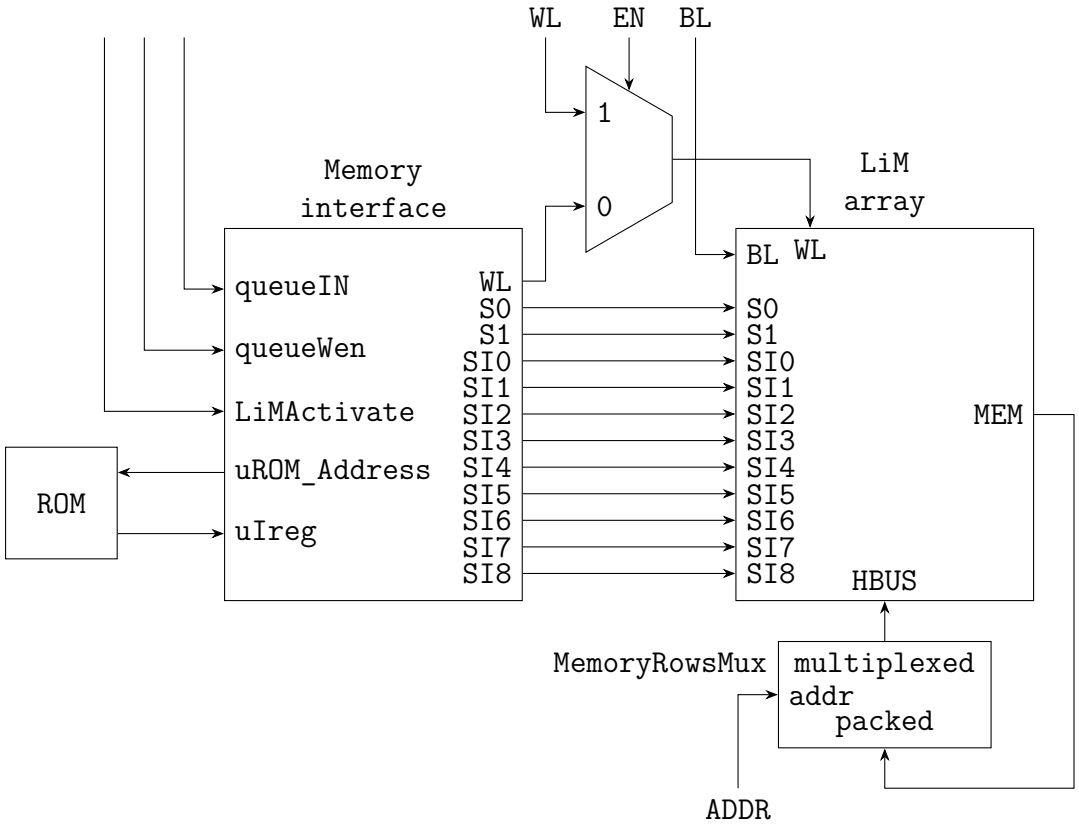


Figure 8.8: LiM SHA-1 architecture: uppermost architectural level. Although not explicitly indicated, the LiM array and the memory interface are driven by the same clock and reset signals.

- `Block_00`: Select the round constant and the round function for computational rounds $0 \leq i \leq 19$.
- `Block_01`: Select the round constant and the round function for computational rounds $20 \leq i \leq 39$.
- `Block_10`: Select the round constant and the round function for computational rounds $40 \leq i \leq 59$.
- `Block_11`: Select the round constant and the round function for computational rounds $60 \leq i \leq 79$.
- `TEMP_EN`: Write to the TEMP buffer.
- `TEMP_LOAD`: Select the round constant as the write input of the TEMP buffer.
- `TEMP_ACC`: Select the output of the accumulation adder as the write input of the TEMP buffer.

- `OPR1`: Accumulate $W[i]$.
- `OPR2`: Accumulate E.
- `OPR3`: Accumulate roundFunction(B, C, D, i).
- `OPR4`: Accumulate rotateLeft(A, 5).

It should be noted that, in the above list, the first two nano-instructions refer to all the previously mentioned IRL blocks, while all others are specific to the IRL block for buffer A, in which the TEMP buffer is progressively updated.

Simulation dashboard The presented set of nano-instructions is sufficient to completely specify the behaviour of the LiM array in any computational cycle. The simulation dashboard may be thus used to indicate, for each algorithm step, both the internal control and the values of the active pins, which are `BL`, `EN`, `WL` and `ADDR`. In the following, unless not otherwise specified, all active pins are kept to their default value (i.e. 0).

Lines 1 to 5 in Algorithm 1 correspond to the following set of algorithm steps in the simulation dashboard.

```
STORE+InputMem_Other,RANGE:16:18
STORE+InputMem_Other,RANGE:19:21
STORE+InputMem_Other,RANGE:22:24
STORE+InputMem_Other,RANGE:25:27
STORE+InputMem_Other,RANGE:28:30
STORE+InputMem_Other,RANGE:31:33
STORE+InputMem_Other,RANGE:34:36
STORE+InputMem_Other,RANGE:37:39
STORE+InputMem_Other,RANGE:40:42
STORE+InputMem_Other,RANGE:43:45
STORE+InputMem_Other,RANGE:46:48
STORE+InputMem_Other,RANGE:49:51
STORE+InputMem_Other,RANGE:52:54
STORE+InputMem_Other,RANGE:55:57
STORE+InputMem_Other,RANGE:58:60
STORE+InputMem_Other,RANGE:61:63
STORE+InputMem_Other,RANGE:64:66
STORE+InputMem_Other,RANGE:67:69
STORE+InputMem_Other,RANGE:70:72
STORE+InputMem_Other,RANGE:73:75
STORE+InputMem_Other,RANGE:76:78
STORE+InputMem_Other,SINGLE:79
```

Line 7 in Algorithm 1 corresponds to the following algorithm step.

```
Buffers_Enable+Buffers_InputTransfer,RANGE:80:84
```

Because of its naturally sequential nature, the TEMP accumulation operation (line 11 in Algorithm 1) is actually implemented by the a set of algorithm steps. For the first twenty computational round (i.e. $0 \leq i \leq 19$), the following description is found in the simulation dashboard.

```
Block_00+TEMP_EN+TEMP_LOAD,SINGLE:80
TEMP_EN+TEMP_ACC+OPR1,SINGLE:80
TEMP_EN+TEMP_ACC+OPR2,SINGLE:80
Block_00+TEMP_EN+TEMP_ACC+OPR3,SINGLE:80
TEMP_EN+TEMP_ACC+OPR4,SINGLE:80
Buffers_Enable+Buffers_InputUpdate,RANGE:80:84
```

Evidently, the `Block_00` nano-instruction must be changed depending on the 20-round computational block, i.e. `Block_01` for $20 \leq i \leq 39$, `Block_10` for $40 \leq i \leq 59$ and `Block_11` for $60 \leq i \leq 79$. Moreover, it is important to point out that in the algorithm step which accumulates the 32-bit word $W[i]$, i.e. `TEMP_EN+TEMP_ACC+OPR1,SINGLE:80`, a proper value of the active `ADDR` must be set, so that the memory rows multiplexer may be properly driven to fetch the desired memory row.

Lastly, the update of all message digest components, i.e. line 20 in Algorithm 1, may be implemented by means of the following algorithm step.

```
STORE+InputMem_Other,RANGE:80:84
```

Chapter 9

AES-128 in DExIMA CAD

This chapter describes the DExIMA CAD implementation of a Logic-in-Memory (LiM) architecture supporting the execution of the Advanced Encryption Standard (AES) algorithm, with the primary concern of showing the new architectural exploration capabilities of DExIMA CAD.

Section 9.1 presents the main motivations behind the choice of such algorithm, the details of which are thoroughly discussed in Section 9.2. The Algorithm-to-Architecture mapping phase is reported in Section 9.3, while Section 9.4 deals with the architectural description process in DExIMA CAD.

9.1 Introduction and motivations

The Advanced Encryption Standard (AES) algorithm [4] is a widely encryption algorithm. While its details are thoroughly presented in Section 9.3, it is important to point out that, besides a cipher, a full AES encryption requires a set of keys, customarily referred to as "round keys", which are derived from an input key by means of the AES key schedule algorithm. As a consequence, a hardware implementation of the AES encryption requires to manage two algorithms at once, namely the cipher and the key schedule.

From the point of view of an in-memory implementation, the AES algorithm is expected to be rather promising, as both the cipher and the key schedule algorithms consist of simple operations, which can be efficiently carried out with basic logic gates, look-up tables and proper interconnection patterns. As a matter of fact, some examples of in-memory AES implementations can be found in the scientific literature: for instance, in [7] the authors presented a modified memory architecture which supports the execution of the AES cipher.

As regards the parallelisation possibilities, it should be stressed that the AES cipher algorithm is particularly devoid of any data dependency, implying that all its steps may be executed in parallel, provided that the required hardware resources

are allocated, leading to a potential speedup in the execution time. On the other hand, the AES key schedule shows some data dependencies, which partially reduce the benefits of parallelisation and unfolding: this is not regarded as a limitation, but rather as a further proof of the flexibility of DExIMA CAD, as it goes to show that a tool specifically conceived with the purpose of parallelising in-memory solutions can implement serial algorithms as well.

The ultimate target of this DExIMA CAD implementation is a LiM architecture supporting a parallel in-memory execution of the AES encryption, derived from the details presented in [4], with some assumptions driving the design flow:

1. the Algorithm-to-Architecture mapping phase is expected to produce an ASIC-like structure, meaning that no prior general-purpose LiM system is modified to support the execution of the required algorithm;

2. the memory array should be as compact as possible, hence a limited memory usage should be enforced in the derivation of its structure.

Several reasons lie behind the above assumptions. Firstly, the ASIC approach should better highlight how DExIMA CAD may be used to describe an *ex novo* LiM architecture. Secondly, limiting memory usage prevents a full expansion of the input key in favour of a on-the-fly computation of the required round key, and is thus foreseen to limit the complexity of the interconnection patterns within the LiM array.

It would be beneficial for the purposes of this work to address several solutions for the implementation of the AES algorithm, so to prove the new architectural exploration capabilities of DExIMA CAD.

Given the above premises, the actual performances of the to-be-designed LiM system are not crucial: several optimisations, e.g. pipelining *et cetera*, may be applied at a later stage, but the current goal is to show the structural description capabilities of the tool.

9.2 Description of the AES-128 algorithm

9.2.1 Introduction, notation and encryption algorithm

The Advanced Encryption Standard (AES) algorithm ciphers a fixed-size input text to produce an encrypted fixed-size output text, which are usually referred to as "plaintext" and "ciphertext", respectively. The plaintext is a 16-byte sequence $b_0, b_1, ..., b_{15}$ which represents the initialisation values of a 4-row, 4-column byte matrix, that is customarily referred to as "state" and may be also seen as a 4-element array of 4-byte columns A_l, with $l = 0, 1, 2, 3$, as presented in Figure 9.1.

State $\underline{\underline{S}}$

$\underline{\underline{S}} = [a_{ij}] = [A_0\ A_1\ A_2\ A_3]$

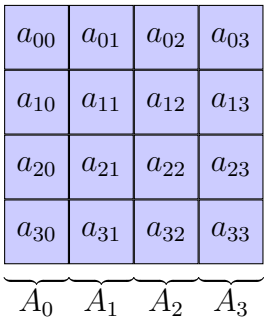


Figure 9.1: Representation of the plaintext and the state in the Advanced Encryption Standard (AES) algorithm.

The encryption process, which applies several operations to progressively transform the state, is accomplished by means of an input key, whose length is determined by the actual variant of the algorithm, i.e. AES-128, AES-192 and AES-256. Besides a variable key length, these variants differ by the number of computational rounds required by the cipher to complete. Before presenting the actual features of the AES variants, some preliminary definitions are required.

Let N_b be the block size, i.e. the length of the plaintext, expressed in number of 32-bit words, N_k be the size of the key, expressed in number of 32-bit words, and N_r be the number of rounds required by the cipher. The values of these parameters are summarised in Table 9.1.

Algorithm	Key length N_k	Block size N_b	Number of rounds N_r
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Table 9.1: Parameters of the Advanced Encryption Standard (AES) variants.

For the purposes of showing the capabilities of DExIMA CAD, the AES-128 variant has been selected, implying that the input key is a 16-byte sequence $k_0, k_1, ..., k_{15}$ which can be conceived as:

- a 4-row, 4-column byte matrix of elements k_{ij};
- a 4-element array of 4-byte columns K_l, with $l = 0, 1, 2, 3$.

To complete the encryption process, the cipher requires a total of N_r 16-byte

round keys, one for each of the N_r computational rounds, which are inferred from the initial key by means of a specific algorithm. Let

$$\underline{\underline{K}} = [k_{ij}] = [K_0 \; K_1 \; K_2 \; K_3]$$

be the initial 16-byte key, r be the computational round and

$$\underline{\underline{K}}^{(r)} = \left[k_{ij}^{(r)}\right] = \left[K_0^{(r)} \; K_1^{(r)} \; K_2^{(r)} \; K_3^{(r)}\right]$$

be the 16-byte round key in round r: the representation of these quantities is presented in Figure 9.2.

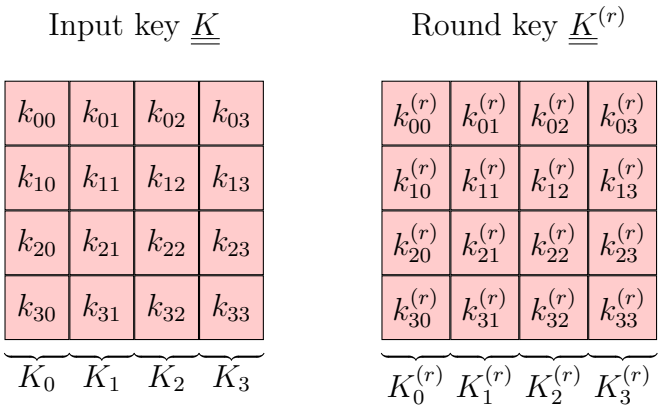


Figure 9.2: Representation of the input key and of the round keys in the Advanced Encryption Standard (AES) algorithm, AES-128 variant.

The AES key schedule algorithm produces $N_\mathrm{b} \cdot (N_\mathrm{r} + 1)$ 32-bit words $W^{(i)}$, which will be used to build each round key $\underline{\underline{K}}^{(r)}$. In the AES-128 variant, the 44 32-bit words $W^{(i)}$ can be related to each round key by means of Algorithm 2.

The pseudo-code of the cipher, which effectively encrypts the plaintext to produce the ciphertext, is presented in Algorithm 3.

In the AddRoundKey step, a byte-wise combination of the state a_{ij} and of the round key $k_{ij}^{(r)}$ is obtained by means of logical XOR operation, as reported in Equation 9.1.

$$a_{ij} \leftarrow a_{ij} \; \oplus \; k_{ij}^{(r)} \tag{9.1}$$

In the SubBytes step, each state byte a_{ij} is applied a non-linear transformation which is referred to as "substitution box", as presented in Equation 9.2.

$$a_{ij} \leftarrow \mathrm{SBox}(a_{ij}) \tag{9.2}$$

In the ShiftRows step, which is depicted in Figure 9.3, a row-dependent byte-wise circular left shift is applied to all rows in the state, as presented in Figure 9.3.

Algorithm 2 AES-128: using the results of the key schedule algorithm to construct the round keys.

```
1: i ← 0
2: while i < 44 do
3:     r ← i/4
4:     if i mod 4 = 0 then
5:         K_0^(r) ← W^(i)
6:     else if i mod 4 = 1 then
7:         K_1^(r) ← W^(i)
8:     else if i mod 4 = 2 then
9:         K_2^(r) ← W^(i)
10:    else if i mod 4 = 3 then
11:        K_3^(r) ← W^(i)
12:    end if
13:    i ← i + 1
14: end while
```

Algorithm 3 AES-128 cipher.

```
1: function CIPHER(Plaintext, K)
2:     S ← Plaintext
3:     KeyExpansion(K)
4:     AddRoundKey(S, K^(0))
5:
6:     r ← 1
7:     while r ≤ 9 do
8:         SubBytes(S)
9:         ShiftRows(S)
10:        MixColumns(S)
11:        AddRoundKey(S, K^(r))
12:        r ← r + 1
13:    end while
14:
15:    SubBytes(S)
16:    ShiftRows(S)
17:    AddRoundKey(S, K^(10))
18:
19:    Ciphertext ← S
20: end function
```

- the bytes from the first row a_{0j} are not shifted;
- the bytes from the second row a_{1j} undergo a 1-byte circular left shift;
- the bytes from the second row a_{2j} undergo a 2-byte circular left shift;
- the bytes from the second row a_{3j} undergo a 3-byte circular left shift.

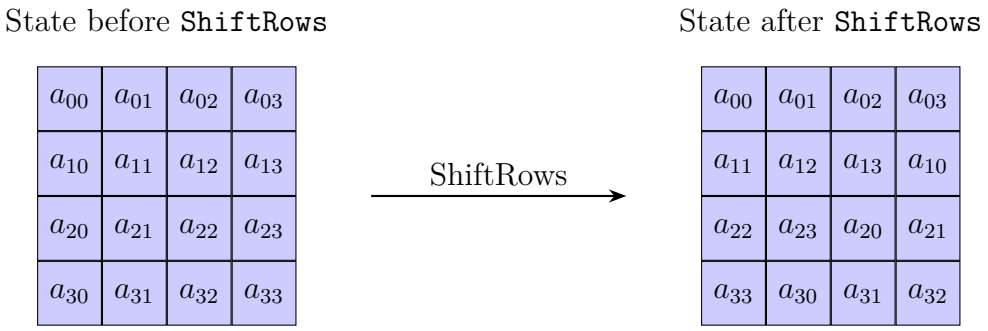


Figure 9.3: General representation of the `ShiftRows` step in the Advanced Encryption Standard (AES) algorithm.

In the `MixColumns` step, each column of the state is processed by means of a transformation which can be expressed in the matrix form of Equation 9.3, considering that all partial products are derived by means of Galois multiplication and are properly summed by means of a logical bitwise XOR operation.

$$\begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix} \leftarrow \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix} \tag{9.3}$$

Given the nature of the operations in Algorithm 3, it is almost straightforward to notice that there is no data dependency within steps AddRoundKey, SubBytes, ShiftRows and MixColumns, implying that these operations can carried out in parallel, i.e. by transforming all state bytes a_{ij} at once, if proper resources allocation choices are made.

9.2.2 Key schedule algorithm

In the AES-128 variant, the key schedule algorithm presented in Algorithm 4 progressively transforms the input key $\underline{\underline{K}}$ to produce the required round keys $\underline{\underline{K}}^{(r)}$.

The key schedule requires the 32-bit round constant

$$\text{Rcon}^{(r)} = [\text{rc}^{(r)}\ \text{0x00}\ \text{0x00}\ \text{0x00}]$$

Algorithm 4 Key schedule algorithm in AES-128.

function KeyExpansion($\underline{\underline{K}}$)
  $W^{(0)} \leftarrow K_0$
  $W^{(1)} \leftarrow K_1$
  $W^{(2)} \leftarrow K_2$
  $W^{(3)} \leftarrow K_3$

  $i \leftarrow 4$
  while $i < 44$ **do**
    TEMP $\leftarrow W^{(i-1)}$
    if $i \mod 4 = 0$ **then**
      TEMP $\leftarrow$ SubWord(RotWord(TEMP)) $\oplus$ Rcon$^{(i/4)}$
    end if
    $W^{(i)} \leftarrow W^{(i-4)} \oplus$ TEMP
    $i \leftarrow i + 1$
  end while
end function

where the quantity $\mathrm{rc}^{(r)}$ is the least-significant byte and is computed according to the recursive definition of Equation 9.4, which, in the case of AES-128, where $r = 0, 1, ..., 10$, leads to the values presented in Table 9.2.

$$\mathrm{rc}^{(r)} = \begin{cases} \text{0x01}, & r = 1 \\ 2 \cdot \mathrm{rc}^{(r-1)}, & r > 1 \text{ and } \mathrm{rc}^{(r-1)} < \text{0x80} \\ 2 \cdot \mathrm{rc}^{(r-1)} \oplus \text{0x11B}, & r > 1 \text{ and } \mathrm{rc}^{(r-1)} \geq \text{0x80} \end{cases} \tag{9.4}$$

r	1	2	3	4	5	6	7	8	9	10
$\mathrm{rc}^{(r)}$	01	02	04	08	10	20	40	80	1B	36

Table 9.2: Values of $\mathrm{rc}^{(r)}$ in AES-128.

Besides the round constants $\mathrm{Rcon}^{(r)}$, the key schedule algorithm requires two additional operations, namely RotWord and SubWord, which transform a 4-byte input column: the former (Equation 9.5) is a byte-wise circular left shift, while the latter (Equation 9.6) applies the substitution box to each byte.

$$\mathrm{RotWord}\left([\mathrm{B}_0\ \mathrm{B}_1\ \mathrm{B}_2\ \mathrm{B}_3]\right) = [\mathrm{B}_1\ \mathrm{B}_2\ \mathrm{B}_3\ \mathrm{B}_0] \tag{9.5}$$

$$\mathrm{SubWord}\left([\mathrm{B}_i]\right) = [\mathrm{SBox}(\mathrm{B}_i)] \tag{9.6}$$

9.3 Derivation of the LiM architecture

An in-memory implementation of the AES-128 encryption phase requires the simultaneous handling of both the cipher and the key schedule procedures. Given the fact that two different algorithms must be implemented, it seems reasonable to partition the design in two main sections.

The first section, hereinafter referred to as "state section", consists of sixteen byte-wide rows and is devoted to store the sixteen state bytes a_{ij} and to manipulate them in the AddRoundKey, SubBytes, ShiftRows and MixColumns steps, as specified by Algorithm 3.

The second section, hereinafter referred to as "key section", consists of sixteen byte-wide rows and is conceived to compute and store the round constants $\underline{\underline{K}}^{(r)}$, by providing an hardware implementation of Algorithm 4. It is important to point out that the key section does not compute the least-significant byte $\mathrm{rc}^{(r)}$ of the round constant $\mathrm{Rcon}^{(r)}$.

A general representation of the sections is presented in Figure 9.4. In either section, the processing capabilities evidently derive from the allocation of specific resources in both the LiM cells and the IRL blocks.

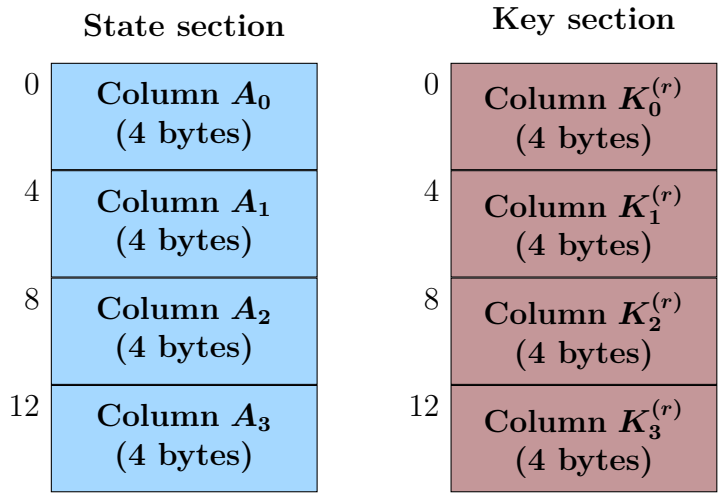


Figure 9.4: LiM AES architecture: general representation of the state and the key sections. The notation presented in Section 9.2 is used to indicate the content of the two sections.

Regarding the overall structure of the complete LiM system, the new architectural exploration capabilities of DExIMA CAD allow two different solutions:

- a single LiM array can contain both the state and the key sections;
- two separate LiM arrays may be allocated, one for the state section and one for the key section.

Given the mentioned limitation of the key section, a component must be allocated, somewhere in the LiM system, to provide the computation of the round constant. As a matter of fact, two solutions are possible: the required resources may be created either within or outside the LiM array.

As a consequence, the section partitioning assumption leads to at least four different architectures which may implement the required algorithm. For the purposes of this study, the considered solutions are summarised in Table 9.3, while their representation is reported in Figure 9.5.

Identifier	Number of LiM arrays	Computation of $rc^{(r)}$
`StateRoundKey1`	1	In-Memory
`StateKey1`	1	Out-Of-Memory
`State1Key1`	2	Out-Of-Memory

Table 9.3: LiM AES architecture: summary of the considered architectural-level solutions.

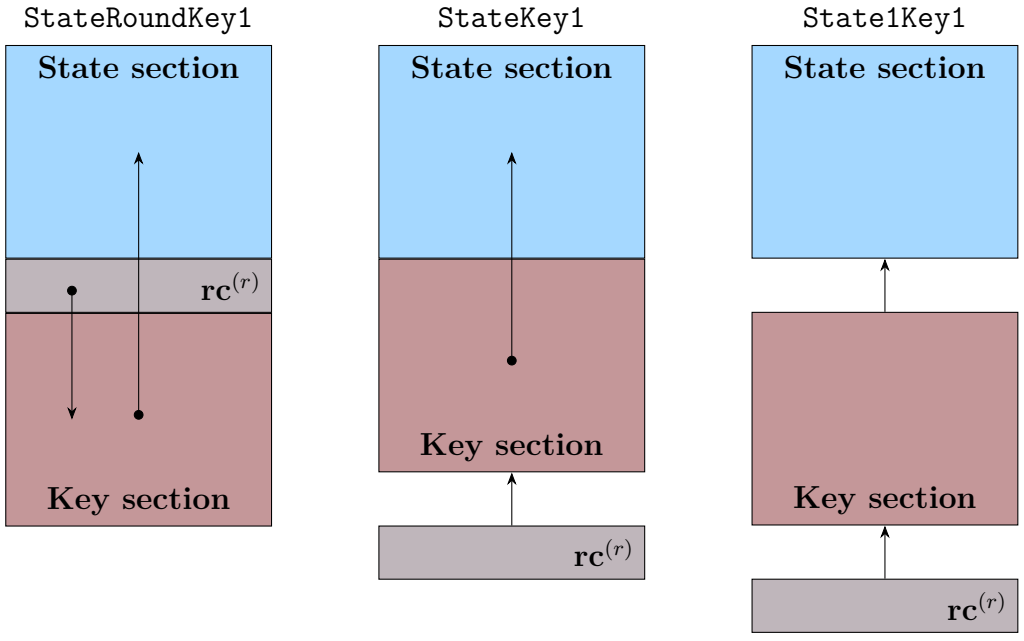


Figure 9.5: LiM AES architecture: representation of the considered architectural-level solutions.

The actual architectural solution does not influence the overall structure of all LiM cells and IRL blocks in each section, but slight differences may be found in the interconnections pattern.

In the `StateRoundKey1` solution, one LiM array is allocated which contains the state section, the key section and additional resources for the computation of $rc^{(r)}$: as it will be described later, such resources consist of a standard memory row and a LiM memory row. A first set of intra-array interconnections connects the round constant LiM row to the key section, enabling the implementation of Algorithm 4, while a second set of intra-array interconnections routes all round key bytes to the corresponding state bytes.

In the `StateKey1` solution, the LiM array contains only the state and the key sections, while the round constant computation resources are allocated outside the memory array and their output is fed to the key section by means of the horizontal bus presented in Chapter 3. Furthermore, one set of intra-array interconnections is required for the AddRoundKey step, linking each state byte with its corresponding round key byte.

In the `State1Key1` solution, two separate LiM arrays are allocated, one for the state section and one for the key section, which communicate with each other with a set of inter-array interconnections. In this solution, the round constant is routed to the key array via the horizontal bus.

9.3.1 State section

AddRoundKey step To enable the fully-parallel execution of the AddRoundKey step, each memory cell of the state section should integrate:

- a XOR gate to implement Equation 9.1;
- a proper interconnection to fetch the corresponding round key byte.

As previously mentioned, the actual interconnection type depends on the specific architectural solution.

SubBytes step According to [4], the SubBytes step would require two separate operations, namely the computation of a multiplicative inverse and an affine transformation: the latter is linear and may be implemented by a proper signal routing scheme and few logic gates, but the former requires a more refined algorithm. As a consequence, to prevent an undesired performance loss, a typical approach in most hardware implementations [7] is to deploy a 8-bit Look-Up Table (LUT). Hence, a fully-parallel execution of the SubBytes step may be achieved if each state byte is associated a IRL block which contains a SBox LUT.

ShiftRows step Since it involves basic byte-wise circular shifts, the ShiftRows step can be easily integrated in the LiM architecture by means of array interconnections. Furthermore, because Algorithm 3 is virtually devoid of any data

dependency, this operation can be scheduled in the same computational cycle of the SubBytes step. More specifically:

1. each state byte is transferred to its associated IRL block for the computation of SBox transformation;

2. to implement the shifts required by the ShiftRows step, the results of the above processing are routed in the state section by means of the interconnection scheme presented in Figure 9.6.

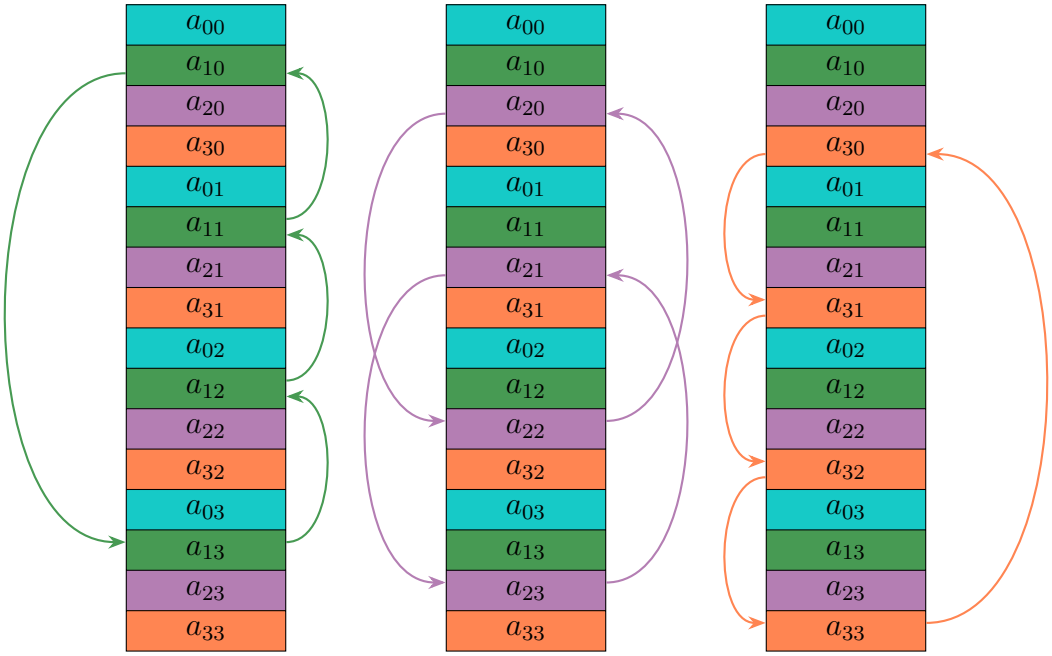


Figure 9.6: LiM AES architecture: implementation of the SubBytes and the ShiftRows steps. The four different colours group together the bytes belonging to the same row. For all array interconnections, the source is a IRL block, with source pin `IRL0`, while the destination is a memory row.

Taxonomy of state cells Because of the interconnection pattern presented in Figure 9.6, multiple types of LiM cells must be allocated in the state section, each integrating different vertical interconnections. The type of state cell may be uniquely identified by the row index and by the vertical interconnection it implements. In this context, an "upward cell" integrates a vertical interconnection with direction `Next`, implementing the upward connections of Figure 9.6. Similarly, a "downward cell" includes a vertical interconnection with direction `Previous`, implementing the

downward connections of Figure 9.6. The types of memory cells in the state section, and the interconnection they implement, are summarised in Table 9.4.

	Upward cells			Downward cells		
	Direction	Displacement	Type	Direction	Displacement	Type
Row 1	`Next`	5	`R1_U`	`Previous`	12	`R1_D`
Row 2	`Next`	9	`R2_U`	`Previous`	8	`R2_D`
Row 3	`Next`	13	`R3_U`	`Previous`	4	`R3_D`

Table 9.4: LiM AES architecture: types of memory cells in the state section.

Besides the six state cell types of Table 9.4, the state section requires a further type, `cell_R0`, for the four state bytes belonging to the first row, which do not require any shift.

MixColumns step To implement the MixColumns step, each IRL block of the state section embeds a dedicated component which combines its four inputs `IN0_1`, `IN1_1`, `IN2_2` and `IN3_3` according to Equation 9.7, where multiplication is indeed Galois multiplication.

$$\texttt{MIXED} = \texttt{IN0\_1} \oplus \texttt{IN1\_1} \oplus 2 \cdot \texttt{IN2\_2} \oplus 3 \cdot \texttt{IN3\_3} \tag{9.7}$$

As all matrix rows in Equation 9.3 are different from one another, to properly feed all `AES_MixColumns` blocks, a proper interconnection pattern must be allocated, as presented in Figure 9.7.

Equation 9.7 may be further simplified by exploiting a property of Galois multiplication, which ultimately leads to Equation 9.8.

$$\texttt{MIXED} = \texttt{IN0\_1} \oplus \texttt{IN1\_1} \oplus 2 \cdot \texttt{IN2\_2} \oplus (2 \cdot \texttt{IN3\_3} \oplus \texttt{IN3\_3}) \tag{9.8}$$

Equation 9.8 clearly shows that, besides four elementary byte-wide XOR gates, a `AES_MixColumns` instance must compute two Galois multiplications. This type of operation may be supported by a look-up table, but a very simple hardware implementation may be achieved if the actual algorithm is considered. In essence, Galois multiplication by 2 is a left-shift operation, but the shifted byte must undergo a XOR operation with constant 0x1B if the MSB of the input quantity is set. In practice, the structure presented in Figure 9.8 may implement the desired behaviour with a rather limited set of logic gates.

LiM cells and IRL blocks The overall structure of all cell types in the state section is presented in Figure 9.9. Regardless of the type, all cells include a XOR gate for the in-memory implementation of the AddRoundKey step, a four-way multiplexer to help schedule the computational tasks and a set of three interconnections.

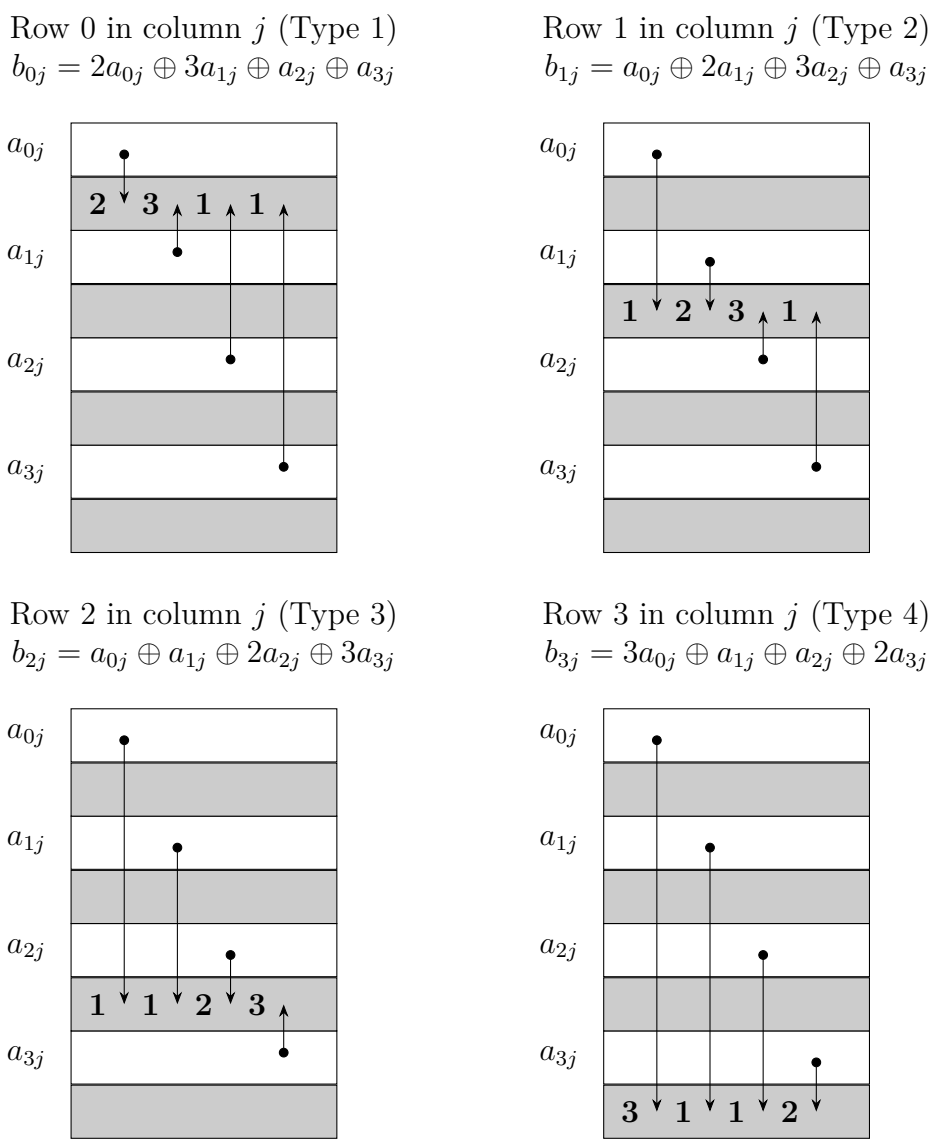


Figure 9.7: Implementation of the `MixColumns` step in the LiM architecture for the Advanced Encryption Standard (AES) algorithm. The source and the destination of each vertical interconnection are indicated by a dot and by an arrow tip, respectively. Each number represents the amount by which each source must be multiplied in the IRL sections.

The structure of all IRL blocks in the state section is reported in Figure 9.10. Each block includes a look-up table for the SubBytes step and a `AES_MixColumns` instance for the MixColumns step.

Assuming that the four-way multiplexer of Figure 9.9 is implemented by means of three two-way multiplexers, the overall complexity of the section is summarised

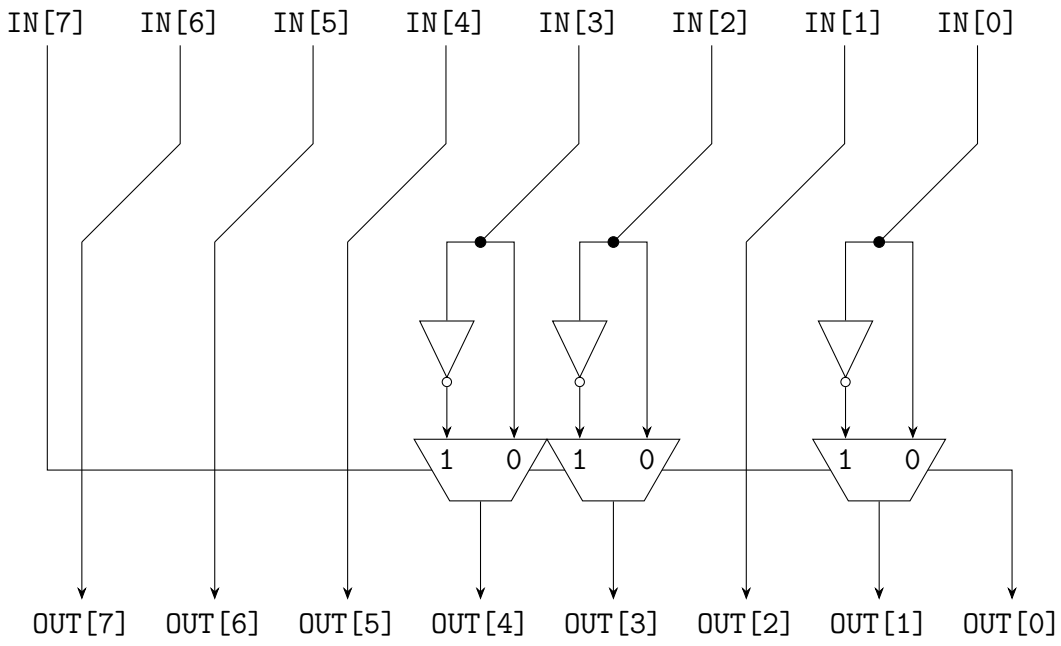


Figure 9.8: LiM AES architecture: internal structure of the `AES_GalMult2` component, which computes the Galois multiplication by 2 of an input byte.

in Table 9.5.

Component type	Number of instances
`XOR2`	16
`MUX21`	48
`AES_LUT_SBox`	16
`AES_MixColumns`	16

Table 9.5: LiM AES architecture: complexity of the state section.

9.3.2 Key section

To derive the structure of the memory cells and of the IRL blocks in the key section the AES key schedule algorithm (Algorithm 4) and the relation between $W^{(i)}$ and the columns in $\underline{\underline{K}}^{(r)}$ (Algorithm 2) should be combined in a hardware-like pseudocode, as the one reported in Algorithm 5.

Lines 9, 10, 11 and 12 in Algorithm 5 are characterised by Read-After-Write data dependencies, implying that the computation of the new round key $\underline{\underline{K}}^{(r)}$ from the previous round key $\underline{\underline{K}}^{(r-1)}$ must be done sequentially, i.e. one column at a time.

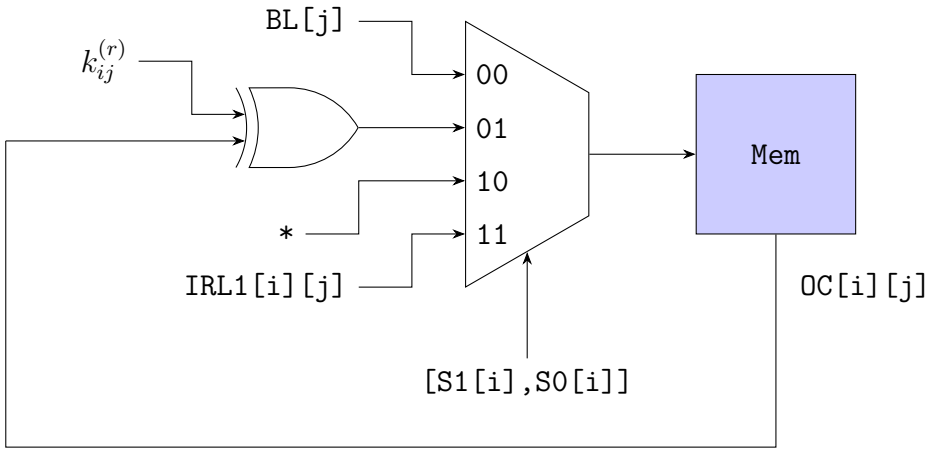


Figure 9.9: LiM AES architecture: LiM cells in the state section. `IRL1[i][j]` is the input pin of a `Next-1-IRL1-to-Row` interconnection, while `*` is a placeholder for a further vertical interconnection pin, which is determined by the pattern presented in Figure 9.6. Lastly, $k_{ij}^{(r)}$ indicates a round key byte connection, which depends on the actual architectural solution (i.e. `StateRoundKey1`, `StateKey1` and `State1Key1`).

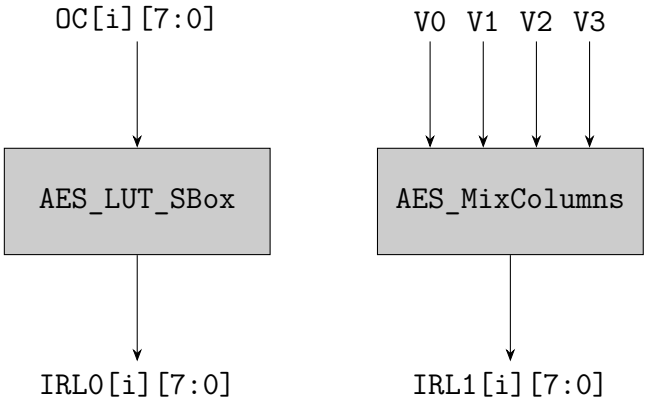


Figure 9.10: LiM AES architecture: IRL blocks in the state section. `V0`, `V1`, `V2` and `V3` represent the interconnection input pins of the pattern presented in Figure 9.7.

The second, the third and the fourth columns (i.e. operations in lines 10, 11 and 12) simply require the allocation of a XOR gate and a set of vertical interconnections, connecting the previous column to the currenty analysed one, e.g. column 0 to column 1, column 1 to column 2 and column 2 to column 3. As a matter of fact, all rows belonging to these columns are not associated to any IRL block, as all operations take place in-row.

Algorithm 5 Hardware-like description of the AES-128 key schedule algorithm.

function KeyExpansion($\underline{\underline{\mathrm{K}}}$)
  $K_0^{(0)} \leftarrow K_0$
  $K_1^{(0)} \leftarrow K_1$
  $K_2^{(0)} \leftarrow K_2$
  $K_3^{(0)} \leftarrow K_3$

  $r \leftarrow 1$
  while $r \geq 9$ **do**
    $K_0^{(r)} \leftarrow K_0^{(r-1)} \oplus \mathrm{SubWord}\left(\mathrm{RotWord}\left(\mathrm{K}_3^{(\mathrm{r}-1)}\right)\right) \oplus \mathrm{Rcon}^{(r)}$
    $K_1^{(r)} \leftarrow K_1^{(r-1)} \oplus K_0^{(r)}$
    $K_2^{(r)} \leftarrow K_2^{(r-1)} \oplus K_1^{(r)}$
    $K_3^{(r)} \leftarrow K_3^{(r-1)} \oplus K_2^{(r)}$
    $r \leftarrow r + 1$
  end while
end function

Besides the above elements, each byte of the first column requires a IRL block with a `AES_LUT_SBox` instance for the computation of the SubWord step, while the RotWord step may be handled with a proper interconnection pattern. Moreover, for the first byte only, a further XOR operation with $\mathrm{rc}^{(r)}$ must be taken.

The above considerations are better visualised in Figure 9.11, which shows all the interconnections originating from the rows in the key section and the required IRL blocks, while the structure of the LiM cells in the key section is presented in Figure 9.12, in Figure 9.13 and in Figure 9.14.

Three different cell types are indeed required to describe the key section, integrating the necessary operators and interconnections:

- `cell_KC0R0` for the first byte in column 0 (Figure 9.12);
- `cell_KC0R1R2R3` for the remaining bytes in column 0 (Figure 9.13);
- `cell_KC1C2C3` for columns 1, 2 and 3 (Figure 9.14).

The overall complexity of the key section is summarised in Table 9.6.

9.3.3 Round constant computation

It has been mentioned that, besides the state and the key sections, some resources must be allocated to compute the quantity $\mathrm{rc}^{(r)}$ according to Equation 9.4. For the first eight computational round rounds, a bit-wise shift register, properly initialized to 0x01 before the encryption starts, may successfully provide the values of $\mathrm{rc}^{(r)}$.

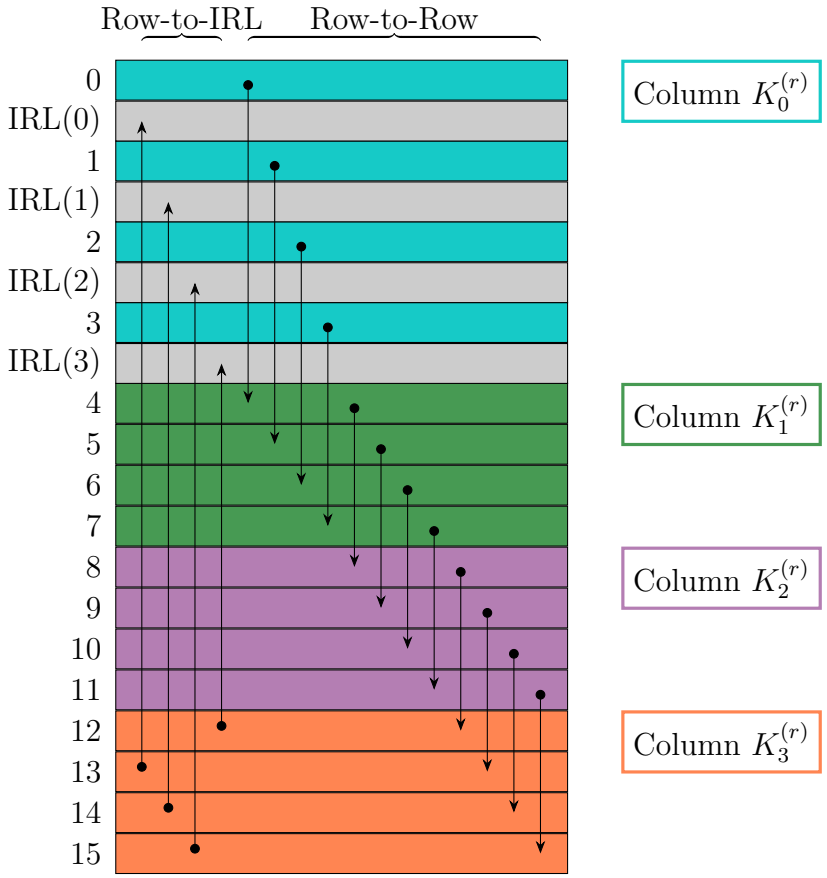


Figure 9.11: LiM AES architecture: interconnections pattern in the key section. The source and the destination of each vertical interconnection are indicated by a dot and by an arrow tip, respectively. Although not explicitly indicated, each IRL block feeds its output to its associated memory row.

Component type	Number of instances
`XOR2`	17
`MUX21`	16
`AES_LUT_SBox`	4

Table 9.6: LiM AES architecture: complexity of the key section.

In round $r = 9$, i.e. when the shifted quantity would be larger than 0x80, constant byte 0x1B should be loaded in the shift register. In the last computational round, i.e. $r = 10$, the usual shift-left operation may be applied to produce the last round constant.

As Figure 9.5 shows, the `StateRoundKey1` solution requires an in-memory shift

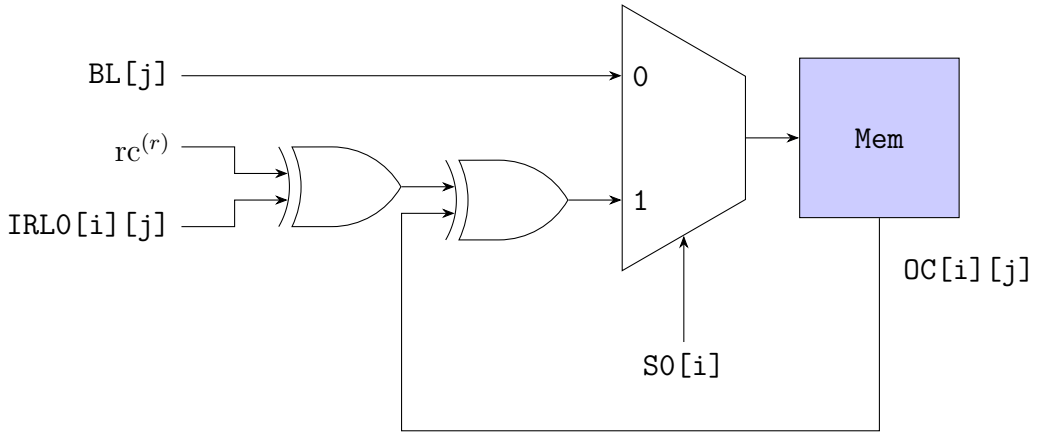


Figure 9.12: LiM AES architecture: LiM cells in the key section, column 0, row 0. `IRL0[i][j]` is the input pin of a `Next-1-IRL0-to-Row` vertical interconnection, while $\mathrm{rc}^{(r)}$ indicates a connection from the round constant computation resources, which depends on the actual architectural solution (i.e. `StateRoundKey1`, `StateKey1` and `State1Key1`).

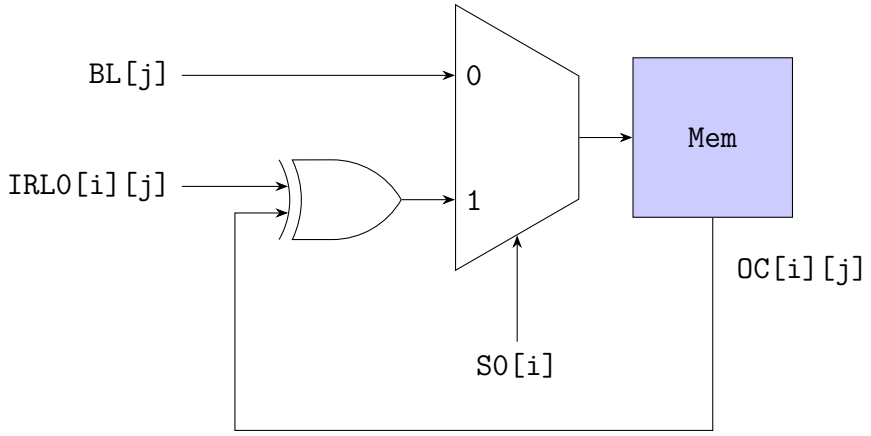


Figure 9.13: LiM AES architecture: LiM cells in the key section, column 0, rows 1, 2 and 3. `IRL0[i][j]` is the input pin of a `Next-1-IRL0-to-Row` vertical interconnection.

register which, in this DExIMA CAD implementation, may be integrated using the horizontal interconnections presented in Chapter 3. Nevertheless, it should be noted that a mechanism must be devised to automatically load the constant value 0x1B in computational round $r = 9$, as this task is not straightforward to fulfill in

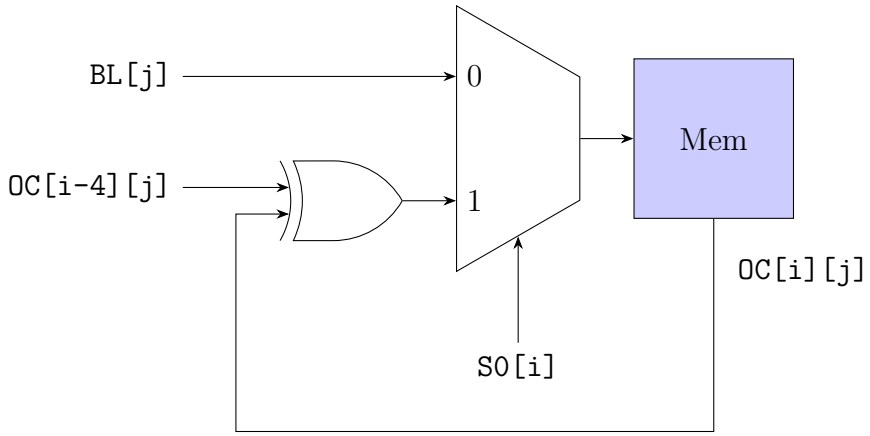


Figure 9.14: LiM AES architecture: LiM cells in the key section, columns 1, 2 and 3. `OC[i-4][j]` is the input pin of a `Prev-4-OC-to-Row` vertical interconnection

DExIMA CAD: indeed, no status signal can be fetched from the LiM architecture by the control unit, which would not know when to force the constant byte 0x1B. A workaround to this problem is readily found by allocating a flip-flop in close proximity to the MSB cell, acting as a localized control element. In addition, a dedicated standard memory row (i.e. with no processing capabilities) is created to store the constant byte 0x1B. The structure of the in-memory shift register is presented in Figure 9.15, for which the following considerations hold:

- a LSB-to-MSB horizontal interconnection is used to configure the memory row as a bit-wise shift register;
- a vertical interconnection is used to fetch the constant 0x1B;
- the flip-flop output is dispatched to all the remaining cells in the row by means of a MSB-to-LSB horizontal interconnection.

On the other hand, in the `StateKey1` and the `State1Key1` solutions, the quantity $\mathrm{rc}^{(r)}$ is computed externally to the memory array, thus a bit-wise shift register may be allocated near the memory array, while the horizontal bus may be used to dispatch the round constant byte to the key section. The structure of this component is reported in Figure 9.16.

9.4 Architectural description in DExIMA CAD

All considerations presented in Section 9.3 define the structure of all array elements and their mutual interconnections. DExIMA CAD can thus be deployed to implement the structure of all LiM cells and IRL blocks, eventually creating the complete

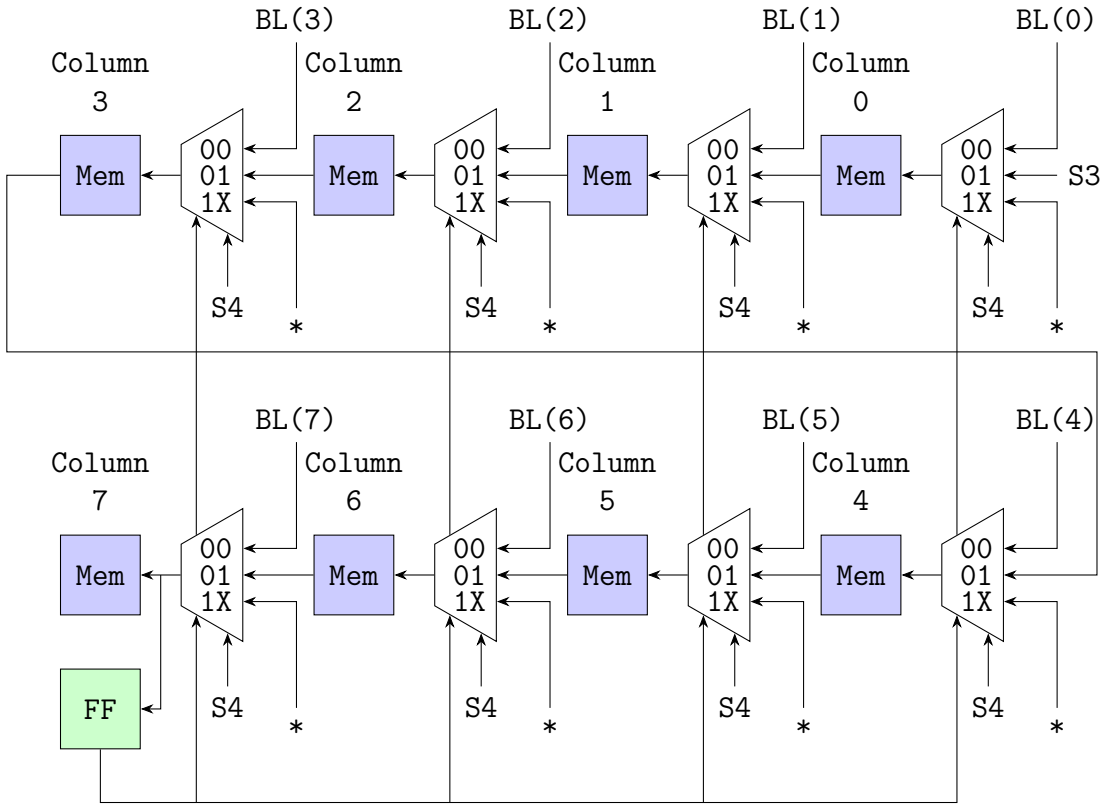


Figure 9.15: LiM AES architecture: LiM cells for the computation of the least-significant byte $\mathrm{rc}^{(r)}$ of the round constant $\mathrm{Rcon}^{(r)}$, `StateRoundKey1` solution. `*` is a placeholder for a `Fixed-16-OC-to-Row` vertical interconnection, which is used to fetch the constant byte 0x1B.

LiM array. From the structural point of view, all three architectural solutions are rather similar: for this reason, Appendix C only reports some implementation examples for the `StateRoundKey1` solution.

Array geometries In the `StateRoundKey1` solution, the only byte-wide LiM array is characterized by the following memory map:

- sixteen byte-wide rows for the state section;
- a standard memory row which stores the constant 0x1B;

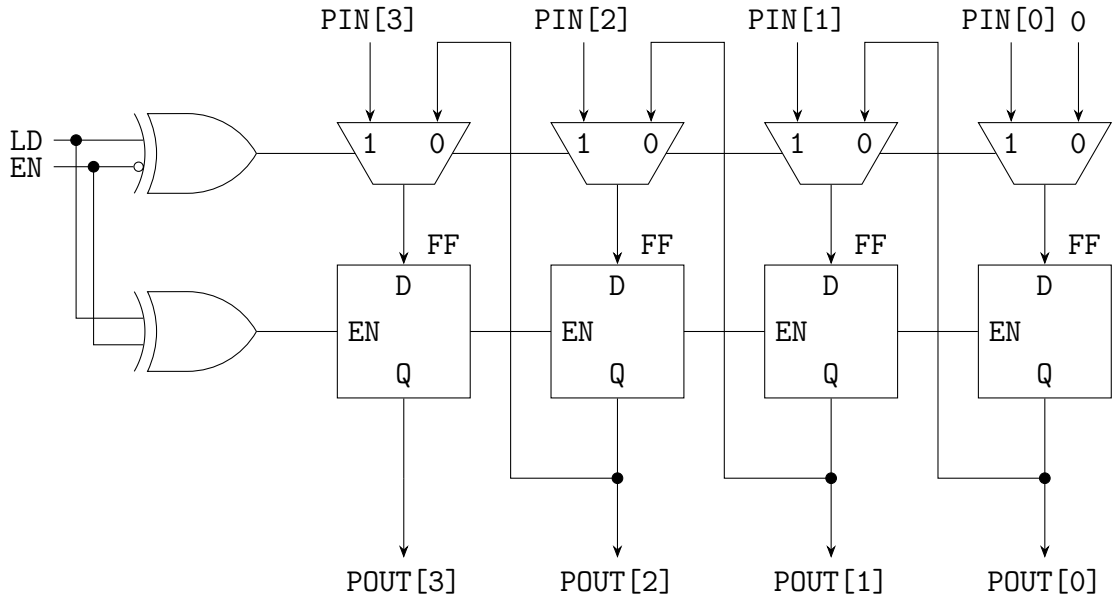


Figure 9.16: LiM AES architecture: structure of the Out-Of-Memory bit-wise shift register. For representation purposes only, a simplified half-byte version of the component is shown.

- an in-memory bit-wise shift register;
- sixteen byte-wide rows for the key section.

As the total number of rows is not a power-of-2 quantity, standard memory rows after the key section extend the LiM array to 64 rows.

In the `StateKey1` solution, the only byte-wide LiM array is characterized by the following memory map:

- sixteen byte-wide rows for the state section;
- sixteen byte-wide rows for the key section.

In the `StateKey1` solution, both byte-wide LiM arrays have sixteen memory rows.

Uppermost architectural level Having determined the structure of the LiM arrays in all three solutions, a definition of the uppermost architectural level must be provided.

In the `StateRoundKey1` solution, only one LiM is required to implement the AES-128 algorithm. To support this unique array instance, the allocation of a single micro-programmed control unit and a single micro-ROM may suffice. A

wordline multiplexer must be allocated to manage the internal control provided by the micro-programmed control unit and the external control provided by the UVM testbench. Nevertheless, no other Out-Of-Memory component is required in this architecture. Ultimately, the uppermost architectural level for the `StateRoundKey1` solutions is reported in Figure 9.17.

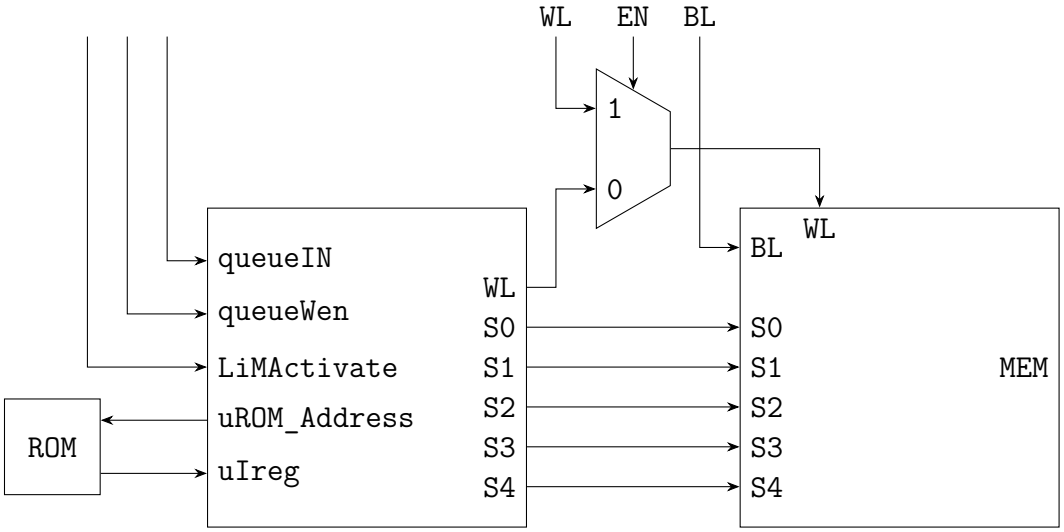


Figure 9.17: LiM AES architecture: uppermost architectural level for the `StateRoundKey1` solution. Although not explicitly indicated, the LiM array and the memory interface are driven by the same clock and reset signals.

With respect to the `StateRoundKey1` solution, the `StateKey1` architecture requires an Out-Of-Memory bit-wise shift register, while only one LiM array is allocated in the uppermost architectural level. The resulting structure is reported in Figure 9.18.

Nano-instructions Regardless of the architectural solution, the same types of nano-instructions may be defined to control the state and the key sections.

The following set of nano-instructions may be applied to any memory cell in the array, regardless of its type.

- `NOP`: No operation.
- `Enable_Row`: Prepare a write operation to the selected memory row.
- `Input_BL`: Select the bitline as the write input of a selected memory row.

All LiM cells belonging to the state section may be controlled by means of the following set of nano-instructions.

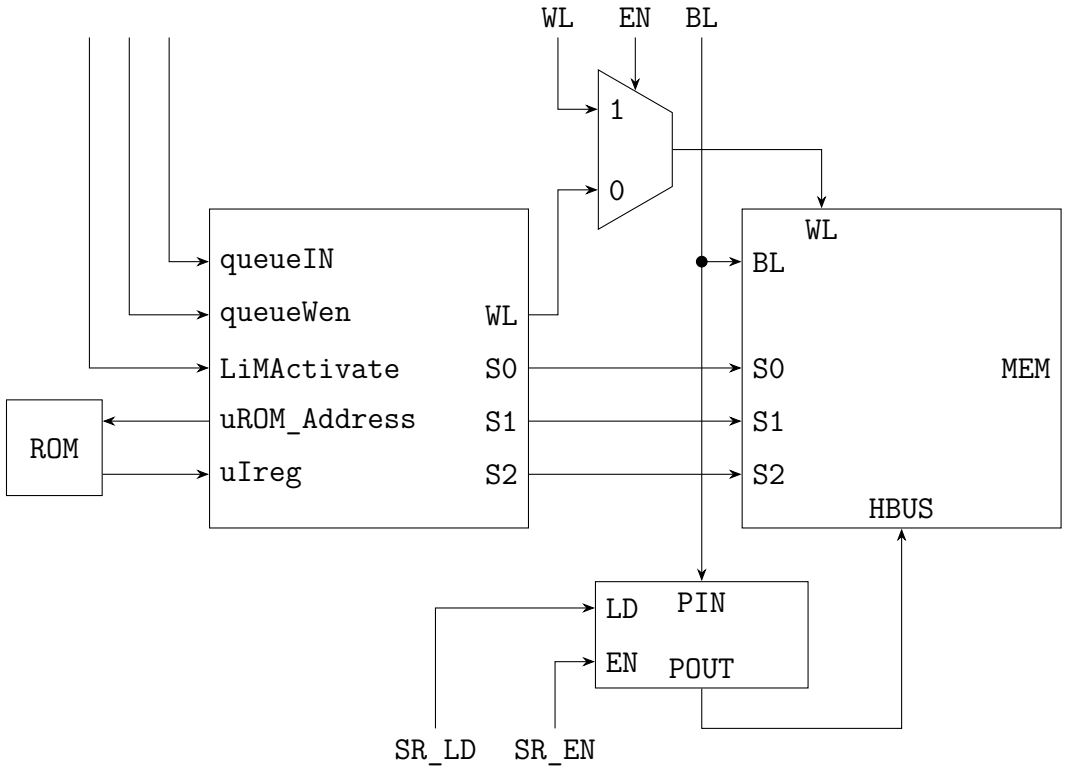


Figure 9.18: LiM AES architecture: uppermost architectural level for the `StateKey1` solution. Although not explicitly indicated, the LiM array, the memory interface and the shift register are driven by the same clock and reset signals.

- `State_Input_AddRoundKey`: Select the result of the AddRoundKey step as the write input of a selected memory row.
- `State_Input_SubBytesAndShiftRows`: Select the result of the SubBytes and the ShiftRows steps as the write input of a selected memory row.
- `State_Input_MixColumns`: Select the result of the MixColumns step as the write input of a selected memory row.

All LiM cells belonging to the key section may be controlled by means of the following set of nano-instructions.

- `Key_Input_Update`: Select the updated round key byte value as the write input of a selected memory row.

The additional nano-instruction `RoundConstant_Update` is required by architectural solution `StateRoundKey1` to enable the shift operation in the in-memory shift register, updating the least-significant byte of the round constant.

Simulation dashboard The presented set of nano-instructions is sufficient to completely specify the behaviour of the LiM array in any computational cycle. The simulation dashboard may be used to indicate, for each algorithm step, the internal control of the LiM array. In the `StateRoundKey1` solution, there is no need to drive any active pin, as all computations take place within the memory array.

All operations involving the state section are detailed in the following.

- AddRoundKey step
  - Scalar control
    `Enable_Row`
    `State_Input_AddRoundKey`
  - Activation pattern
    `RANGE:0:15` in the `StateRoundKey1` solution
    `RANGE:0:15` in the `StateKey1` solution
    `ALL` in the `State1Key1` solution
- SubBytes and ShiftRows steps
  - Scalar control
    `Enable_Row`
    `State_Input_SubBytesAndShiftRows`
  - Activation pattern
    `RANGE:0:15` in the `StateRoundKey1` solution
    `RANGE:0:15` in the `StateKey1` solution
    `ALL` in the `State1Key1` solution
- MixColumns step
  - Scalar control
    `Enable_Row`
    `State_Input_MixColumns`
  - Activation pattern
    `RANGE:0:15` in the `StateRoundKey1` solution
    `RANGE:0:15` in the `StateKey1` solution
    `ALL` in the `State1Key1` solution

The round key update, i.e. the management of the key section, is detailed in the following.

- `StateRoundKey1` solution
  - Scalar control
    `Enable_Row`
    `Key_Input_Update`

  - Activation pattern
    `RANGE:18:21` for the first column
    `RANGE:22:25` for the second column
    `RANGE:26:29` for the third column
    `RANGE:30:33` for the fourth column

- `StateKey1` solution

  - Scalar control
    `Enable_Row`
    `Key_Input_Update`
  - Activation pattern
    `RANGE:16:19` for the first column
    `RANGE:20:23` for the second column
    `RANGE:24:27` for the third column
    `RANGE:28:31` for the fourth column

- `State1Key1` solution

  - Scalar control
    `Enable_Row`
    `Key_Input_Update`
  - Activation pattern
    `RANGE:0:3` for the first column
    `RANGE:4:7` for the second column
    `RANGE:8:11` for the third column
    `RANGE:12:15` for the fourth column

Chapter 10

Results

The purpose of Chapter 10 is to present a set of results, which are deemed useful to show the effectiveness of all new DExIMA CAD functionalities.

The new modules presented in Chapter 3, Chapter 4 and Chapter 5 are supposed to largely increase the structural description and the architectural exploration capabilities of DExIMA CAD, generating a plug-and-play LiM architecture and its associated UVM testbench. When the design flow is complete, DExIMA CAD may generate the source code for the complete LiM system and may configure its UVM testbench. A commercial HDL simulator, e.g. QuestaSim, may be used to simulate the design, in a fully-automated way. In fact, the top-level analyzer is also able to generate a specific QuestaSim simulation script, so to reduce the necessary interactions with the designer. The effectiveness of the new functionalities may be shown by verifying if the DExIMA CAD design process leads to a functionally-sound architecture. This point is addressed in Section 10.1.

A QuestaSim simulation of the synthesized LiM architecture would not be sufficient to prove the new functionalities in DExIMA CAD, as the most crucial component in the LiM Development Toolchain is the custom estimator, i.e. DExIMA Backend. With respect to the past version of DExIMA CAD, the array interconnections module presented in Chapter 3 changes the procedure for generating the DExIMA Backend input file. It is thus necessary to verify if DExIMA Backend produces reasonable estimates when analyzing the new `.dex` description. This point is addressed in Section 10.2, in which the main figures of merit of a LiM architecture are estimated with both DExIMA Backend and with a well-established EDA tool, i.e. Synopsys Design Compiler.

10.1 Simulation results

The primary aim of Section 10.1 is to report the results of the complete DExIMA CAD design flow for the LiM SHA-1 architecture presented in Chapter 8. In addition, additional remarks related to all other DExIMA CAD implementations will be presented.

LiM SHA-1 The DExIMA CAD description of the LiM SHA-1 architecture is synthesized, leading to the generation of the source code and the configuration of the DExIMA-controlled UVM testbench. The produced description is then simulated with QuestaSim and, to verify if it is functionally sound, it is important to determine whether the produced message digest is correct. To fulfill this task, the numerical example provided in [3] is used as a draft verification reference.

The example provided in [3] presents the values of the five temporary buffers in all computational steps: their final values are reported in the following.

- Final value of A: `0x42541B35`.
- Final value of B: `0x5738D5E1`.
- Final value of C: `0x21834873`.
- Final value of D: `0x681E6DF6`.
- Final value of E: `0xD8FDF6AD`.

As the five message digest components are initialized as indicated in Section 8.1, their expected final values are the following.

- Final value of H_0: `0x67452301 + 0x42541B35 = 0xA9993E36`.
- Final value of H_1: `0xEFCDAB89 + 0x5738D5E1 = 0x4706816A`.
- Final value of H_2: `0x98BADCFE + 0x21834873 = 0xBA3E2571`.
- Final value of H_3: `0x10325476 + 0x681E6DF6 = 0x7850C26C`.
- Final value of H_4: `0xC3D2E1F0 + 0xD8FDF6AD = 0x9CD0D89D`.

To prove the correct behaviour of the LiM system, a snapshot from the QuestaSim simulation is reported in Figure 10.1. For representation purposes, only the five memory rows associated to the message digest components are reported: despite this simplification, it is immediate to notice a match between their expected and simulated message digest components. It is thus possible to conclude that the DExIMA CAD source code generation procedure is correct and that array interconnections are properly handled.

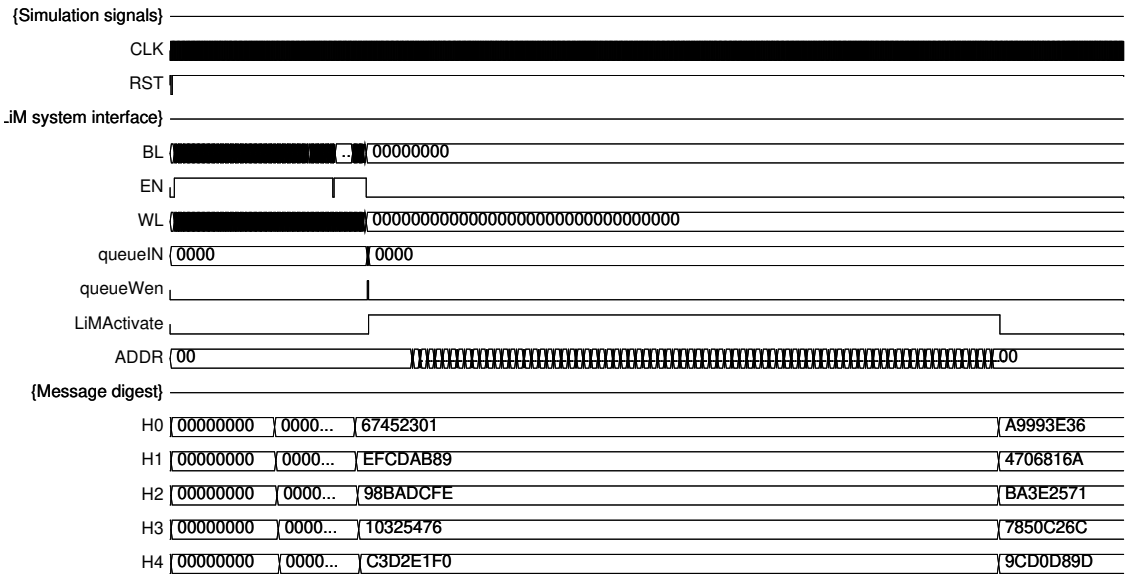


Figure 10.1: LiM SHA-1 architecture: message digest components in the QuestaSim simulation.

All other DExIMA CAD implementations The functional verification procedure presented in the previous paragraph has been carried out for all other DExIMA CAD implementations, to verify whether the results produced by the LiM architecture are correct: in all cases, a positive outcome has been observed.

It is thus possible to conclude that the design flow in DExIMA CAD is able to produce functionally-sound architectures, which are ready to be simulated and synthesized with the most common EDA tools, e.g. QuestaSim and Synopsys Design Compiler. A rapid simulation of the produced design is made possible by the algorithm description facilities and the simulation dashboard, which ease the definition of the simulation-time behaviour of the complete LiM system, and by the top-level analyzer, which carries out its analysis and synthesis tasks to generate the design and to configure its UVM testbench. A positive simulation outcome is an immediate proof that the structural description functionalities are able to handle all different types of array interconnections, implementing the data movement patterns required by a LiM array.

10.2 Numerical results

The introduction to Chapter 10 has indicated the need to verify if the estimates produced by DExIMA Backend are reasonable. Before presenting the actual results, it is important to address a few points.

The underlying component models in DExIMA Backend are currently based on

the NanGate 45 nm library. As a consequence, the synthesis process in Synopsys Design Compiler should refer to this library as well. Since all DExIMA Backend models discard any cell-level parasitic, a parasitic-free variant of the NanGate 45 nm is used in the synthesis process. This approach has a major limitation: Synopsys Design Compiler may not estimate the area occupation of the synthesized netlist. For this reason, only the static power (P_{stat}), the dynamic power (P_{dyn}) and the critical path (T_{cp}) will be involved in the comparison.

All the results presented in Section 10.2 refer only to the actual LiM arrays and not to the complete LiM system. This scope-related choice is motivated by the fact that most of the new DExIMA CAD capabilities derive from the array interconnections module, which only modifies the internal structure of a LiM array and not that of the complete architecture. Moreover, both Synopsys Design Compiler and DExIMA Backend will provide worst-case power consumption estimates, which discard the actual switching activity in the LiM array.

A set of comparisons is presented in tabular form in Table 10.1, Table 10.2, Table 10.3, Table 10.4, Table 10.5 and Table 10.6. The same numerical data are also depicted by the radar plots presented in Figure 10.2, Figure 10.3, Figure 10.4, Figure 10.5, Figure 10.6 and Figure 10.7.

For all presented radar plots, the left-hand side reports the estimates produced by Synopsys Design Compiler, while the right-hand side reports the results of DExIMA Backend: in both cases, a triangle is constructed, whose vertexes are the produced numerical values. The area of each of two the triangles does not have a meaning *per se*, but the shape and the area difference is a visual indication of how different the estimates of DExIMA Backend are from those of Synopsys Design Compiler.

	Synopsys Design Compiler	DExIMA Backend	Variation
T_{cp}	0.627 ns	0.785 ns	+25.17%
P_{stat}	1.624 mW	3.233 mW	+99.11%
P_{dyn}	1.518 mW	4.271 mW	+181.31%

Table 10.1: LiM FIR array: comparison between the main figures of merit produced by Synopsys Design Compiler and DExIMA Backend.

	Synopsys Design Compiler	DExIMA Backend	Variation
T_{cp}	0.234 ns	0.212 ns	-9.25%
P_{stat}	71.672 µW	95.434 µW	+33.15%
P_{dyn}	120.938 µW	136.179 µW	+12.60%

Table 10.2: LiM ones counter array: comparison between the main figures of merit produced by Synopsys Design Compiler and DExIMA Backend.

	Synopsys Design Compiler	DExIMA Backend	Variation
T_{cp}	2.144 ns	2.769 ns	+29.13%
P_{stat}	7.994 mW	14.481 mW	+81.14%
P_{dyn}	13.533 mW	19.955 mW	+47.45%

Table 10.3: Hybrid-SIMD array: comparison between the main figures of merit produced by Synopsys Design Compiler and DExIMA Backend.

	Synopsys Design Compiler	DExIMA Backend	Variation
T_{cp}	1.001 ns	0.775 ns	-22.56%
P_{stat}	4.476 mW	5.187 mW	+15.89%
P_{dyn}	8.729 mW	7.713 mW	-11.63%

Table 10.4: LiM SHA-1 array: comparison between the main figures of merit produced by Synopsys Design Compiler and DExIMA Backend.

The obtained numerical results show an acceptable compatibility between the estimates produced by Synopsys Design Compiler and by DExIMA Backend. The latter always underestimates or overestimates the target quantities, but in most cases the variation with respect to Synopsys Design Compiler is acceptable (e.g. less than fifty percent in absolute value). This consideration is particularly fit to limited-complexity LiM arrays, e.g. the LiM ones counter array or the LiM SHA-1 array, while more complex structures, e.g. the Hybrid-SIMD array, may cause larger deviations in the power estimation process. In truth, dynamic power consumption is almost always largely overestimated: the reason for this behaviour lies in the way DExIMA Backend computes this quantity and is not related to the new DExIMA CAD functionalities.

It is important to highlight that the mutual interaction between DExIMA CAD and DExIMA Backend is quite effective in identifying the critical path in the LiM array, even after array interconnections are integrated in the DExIMA Backend description. In fact, among the three target quantities, the critical path T_{cp} is consistently well-estimated by DExIMA Backend.

	Synopsys Design Compiler	DExIMA Backend	Variation
T_{cp}	0.372 ns	0.551 ns	+48.03%
P_{stat}	2.389 mW	2.182 mW	-8.69%
P_{dyn}	2.177 mW	4.811 mW	+121.03%

Table 10.5: LiM AES-128 array, `StateRoundKey1` solution: comparison between the main figures of merit produced by Synopsys Design Compiler and DExIMA Backend.

	Synopsys Design Compiler	DExIMA Backend	Variation
T_{cp}	0.372 ns	0.551 ns	+48.03%
P_{stat}	2.219 mW	1.986 mW	-10.49%
P_{dyn}	1.900 mW	3.974 mW	+109.22%

Table 10.6: LiM AES-128 array, `StateKey1` solution: comparison between the main figures of merit produced by Synopsys Design Compiler and DExIMA Backend.

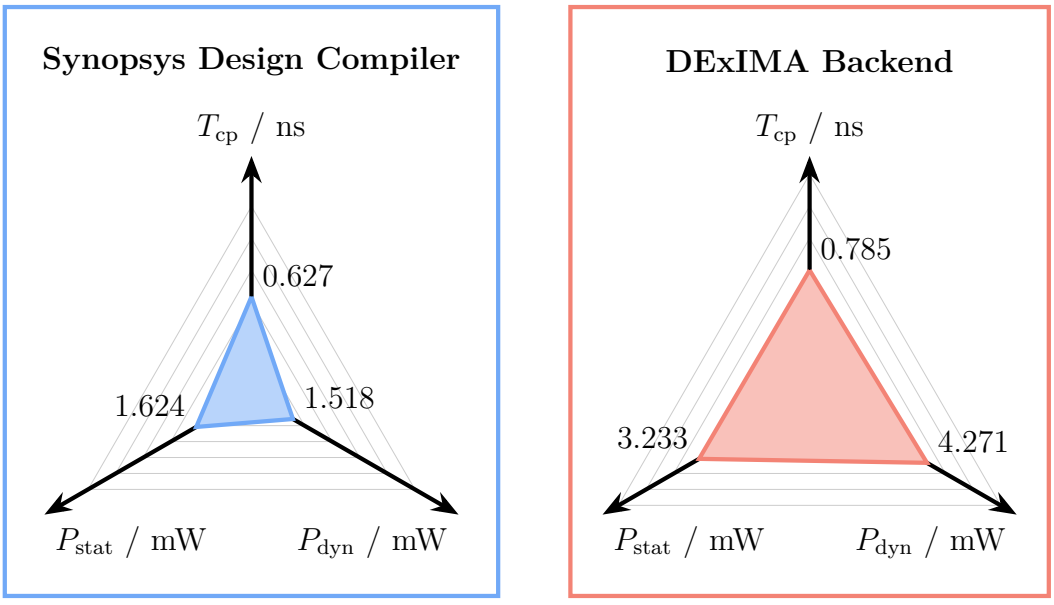


Figure 10.2: Visual representation of the comparison between Synopsys Design Compiler and DExIMA Backend for the LiM FIR array. T_{cp} is the critical path, P_{stat} is the static power and P_{dyn} is the dynamic power.

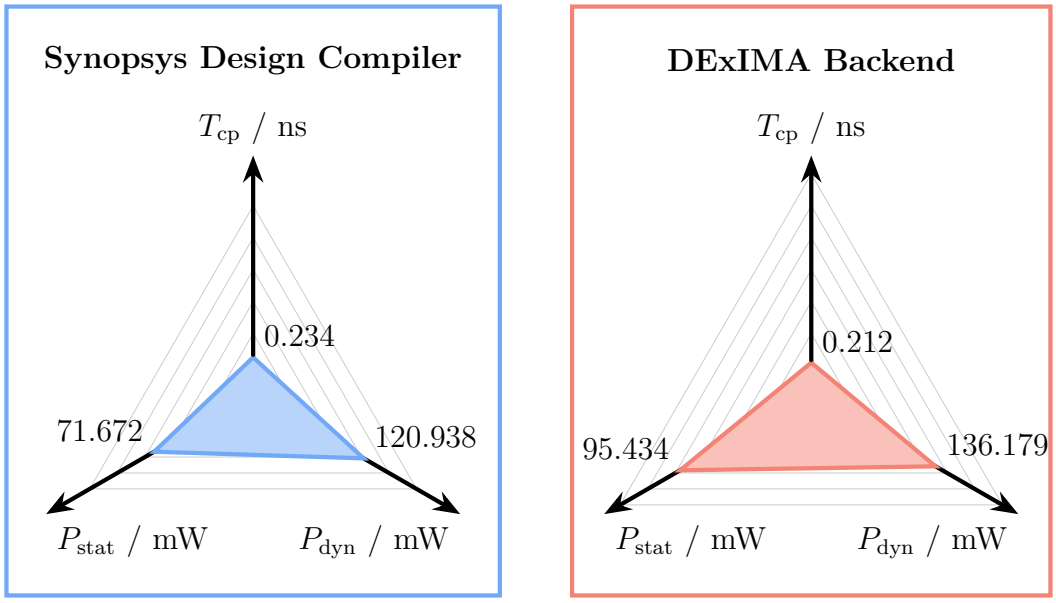


Figure 10.3: Visual representation of the comparison between Synopsys Design Compiler and DExIMA Backend for the LiM ones counter array. T_{cp} is the critical path, P_{stat} is the static power and P_{dyn} is the dynamic power.

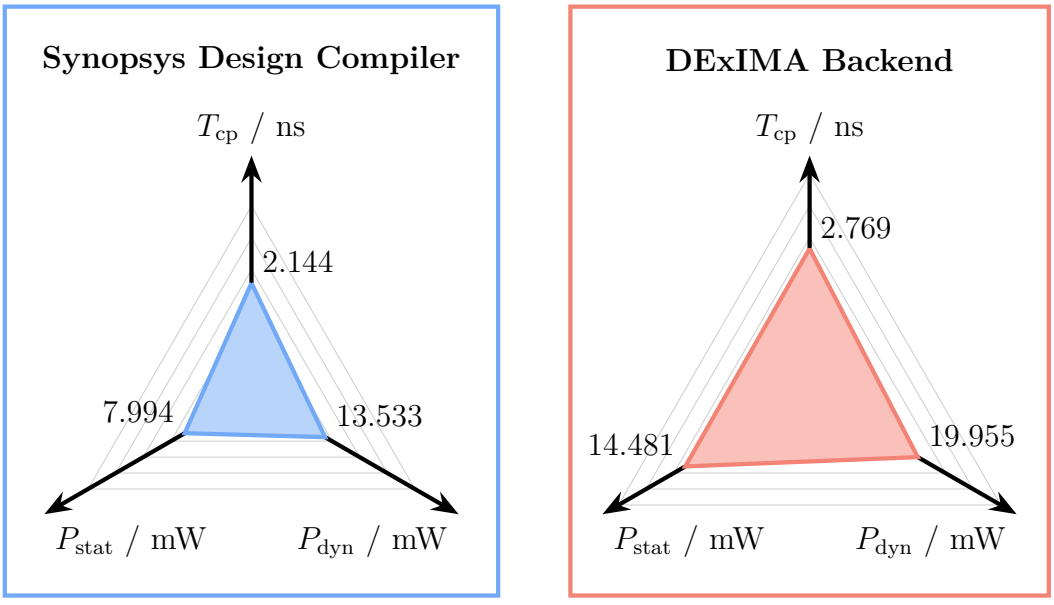


Figure 10.4: Visual representation of the comparison between Synopsys Design Compiler and DExIMA Backend for the Hybrid-SIMD array. T_{cp} is the critical path, P_{stat} is the static power and P_{dyn} is the dynamic power.

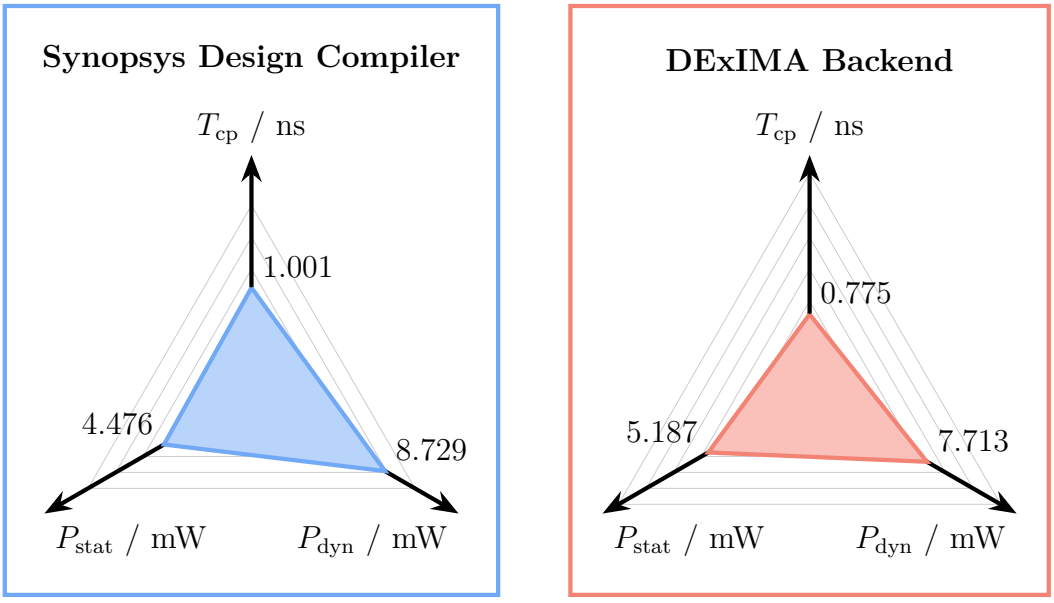


Figure 10.5: Visual representation of the comparison between Synopsys Design Compiler and DExIMA Backend for the SHA-1 array. T_{cp} is the critical path, P_{stat} is the static power and P_{dyn} is the dynamic power.

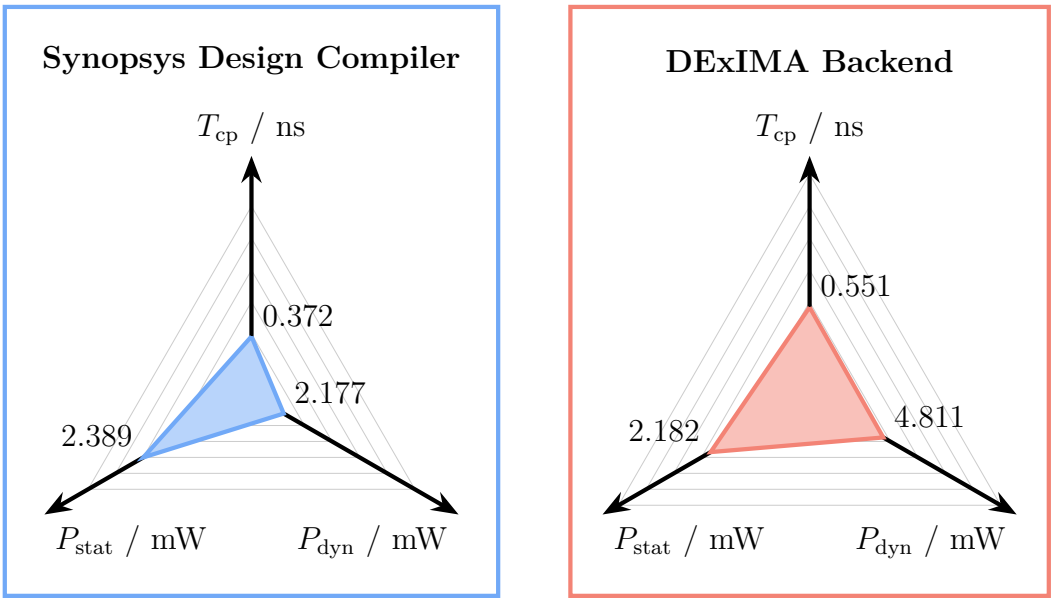


Figure 10.6: Visual representation of the comparison between Synopsys Design Compiler and DExIMA Backend for the LiM AES-128 array, `StateRoundKey1` solution. T_{cp} is the critical path, P_{stat} is the static power and P_{dyn} is the dynamic power.

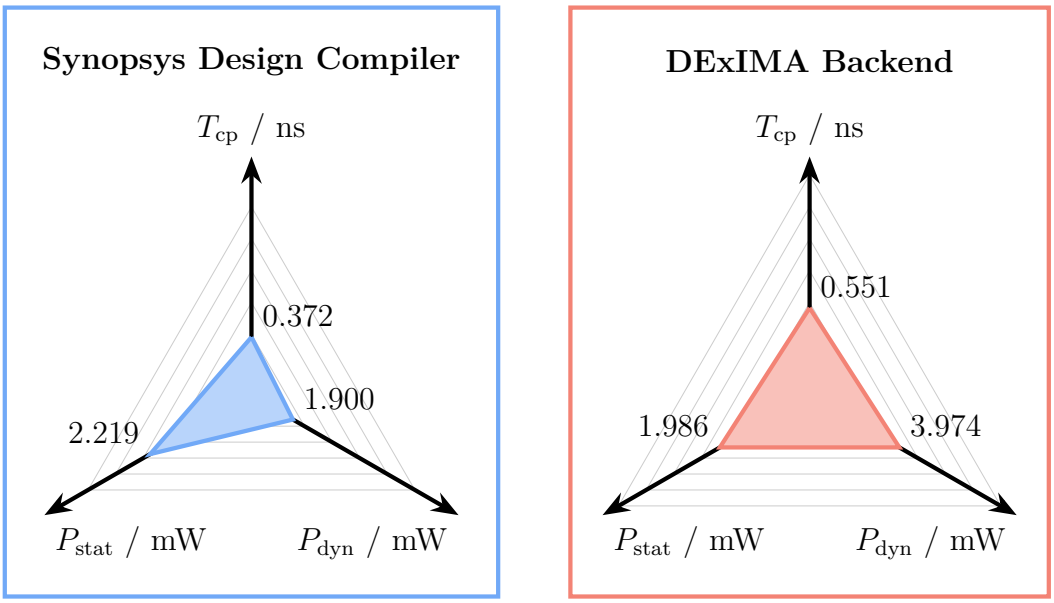


Figure 10.7: Visual representation of the comparison between Synopsys Design Compiler and DExIMA Backend for the LiM AES-128 array, `StateKey1` solution. T_{cp} is the critical path, P_{stat} is the static power and P_{dyn} is the dynamic power.

Chapter 11

Conclusions and future developments

The structural description capabilities of DExIMA CAD have been largely enriched by the introduction of array interconnections. The implementation of arbitrary data movement patterns is vital to the functionality of a LiM array, and array interconnections can efficiently answer this specific need, as thoroughly proved by all proposed implementations.

The revisited DExIMA CAD is endowed with more flexible architectural exploration capabilities. With respect to its past version, it may now be used to explore a larger variety of architectural solutions, thanks to the possibility of allocating multiple LiM arrays and other arbitrarily-complex sub-systems. The large variability of the uppermost architectural level is supported by the top-level analyzer with its analysis and synthesis tasks, which include, but are not limited to, the automatic configuration of a UVM testbench and the generation of the HDL code for the complete system. The top-level simulation dashboard and the new version of the algorithm description module ease the simulation process and embed it in the toolchain environment.

All presented LiM structures would have been unattainable by the previous version of DExIMA CAD. Their implementation highlights the increased breadth of the structural description functionalities and the architectural exploration capabilities of DExIMA CAD, which has become more versatile and efficient in supporting the implementation of different LiM systems and algorithms. For all proposed LiM structures, the DExIMA CAD estimates are compatible with the figures of merit produced by Synopsys Design Compiler, further proving the effectiveness of the tool in handling a variety of LiM structures.

Lots of future developments may further increase the flexibility of DExIMA CAD. For instance, all introduced modules have been conceived to depend as little as possible on the GUI overlay, with the primary concern of fostering future

scripting functionalities, which may automatize specific description tasks or enable parametric performances analyses. Moreover, DExIMA CAD has been so far using a one-dimensional memory model, in which the word length is equal to the number of columns in the array. As a further development, a two-dimensional model may be introduced, where the word length may be different from the number of array columns. Such implementation is expected to move from a purely functional to a more structural memory model, which is better suited to describe more complex LiM arrays. Should this model be implemented, an additional dimension must be introduced in the management of array interconnections, and the algorithm description module should be slightly revisited, in order to guarantee the required degree of SIMD processing within the LiM array.

With the developments presented in this thesis, the revisited DExIMA CAD is more versatile in the description of a large variety of LiM structures and in the implementation of several algorithms. It offers a valid support to the LiM design flow, helping the exploration of increasingly complex LiM systems.

Appendix A

Hybrid-SIMD architecture

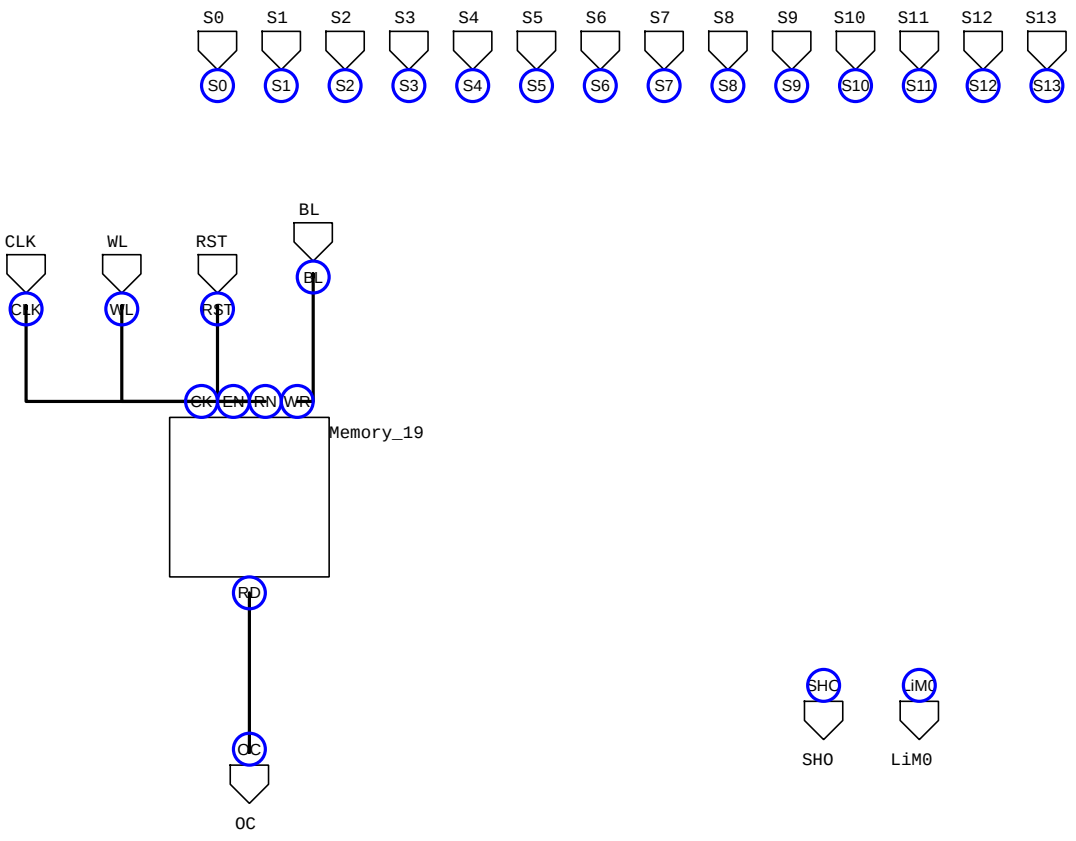


Figure A.1: Hybrid-SIMD architecture: DExIMA CAD implementation of the standard memory cells.

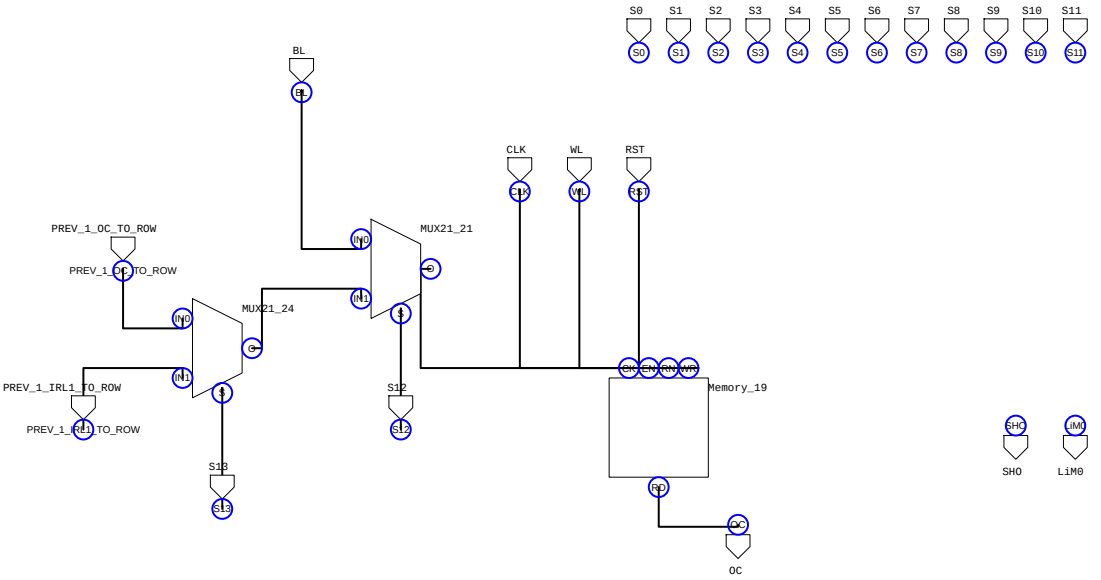


Figure A.2: Hybrid-SIMD architecture: DExIMA CAD implementation of the semi-standard memory cells.

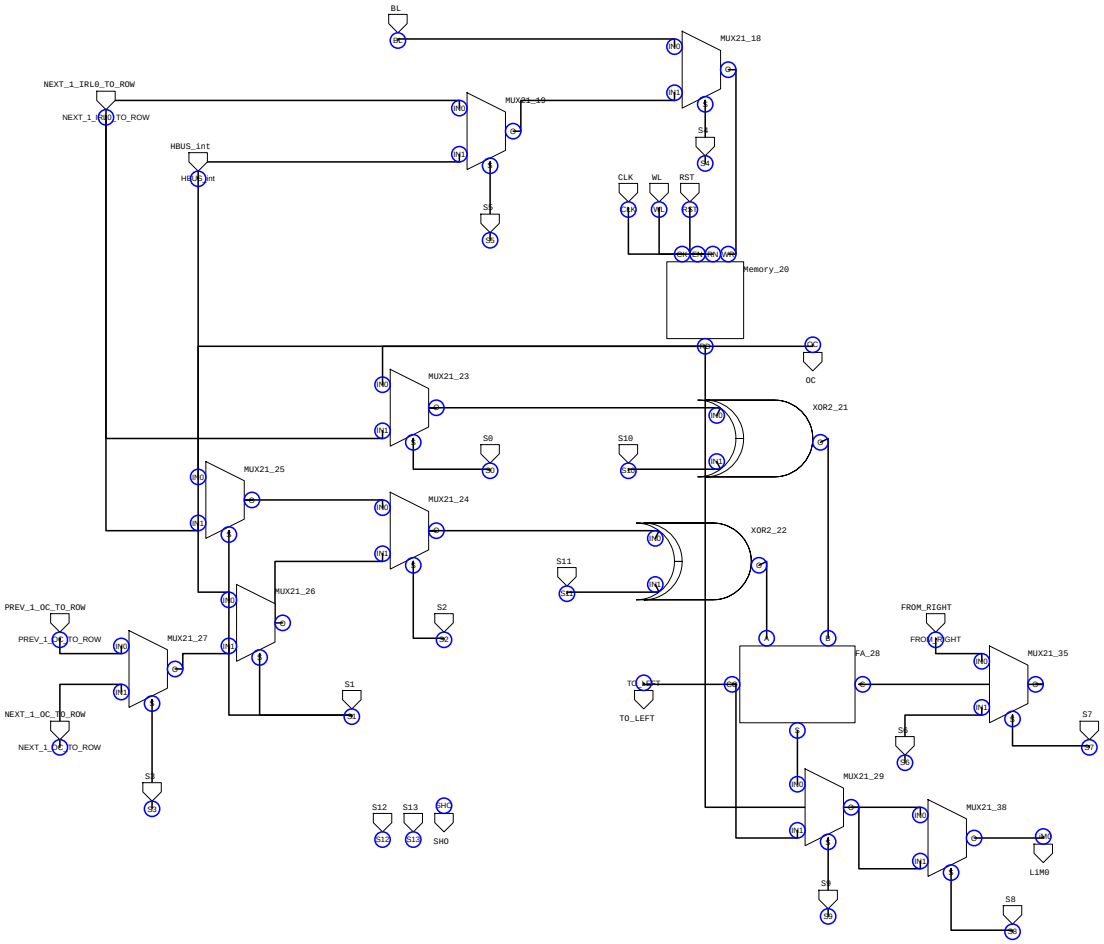


Figure A.3: Hybrid-SIMD architecture: DExIMA CAD implementation of the arithmetic/logic LiM cells.

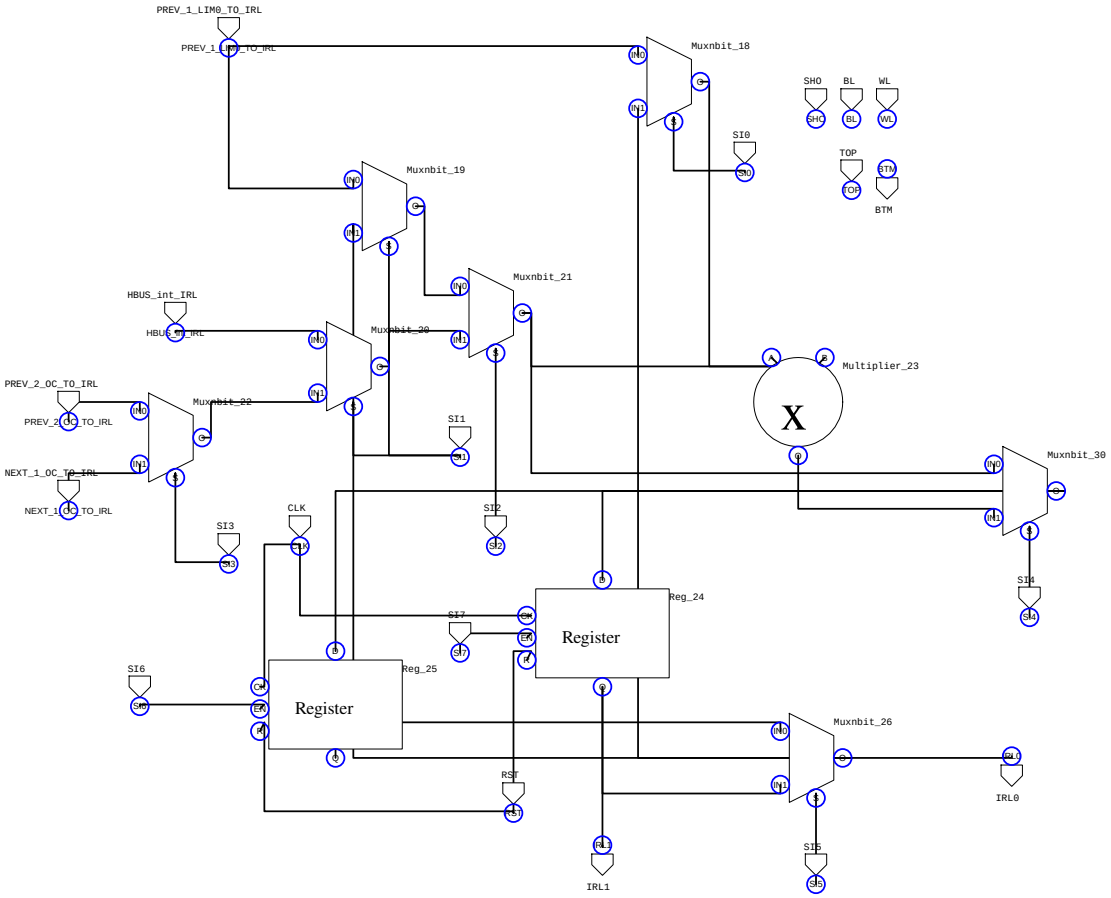


Figure A.4: Hybrid-SIMD architecture: DExIMA CAD implementation of the IRL blocks.

Appendix B

LiM SHA-1 architecture

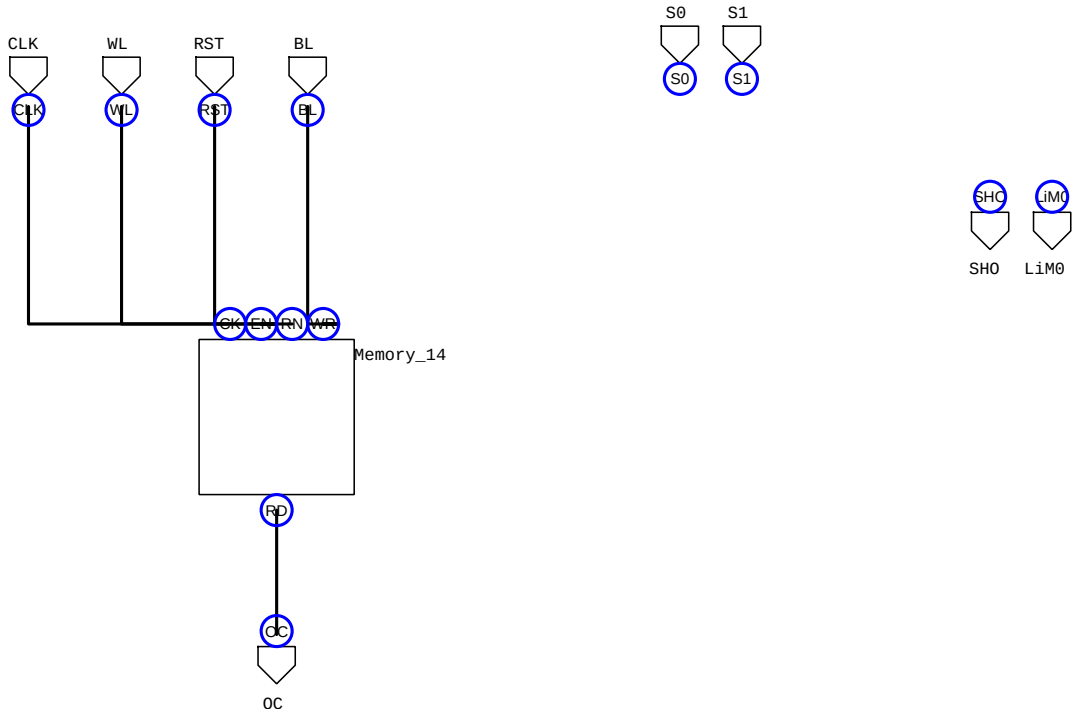


Figure B.1: LiM SHA-1 architecture: DExIMA CAD implementation of the LiM cells for rows 0 to 15.

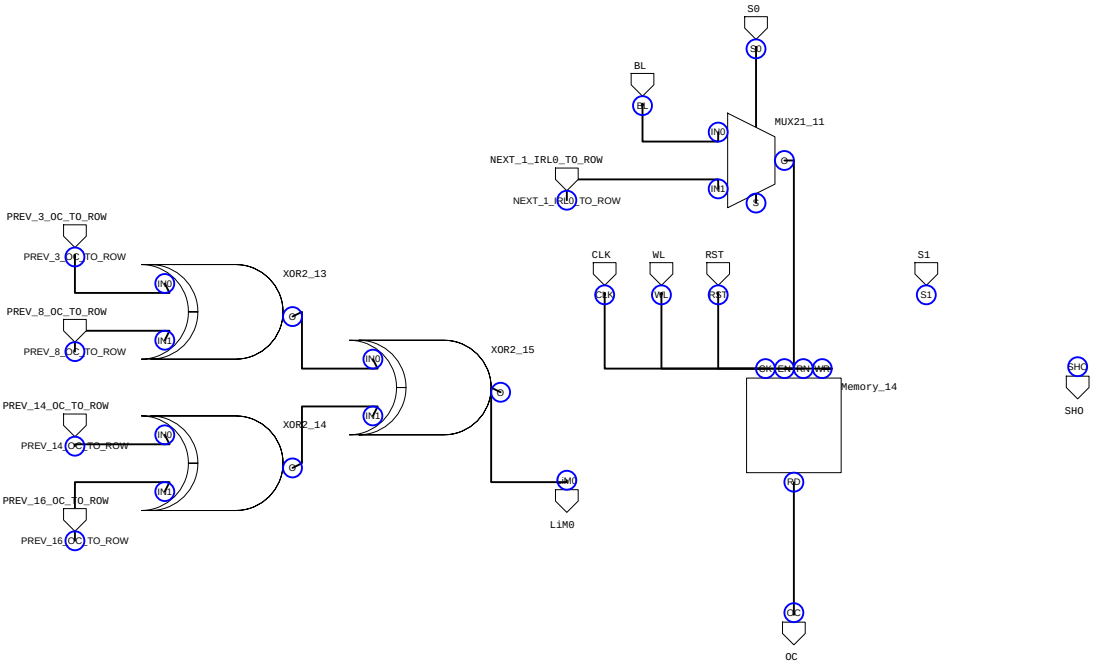


Figure B.2: LiM SHA-1 architecture: DExIMA CAD implementation of the LiM cells for rows 16 to 79.

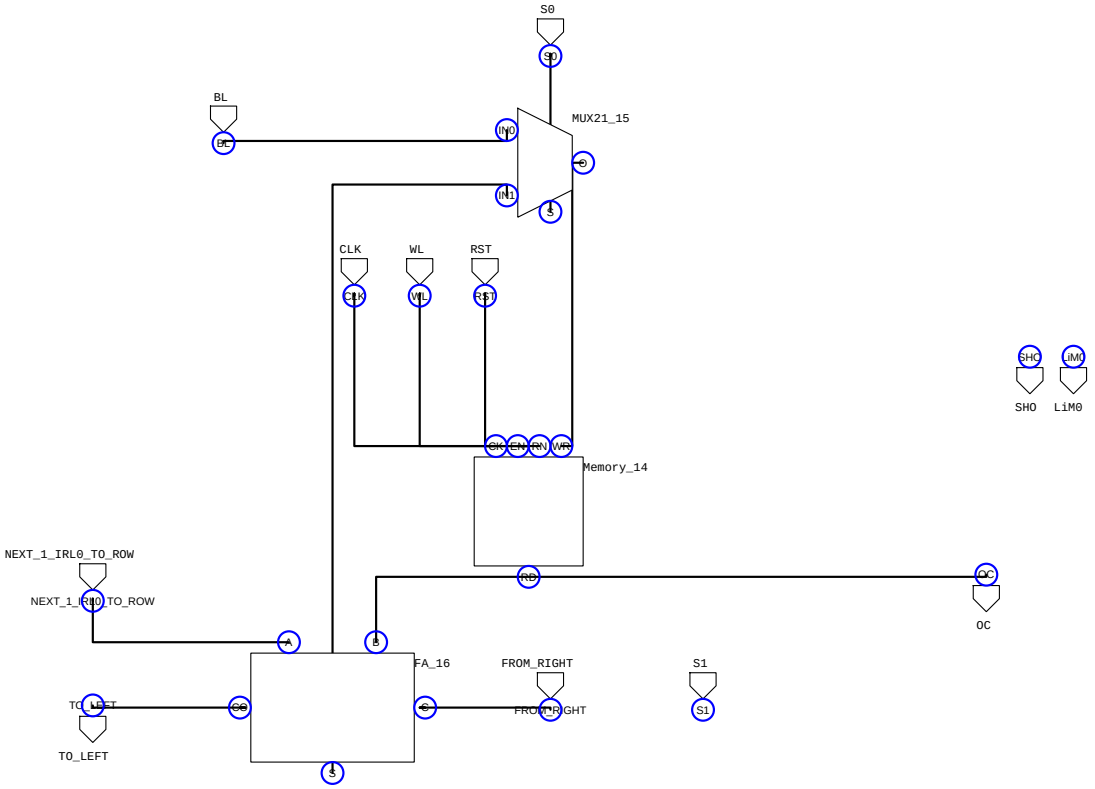


Figure B.3: LiM SHA-1 architecture: DExIMA CAD implementation of the LiM cells for rows 80 to 84.

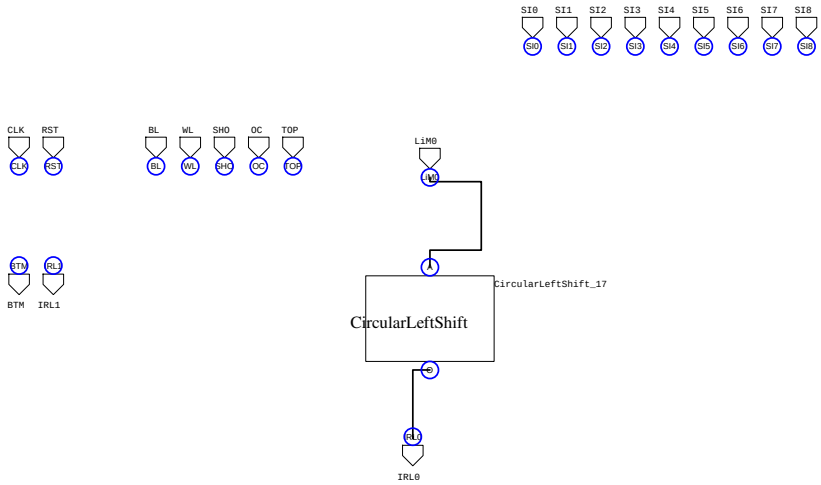


Figure B.4: LiM SHA-1 architecture: DExIMA CAD implementation of the IRL blocks for rows 16 to 79.

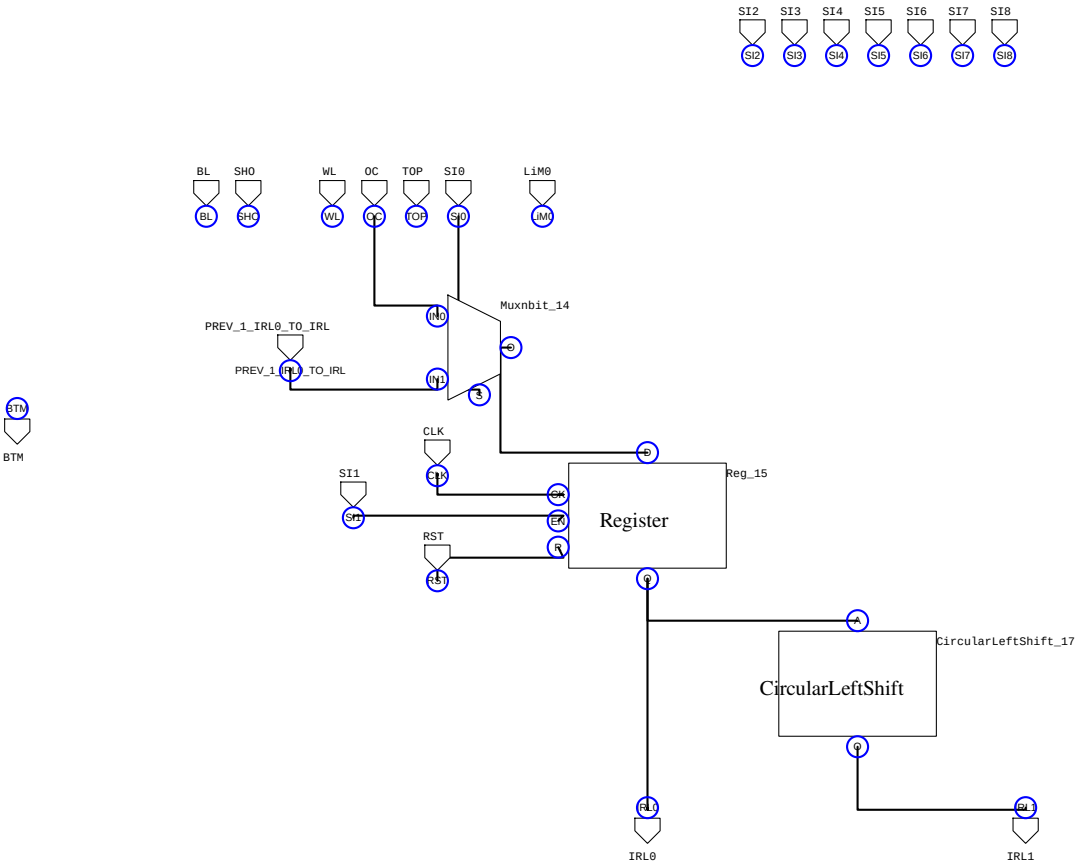


Figure B.5: LiM SHA-1 architecture: DExIMA CAD implementation of the IRL blocks for row 81.

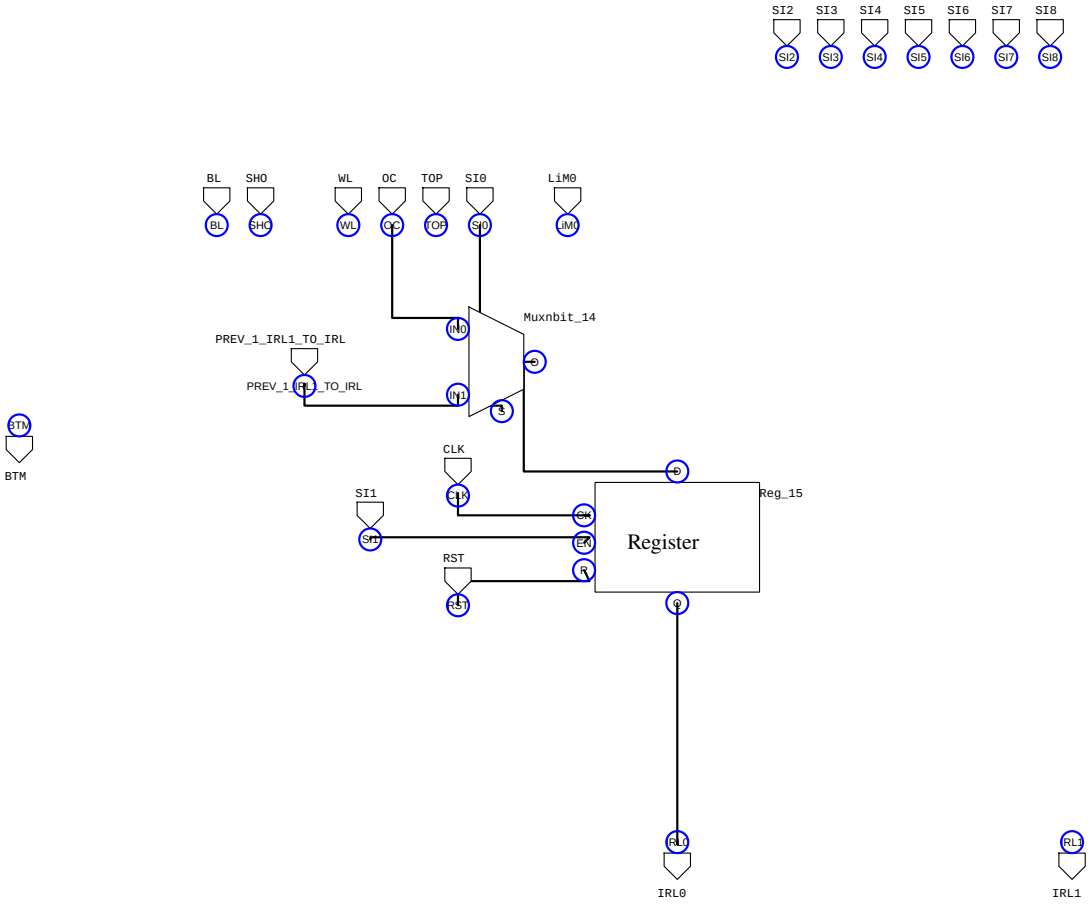


Figure B.6: LiM SHA-1 architecture: DExIMA CAD implementation of the IRL blocks for row 82.

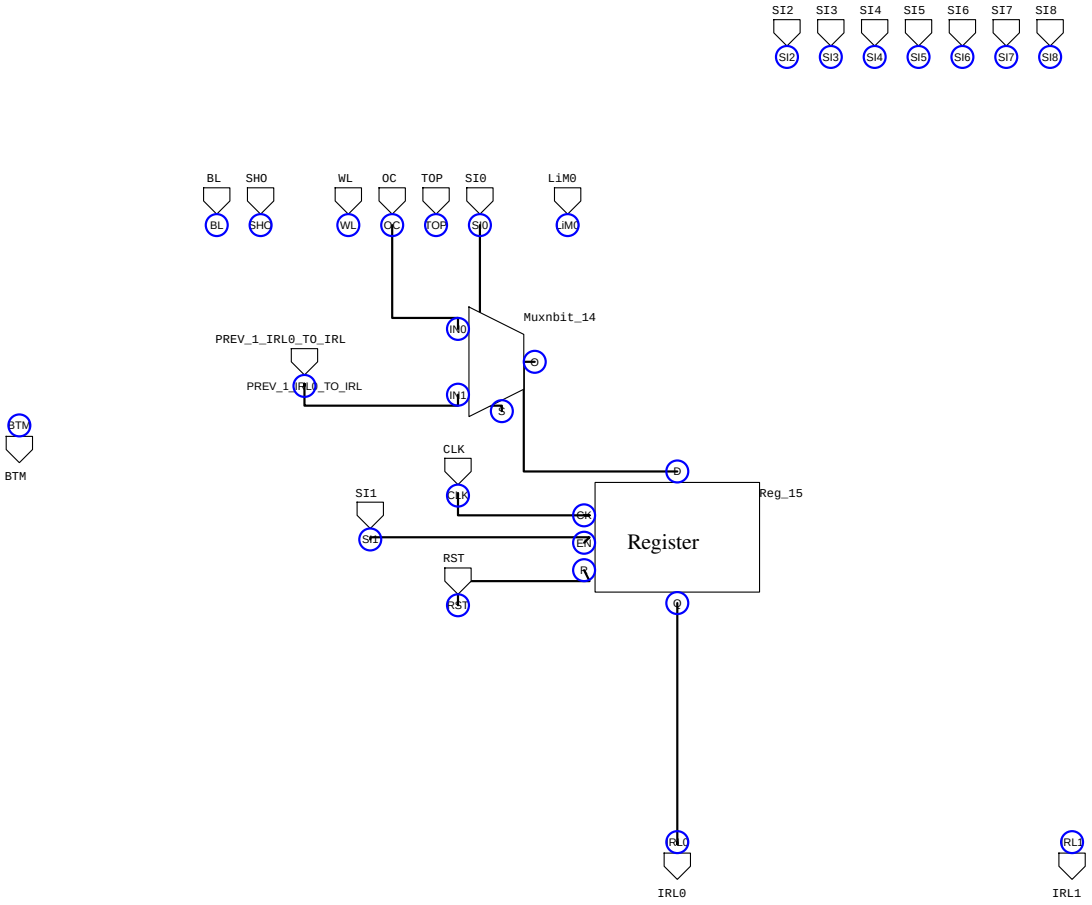


Figure B.7: LiM SHA-1 architecture: DExIMA CAD implementation of the IRL blocks for rows 83 to 84.

Appendix C

AES-128 architecture

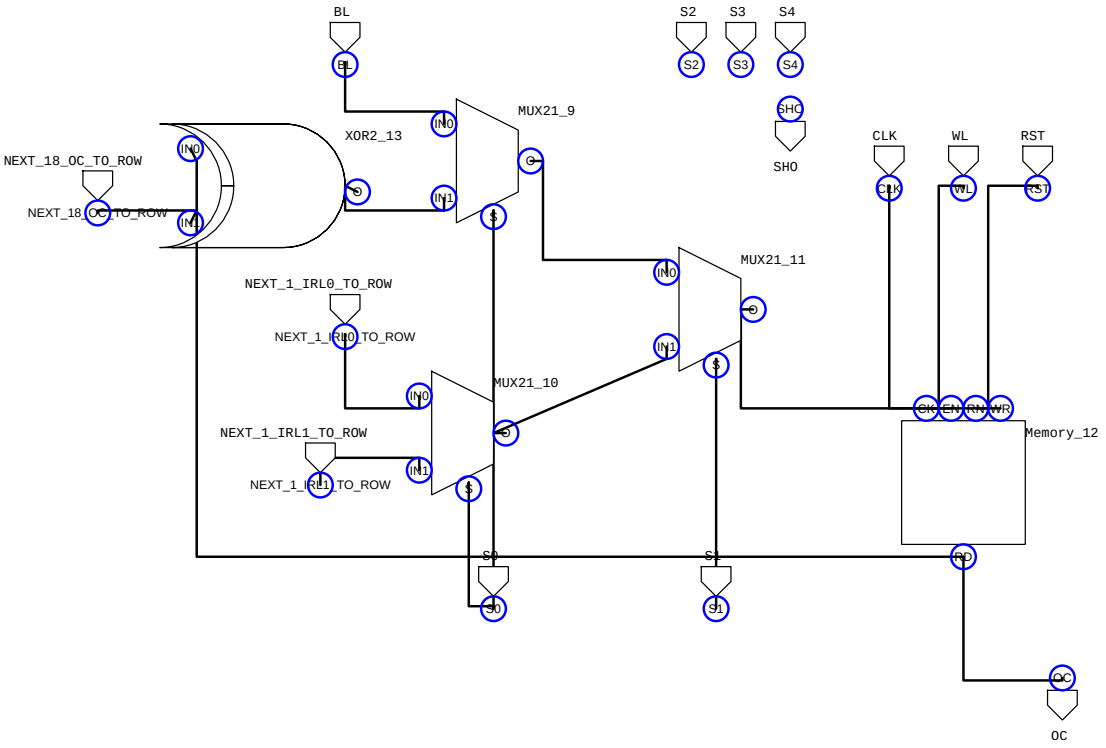


Figure C.1: AES-128 architecture, `StateRoundKey1` solution: DExIMA CAD implementation of the LiM cell presented in Figure 9.9, type `cell_R0`.

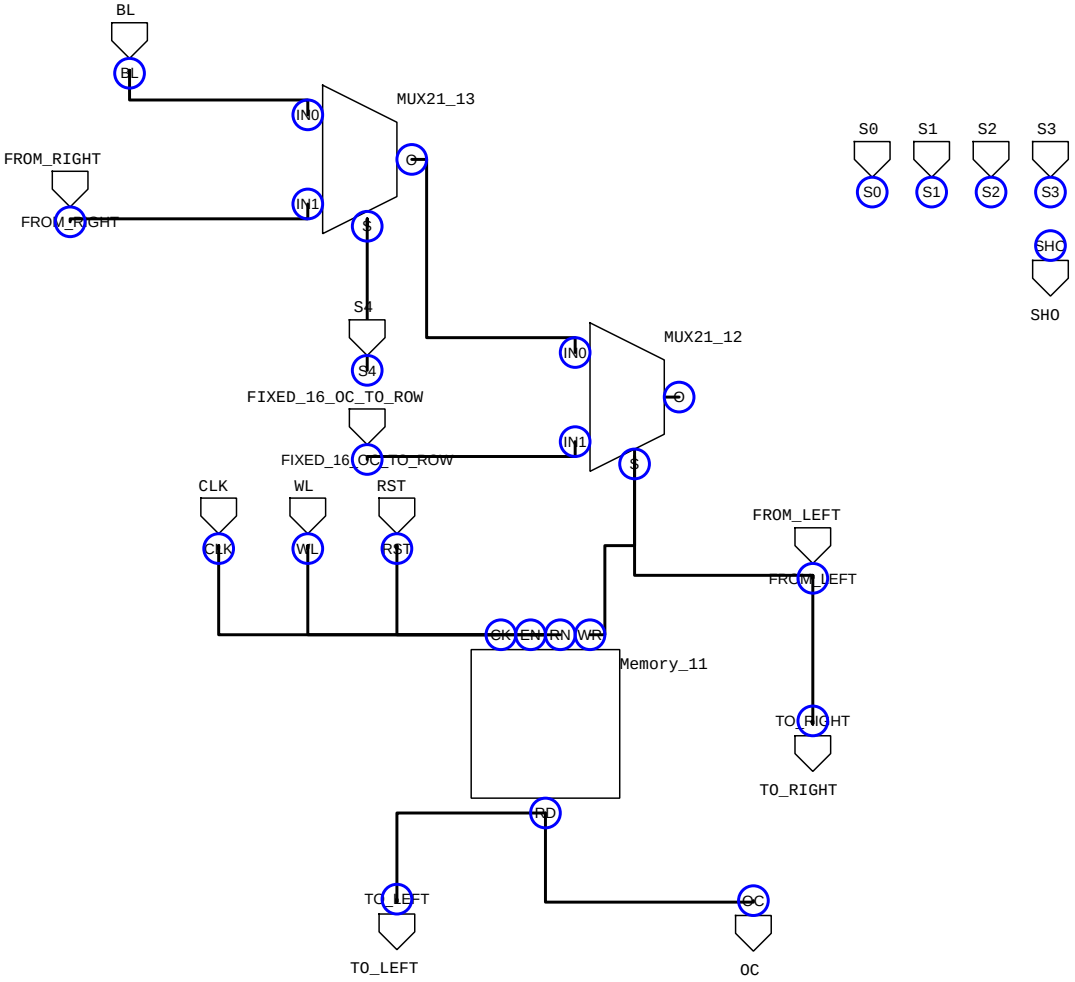


Figure C.2: AES-128 architecture, `StateRoundKey1` solution: DExIMA CAD implementation of the LiM cells presented in Figure 9.15.

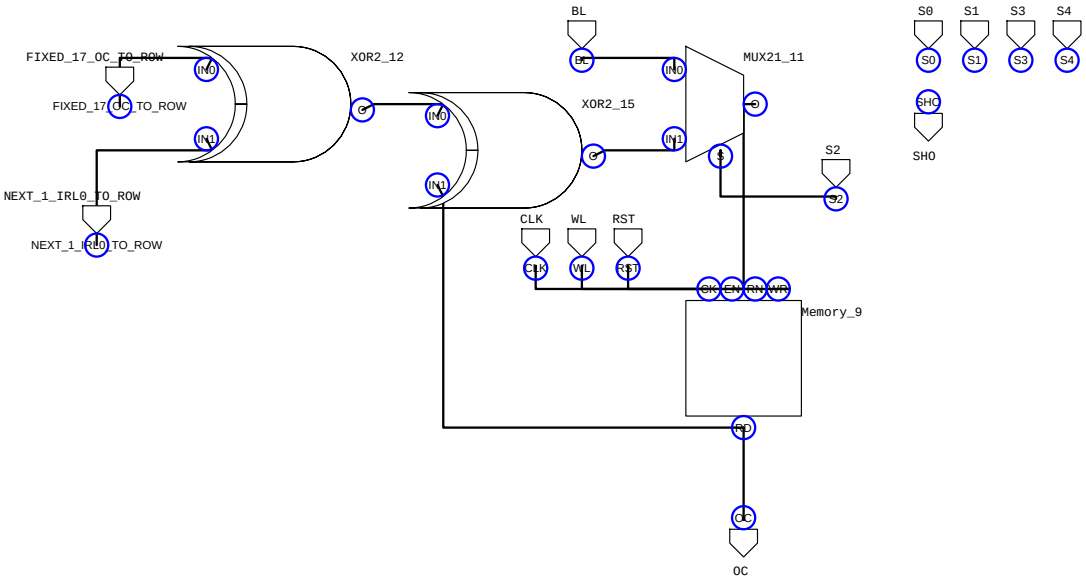


Figure C.3: AES-128 architecture, `StateRoundKey1` solution: DExIMA CAD implementation of the LiM cell presented in Figure 9.12.

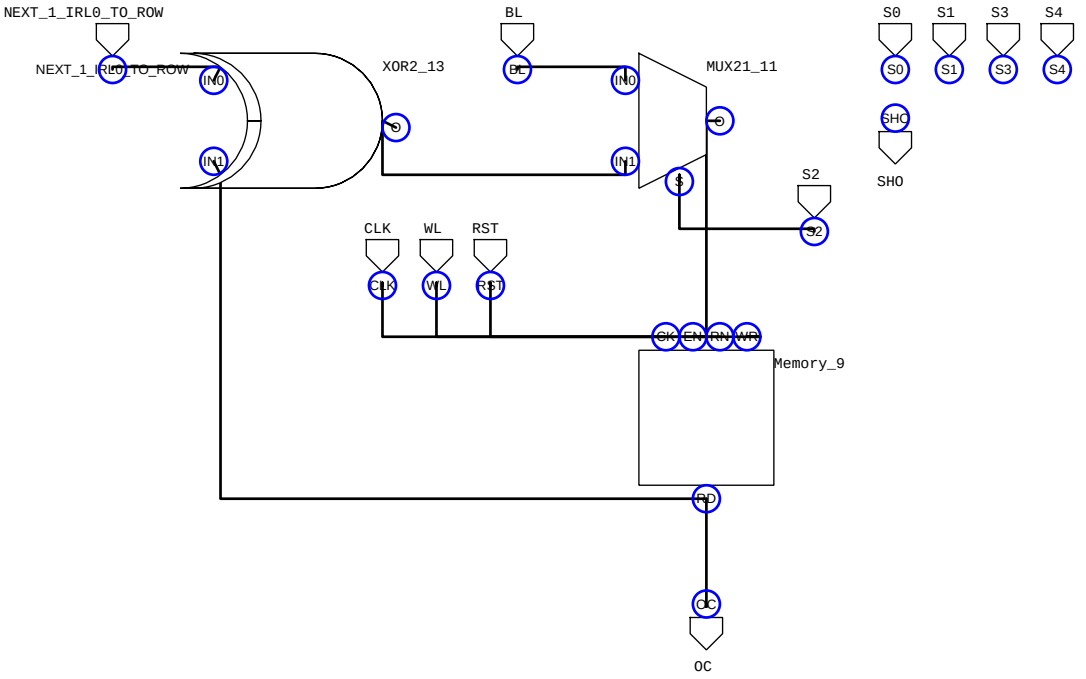


Figure C.4: AES-128 architecture, `StateRoundKey1` solution: DExIMA CAD implementation of the LiM cell presented in Figure 9.13.

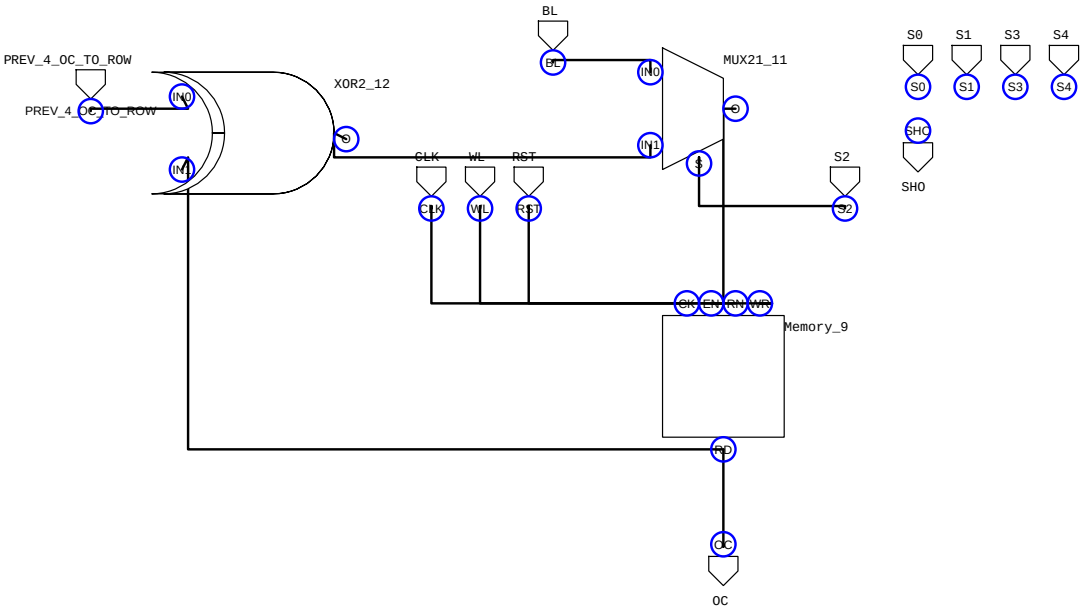


Figure C.5: AES-128 architecture, `StateRoundKey1` solution: DExIMA CAD implementation of the LiM cell presented in Figure 9.14.

Acronyms

AES Advanced Encryption Standard

ASIC Application-Specific Integrated Circuit

BNN Binary Neural Network

CiM Computation-in-Memory

CLiMA Configurable Logic-in-Memory Architecture

CnM Computation-near-Memory

CNN Convolutional Neural Network

CPU Central Processing Unit

CSV Comma-Separated Values

CwM Computation-with-Memory

DFS Depth-First Search

DUT Device Under Test

EDA Electronic Design Automation

FA Full-Adder

FIR Finite Impulse Response

GUI Graphical User Interface

HDL Hardware Description Language

HLS High-Level Synthesis

I/O Input/Output

IRL Intra-Row Logic

LHS Left-Hand Side

LiM Logic-in-Memory

LSB Least-Significant Bit

LUT Look-Up Table

MSB Most-Significant Bit

MVM Matrix-Vector Multiplication

RCA Ripple-Carry Adder

RHS Right-Hand Side

ROM Read-Only Memory

SHA Secure Hash Algorithm

SIMD Single-Instruction, Multiple-Data

UVM Universal Verification Methodology

VHDL Very high speed integrated circuits Hardware Description Language

Bibliography

[1] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. "CACTI 6.0: A Tool to Model Large Caches". (209), [Online]. Available: https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf.

[2] H. S. Stone, "A Logic-in-Memory Computer", *IEEE Transactions on Computers*, vol. C-19, no. 1, pp. 73–78, 1970. DOI: 10.1109/TC.1970.5008902.

[3] N. I. of Standards and Technology, "Secure Hash Standard", U.S. Department of Commerce, Washington, D.C., Tech. Rep., 1993. DOI: 10.6028/NIST.FIPS.180.

[4] N. I. of Standards and Technology, "Advanced Encryption Standard (AES)", U.S. Department of Commerce, Washington, D.C., Tech. Rep., 2001. DOI: 10.6028/NIST.FIPS.197.

[5] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX", in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[6] N. I. of Standards and Technology, "Secure Hash Standard (SHS)", U.S. Department of Commerce, Washington, D.C., Tech. Rep., 2015. DOI: 10.6028/NIST.FIPS.180-4.

[7] M. Xie, S. Li, A. O. Glova, J. Hu, and Y. Xie, "Securing Emerging Nonvolatile Main Memory With Fast and Energy-Efficient AES In-Memory Implementation", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 11, pp. 2443–2455, 2018. DOI: 10.1109/TVLSI.2018.2865133.

[8] N. Piano. "DExIMA: A Design Explorer for In-Memory Architectures". (2019), [Online]. Available: http://webthesis.biblio.polito.it/id/eprint/12547.

[9] G. Santoro, G. Turvani, and M. Graziano, "New Logic-In-Memory Paradigms: An Architectural and Technological Perspective", *Micromachines*, vol. 10, no. 6, 2019, ISSN: 2072-666X. DOI: 10.3390/mi10060368. [Online]. Available: https://www.mdpi.com/2072-666X/10/6/368.

[10] M. Andrighetti, G. Turvani, G. Santoro, *et al.*, "Data Processing and Information Classification—An In-Memory Approach", *Sensors*, vol. 20, no. 6, 2020, ISSN: 1424-8220. DOI: 10.3390/s20061681. [Online]. Available: https://www.mdpi.com/1424-8220/20/6/1681.

[11] A. Coluccio, M. Vacca, and G. Turvani, "Logic-in-Memory Computation: Is It Worth It? A Binary Neural Network Case Study", *Journal of Low Power Electronics and Applications*, vol. 10, no. 1, 2020, ISSN: 2079-9268. DOI: 10.3390/jlpea10010007. [Online]. Available: https://www.mdpi.com/2079-9268/10/1/7.

[12] J. Lowe-Power, A. M. Ahmad, A. Akram, *et al.*, "The gem5 simulator: Version 20.0+", *CoRR*, vol. abs/2007.03152, 2020. arXiv: 2007.03152. [Online]. Available: https://arxiv.org/abs/2007.03152.

[13] A. Marchesin. "Octantis - A High-Level Explorer for Logic-in-Memory architectures". (2020), [Online]. Available: http://webthesis.biblio.polito.it/id/eprint/15852.

[14] A. Marchesin, G. Turvani, A. Coluccio, *et al.*, "Octantis: An Exploration Tool for Beyond von Neumann architectures", in *2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2021, pp. 1–5. DOI: 10.1109/DTIS53253.2021.9505135.

[15] L. Mendola. "DExIMA A synthesis tool and performance estimator for Logic-in-Memory architectures". (2021), [Online]. Available: http://webthesis.biblio.polito.it/id/eprint/17852.

[16] A. Coluccio, U. Casale, A. Guastamacchia, *et al.*, "Hybrid-SIMD: A Modular and Reconfigurable Approach to Beyond von Neumann Computing", *IEEE Transactions on Computers*, vol. 71, no. 9, pp. 2287–2299, 2022. DOI: 10.1109/TC.2021.3127354.

[17] A. Coluccio, A. Ieva, F. Riente, M. R. Roch, M. Ottavi, and M. Vacca, "RISC-Vlim, a RISC-V Framework for Logic-in-Memory Architectures", *Electronics*, vol. 11, no. 19, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11192990. [Online]. Available: https://www.mdpi.com/2079-9292/11/19/2990.

[18] A. Naclerio. "HLS techniques for high performance parallel codes in Logic-in-Memory systems." (2022), [Online]. Available: http://webthesis.biblio.polito.it/id/eprint/22828.