POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

DExIMA
A synthesis tool and performance estimator for Logic-in-Memory architectures

Supervisors
Prof. Maurizio Zamboni
Prof.ssa Mariagrazia Graziano
Ph.D Giovanna Turvani

Candidate
Loris Mendola
ID: 263630

Academic Year 2020-2021

Summary

In this thesis, we will produce a detailed analysis of the tool called DExIMA, from the high-level description to the more detailed one. The tool was previously developed by Nicola Piano [6] in its first version. In this thesis, we will analyze the first version of the program, and we will successively rewrite it completely from zero, changing the structure and the modes of computations of the program. All this work is done to improve several aspects of the code and the interface with the user. Developed in C++, DExIMA word stands for ***D**esign-**Ex**plorer for **I**n-**M**emory **A**rchitecture*, because it is used to explore the solution spaces of the possible Logic-in-Memory (LiM) approach. The first part of this thesis is dedicated to the description of what a LiM is, and how it works, considering the state of the art of this technology and its applications. After the explanation of LiM concept, there is a chapter describing the motivation of this thesis and the general characteristics of DExIMA. Following the general chapters, there are the ones related to the tool. The core of the thesis is dived into the following parts:

DExIMA: This chapter is dedicated to the architecture of the program and explains to the user which are the external components involved in the program, and to understand how to approach it.

DExIMA Language: This chapter is a guide for the user since it explains the syntax used in the configuration file of DExIMA.

DExIMA Files Descriptions: This chapter describes the fields and the information of the output files of DExIMA and how to interpret it.

DExIMA Hardware Models: This chapter describes in detail the models of the gates realized in DExIMA from a low-level point of view.

DExIMA Data Structure: This chapter is related to the class structure of the program and the function that these classes have inside the program.

Compilation Process: This chapter describes how the Compilation process involves, and the event triggered when it parses the input configuration file.

Performance Computation: This last chapter is related to how DExIMA works and explains how the performance is computed inside the tool.

The last part of this thesis is dedicated to some results obtained from the tool, from a general point of view, and a specific case of study implementing a Binary Neural Network. In the end, there is a chapter dedicated to the future improvement of the tool. There is also an Appendix with some useful data related to the language and the components library.

Contents

List of Figures

Chapter 1

State of the Art

1.1 Introduction

Today the digital integrated circuits take several improvements in terms of performance, especially in the last decades. This is due to the new technologies and the scaling of our circuit. The scaling improves several characteristics of our logic gates like the reduction of the delay, area, and power consumption. But the scaling introduces also problems related to the quantum effects to take into considerations. But the improvements increase differently in the case of memory that results slower than our computation units. As we know the *Von Neumann* paradigm is the foundation of all modern computing systems. The *Von Neumann* paradigm consists of having an architecture with two main components: the central processing unit (CPU), and Memory. These two components exchange data between themselves using a bus. Now the problem occurs, because due to the Technology improvements, the CPU began faster than the memory, making the memory unable to provide data as fast as the CPU is able to compute them. This problem is called *Von Neumann bottleneck* or *Memory Wall*. Another problem related to the *Von Neumann* architecture concerns data intensive algorithms, where the data exchange between the CPU and memory is one of the main contributions to power consumption. The bus line connected to the memory can have a high capacity load, that generates high consumptions. To avoid this problem a new approach is proposed, the Logic-in-Memory (LiM) approach. The idea is to integrate some simple logic inside the memory. Doing the computation directly inside the memory avoids the exchange of data to the CPU. This reduces the *Memory Wall*, creating more parallelizable architecture and a reduction of power consumption.

1.2 Computing approach Taxonomy

The literature regarding the computation with memory is very rich, but we can classify it by observing the role of that memory regarding the computation operation. According to [1] we can dived it into four main categories:

- **Computation-near-Memory (CnM)**
- **Computation-in-Memory (CiM)**
- **Computation-with-Memory (CwM)**
- **Logic-in-Memory (LiM)**

In Figure 1.1 these four categories are presented graphically.

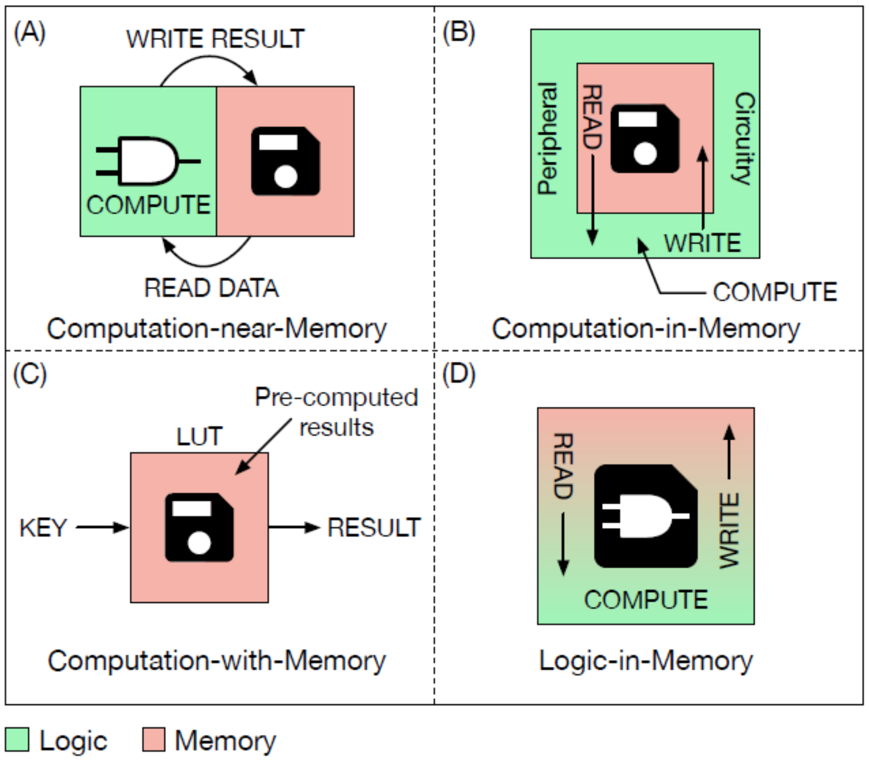


Figure 1.1: In-memory computing aproaches. (**A**) CnM (**B**) CiM (**C**) CwM (**D**) LiM. Source:[1]

Now, these different approaches are being described one by one, describing how the logic and the memory part interact between themselves.

Computation-near-Memory The Computation near Memory approach is characterized by the fact that the logic and the memory unit are well separated. The two units are very close to each other. These two units are stacked one to the top of the other in a 3D structure using the *Through Silicon Via* (TSV) technology. From a system point of view, we have no variations, this approach is involved using only the Technology. The advantage of this configuration is the bus length reduction, implying less delay and power consumption.

Computation-in-Memory This approach does not modify the logic array of the memory but exploits the logic operation using the analog part of the memory. In particular sense amplifiers of the memory are used, realizing simple logic operations like the AND or the OR. Moreover, the decoder of the memory is also modified to permit to do more than one read at a time and performs row-wise and column-wise operations (i.e. operations between different rows/columns). Also, the match lines of a CAM can be exploited to do this type of operation. The advantage of this approach is, from a technology point of view, to create a dedicated special design of the memory. On the other hand, the set of available operations is very limited. RRAM (Resistive Random-Access Memory) and MRAM (Magnetoresistive Random-Access Memory) technologies like the PIMA architecture [2] and [5] belong to this category.

Computation-with-Memory This computation approach uses the memory like a Look-up table (LUT), containing the precomputed-result inside of it. The LUTs are used to perform a boolean logic function with two or more inputs, storing its truth table. The memory is used as a Countable-Adress-Memory (CAM) to store the result of the computation. The working principle is simple: the LUT is accessed by a combination of inputs, the LUT retrieves an address that is used to access the CAM to obtain the final result. An example of this approach is exploited in [3], where a ReCAM (ReRAM-based CAM) to compute a DNA alignment is used.

Logic-in-Memory The last approach is what we are focusing on in this thesis and the philosophy of how DExIMA works. The LiM approach uses logic elements directly integrated into the cell. In this case, we have a custom memory, where each cell has a specific logic. The read and write operations can be performed locally using the logic, without moving data outside the array. This is an advantage in terms of speed and power consumption. The other advantage is the possibility to use different emerging technologies beyond the classic CMOS to construct the memory cells, like MTJ (Magnetic Tunnel junction). An example of how LiM work is in [4].

1.3 Configurable Logic-In-Memory Architecture (CLiMA)

An interesting example of a LiM approach is the CLiMA architecture that stands for *Configurable Logic-In-Memory Architecture* [1]. The main aspect related to this architecture is configurability, hence flexibility. The CLiMA architecture is composed of an in-memory computation LiM and/or CiM, the CLiM array, and a CnM unit. The usage of these three units is decided by the type of operation to perform. The operation varies based on the operation complexity and data movement. From these considerations, we decide to perform the operation inside the memory or not.

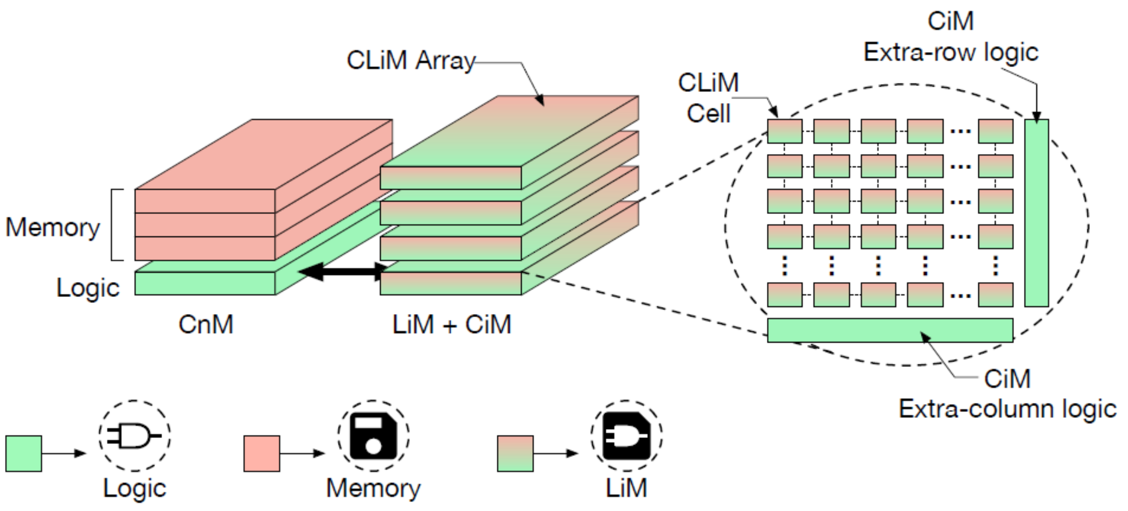


Figure 1.2: Conceptual structure of Configurable Logic-in-Memory Architecture (CLiMA). Source:[1]

In fact, the operations that can be performed in the memory are associated with the CLiM arrays. Instead, the operations that cannot be performed inside the memory, are performed by the CnM unit. Figure 1.2 shows the structure of the memory already described. The main block of the CLiM array is the CLiM cell, which is a 1-bit configurable cell, that can be used to perform different types of logic operations. The connections are configured to create more complex operations like the addition or the multiplication using multiple cells. Now we are going to see how these cells are composed and allocated to the array.

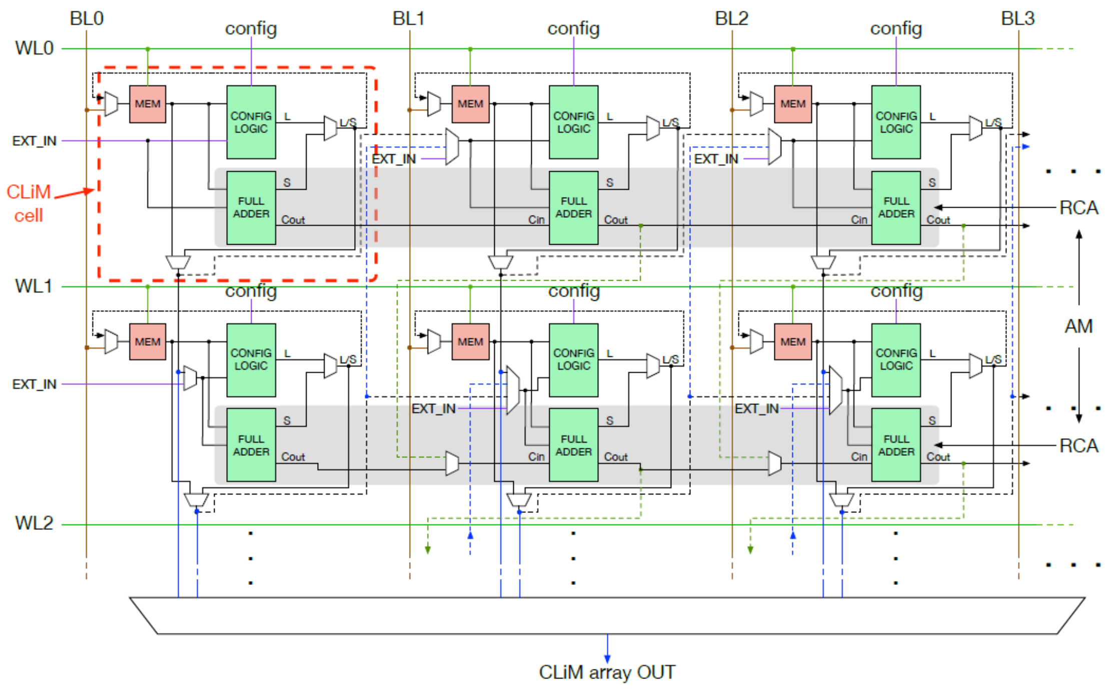


Figure 1.3: Configurable Logic-in-Memory Cell Array. Source:[1]

Figure 1.3 shows the array of CLiM cells, how it is possible to see, each cell is composed of a memory cell, that is connected to the word line (WL) and bit line (BL), and a configurable logic block. There is also a Full-Adder in each cell that is very useful to perform different types of operations. Each cell contains some multiplexer to connect the cells between them. The cells are connected to perform operation involving the whole row or column. In Figure 1.3, all the Full-Adders of the same row are highlighted in blue. It is possible to verify that, connecting the carry output to the next Full-Adder of the same row, we can create a Ripple Carry Adder (RCA). Using more RCAs, we can create an Array Multiplier (AM) connecting more columns. These are the complex operations that we can perform using the composition of more CLiM logic cells. The trade of using these operations is that the configuration of these arithmetic circuits is not so convenient, for this reason, if we need to do fast operations, it is better to use logic outside the memory. Thanks to this configurability this type of architecture is very useful in data intensive applications like the Convolutional Neural Network (CNN). In [1] it is showed how to use the CLiMA to perform the computation of CNN in a very efficient way.

Chapter 2

LiM Architecture

Our LiM Memory is a complex component that merges storage and computation together. Additionally, some external units may be required for complex computations. For this reason, we classify the logic inside the memory into two categories:

- **Intra-Cell Logic**: the logic present inside the LiM cell
- **Inter-Cell Logic**: the logic external to the LiM cell

The LiM cell of our memory is composed of a memory cell and an Intra-Cell logic. It can be interconnected with the other cells using the logic or directly using the memory inputs/outputs.

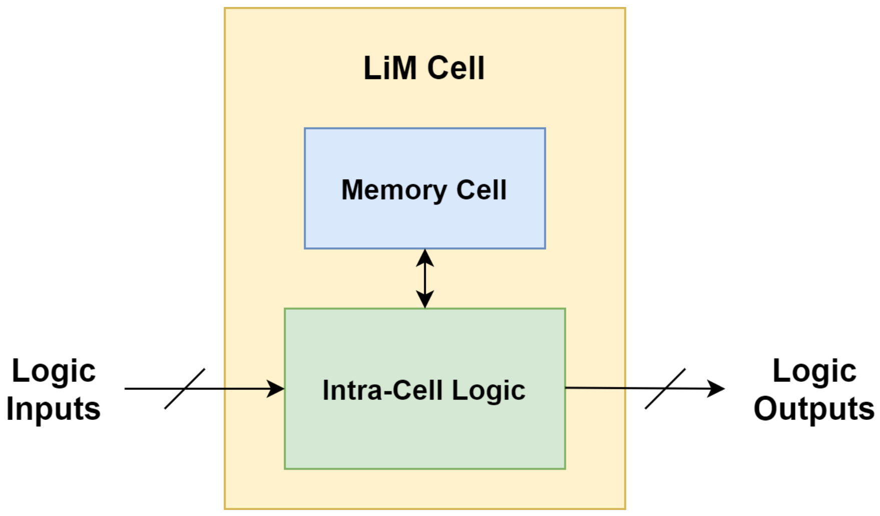


Figure 2.1: LiM Cell

In Figure 2.1, an example of LiM cell is shown, highlighting the two components and grouping all the interconnection wires that communicate with the outside of the cell by the Inputs and Outputs Logic. There is internal communication between the two parts. In order to explain how Inter-Cell logic is inserted, we can see Figure 2.2, where some LiM cells directly interact between themselves or pass through Inter-Cell logic. There are no restrictions about the link of these components, in fact, the first cell in figure 2.2 can be connected with any component of the scheme.

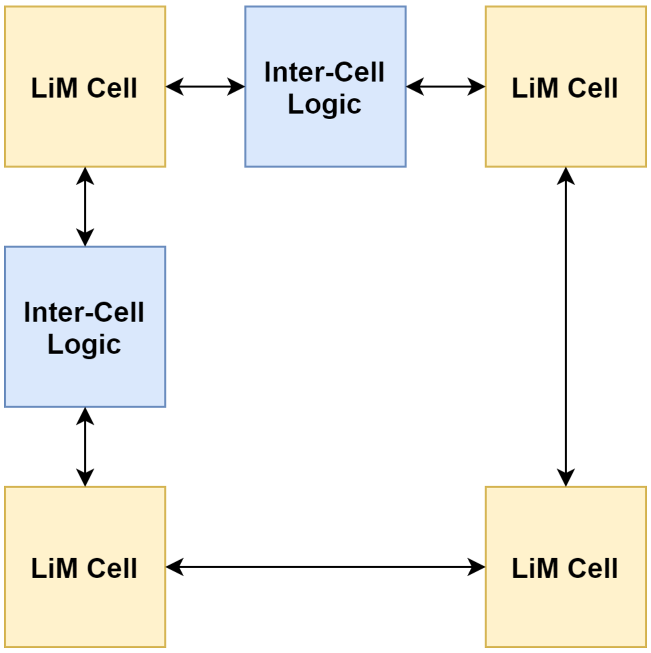


Figure 2.2: LiM Cell interconnection between Inter-Cell Logic

Also, in this case, the logic inside the Inter-Cell block can be very simple, like a single gate or a more complex one.

2.1 LiM Structure

From now, a punctual description of the LiM architecture is provided. The operations that a LiM usually performs involve the use of an entire row or column. As we have already seen for the CLiMA architecture, we can use a group of rows and columns to exploit complex operations, like the sum or the multiplication. For this reason, we reference to Inter/Intra logic of an entire row or column.

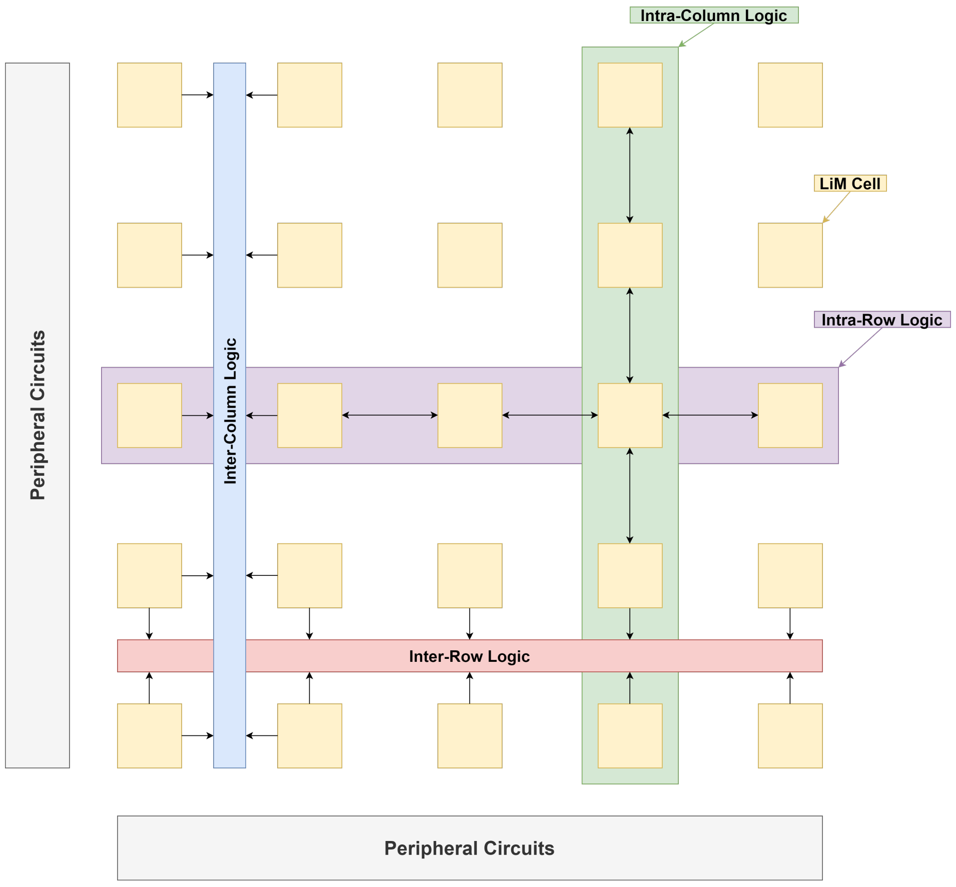


Figure 2.3: LiM Architecture

The Figure 2.3 shows the entire LiM structure. The yellow boxes represent the LiM cell containing logic inside. We have also the control circuitry of rows/columns access operations. Similar to Intra-Column logic, which interests the whole column of Intra-Cell logic, the same considerations can be done for the Intra-Row logic. There are also two layers that consider Inter-Columns and Inter-Row logic interacting with the adjacent cells. All this component creates our LiM structure and it can be customized to perform the operations of the designed algorithm.

Chapter 3

Motivations

The DExIMA tool was born from a Nicola Piano's idea [6]. The good of the tool is to explore different LiM architectures using a high-level description, without writing the entire HDL description at register transfer level (RTL). In this way, it is possible to understand which is the most feasible LiM architecture for our algorithm with lower effort. The tool is a performance estimator, that computes the four main figures of merit of our circuit:

- Area
- Static Power
- Dynamic Power
- Delay (Critical path)

The tool is able to compute the performance specifying the algorithm that we want to execute. In the following section, we explain how the program works and the problems that we fixed starting from the original version of DExIMA. From now on the first version of DExIMA is called 1.0, while the newer one 2.0.

3.1 DExIMA 1.0 High-Level Description

We start from the structure of the program that is composed of three different compilers, that use different configuration files in input. Each file has different behavior and it's used in different parts of the architecture. All the Files are used to compute the performance in output together with the Hardware Models.

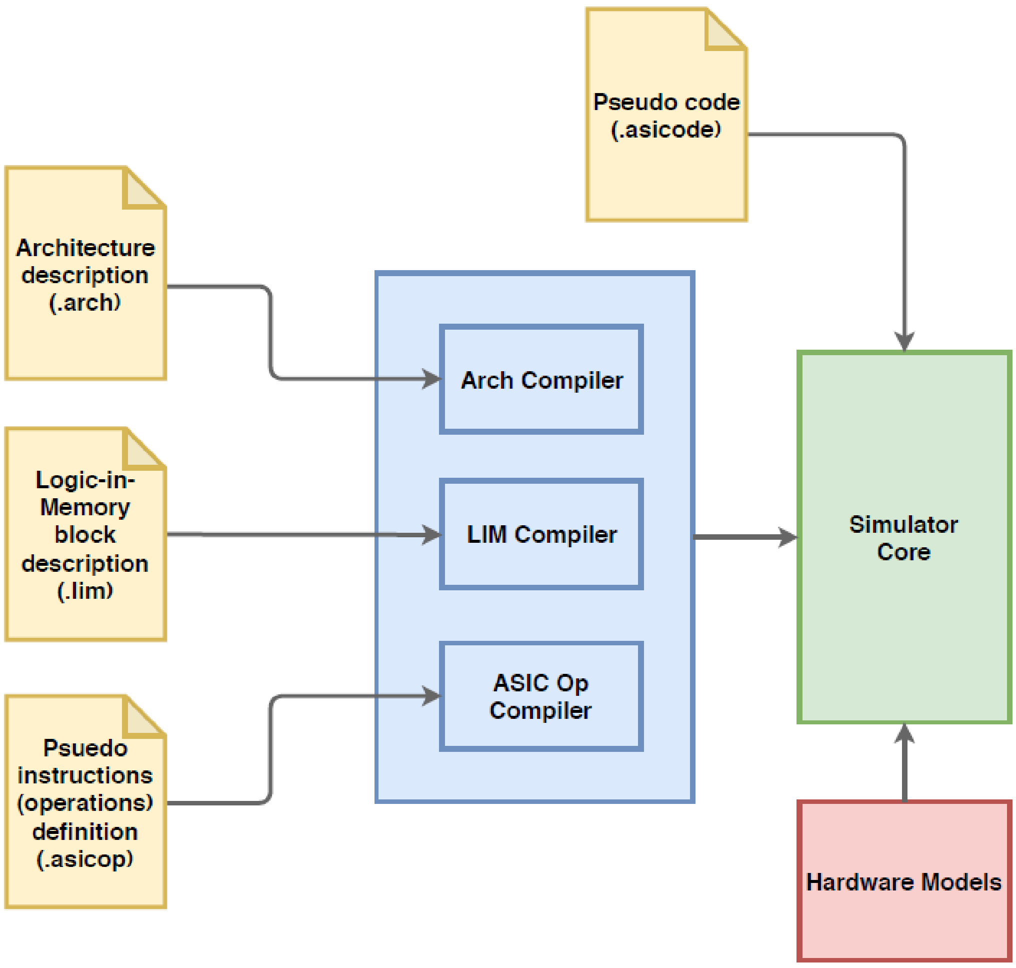

Pseudo code (.asicode)
Architecture description (.arch)
Arch Compiler
Logic-in-Memory block description (.lim)
LIM Compiler
Simulator Core
Psuedo instructions (operations) definition (.asicop)
ASIC Op Compiler
Hardware Models

Figure 3.1: DExIMA 1.0 High-level Structure. Source:[6]

These files have the following extensions:

- **.arch**: This file is used to instantiate and connect the components outside the memory
- **.lim**: This file contains the description of the LiM organization and the operation that the LiM can perform
- **.asicop**: This file contains the pseudo-instruction set of the operations executed by the components described in the .arch file
- **.asicode**: This file contains the sequence of instructions described in the .lim and .asicop files.

The program is composed of a component library related to the ASIC operations, and other predefined libraries of cells that can be used to construct our custom LiM. All the gate models are based on the NAND with two inputs and hierarchy generate to create all the core complex models.

3.2 Motivations

This section contains the explanation of all the issues of the DExIMA 1.0 and the motivation that led to the rewriting of the code completely from zero.

3.2.1 Configuration Files Writing Effort

The first problem that occurs when we approached to DExIMA 1.0 was the effort in writing the code. If we need to create an architecture with many components, in DExIMA these components should be created one by one. To create a component, its name is specified in a different line of the configuration file, in fact, if we need n gates we need to write n lines of code. It is the same if we need to connect two or more components, if we need to do n connections we need n lines of code. This concept concerns all parts of the code of the configuration files. This method is not feasible if we have a high amount of gates. As a matter of fact we need an external script to create the configuration files would frustrate part of the advantages of using DExIMA to evaluate the performance of our architecture in a fast way. Moreover, it is not possible to generalize the code using variables.

3.2.2 Error checking problem

Another important aspect related to the interface with the user was the error checking. The program does not have a complete syntax manual, so its easy to commit some errors when we write the configuration file. The problem is that in most cases the error is not reported, meaning that we occur in errors at the run time of the program generating, for example, segmentation-faults. When this problem occurs, the user doesn't have any idea on which instruction generates the error, implying waste to find the bug.

3.2.3 Configuration files separation

Each file works completely separated and it is not possible to connect components of .arch file with the LiM structure. A memory interface does not exist and is not possible to compute the power consumption related to memory accesses by an external component. More than one LiM memory instance are not allowed, because the only one permitted is described in the .lim file.

3.2.4 Random Behaviour

Random behavior happens sometimes, especially when the LiM part is used. Running the same configuration files generates different results in the output. Variations can be both small and so big as well, that they can vary completely the outputs.

3.2.5 Simulation Times

Simulation times are very important when we exploit architectures that can be big. Simulation times can be very long, because they grow exponentially, requiring even hours. This is due to the model computations and object creation that have a pyramid structure. In this structure, the high-level component is dismantled with a high number of sub-components and finally to a large number of NAND2 gates, where the performance of each is recomputed each time.

3.2.6 Models Efficiency

The models inside the program are all based on the NAND2, but each time a gate is connected to another one, the output fanout is simply incremented by 1, meaning that the load of the gate is increased by the input capacitance of one NAND2. This model does not consider that some more complex gates can have inputs with more than one NAND2 connected. Many errors could be introduced, depending on which component is being used. The other elementary gates that are composed of multiple NAND2 gates result in an over-estimating value of the area and static power. Moreover, the memory cells are not created by the program. Another external tool is employed to compute the memory performance, meaning that the connections to the memory cell create wrong performance results. Moreover, there is no possibility to use different timing parameters (such as the clock to output, setup, and hold times of FlipFlop) besides the critical path of each internal component. Is not possible also to discriminate between a read and a write operation of the memory in terms of timing and power. In the Memory part, we can create the memory cell using only the components of the memory library and it is not possible to insert all the models inside the program. Moreover, switching activity estimation is not considered in dynamic power computation.

3.2.7 Simulation parameters orientation

The simulation of the circuit is bounded by the information set in the code. Simulation results cannot be varied. For example, we cannot change the circuit Voltage or vary the performance changing the clock period. Moreover the output of the program groups all the information computed in few fields, so it is not possible to discriminate all the contributions parameters of circuit components. Imagine having in the output the total performance of the memory, but we want to know which is the area of the array and the area of the interface circuitry. With only the total value, it is not possible to know how much area is dedicated to the array, and how much area to the circuitry. The same considerations can be done for the single instruction performances because it is not possible to isolate it from the complete algorithm.

3.2.8 Insertion of a new model

All the models must be designed with NAND2 gates. Moreover, to insert a new model, we need to modify several files to define completely the component behaviour.

3.2.9 Absence of Documentation

The tool is provided without documentation, creating some difficulties when someone tries to modify or check some information needed for the program behavior.

3.2.10 Conclusions

In conclusion, we decided to re-design the whole program, because modifying and upgrading some important parts would have implied resolving the problems previously cited in a non-efficient way, due to the structure of the program that cannot be changed. By re-designing it, we can create a more precise program inheriting only a part of the syntax of the configuration files already done.

Chapter 4

DExIMA

DExIMA is the core of this thesis. DExIMA is a powerful tool intended to explore and evaluate the performance of a logic circuit. It can build a custom LiM memory and evaluate the performance in a very fast way. The process to build a LiM memory can be very time consuming with HDL representation. For these reasons, DExIMA provides the user a library of components useful for the general architecture and allows us to design our architecture using a high-level methodology. DExIMA synthetizes the circuit and computes the performance of our algorithm. In this sense, it can be seen as a logic circuit creator with the possibility to embedded the test bench of our algorithm.

4.1 DExIMA Structure

Let's start with the top entity of the entire program that controls and manages all the process operations. The program's main class is called *Dexima*. The class contains some objects that exchange information: its structure is shown in Figure 4.1 below.

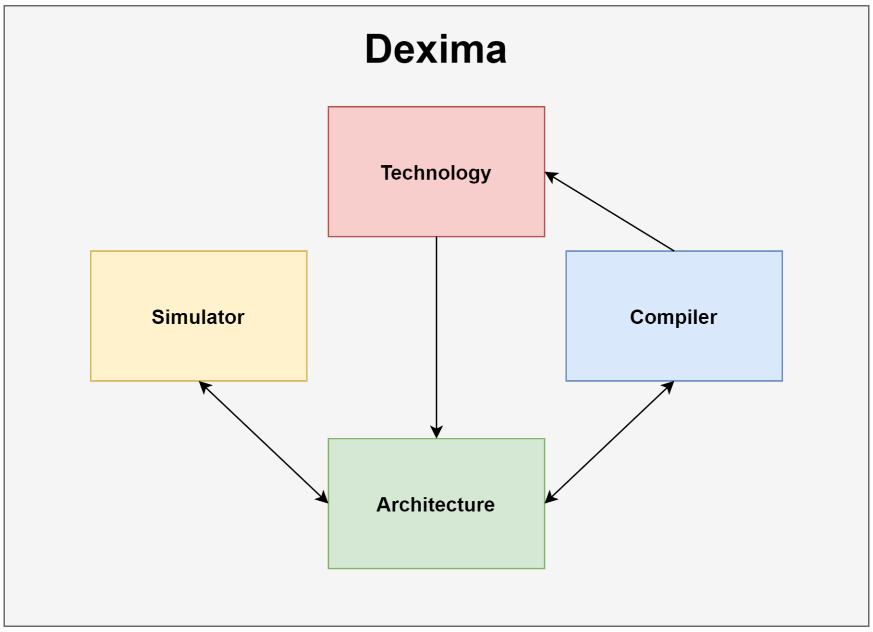


Figure 4.1: Dexima class structure

The class is composed of four objects:

- **Technology**: used to store and compute all the technology parameters for the models.
- **Compiler**: used to compile the DExIMA code and to fill the *Architecture* object.
- **Simulator**: used to manage the architecture simulation process.
- **Architecture**: used to store all the components of the circuit and the performance data of them.

The *Dexima* class in Figure 4.1 describes the objects interaction. In particular, the core of the class is the *Architecture* class, which stores the whole components of our architecture. The *Compiler* interacts directly with the *Architecture* and *Technology* objects. The *Compiler* sets and stores some technology parameters specified by the user in the *Technology* object. The *Compiler* also gets the empty *Architecture* object in input and returns the full object. This is the motivation of the double arrow. The *Simulator* has a management role and Simulates the circuit, so the interaction is only with the *Architecture* object. An important aspect of *Dexima* class is that the objects created are unique. All the classes of the program get only the reference of the object to reduce the memory occupation. There is a twofold advantage in doing it: the occupation of less amount of memory and there is also no need to copy the objects inside the functions.

4.2 Process Flow

In this section, we will see a critical aspect for the user: what the program expects in input and what the program returns in output. The program receives in input a file with *.dex* extension that contains all the information to instantiate and simulate our circuit. The output of the program is a file with *.dof* extension that is the abbreviation of *DExIMA Output File*, and contains the performance results of the circuit.
The process is split into three main steps:

- Compilation Step
- Technology Parameters Computation Step
- Simulation Step

4.2.1 Compilation Step

The first step is the Compilation Step, which is described in Figure 4.2. The *Compiler* gets in input the *.dex* file, and the reference of the *Architecture* object created inside the *Dexima* class. The *Architecture* object is initially empty. The *Compiler* parses the file and fills the *Architecture* object with all the information written in the *.dex* file. The compiler sets also some technology parameters specified in the file. In output, we have a file with the same name of the input file but with the *.log* extension. The log file contains the summary of the created architecture. The structure and the information contained in the log file will be described in detail.

4.2.2 Technology Parameters Computation Step

This step is located between the Compilation Step and Simulation step. It is very important because it sets and computes all the parameters of the technology used later in the simulation process, shown in Figure 4.3. When the compiler gets the parameters related to the Technology, the *Technology* object stores these parameters. After that, the technology object extracts

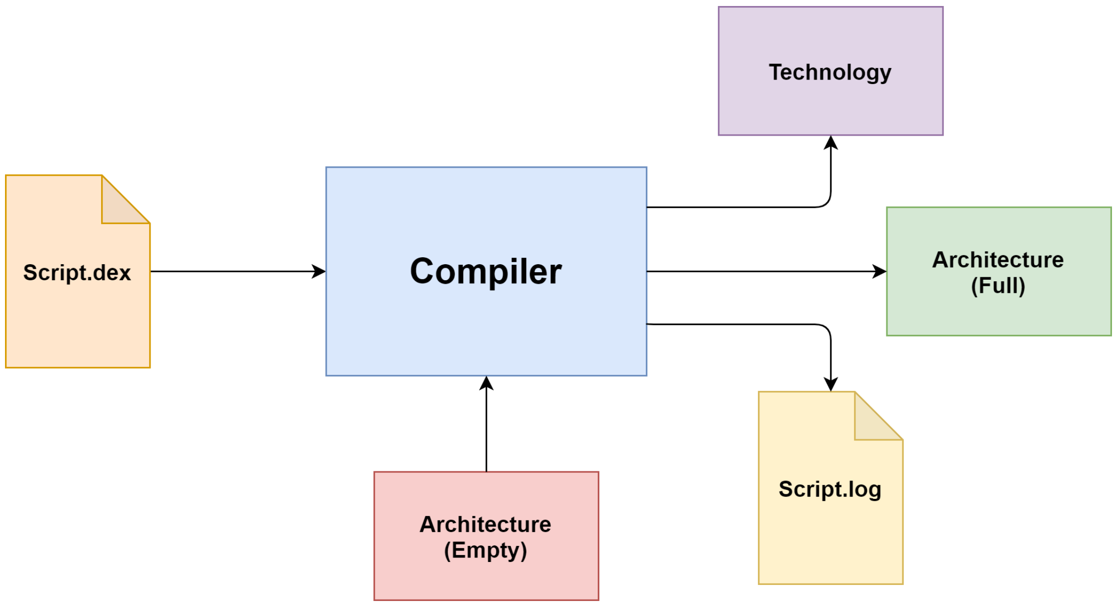


Figure 4.2: Compilation Step process flow

the technology parameters from a TechFile. In the folder, there are several TechFiles available. The object gets one of these and loads in main memory

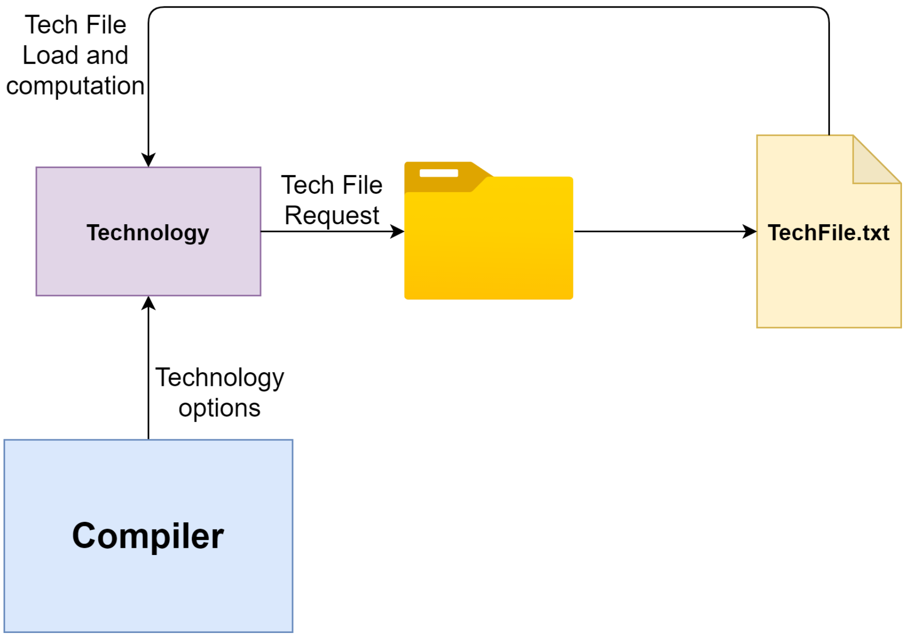


Figure 4.3: Technology load and computation process

the parameters written inside. The following step consists of the computation of the parameters derived from the base values written in the TechFile (that we will use often during the performance computation). These parameters need to be computed at this specific point since they will later be used to manage the interfaces between the gates (fanin values).

4.2.3 Simulation Step

The Simulation Step is managed by the *Simulator* class. It uses the methods of the *Architecture* class to run in the correct sequence the process steps for the simulation of the circuit. The process flow is described in Figure 4.4. We

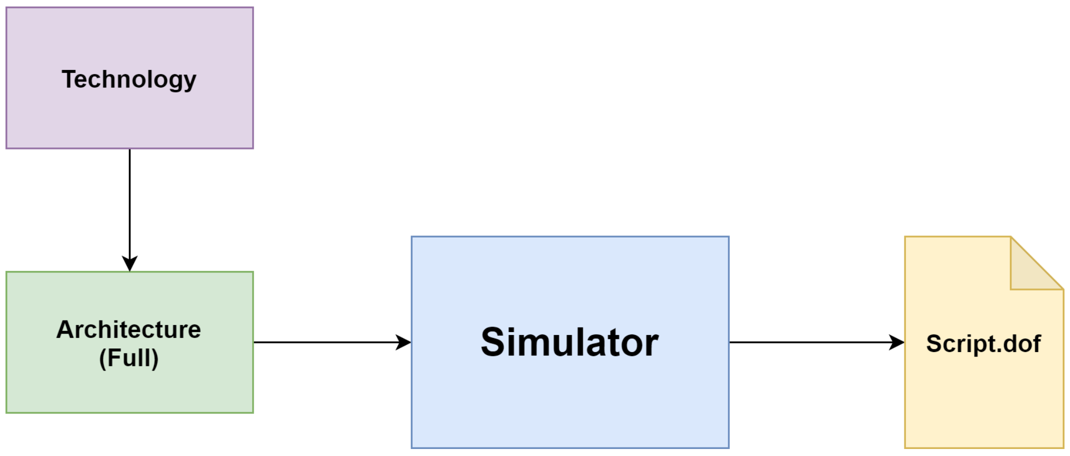


Figure 4.4: Simulation Step process flow

have in input the reference of the full *Architecture* object and in the output the *.dof* file with the performance of our circuit. The *Technology* class is used from the *Architecture* class to compute the Technology parameters during the Simulation process. In summary the *Simulator* calls the *Architecture* functions and, finally, fetches the results from all the objects. These results are written in the .dof file.

Chapter 5

DExIMA Language

We start by describing the DExIMA language and how to use it. In this way it is more convenient to know the high-level description, and after it will be easier to explain all the hidden processes of the program that interprets the instruction.

5.1 Input File description

The input file is the *dex* file. Inside of it, the followings information are specified:

- Simulation constrain
- Components of the circuit
- Memories of the circuit
- Memories Architectures
- Components links
- Instruction set
- Algorithm code description

This information defines the complete description of our circuit. The language is a *line-based* language, meaning that each instruction must start and end in the same line. The file is composed of sections as shown in Figure 5.1. Each section starts with keyword **begin** and ends with keyword **end**. After that, we specify the name of the section to open/close. Nested sections are needed in some part of the code for the correct execution of the instructions.

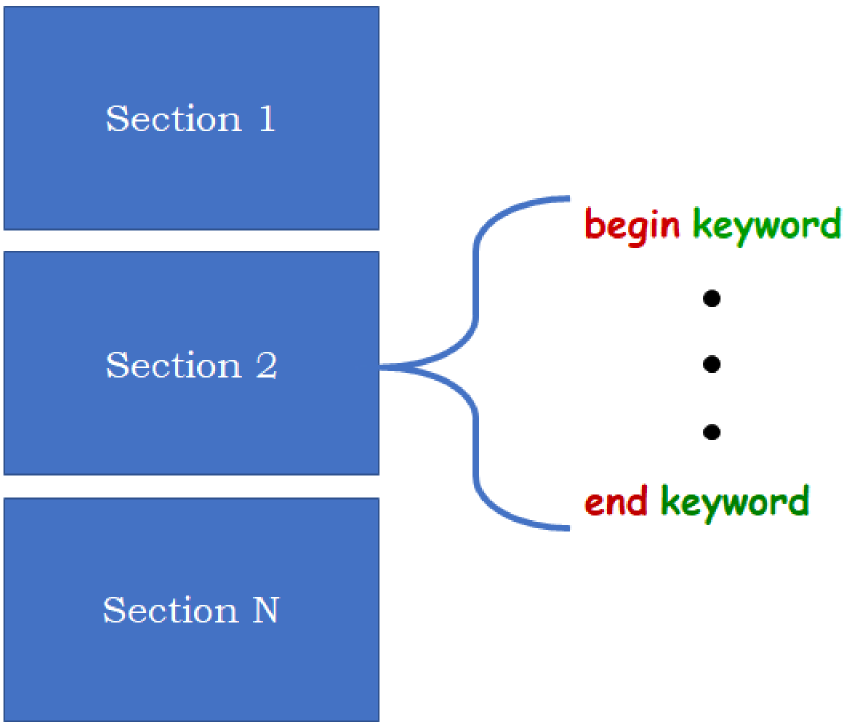


Figure 5.1: Section organization of dex file

The sections have a predefined order. The sequence must be:

- **Constants**: Used to define constants values usable in the code
- **Init**: Used to instantiate the components of the circuit
- **Memory**: Used to specify all the information about memory
- **Map**: Used to link the components between them
- **Instructions**: Used to define the instruction set
- **Code**: Used to define the algorithm

The number of sections varies with a dependence on the number of memories. The number of Memory sections is equal to the number of memories in our architecture. The summary of section order is shown in Figure 5.2.

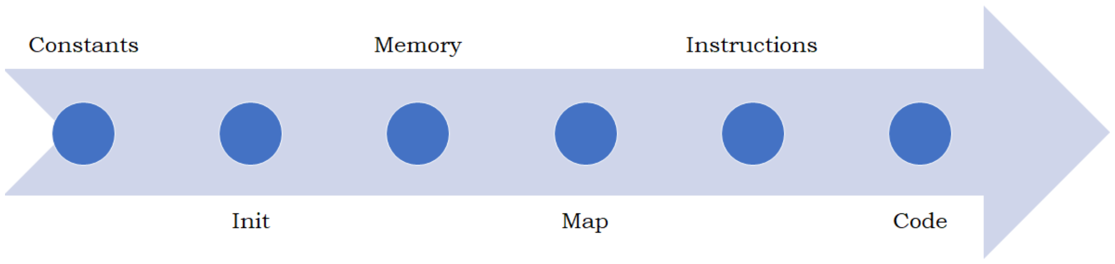


Figure 5.2: Sections order of dex file

5.2 Comments

Line comments are allowed in the code. The comment lines start with symbol #. All parts of the line at the right of the symbol are ignored by the compiler. An example of DExIMA comment is shown below:

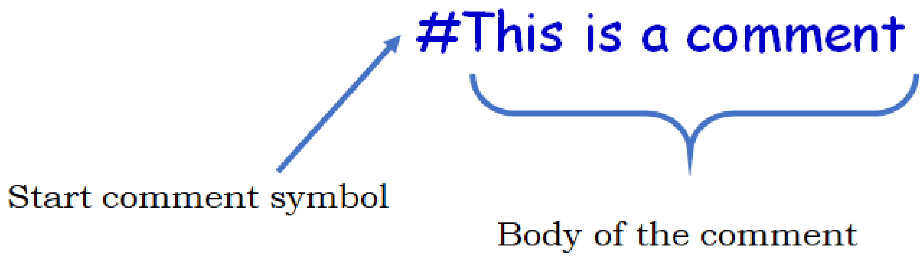


Figure 5.3: Comment block example

```
#This is a comment
Line of code #This is another comment
```

Listing 5.1: Comment example

5.3 Constants Section

The constants section is used to define constants that are usable inside the code. There are three types of constants:

1) **INT**: Integer constant

2) **FLOAT**: Floating point constant

3) **STRING**: String constant

To instantiate a constant, we use the specific constant keyword followed by the name of the constant. At the end of the line, we insert the value. These three fields are separated by spaces or tabs like shown in 5.4.

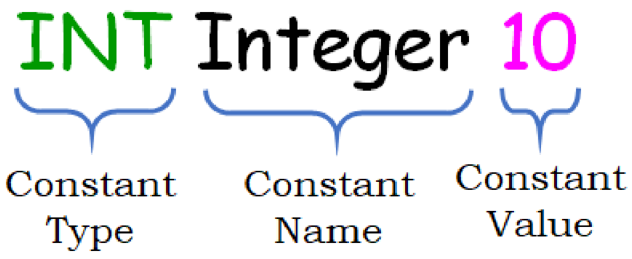


Figure 5.4: Constants section syntax

Now we see a practical example:

```
begin  constants

    #Definition of an integer constant
     INT integer 20

    #Definition of a float constant
     FLOAT pi 3.14

    #Definition of string constant
     STRING str "This is a string"

end  constants
```

Listing 5.2: Constants example

Now we analyze the example 5.2, we instantiate an integer named integer of value 20, and a float named pi of value 3.14. The string needs double quotes to specify the start and the end of the string. To use a constant in our code, we use the symbol $ in front of the constant name we want to expand. The constants in DExIMA are very powerful: they can be used everywhere in the code and can be used in a nested form. They can directly call a command or open and close a section. The value of the constant is substituted in the place where they are expanded, in order to clarify better this concept below there is a practical example:

```
begin  constants
```

```

    #Define an integer
     INT num 12

    #Create a string with the num inside
    #The result value of string is "Gate12"
     STRING str "Gate$num"

    #Create command string
     STRING command "FLOAT var 7.4"

    #Instantiate a constant indirectly
    $command

end  constants
```

Listing 5.3: Nested constants example

The example 5.3 shows two cases. In the first one, we insert an integer inside the value of the string, in another case we use a string to call a command. All the constants are treated as strings in DExIMA, but the main difference between these three types is that the value is checked by the compiler. If, for example, we try to instantiate a float variable inside an INT constant or a char, the compiler returns an error.

5.3.1 Built In Constants

There is also a fourth type of constant that has a special meaning. The built-in constants cannot be used in the code like previous, using the $ symbol. They act directly in the simulation process. The built-in constants are instantiated with the keyword **BUILT_IN**, but the names cannot be arbitrary, it must be chosen from a list of supported types. The built-in constants can be omitted in the section because they have defaulted predefined values. The supported built-in constants are:

- **VDD**: used to define the supply voltage, expressed in [V] Volts. The default value is specified in the Technology File.
- **CLOCK**: used to specify the clock period, expressed in [ns] nanoseconds. The default value is the critical path of the circuit.
- **AR**: used to specify the Aspect ratio of the minimum sized N-mos transistor. The default value is specified in the Technology File.

- **SF**: used to specify the stack factor, the stack factor is a model parameter to compute the stack effect. If we don't want to consider the stack effect, we put it to zero. The default value is 2.
- **NODE**: used to set the technology node of our Technology. The default value is 45 nm.
- **TECH**: used to specify the technology type. The possible technologies are HP "High Performance", LOP "Low Operating Power", LSTP "Low Standby Power". The default technology is LOP.
- **SWITCHING**: used to enable the computation of the switching activity of the gates. The value can be ON or OFF. The default value is OFF.
- **PROB**: used to set the input probability of the gates in the switching activity computation. The default value is 0.5.

In the table A.3 there are all the supported technologies. To clarify how built-in constants are used, an example is shown in Listing 5.4:

```
begin  constants

    #Define the supply voltage of 0.9 V
     BUILT_IN VDD 0.9

    #Define the clock period of 2 ns
     BUILT_IN CLOCK 2

    #Define an aspect ratio of 4
     BUILT_IN AR 4

    #Define a stack factor of 1
     BUILT_IN SF 1

    #Define the technology node of 32 nm
     BUILT_IN NODE 32

    #Define the HP technology
     BUILT_IN TECH HP

    #Enable the computation of switching activity
     BUILT_IN SWITCHING ON

    #Define the input probability
    BUILT_IN PROB 0.6

end constants
```

Listing 5.4: Built in Constants example

The supply voltage can be modified but it is not a good practice because the Technology parameters are computed in a particular bias point defined by the default voltage. This implies that changing the voltage can cause unexpected or wrong values of performance because it can be used for future implementations. However, we insert this feature.

5.4 Init Section

After the constant section, there is the Init section. This section is used to instantiate the components and the memories that will be used in our circuit. To instantiate a component, the syntax is the name of the model followed by the name of the instance. In the instance name, we open a round bracket where we specify the parameters needed for the model, like in Figure 5.5.

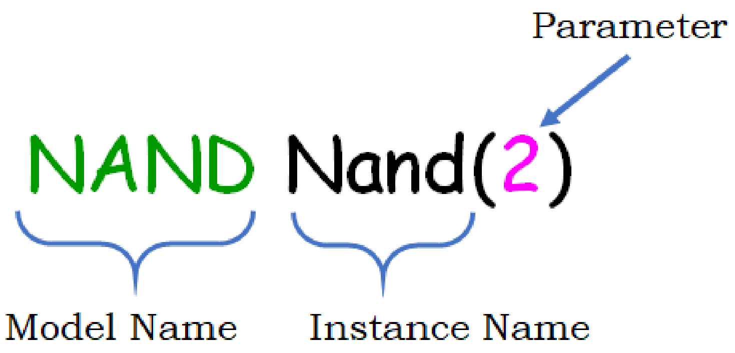


Figure 5.5: Init section syntax

Nand is the name of the instance and *2* is the parameter, that in this case indicates the number of inputs. Each model has different names and different types of parameters. The arguments specified in the brackets can be also a string or empty.
The example 5.5 shows some of the simple gates. All the other gate specifications can be found in the appendix.

```
begin init

```

```
    #Instantiate a And with four inputs
     AND And(4)

    #Example of no parameter component
    #Instance of Inverter
     NOT Inverter()

    #Instance of a Register of D Flip Flop with 8 bit
     FF Register(8)

    #Create a latch SR
     LATCH_SR Latch()

end  init
```

Listing 5.5: Base components examples

It is possible to use the constants to parameterize the component. For example, we can choose the name of the component or the number of inputs like the example shown below 5.6:

```
begin  init

    #Create a Nor with int_var inputs
     NOR Nor($int_var)

    #Create a Or with name contained in string_var
     OR $string_var(3)

end  init
```

Listing 5.6: Create instance using constants

5.4.1 Special components

There are also special components that work differently from other components. Four special components are available:

- **DRIVER**: The Driver is an adaptive component, the device is created in the function of the load and has three different operation modes.
- **CK_DRIVER**: The clock driver is a type of driver used to compute the performances of the clock, and it is evaluated separately from the other components.
- **LOAD**: The load component is used to emulate a capacity load.

- **LIM**: Instantiates a memory and is responsible for the memory interface circuits.

Now we see a practical use and a more detailed description of these special components, for the LIM we have a dedicated section.

5.4.2 Driver

DRIVER has three operating modes:

- **Auto**: in this mode, the driver is synthesized using the optimum number of stages to drive a big capacity load.
- **Buffer**: in this mode, the driver has an optimum even number of stages to not change the logic value in the input.
- **Inverter**: in this mode, the driver has an optimum odd number of stages. It is used to invert the logic value in the output.

To instantiate the Driver component, we need also the multiplicity factor that sets the successive stage size concerning the previous one. DExIMA applies the logical effort method and computes the best number of stages of the Driver taking into account the mode of operation. Now we see an example of the use of this component:

```
begin  init

    #Create a driver in auto mode (A) with effort equal to 4
     DRIVER Driver_auto(A,4)

    #Create a driver in buffer mode (B) with effort equal to 2
     DRIVER Buffer(B,2)

    #Create a driver in inverting mode (I) with effort equal to
   3.4
     DRIVER Driver_inverter(I,3.4)

end  init
```

Listing 5.7: Drivers examples

5.4.3 CLock Driver

The clock driver is a specific type of Driver. This Driver is set in Auto mode. The computation of the parameters like dynamic energy is computed differently because it switches in each clock step of the algorithm and the commutation is doubled compared to the normal gates. Like the normal drivers, it is possible to set the effort parameter. More than one clock driver can be used in the circuit and at the end of the computation, we get a detailed report for each clock driver instantiated. Now we see a simple example of how instantiating a clock driver:

```
begin  init

   #Create a clock driver of effort 4
    CK_DRIVER Clock_driver(4)

end  init
```

Listing 5.8: Clock Driver example

5.4.4 Load Component

The load component is used to test or emulate a capacity load. This component is a special component. The performance does not affect the circuit because it has a zero value for the area, dynamic energy, static power, and delay. The component varies the performance of the gate on which is connected. Load does not have outputs, but only inputs. The parameters needed for the component are the value of the capacitance expressed in pF and the parallelism of inputs we want to use.

```
begin  init

   #Create a load of 2.4 pF with 8 input parallelism
    LOAD Load(2.4,8)

end  init
```

Listing 5.9: Load component example

5.4.5 LiM Component

The LiM component is the most important. It is used to instantiate a memory. We can instantiate how many memories we want, and the parameters needed for the components are two: the address parallelism, and the input/output port parallelism. The default memory in DExIMA has two ports, one for the Read operation and one for Write operation. The other internal parameters will be specified in the *Memory* section. The order of how we create this component is important because it is used later to define a dedicated section (the *Memory* section) where the name of the section is not "Memory" but the name of the instance we create. For example, if we create three memories called Ram, SRam, and Dram in this order, later we will have three sections with the same names that must be created in the same order. To create a memory we use the keyword *LIM*, in 5.10 there is an example of memory creation.

```
begin  init

    #Remember the order is important

    #Create a memory with 8 bit address and
    #8 bit input/output  parallelism
     LIM Ram(8,8)

    #Create a memory with 12 bit address and
    #64 bit input/output parallelism
     LIM Lim(12,64)


end  init
```

Listing 5.10: Memories creation example

5.5 For Control Flow

Before going ahead, the For loop is introduced. The For loop is very similar to the modern programming language and it is very useful when we need to do a lot of operations of the same category. In Figure 5.6 there is a description of the syntax used. The information needed by the control flow is, first of all, the name of the iterator, that in the figure is *i*. This will be the parameter that in the cycle changes at each step. Later, we find the *range* function that receives in input three parameters: the start value, the increment and, the stop value. The start and stop values are included in the loop: if, for example, we have the parameters (0,1,10) we have 11 iterations with values 0,1,2...10. The float values are forbidden since it is very dangerous when used with instance names because a non-infinite precision can cause an unexpected instance name. Negative values and decrements cycles like $(10, -1, 1)$ are

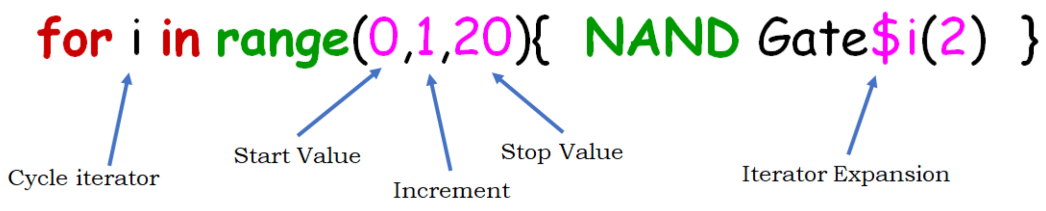


Figure 5.6: For control flow description

allowed too. When the user inserts a loop with no end like this $(1,-1,10)$ the compiler reports the error. The last part of the syntax is the body of code: we insert the instruction we want to loop between curly brackets. The iterator can be used only inside the curly brackets and the syntax for the use is equivalent to the constant expansion syntax. As it was pointed out at the beginning of the language description, this language is *line-based*, so it is not possible to break the loop with a newline. It is possible to use in a nested way to have multiple loops inside. It is very useful for example in a bidimensional array loop. Some examples of loop use are shown in 5.11 5.11.

```
begin  init

   #Example of instantiating of n gates Nand
   for i in  range(1,1,$n){  NAND Gate$i(2) }

   #Example of instantiating of n gates Nor decreasing loop
   for i in  range($n,-1,1){  NOR Nor$i(2) }

   #Example of nested loop
   for i in  range(1,1,10){ for j in  range(0,2,10){
  XOR Xor$i$j(2) } }

end  init
```

Listing 5.11: For loop examples

5.6 Map Section

The Map section is used to connect the components to each other. The components before the connection must be instantiated in the Init section. The syntax used is shown in Figure 5.7. The line is divided into two parts by the right arrow "->" called link operator. The link operator shows the direction of the link from the left to the right. The left part is reserved for the output port and the right part for the input port. To specify the port to connect, we write the name of the instance followed by the "." dot operator that is used to specify the port name of the instance. If we try to specify a port that does not exist for the component or is not of the correct type (input/output) in the corresponding left/right part the compiler generates an error. The input/output port can have a multiplicity of one or more. For example, the elementary gates like Nand or Nor gates have a multiplicity port of one, a more complex model like a register can have an input/output multiplicity higher than one. To connect a specific wire and not the total wires we can use the index operator "[ ]". We include the index number between the two square brackets of the index operator. All the index starts from 0 to the max value minus one. For example, for 8-bit parallelism, the index goes from 0 to 7. Is important that the parallelism of left and right part must be the same if we have one bit on the left, we need one bit on the right part. If we don't specify the index, DExIMA interpreters the port of full parallelism, so we can connect the whole bus. If the bus width doesn't match between the two ports, we can use a for loop to connect only the wires we need to connect varying the index. Now we see a couple of examples that explain these functions:

```
begin  map

    #Connection of a single wire gates
    Nand.OUT -> Nor.IN0

    #Connection of a single wire of a multiple parallelism
    FlipFlop1.Q[0] -> FlipFlop2.D[1]

    #Connection of entire bus
    FlipFlop1.Q -> FlipFlop2.D

    #Connect part of the wires using for
    for i in  range(0,1,3){ FlipFlop1.Q[$i] -> FlipFlop2.D[$i] }

end  map
```

Listing 5.12: Map section examples

5.7 Math Environment

The Math Environment is a feature of the code used to make math operations. The use of Math Environment is shown in Figure 5.8. To interpret

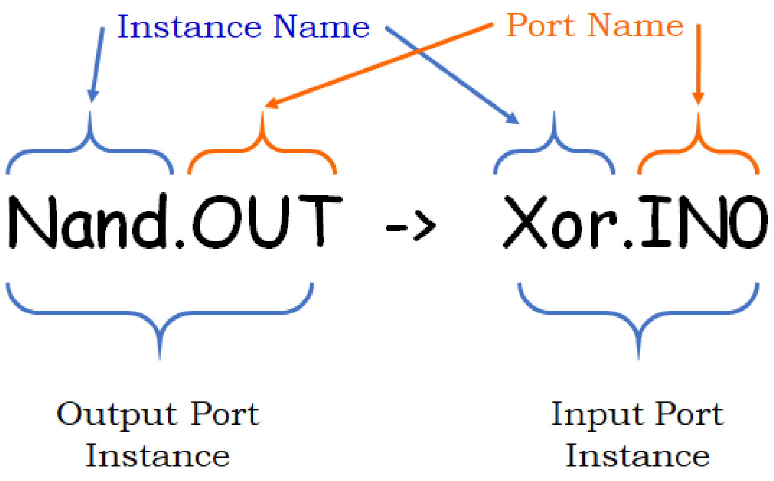


Figure 5.7: Map section syntax

a mathematical operation with the Math Environment we use the symbols "$(" and ")$". All the numbers, operators, or brackets must be separated by at least one space to be interpreted correctly. To group the operations, only round brackets are allowed. The following operations are available:

- "+": Sum operation
- "-": Subtraction operation (negative number)
- "*": Multiplication operation
- "/": Division operation
- "^": Power operation

The available operations are shown also in Figure 5.8. The Math Environment uses the Shunting-Yard algorithm to parse and do the operations, something similar to what is done in [7]. The computations are done in floating-point, but only when the result is ready, it is transformed into an integer value using truncation. The output is an integer that is conformed with the ecosystem of DExIMA, where all the operations and commands support only integer numbers. The result is replaced with a string in the position where the command is invoked. Some examples of the Math Environment are reported in 5.13. For example, suppose to connect a chain of gates. In a normal situation, it is hard to do this operation, but with the Math Environment it is very simple like the example 5.13 below:

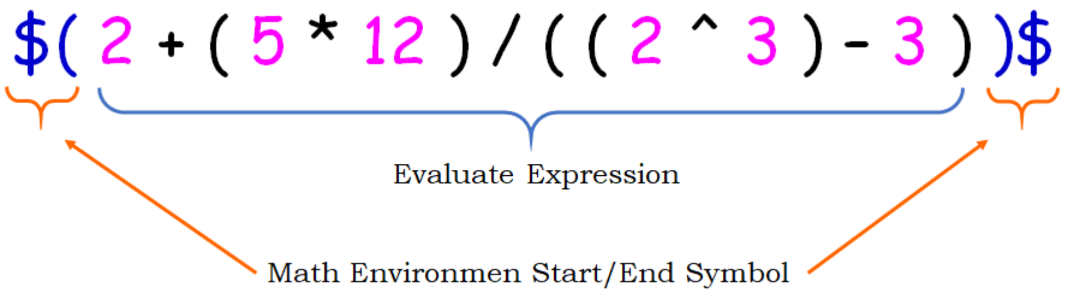


Figure 5.8: Math Environment syntax

```
begin  map

   #Connect a chain of gate
   for i in  range(0,1,10){ Gate$i.OUT -> Gate$( $i + 1 )$.IN }

end  map
```

Listing 5.13: Math Environment example

5.8 Memory Section

The memory section specifies all the information related to the memories instantiated in the Init section. The Memory section has some subsections inside and also in this case an order must be respected.
The memory subsections are:

- **Memdef**: Used to specify the array shape and the memory cells
- **Logic**: Used to specify the inter row/column logic
- **Cells**: Used to create the custom LiM cells
- **Map**: Used to link the instances inside the memory

The Figure 5.9 explains better the order of the sections.
From now until the end, the examples are related to a Memory called "Lim".

5.8.1 Memdef Section

The memdef section is the entry point inside the Memory Section. The information needed is:

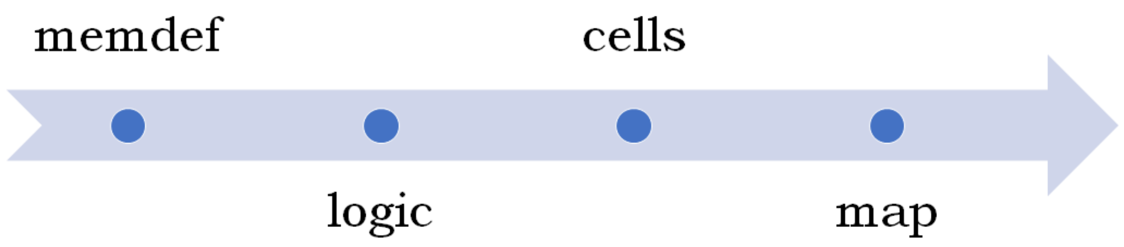


Figure 5.9: Memory sections order

- **Rows**: The number of rows in the memory array
- **Columns**: The number of columns in the memory array
- **Type**: The type of memory cells of the array

Now see a practical example:

```
begin Lim

    begin  memdef

        #Number of rows
         ROWS 16

        #Number of columns
         COLUMNS 16

        #Type of memory
         TYPE FLIPFLOP

    end  memdef

end Lim
```

Listing 5.14: Memdef definition

Up to now, it is possible to use only *FLIPFLOP* type. This type of memory uses Flip Flops instead of classic memory cells. In future releases, we will provide other types like SRam and DRam for example. When the rows and columns are specified, the compiler starts a software routine to check if the internal array matrix is consistent with the memory interface chosen in the Init section, when the LIM model is invoked. For example, a memory with only one column having an input/output parallelism of a value higher than one is not allowed, because when DExIMA synthesizes a memory it doesn't know how to construct the memory interface circuitry. If the parameters chosen in the memdef section are not consistent, the compiler reports an error and, if possible, a suggestion to resolve the inconsistency.

5.8.2 Logic Section

The logic section is similar to the Init section. The syntax is the same, but the components instantiated inside are used only inside the reference memory. Moreover, the name of the instances can be the same as the components instantiated inside the init section because these components live in different namespaces. DExIMA has three different namespaces. We have seen two of them. The largest is the Architecture namespace, the components are instantiated in the Init section. The namespace related to the Lim is filled in the Logic Section, the last namespace will be filled in the Cells section. So we can have the same instance name but in three different namespaces. The hierarchy of namespace is shown in Figure 5.10.

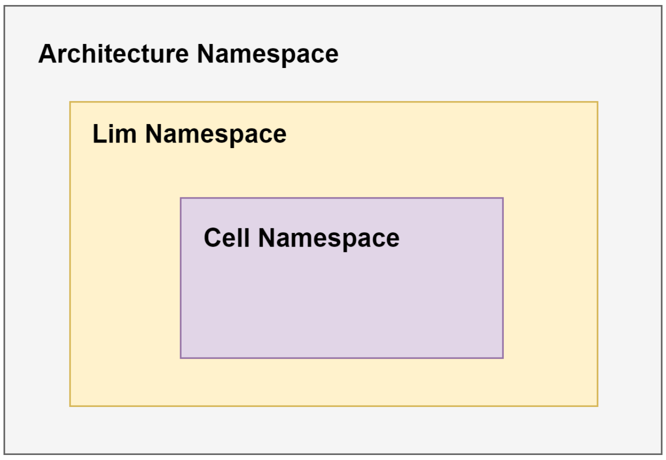


Figure 5.10: DExIMA Namespaces

A practical example is:

```
begin Lim

    begin  memdef

        ROWS 16
        COLUMNS 16
```

```
            TYPE FLIPFLOP

      end  memdef

      begin  logic

            #Nand with the same name
            #of one created in the init section
            NAND Nand(2)

      end  logic

end Lim
```

Listing 5.15: Logic section example

5.8.3 Cells Section

The cells section is similar to the Init/Logic Sections but in this case, we work in the last namespace. The Cell namespace is related to the smart Cell, composed of Memory and a logic chosen by the user to create the custom LiM. We use the link operator to push a logic element inside a cell. In Figure 5.11 there is the example of the syntax used.

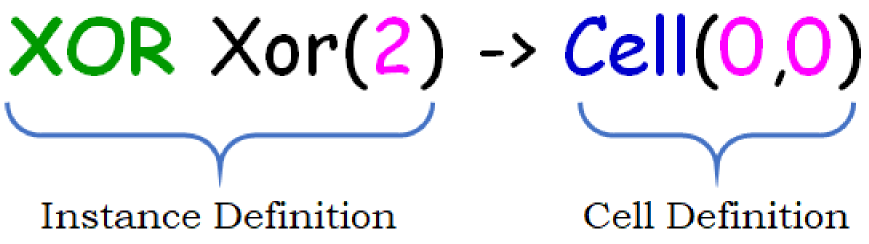


Figure 5.11: Cells section syntax

On the left, we create a component like the previous one, but now we add the keyword *Cell* on the right using the link operator and specify between round brackets the two coordinates to identify uniquely a cell inside the memory array. The indexing starts from zero to the number of rows/columns minus one. By default in every cell, there is only the memory cell, but in this section, we can choose to add inside it all the logic we want. When we push inside the cell a Nand for example, we are free to add how many components we want. For example, if we want to fill an entire memory 16x16 with Nand gate we can look at the example 5.16 (all the previous sections are omitted to focus on what we want to show).

```
begin Lim

    begin  cells

        #Fill the memory with Nand Gate
        for i in  range(0,1,15){ for j in  range(0,1,15) {
    NAND Nand(2) -> Cell($i,$j) } }

    end  cells

end Lim
```

Listing 5.16: Cells section example

5.9 Memory Map Section

The Memory Map Section is similar to Map Section. The syntax is the same. The main difference is that the component linked in this section are only the component belonging to the second and third namespace. Now we look at how to reference components contained inside the memory cell (third namespace). We need to specify in the name the coordinates (x,y) of the cell in the memory array. To access the Memory cell and not the logic inside we use the keyword *Memory*.

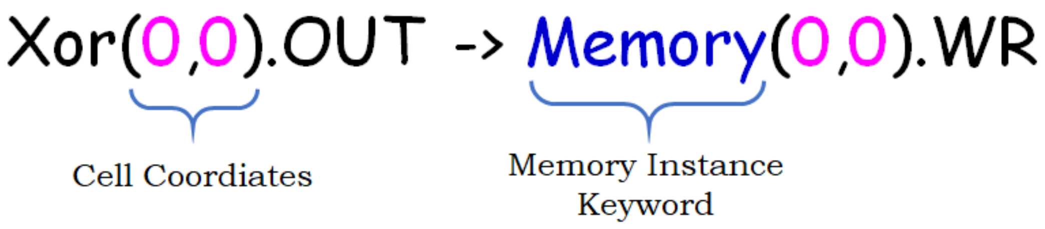


Figure 5.12: Memory Map section syntax

The other components outside the cells are specified like previous. It is important to underline that in order to use a component inside this section, this must be instantiated in the logic or cell section of the same memory.

```
begin Lim

```

```
    begin  map

        #Connection of cell component with out of cell component
        Nand(0,0).OUT -> Xor.IN0

        #Connection of memory output with a component inside
        #another cell
        Memory(2,4).RD -> Xor(0,1).IN2

    end  map

end Lim
```

Listing 5.17: Memory Map section example

5.10 Instructions Section

At this point, we have created all the components of the circuit and we created also the connections between them. Now we need to create an Instruction Set that our architecture can perform. First of all, we define two types of instructions:

- **INSTRUCTION**: Creates an instruction where the considered components are the gates of the first namespace, the Architecture Namespace.
- **LIM_INSTRUCTION**: Creates an instruction where the considered components are the gates of the second and third namespace, the namespace related to the LiM.

To create an instruction we simply use these two keywords followed by the name of instruction we choose. For the LiM instruction, we need to specify the name of the LiM who is referenced before the instruction name. Now we see an example of these facts in 5.18.

```
begin  instructions

    #Create an instruction called subtraction
     INSTRUCTION subtraction

    #Create a Lim instruction called sum
    #referenced with Lim memory
     LIM_INSTRUCTION Lim sum

```

```
end  instructions
```

Listing 5.18: Instructions creation example

Also in this case, like the memories, the order of definition of the instructions is important, in fact, after the definition of the names we need to describe each instruction in detail with the same order. To describe each instruction, we open a section called with the same name of the instruction (inside the instructions section) after all the instructions definitions. First of all, when we open a particular instruction section we need to specify a parameter called **PIPELINE**. This parameter can be both zero and more than zero, it specifies how many pipeline stages the instruction has. From the path's point of view, the number of paths is the value of pipeline plus one.
Now we see a practical example of the definition of instructions and pipeline:

```
begin  instructions

     INSTRUCTION subtraction

     LIM_INSTRUCTION Lim sum

    begin subtraction

        #Pipeline definition
         PIPELINE 0

        #Need more code

    end subtraction

    begin sum

        #Pipeline definition
         PIPELINE 1

        #Need more code

    end sum

end  instructions
```

Listing 5.19: Instructions definitions and pipeline example

The code shows in 5.19 shows how to open instruction and set the pipeline value, but the code to complete an instruction description is not enough, as written in the comments.
Each instruction is described completely using at least two subsections:

- **Power**: used to compute the power performance of the instruction
- **Path**: used to compute the timing performance of the instruction

The Power section is only one, instead of the Path section that is equal to the pipeline value plus one. The path section has an index that starts from zero to the pipeline value. They are indexed using the syntax "*path[index]*" and must follow the growing index order.

5.10.1 Power Section

The Power section is used to compute the performance of a group of gates in terms of power. The power section needs a list of instances. When the instruction is called, the dynamic energy dissipated is the sum of all the dynamic energies dissipated of each component in the list. To include our component we specify the name of the instance. It is possible to call the same instance multiple times if needed. We need to remember that the usable components inside the section are the components included in the referenced namespace of the instruction. An example is represented in 5.20.

```
begin  instructions

     INSTRUCTION subtraction

    begin subtraction

         PIPELINE 0

        begin  power

            #List of power gates used
            Gate1
            Gate2
            for i in  range(3,1,10){ Gate$i }

        end  power

        #Need more code

    end subtraction

end  instructions
```

Listing 5.20: Power Section example

5.10.2 Power Attributes

The attributes are special options used to modify the computation behavior of the dynamic energy. We have two types of attributes:

- **Read**: Used to compute the dynamic energy for the Read operation.
- **Write**: Used to compute the dynamic energy for the Write operation.

These two attributes are usable only for the LiM memories and for the memory cells. To use the attribute we use the link operator, specifying the instance name on the left and the attribute name on the right. An example is shown in 5.21.

```
#Architecture instruction
begin power

    Lim
    Sram -> Read
    Lim2 -> Write

end power

#Lim instruction
begin power

    Memory(0,0)
    Memory(1,0) -> Read
    Memory(0,2) -> Write

end power
```

Listing 5.21: Power attributes examples

5.10.3 Path Section

The Path Sections are used to compute the timing performance of the instruction. Inside a path section, we insert a list of components taking part in an eventual critical path. The section computes the sum of all the propagation delays of the instances. We have different paths related to the pipeline index so we have different possible critical paths. DExIMA evaluates which is the critical path between them and which instruction is responsible.

```
begin instructions

```

```
     INSTRUCTION subtraction

    begin subtraction

         PIPELINE 1

        begin  power
            #Power instances
        end  power

        #Path 0 list
        begin  path[0]

            Nand
            Xor

        end  path[0]

        #Path 1 list
        begin  path[1]

            Nor
            Inverter

        end  path[1]

    end subtraction

end  instructions
```

Listing 5.22: Path Section example

5.10.4 Parallel Paths

There is a possibility to have multiple parallel paths in each pipeline stage. To implement this function, we can use the keyword **break**. This keyword is inserted in a line without anything else (substitute an eventually instance name) to break the upper part to the lower part. All the parallel paths are included between two break lines or the start/end of the path section. See the example 5.23 to have a better understanding of the use of this keyword.

```
begin  path[0]

    #First parallel path
    Gate0
    Gate1

    #Second parallel path
    break
    Gate0
    Gate2

    #Third parallel path
    break
    Gate3
    Gate4

end  path[0]
```

Listing 5.23: Parallel Paths example

In the results related to the path, DExIMA highlights only the largest delay parallel path.

5.10.5 Timing Attributes

The timing attributes have the same logic as power attributes, but in this case, they are used inside a path section, evaluating different aspects related to the timing of a component. The available timing attributes are:

- **Read**: Used to compute the delay of the Read operation.
- **Write**: Used to compute the delay of the Write operation.
- **Clock_to_output**: Used to compute the Clock to output delay.
- **Setup**: Used to compute the Setup time.
- **Hold**: Used to compute the Hold time.
- **Contamination**: Used to compute the Contamination delay.

The Read/Write attributes are used by the LiM memories and the memory cells. The other attributes, except the Contamination one, are used by the Flip Flops components. The syntax is similar to the power attributes. The timing attributes are very useful especially when we need to consider also the clock to output and setup time including in the critical path computation. Now we see the example 5.24 of the use:

```
begin  path[0]

   #Clock to output
   FlipFlop1 -> Clock_to_output
   Nand
   Not
   FlipFlop2 -> Setup

end  path[0]
```

Listing 5.24: Timing attributes example

5.10.6 Path Section Example

A practical example is proposed to clarify the concepts. Firstly, we draw the circuit and all the path of the circuit like shown in Figure 5.13.

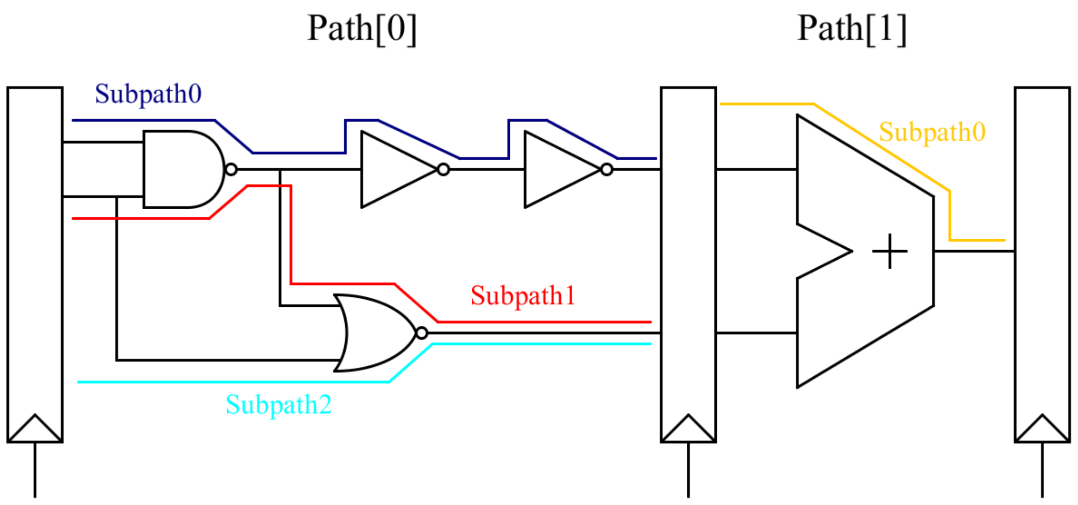


Figure 5.13: Multiple path circuit example

We highlighted all the paths and sub-paths in the circuit. We can identify 3 pipeline stages that imply 2 paths. The first path has three sub-paths, the second has only one. The resulting code is:

```
begin  path[0]

   #Subpath0
   FlipFlop1 -> Clock_to_output
   Nand
   Not1
```

```
    Not2
    FlipFlop2 -> Setup

    break

    #Subpath1
    FlipFlop1 -> Clock_to_output
    Nand
    Nor
    FlipFlop2 -> Setup

    break

    #Subpath2
    FlipFlop1 -> Clock_to_output
    Nor
    FlipFlop2 -> Setup


end  path[0]

begin  path[1]

    FlipFlop2 -> Clock_to_output
    Adder
    FlipFlop3 -> Setup

end  path[1]
```

Listing 5.25: Multiple path circuit code

It's obvious that if we know in advance that one of the subpaths is the critical path, it is useless to insert all the paths since only the critical one is considered.

5.11 Code Section

The code section is the last section of the *dex* file. It is used to specify the algorithm that the circuit performs using the instruction set defined before. The syntax is very simple, we write the name of instruction followed by an integer that tells the compiler how many times that instruction is executed. The order of instruction is the same as the algorithm. There is the possibility to do multiple instructions at the same time. To do concurrent instructions we simply do a list separated by a comma in the same line. At the end of the line, we specify the multiplicity of the concurrent instruction. We show in 5.26 an example of code.

```
begin  code

    #Single instructions
    multiplication 12
    subtraction 4
    division 3
    multiplication 45

    #Concurrent instructions
    multiplication, division 7
    subtraction, multiplication, division 13

end  code
```

Listing 5.26: Code section example

5.11.1 Timing and Power Interpretation

The energy is computed through this formula:

$$E_{line} = \sum_i E_i \cdot n \tag{5.1}$$

Where E_{line} is energy dissipated in one line of the code and E_i is the energy of the $i-th$ instruction. n represents the multiplicity inserted at the end of the line. For the timing interpretation, the number of clock steps required by one line of code is computed with the formula:

$$N_{line} = \max_i(N_i) \cdot n \tag{5.2}$$

where N_i is defined by:

$$N_i = Pipe_i + 1 \tag{5.3}$$

The N_i is the number of clock cycles needed to complete the $i-th$ instruction. It is equal to the number of Pipeline factor $Pipe_i$ of $i-th$ instruction plus one. The computation is simply the clock step number of the slowest instruction in the line multiplied by the multiplicity factor. It is intuitive now because if we have a different pipe stage of the instruction we need to wait until the time of the instruction which has the highest number of pipeline stages.

Chapter 6

DExIMA Files Descriptions

DExIMA is a software that interacts with the user using files. The input of our program is both a file and also an output. This is an advantage because it eases the interpretation of the results for very complex designs. The Files used in the DExIMA program are:

- **Input dex File**: It is the script where the information is inserted to compute the performance of our architecture.
- **Output log File**: It is the output we get after the compilation process and contains the information about the instances created.
- **Input TechFile**: It is the file used by the program to load all the parameters used by the specific technology.
- **Output dof File**: It is the output file where are written all the performance information of the circuit.

The dex file is widely discussed in the Language Chapter, in this small chapter, we see the other inputs/outputs files inside the program.

6.1 Log File

The log file is generated after the compiling stage. It is very useful to check and understand if the architecture we have in mind is the same as what we have written in the dex file and what the compiler interpreted from it.

6.1.1 General Information

The first part of the log file gives to the user very general information like:

```
The compiled architecture contain:

Architecture modules: 119
Number of Lim: 1
Number of models: 12
Number of instructions: 3
```

Figure 6.1: Log File General Info

The Architecture modules are located in the first namespace, including the LiM modules. After, there are the memories we instantiated. The number of models is equivalent to the number of Printers used in our architecture. For example, if we use Nand gate and Inverters in our architecture, we have 2 models. If we insert a memory, the number increases by 2 because we have the model of the memory and the model of the memory cells. In conclusion, we have the instructions that can be used in our instruction set.

6.1.2 LiM Data

The second part is a list of more detailed information about the memory instantiated. The info written in the log file related to the memories is shown in Figure 6.2. The first information is the Name of the Lim. After, we list the geometry of the memory related, which are: the number of rows, columns, number of cells, and their type, in this case, FLIPFLOP. The information about the periphery of the memory is explained using the parallelism of the address port and the data parallelism. The Cell modules are the sum of the memory cells and the logic inside them. In this example, we have $385 - 256 = 129$ numbers of logic modules inserted inside the cells. On the contrary, the total modules are composed of the sum of the cells section with those that are part of the Inter-cell logic. In this case, we have $397 - 385 = 12$ Inter-Cell logic modules. These chunks of information are repeated for each memory.

```
Lim detailed information:

Name of Lim: Lim
Out of memory cell modules: 12
Memory rows: 16
Memory columns: 16
Number of cells: 256
Type of memory: FLIPFLOP
Address bus parallelism: 4
Data bus parallelism: 16
Cell modules: 385
Total modules: 397
```

Figure 6.2: Log File Memories Info

6.1.3 Instructions Data

Also in this case we have a list of information related to each instruction. We have the name of the instruction and the type. The only difference between the two types is that for the LiM instruction there is present also the reference LiM. The number of instances inserted in the power section is reported also. This number takes into account multiple instances: if, for example, we insert four times the same instance we increase this number by four. After the specification on the number of paths, that is equal to the pipeline value, there is a list of the information about all the path/subpath. For each path, there is a list of each subpath with the number of modules specified inside. The same considerations made in the power section can be done for the enumeration of instances. The Figure 6.3 show what explained before.

6.1.4 Code Data

In the part related to code, we have simply the information of how many times that instruction is used in our algorithm. An example of this part is shown in Figure 6.4.

```
Instructions detailed information:

Name of Instruction: Division
Type of instruction: INSTRUCTION
Number of paths: 3
Number of power instances: 3
Path Name: path[0]
Subpath0 modules: 1
Subpath1 modules: 2
Path Name: path[1]
Subpath0 modules: 1
Subpath1 modules: 3
Path Name: path[2]
Subpath0 modules: 1

Name of Instruction: Shift
Type of instruction: LIM_INSTRUCTION
Lim reference: Lim
Number of paths: 1
Number of power instances: 267
Path Name: path[0]
Subpath0 modules: 267
```

Figure 6.3: Log File Instructions Info

```
Code:

Code multiplicity:
Division: 49
Multiplication: 137
Shift: 40
```

Figure 6.4: Log File Code Info

6.1.5 Final Information

The last part shows the number of clock steps used to complete the algorithm and the total number of modules of the entire architecture, including modules inside and outside of memory. Other useful information is the occupied memory, that is the memory occupied by all the program, not only to the created architecture. The value of memory occupied by the program when we do not instantiate any components is about 5 MB. This means that we have an offset of 5 MB to the dimensions of our architecture. If we have a big architecture that occupies much more than 5 MB, the offset is negligible. In this case, the memory occupied by the architecture can be considered correct with an addiction of small error. If the architecture is small, to the order of some kB, we need to consider the offset and the fact that the 5 MB are noisy (can go from 4.8 to 5.2). In the end, we have the time needed to compile the dex file.

```
Total clock steps: 186
Total modules: 516
Occupied memory: 5.65 MB
Compilation time: 2.39 s
```

Figure 6.5: Log File Final Info

6.2 Dof File

The Dof File contains the performance data of the simulated circuit. We recall that the Dof file is the output of the simulator at the end of all the processes inside the program.

6.2.1 General Information

In the first part, we have the total performance of the whole circuit. In the Figure 6.6 we can see an example. The first information we have is the working clock period and the related frequency. The clock period displayed is the value used by the tool to compute all the derived parameters, like the execution time of the total algorithm or the computation of the dynamic power. If the user defines the clock period using a built-in constant, the period is set, if not defined the clock period is chosen using the critical path of the circuit. If, for example, the clock period chosen by the user is lower than the critical a message appears:

```
Warning! The clock period used is lower than the critical path
```

```
Simulation results

Clock period: 2.2 ns
Frequency: 454.545 MHz
Critical Path Instruction: subtraction
Critical Path name: path[0]
Critical Path: 1.08162 ns
Area: 261461 um^2
Dissipated dynamic energy: 194.984 nJ
Dissipated static energy: 27.5022 nJ
Total dissipated energy: 222.486 nJ
Static power: 10.8421 mW
Execution time: 2.5366 us
Average dynamic power: 76.8682 mW
Total power: 87.7104 mW
Total clock steps: 1153
```

Figure 6.6: Dof File General Info

The computation considers a clock period lower than the allowed one. Now we have defined how our clock period value is set. The critical path displayed includes also more info indicating who is responsible for our critical path. The information is the instruction responsible for the critical path (in this case called `subtraction` and the name of the path `path[0]`). The value of an area is the sum of all area occupation of the architecture, the value is expressed always in μm^2. All the other units are adaptive in the file, this means that they are not fixed, exempt for the area that has the same time unit. The smallest unit is $femto$ and the highest is *Giga*. All the numbers are expressed in the range $1 - 1000$. We find also the energies related to the dynamic and static dissipation. The power is divided into dynamic and static. The Execution time is the time needed to complete the algorithm. In fact, we can see the presence of the clock steps needed by making the product between the clock period and the clock steps we obtain this time.

6.2.2 LiM Data

In the LiM part, we have the information related to each Memory, in this case, we do not need too much information because most of it depends on which part of the memory logic we are focusing on. The only invariant info is that we take the area of the memory and the static power dissipation like shown in Figure 6.7. The area and static power values consist of the sum of circuit interface, memory cells, Intra-Cell, and Inter-Cell logic. In the end, we also have the parameters related only to the memory circuitry interface, excluding so the other parts of the memory.

```
Memory information

Memory: Lim
Memory area: 258642 um^2
Memory static power: 10.6853 mW
Memory interface area: 48660.4 um^2
Memory interface static power: 1.97333 mW
```

Figure 6.7: Dof File Memory Info

6.2.3 Clock Drivers Data

Now we have an additional part concerning the log File because this part is used to know how the clock dissipates and the information about each clock driver used.

```
Clock information

Clock Driver: ClockDriver
Dissipate energy: 40.9172 nJ
Static Power: 156.81 uW
Area: 2818.94 um^2
Delay: 74.6484 ps
```

Figure 6.8: Dof File Clock Driver Info

The example in Figure 6.8 shows the four main parameters for characterizing a component. The difference is that the clock driver is active for each clock step and the switching activity is doubled compared to the standard gates.

6.2.4 Instructions Data

The next section is used to know the performance of each instruction. Theoretically, the only information about an instruction needed is the path delays and the dissipated dynamic energy. But we add also the parameters of static power and area because are useful to know one or a group of components how much area and static power they consume. All the parameters, except for the delays, are taken into account when we insert a name of a component inside the power section. All the component parameters in the power section are accumulated and written after in the dof file. Remember that there is the possibility to insert a component multiple times and also in this case the parameters are accumulated.

```
Instruction: FlipFlop
Dissipated energy: 3.62085 fJ
Static Power: 73.9347 nW
Area: 1.70472 um^2
Critical path: 86.0781 ps
Critical path name: path[0]
Path delays
path[0] -> 86.0781 ps
path[1] -> 58.03 ps
path[2] -> 36.6322 ps
path[3] -> 8.58404 ps
```

Figure 6.9: Dof File Instruction Info

In Figure 6.9 there is an example where we insert only a Flip Flop component inside an instruction, and four paths to exploit all the timing attribute that this component has. The value of delays for each path is the maximum value between all the subpaths. If a path indicates a null value of zero, it is represented by $0fs$ because fs is the smallest unit usable.

6.2.5 Technology Information

The last part of the File includes all the parameters used by the technology. This can be very useful when we want to compare different technology and know which parameters are used to compute the model performances.
In Figure 6.10 there are all the parameters. The first information is the name of the technology file used, in this example `LOP_45.txt`. After that, we

```
Technology internal parameters

Technology file: LOP_45.txt
Enabled the switching activity computation
Input probability: 0.5
Interconnection overhead: 15%
Standard Cell overhead: 0%
Stack factor: 2
Vdd: 0.9 V
Aspect ratio: 10
Cox: 2.4665 uF/m^2
Leff: 29.1 nm
Beta: 1.85
Diffusion lenght: 72.75 nm
C bottom n: 167.139 pF/m
C bottom p: 199.009 pF/m
C sidewall n: 821.668 pF/m
C sidewall p: 707.241 pF/m
C interconnections: 183.13 pF/m
Unitary Mos width: 0.291 um
Cin n mos: 0.0785909 fF
Gamma: 1.3705
Rho: 0.74081
Ion: 543.14 uA/um
Ioff: 3.1186 nA/um
Igate: 24.29 nA/um
Ion unitary mos: 158.054 uA
Ioff unitary mos: 0.907513 nA
Igate unitary mos: 7.06839 nA
```

Figure 6.10: Dof File Technology Info

can find the value of the probability associated with the inputs when the switching activity calculation of the circuit is required. The percent values are related to the area overhead used by the Interconnections and the area overhead considering a standard cell design. To be more clear:

$$Area = A_0 \cdot (1 + Interc_{over}) \cdot (1 + StdCell_{over}) \tag{6.1}$$

Where A_0 is the area without overhead, $Interc_{over}$ is the interconnection overhead, and $StdCell_{over}$ is the standard cell overhead.
All the values written after are all derived from the base parameters written in the TechFile. For example, we have all the capacity values computed (all the capacity are described in the chapter related to the models). At the end of this chapter, all the parameters will be explained and easily comprehensible.

6.3 Technology File

To conclude this chapter we see how the technology file is formatted (TechFile) and how to add the model parameters that can be used by DExIMA. In Figure 6.11 there is the TechFile used in the previous example of the Dof file. The file is a simple text file where we have two columns. The left column represents the name of the variables. This name must be respected, because if we insert a wrong name, DExIMA returns an error indicating a group of variables in the TechFile (including the wrong one). The right column contains the values of the variables, the units are the composition of the fundamental units without using multiple of them. For example, some capacity values are usually specified in pF/m but in the tech file are in F/m. The File is simply converted in a map having the name of the variable as a key and a float variable for the value, and used after to compute the derived parameters stored directly in the Technology class.

```
Year 2005
Lgate 45.1e-9
Xj 20e-9
Gamma 0.8
Inter_over 0.15
Cell_over 0
Aspect_ratio 10
Beta 1.85
Vdd 0.9
Cox 2.4665e-06
Ion 543.14
Ioff 3.1186e-3
Igate 24.290e-3
CJ0N 2.7e-3
CJ0P 3.3e-3
CJSWN 9.2e-10
CJSWP 8.0e-10
CGD0N 1.35e-10
CGD0P 1.0e-10
MJN 0.38
MJP 0.45
MSWN 0.22
MSWP 0.265
PBN 0.85
PBP 0.87
PBSWN 0.67
PBSWP 0.76
C_Interc 1.8313e-10
```

Figure 6.11: TechFile LOP_45.txt

Chapter 7

DExIMA Hardware Models

7.1 Introduction

This chapter shows how the gate models are built and described. The models are created in a hierarchical approach: the basic gates are described at the transistor level and modeled. The more complex models are constructed starting from the base models. For example, an And gate can be seen like the cascade of a Nand gate and an Inverter. But the Nand and the Inverter have a low-level description, constructed from the transistor configuration. The description of the models starts with the definition of the transistor capacitance and defines some parameters in common to save compilation time. All the parameters will be normalized to a reference value, so the computations are less complex and permit to speed up of the software.

7.2 Transistors Capacitance

Let's start with the main capacitance of a transistor, the input capacitance. The input capacitance is the result of the capacity of the gate MOS structure, and the overlapping between the drain/source diffusion and the gate structure. All the formulas have a double subscript used for the n and p mos transistors. With the n/p notation, we can write a single formula, instead of two different ones (one with n and one with p). All the formulas and more detailed information can be found in [8].
The gate capacitance is obtained by the formula:

$$C_{MOS,\,n/p} = C_{OX} \cdot W_{n/p} \cdot L_{eff} + 2 \cdot W_{n/p} \cdot C_{overlap,\,n/p} \tag{7.1}$$

Where $W_{n/p}$ is the width of the transistor, L_{eff} the effective length of the gate, and $C_{overlap,\,n/p}$ is a technology parameter for the overlapping capacitance.
After the gate capacitance, we need to define also the capacitance related to the drain/source junctions. The formula for the junction capacitance is:

$$C_{j\;n/p} = C_{bottom\;n/p} \cdot W_{n/p} + C_{sidewall\;n/p} \cdot perimeter_{n/p} \tag{7.2}$$

Where each member of the equation is computed from these formulas:

$$C_{bottom\;n/p} = C_{j0\;n/p} \cdot \left(1 + \frac{V_{DD}}{2 \cdot P_{b\;n/p}}\right)^{-M_{j\;n/p}} \cdot L_S \tag{7.3}$$

$$C_{sidewall\;n/p} = C_{jsw0\;n/p} \cdot \left(1 + \frac{V_{DD}}{2 \cdot P_{bsw\;n/p}}\right)^{-M_{jsw\;n/p}} \tag{7.4}$$

$$perimeter_{n/p} = 2 \cdot L_S + W_{n/p} \tag{7.5}$$

Where L_S is the diffusion length and the other parameters are technology constants. First, we rewrite the complete expressions of the input capacitance of n and p mos:

$$C_{MOS,\,p} = W_p \cdot \left(C_{OX} \cdot L_{eff} + 2 \cdot C_{overlap,\,p}\right) \tag{7.6}$$

$$C_{MOS,\,n} = W_n \cdot \left(C_{OX} \cdot L_{eff} + 2 \cdot C_{overlap,\,n}\right) \tag{7.7}$$

Now we compute the ratio between these two quantities:

$$\frac{C_{MOS,\,p}}{C_{MOS,\,n}} = \frac{W_p}{W_n} \cdot \frac{\left(C_{OX} \cdot L_{eff} + 2 \cdot C_{overlap,\,p}\right)}{\left(C_{OX} \cdot L_{eff} + 2 \cdot C_{overlap,\,n}\right)} = \frac{W_p}{W_n} \cdot \rho \tag{7.8}$$

The capacity ratio, excluding the width ratio, is called ρ, and it is an important parameter. We can rewrite the equation to get the input capacity of the p mos in function of the parameters of the n mos using the equation:

$$C_{MOS,\,p} = \frac{W_p}{W_n} \cdot \rho \cdot C_{MOS,\,n} \tag{7.9}$$

7.3 Inverter Gate Reference

All parameters of the model will be normalized in relation to the inverter gate. To be more specific normalized to the n mos of the inverter. The circuit is well known in Figure 7.1.

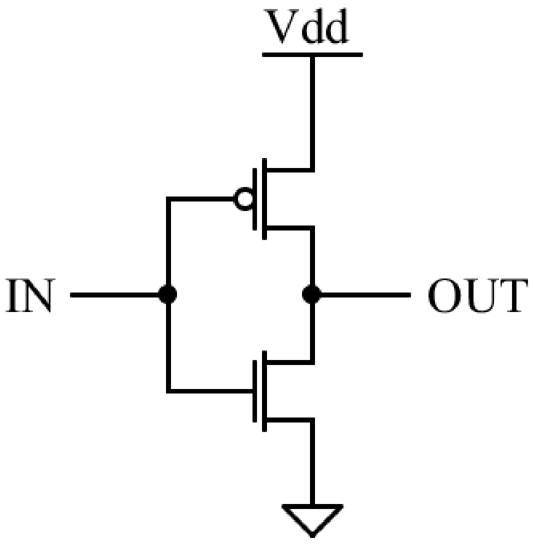


Figure 7.1: Inverter CMOS

We start with the dimension of the transistors, the W_n of the n mos is equal to the minimum aspect ratio defined in the TechFile or specified by the user in the constant section with the built-in constant AR. The formula is very simple:

$$W_n = AR \cdot L_{eff} \tag{7.10}$$

Now in order to choose the dimension of the p mos, we use the standard rules of microelectronics to have the same current in the n and p mos. The difference in the current of p mos and n mos is usually the mobility of electrons and holes. The p mos is usually slower due to the lower mobility of the holes. In general, the current computation of the two transistors is not only influenced by the mobility but also the other parameters that need to be taken into account. So, to have the same current flowing in the p and n mos, we need to do the p mos wider beta times than n mos with the parameter beta equal to:

$$\beta = \frac{I_{on,n}}{I_{on,p}} \tag{7.11}$$

Now have all the ingredients to choose the dimension of the p mos to have the same amount of current:

$$W_p = \beta \cdot W_n \tag{7.12}$$

The value of the normalized widthis defined as:

$$w \overset{\text{def}}{=} W_{norm} = \frac{W}{W_{n,INV}} \tag{7.13}$$

From now we use the notation w to indicate a normalized width. As we can see in the previous definition, we normalize a generic W respect to the width of the n mos of unitary inverter $W_{n,INV}$. With these considerations we get that the normalized value of widths on the inverter are:

$$w_n = 1 \qquad\qquad w_p = \beta \tag{7.14}$$

From this point, all the gate figures will show in this notation the width of the transistors. The same can be done for the capacitance. We define the normalized capacitance value like:

$$c \overset{\text{def}}{=} C_{norm} = \frac{C}{C_{MOS\,n,\,INV}} \tag{7.15}$$

In this case, the computation of the input capacitance of the inverter is a little bit different, in fact, the value of p mos capacitance is not like the width. To find it, we use the equation 7.9. In the formula, we substitute the generic n mos with the unitary inverter n mos and we obtain:

$$C_{MOS,\,p} = \frac{W_p}{W_{n,INV}} \cdot \rho \cdot C_{MOS\,n,\,INV} = w_p \cdot \rho \cdot C_{MOS\,n,\,INV} \tag{7.16}$$

The first result we get is that the p mos capacitance is a function of its normalized width and the inverter's n mos capacitance. If we express this capacitance in a normalized way, we obtain:

$$c_p = w_p \cdot \rho \tag{7.17}$$

Now it's clear that we define all these constants and the normalized approach because they simplify all the definitions of the components and reduce the number of calculations to do. To further simplify the notation, we usually use for the p mos width values that are multiples of β, so we defined the parameter:

$$\gamma \overset{\text{def}}{=} \beta \cdot \rho \tag{7.18}$$

To define uniquely a gate, we need to know the gate behavior in input and output, treating the component as a black box. Firstly, we need to define the fanin and fanout of our component. Different from the classical approach, the fanout, and fanin do not have the reference to the Inverter gate but only to the n mos transistor of the inverter. This simplifies the computation in a relation to how all the previous parameters are defined. The definitions are:

$$fanin \overset{\text{def}}{=} \sum_{port\ inputs} c_i \tag{7.19}$$

$$fanout \overset{\text{def}}{=} \sum_{port\ outputs} c_i \tag{7.20}$$

The fanin is equal to the sum of all the normalized capacitance of a certain input port. The fanout is equal to the sum of the normalized capacitance connected to the output port of the gate. To characterize an output of a gate we need also to take into consideration the output parasitic capacitance of the port.

7.4 Performance Equations

In this section, we explain which formulas DExIMA uses for the performance computation and which parameters of the model are needed to compute it. The first figure of merit we analyze is the delay of the gate. The formula used for the delay computation is:

$$\tau = \frac{V_{DD}}{I_{ON}} \cdot C_{TOT} \tag{7.21}$$

where V_{DD} is the supply voltage, I_{ON} is the on current, and C_{TOT} the total output capacitance.
The I_{ON} is defined like:

$$I_{ON} = I_{DS}\left(V_{GS} = V_{DD}, V_{DS} = V_{DD}, W = W_{ref}\right) \tag{7.22}$$

And the W_{ref} is usually 1 μm, the current is measured in $[\mu A/\mu m]$. The C_{TOT} is the sum of output parasitic capacitance and the capacitance due to the load.
The equation is:

$$C_{TOT} = C_{OUT} + C_L = C_{OUT} + fanout \cdot C_{MOS,n} \tag{7.23}$$

The C_{OUT} is dependent on the type of gate, and it is measured in [F], also the C_L must be in [F]. The load capacitance is equal to the fanout of the gate that is a normalized capacitance, multiplied by the reference capacitance to denormalize the value.
The dynamic energy is computed with the formula:

$$E_{dynamic} = \frac{1}{2} \cdot C_{TOT} \cdot V_{DD}^2 \tag{7.24}$$

All the parameters were already defined. The factor 1/2 is inserted because in one clock step we charge or discharge the output capacitance. For the clock instead, we don't have the factor 1/2 because in one cycle we charge and discharge the output capacitance.
The static power is computed by the equation:

$$P_{static} = I_{leak} \cdot V_{DD} \tag{7.25}$$

The I_{leak} is the leakage current of the gate. It is composed by two parts:

$$I_{leak} = I_{off} + I_{gate} \tag{7.26}$$

The I_{off} is the subthreshold current got by the formula:

$$I_{off} = I_{DS}\left(V_{GS} = 0, V_{DS} = V_{DD}, W = W_{ref}\right) \tag{7.27}$$

It is usually measured in $[nA/\mu m]$. The gate leakage current is defined in a dual way compared to the n/p mos. Figure 7.2 explains the work condition of the gate leakage current. The three current values I_{on}, I_{off} and I_{gate} are computed from a tool developed in Politecnico di Torino called TAMTAMS [9]. This tool is available online and it is a technology simulator based on low-level parameters. The model used for the computations is the MASTAR model [10] developed by STMicroelectronics and is freely distributed on ITRS organization website [11]. As a consequence, we obtain three currents with static numerical values, that does not change according to the parameters (are not functions implemented directly inside DExIMA). One

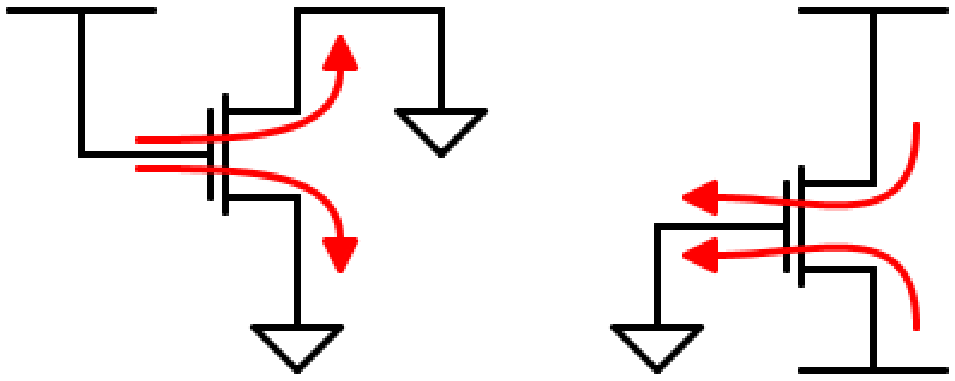


Figure 7.2: Gate current conditions

of the parameters that could vary is the supply voltage, which cannot be used since the current is computed in the point of default voltage, which is indicated in the TechFile. The last important parameter is the area that is simply the sum of all the transistor's dimensions of the gate. It is added also to the area computation a 15% of the value to take into consideration the interconnection overhead. It is also possible to insert in the TechFile a parameter that takes into consideration an overhead related to the standard cell layout.

7.5 Switching Activity Evaluation

The switching activity is a parameter that we use to have a measure how a signal toggles in our circuit nodes. It is defined by the equation:

$$\alpha(n) = p^1(n)p^0(n) + p^0(n)p^1(n) \tag{7.28}$$

where α is the switching activity and n is the node of interest. The $p^1(n)$ and $p^0(n)$ are the probabilities of having a logic value of one or zero at the node n respectively. The equation can be rewritten in the form:

$$\alpha(n) = 2p^1(n)(1 - p^1(n)) \tag{7.29}$$

This equation says that the switching activity depends on the probability to have a value of logic 1 in the node n. In reality, the switching activity is a function of several parameters, but for our computation, we focus on the dependency of the logic boolean function of our gate like in [12]. To have an idea of the complete probabilistic model of the switching activity it is possible to check [13]. First of all, we can rewrite the equation for the dynamic dissipation considering also the switching activity:

$$E_{dynamic} = \frac{1}{2} \cdot C_{TOT} \cdot V_{DD}^2 \cdot \alpha \tag{7.30}$$

For example, we have a gate with fixed input values and the logic values can be 1 or 0. This means that the associated input probability is p = 1 or p = 0. If the inputs do not change, it means that there is no commutation in the output. This is reflected by the value that the switching activity assumes, which is zero. This means that we do not have dynamic power dissipation. On the contrary, if we have a signal like a clock, that switches every clock cycle, we have a switching activity of 1. To be more precise, with this formula the clock switching activity is 2 because we have two transitions in the same cycle and not only one. This characteristic, if we neglect the glitches, it is only for the clock signal. To compute the output probability of a logic gate we need to know the value of the probability in all of its inputs, by default DExIMA puts this probability to 0.5. This means that we have the same probability of having the value 0 or 1 in the input. Having the input probability and knowing the logic function, we can compute the output probability. For example, if we have 2 And gates, we have an output of 1 only if both inputs are to 1. This means:

$$P_{OUT} = P_1 \cdot P_2 = 0.5 \cdot 0.5 = 0.25 \tag{7.31}$$

and the value of the switching activity is:

$$\alpha_{OUT} = 2 \cdot P_{OUT} \cdot (1 - P_{OUT}) = 2 \cdot 0.25 \cdot (1 - 0.25) = 0.375 \tag{7.32}$$

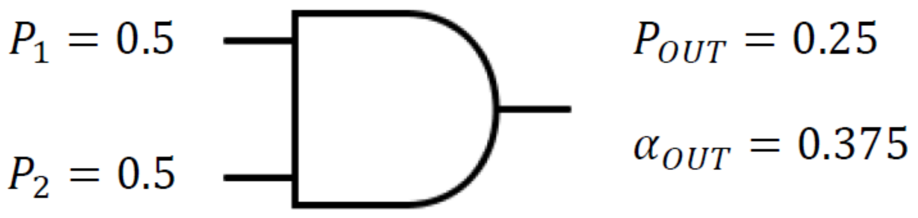


Figure 7.3: And Gate Output Switching Activity

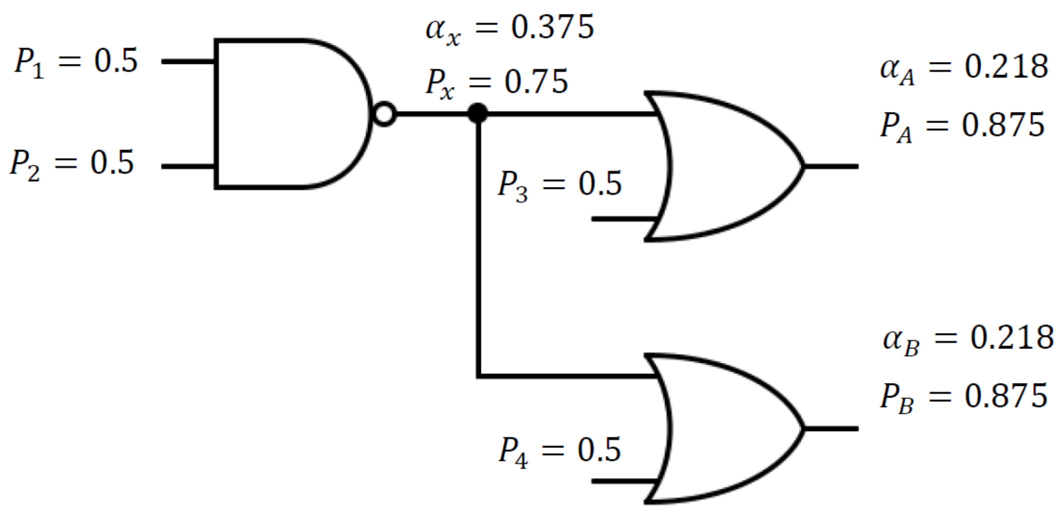


Figure 7.4: Probability propagation along the circuit

With this method it is possible to compute the switching activity only in some situations, propagating the probability when we connect gates between themselves. In fact, for example, if we have multiple gates connected, we can propagate the probability and compute the switching activity like in Figure 7.4. In this case, the computation is simple, but it could be more complex and problematic when we have a sequential circuit in which time takes part in the computation, and there are also problems when having some loops inside our architecture. To propagate the probability, we need also to add more complexity in the data structure because we need to remember the whole architecture links and create also some algorithms in order to propagate all the probability in the circuit. This means that the tool becomes more complex and slower. For all these reasons we choose to not implement the propagation of probability but to take an approach of "isolated gate", meanings that the switching activity of each gate is computed putting in inputs the user-chosen probability that is independent of how is connected to the gate. The result is the worst case approach because the dissipated energy gate is higher than the propagated probability. The values of all the probability for each gate are shown below in 7.1. But not all these formulas are used because we have a hierarchical approach, so only the gate in the section of the Elementary gate models has this type of computation, the other simply uses the isolate gate computation of the components.

Gate	Output Probability
NOT	$1 - P_1$
AND,n	$\prod_{i=1}^{n} P_i$
OR,n	$1 - \prod_{i=1}^{n}(1 - P_i)$
NAND,n	$1 - \prod_{i=1}^{n} P_i$
NOR,n	$\prod_{i=1}^{n}(1 - P_i)$
XOR,2	$P_1(1 - P_2) + (1 - P_1)P_2$
XNOR,2	$1 - [P_1(1 - P_2) + (1 - P_1)P_2]$

Table 7.1: Table of output probabilities

7.6 Stack Effect Model

The stack effect is an effect that helps to reduce the leakage current of a gate. It happens when we put multiple transistors in series. When we have a transistor in the off state, the leakage current is the subthreshold current. When we put two transistors in series, the current that flows in the devices are smaller and it is a function of some technology parameters like the Drain Induced Barrier Lowering (DIBL). For two transistors in series, it is not so difficult to compute the reduction that can be also in the order of ten times and more. When we have more than two transistors this effect is more relevant, but it is hard to compute each time the exact value. To avoid this problem and take into consideration this effect we created a worst case model to emulate this effect. The principle is very simple: suppose to have multiple transistors in series like the in Figure 7.5. The approximation is to have a homogeneous partition of the voltage across the transistors. The reduction is related to the partition with the formula:

$$I_{stacked} = \frac{1}{n^{\alpha}} \cdot I_{off} \tag{7.33}$$

Where n is the number of stacked stages and α is the stack factor. The default value is 2. With this value, the current is higher than the typical stack effect values. If we choose a stack factor of 0 we do not consider this effect. If we make a comparison of the values got in [14], the value obtained with two transistors stacked is 1/8.4, our model return 1/4 using a stack factor of 2. Three transistors stacked got a value of 1/15 versus the 1/9 got with our model.

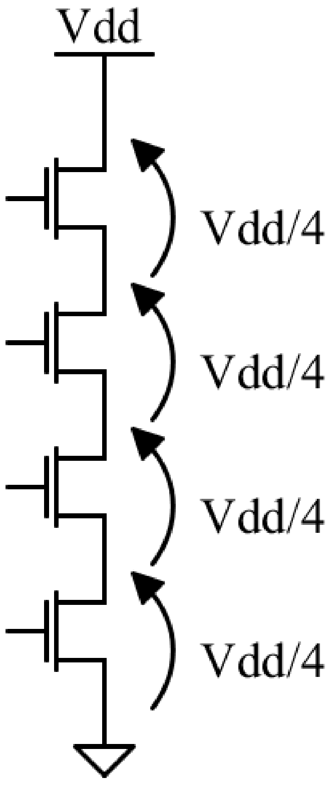


Figure 7.5: Transistors Stacked

7.7 Interconnections Model

The interconnections model is implemented only for the connections between two cells of the memory. This because the problem is to evaluate the length of the wire when we connect two gates. The problem when we connect two components between themselves is to determine their distances. We only know the area of the cells and using this, we are able to determine the distances between two cells. If we want to know the distances between components that are not inside a cell of the array, we need to know their precise position in the space. To get this information, we need to do a kind of virtual place and route of our components, but this is a tricky operation. This is why the interconnection is taken into account only by two components contained inside the cells of the memory array. The same discussion can be done with gates inside the memory but out of the cells. But for the connection between two cells, this can be done with a smaller effort. When we connect two components that are inside a cell (Memory or Logic) we can compute the distance expressed in number of cells. The wire length computation uses a simple Manhattan method, doing the sum of the difference between the two coordinates. The length is simply:

$$l = |x_2 - x_1| + |y_2 - y_1| \tag{7.34}$$

where the two cells have coordinates (x_1, y_1) and (x_2, y_2), and the l is the normalized length. The normalized length is expressed in the number of cells, this is useful to generalize the computation, because if we change the type of memory cell, this parameter is independent, and the effective length is computed after, as soon as we have defined the type of cell. Now we see how it computes the effective length having the normalized one. The model is the following: we only know the area of the cell, independently from the type of the cell self. We call this area A and we suppose it is a square. Consequently, the side of the square is $a = \sqrt{A}$. If we imagine having all the cells adjacent to the nearest cells in the memory, the distance between the two centers of these squares is exactly a. Now, to compute the effective length we do:

$$L = l \cdot a \tag{7.35}$$

Figure 7.6 is show what we explained before.

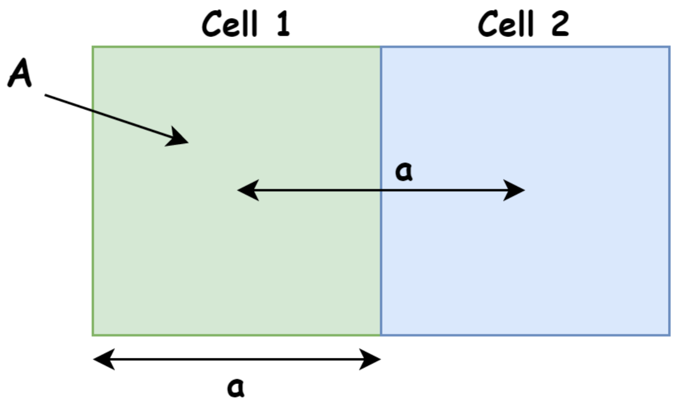


Figure 7.6: Cell's Pitch

The model used for the interconnection is explained in [15], which is exploited in TAMTAMS. From TAMTAMS we get the value of capacity per unit length of the interconnections and it is inserted in the TechFile. Now it is possible to get the value of the capacity by multiplying it by the length:

$$C_{inter} = C_{inter,unitlength} \cdot L \tag{7.36}$$

now we normalize this value of the capacity to get the value of fanout to add to the output of the component involved in the connection.

$$fanout = c_{inter} = \frac{C_{inter}}{C_{MOS,n}} \tag{7.37}$$

7.8 Elementary Gate Models

In this section, the available models in DExIMA are explained.

7.8.1 Inverter

We start from the simplest gate, the inverter. We write all the steps only for this first case, for the others, we write only the result.

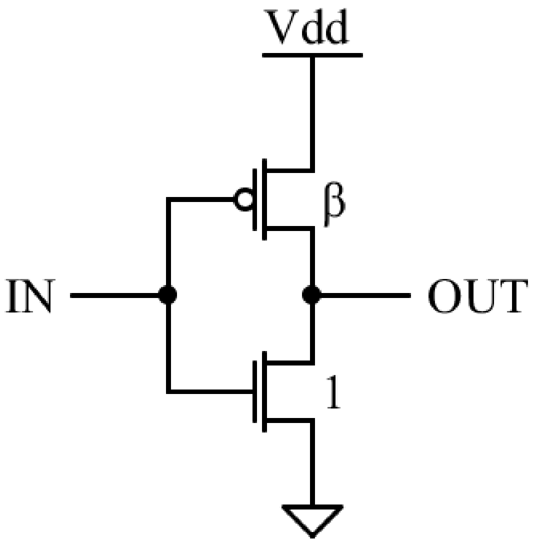


Figure 7.7: Inverter gate

The two normalized capacitance in input are:

$$c_n = 1 \qquad c_p = w_p \cdot \rho = \beta \cdot \rho = \gamma \tag{7.38}$$

The result fanin is:

$$fanin = 1 + \gamma \tag{7.39}$$

The output capacitance is:

$$C_{OUT} = C_{jn}(W_n) + C_{jp}(\beta W_n) \tag{7.40}$$

From this point to simplify the notation with the symbol W_n, we refer to the n mos of the unitary inverter. For the leakage current computation, the process is very simple: we compute the leakage current for each configuration of the inputs and make the average. In this case, we have only two combinations of inputs, and making the computation we get:

$$I_{leak} = \frac{1}{2}\left[I_{gate,p} + I_{off,p} + I_{gate,n} + I_{off,n}\right] \tag{7.41}$$

The area is simply:

$$A = L_{MOS} \cdot W_n \cdot (1 + \beta) \tag{7.42}$$

The L_{MOS} is the length of the total mos device. We have the factor $1 + \beta$ that is the sum of the normalized width in the input. In this and the other devices, we always find this term.

7.8.2 Nand gate

The parameters are computed for a generic gate with n inputs. The reference Nand 2 is shown in Figure 7.8. We use the dimension of the transistors to have the same characteristic as the reference inverter. The dimension of the p mos remains the same because we simply add another transistor in parallel. Differently, for the n mos, we increase the dimensions, because the resultant transistor obtained by the series, is like a single transistor with longer L. To avoid this, we increase the width to maintain the same aspect ratio. This implies that the dimension of the n mos is n. The value of fanin of a generic input is:

$$fanin = n + \gamma \tag{7.43}$$

For the output capacitance, we have only one n mos connected, and n p mos connected in parallel, implying an output capacitance of:

$$C_{OUT} = C_{jn}(nW_n) + n \cdot C_{jp}(\beta W_n) \tag{7.44}$$

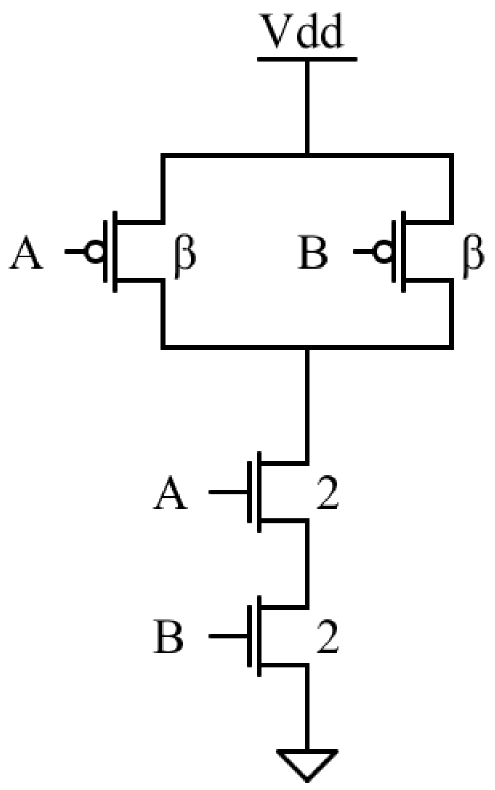


Figure 7.8: Nand 2 inputs gate

To compute the leakage current, we have to consider all the combinations of inputs. This can be complex for a high number of inputs. For this reason, we try to compute it for a low number of inputs, and we were able to extract a mathematical relation that links the number of inputs and the leakage current. The computations are done using the same technique in [14]. The formula we get from the subthreshold current consider the stack effect is:

$$I_{off} = \frac{1}{2^n} \cdot \left[n \cdot I_{off,p} + I_{off,n} \sum_{k=1, n_z \neq 0}^{2^n - 1} \frac{1}{n_z^{\alpha}} \right] \tag{7.45}$$

Where n_z is the number of zeros of equivalent binary input configuration. If we compute for $n = 2$ and without stack $\alpha = 0$ we get:

$$I_{off} = \frac{1}{4} \cdot \left[2 \cdot I_{off,p} + 3 \cdot I_{off,n} \right] \tag{7.46}$$

The other contribution of the leakage current is the gate current, the formula we get is:

$$I_{gate} = \frac{1}{2^n} \cdot \left[\sum_{k=1, n_z \neq 0}^{2^n-1} I_{gate,p} \cdot n_z + (2^n - 1) \cdot I_{gate,n} \right] \tag{7.47}$$

that can be rewritten in a simple way:

$$I_{gate} = \frac{1}{2^n} \cdot \left[\frac{2^n}{2} \cdot n \cdot I_{gate,p} + (2^n - 1) \cdot I_{gate,n} \right] \tag{7.48}$$

By summing these two components, we get the total leakage current. If we try, for example, to use $n = 3$ we get exactly the formulas in [14]. The only differences are: firstly the value of the stack effect that was explained previously, and then there are also terms related to fractions of gate leakage that, we do not consider for the model. The gate leakage fraction consists of considering the potential conditions represented in Figure 7.2, only to the drain or the source.
Finally, the area of the gate is:

$$A = L_{MOS} \cdot \left(n^2 \cdot W_n + n \cdot W_p \right) = L_{MOS} \cdot n \cdot W_n \cdot (n + \beta) \tag{7.49}$$

7.8.3 Nor gate

The Nor gate is the dual component of the Nand gate. Also, in this case, we consider the generic gate with n inputs. In Figure 7.9 we see the version of the circuit with 2 inputs. The fanin of the gate is higher than the Nand gate because the part of the series transistor is the p mos that are β times greater. The value is:

$$fanin = 1 + n \cdot \gamma \tag{7.50}$$

The value of the output parasitic capacitance is:

$$C_{OUT} = n \cdot C_{jn}(W_n) + C_{jp}(n\beta W_n) \tag{7.51}$$

If we observe the structure of the gate, it is the dual version of the Nand gate, for this reason, it is simple to get the value of the leakage current. The method is to replace the index n with the p and vice versa.
The off current is:

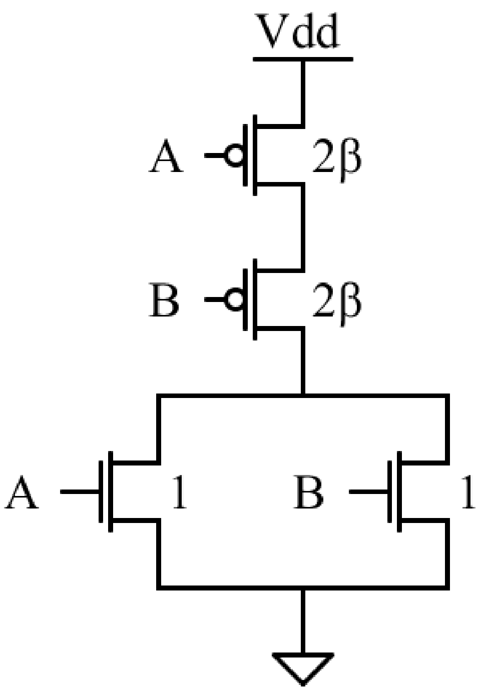


Figure 7.9: Nor 2 inputs gate

$$I_{off} = \frac{1}{2^n} \cdot \left[n \cdot I_{off,n} + I_{off,p} \sum_{k=1, n_z \neq 0}^{2^n-1} \frac{1}{n_z^{\alpha}} \right] \tag{7.52}$$

and the gate current is:

$$I_{gate} = \frac{1}{2^n} \cdot \left[\frac{2^n}{2} \cdot n \cdot I_{gate,n} + (2^n - 1) \cdot I_{gate,p} \right] \tag{7.53}$$

To conclude the area is:

$$A = L_{MOS} \cdot n \cdot W_n \cdot (1 + n \cdot \beta) \tag{7.54}$$

7.8.4 Xor/Xnor Core

The Xor and Xnor gates in DExIMA are also base gates. They are composed of two parts. The first part is the core of the gate and it is responsible for the behavior of the gate. This part is called *Core* and it is the same for the two gates. The circuit of the Xor and Xnor core is shown in Figure 7.10.

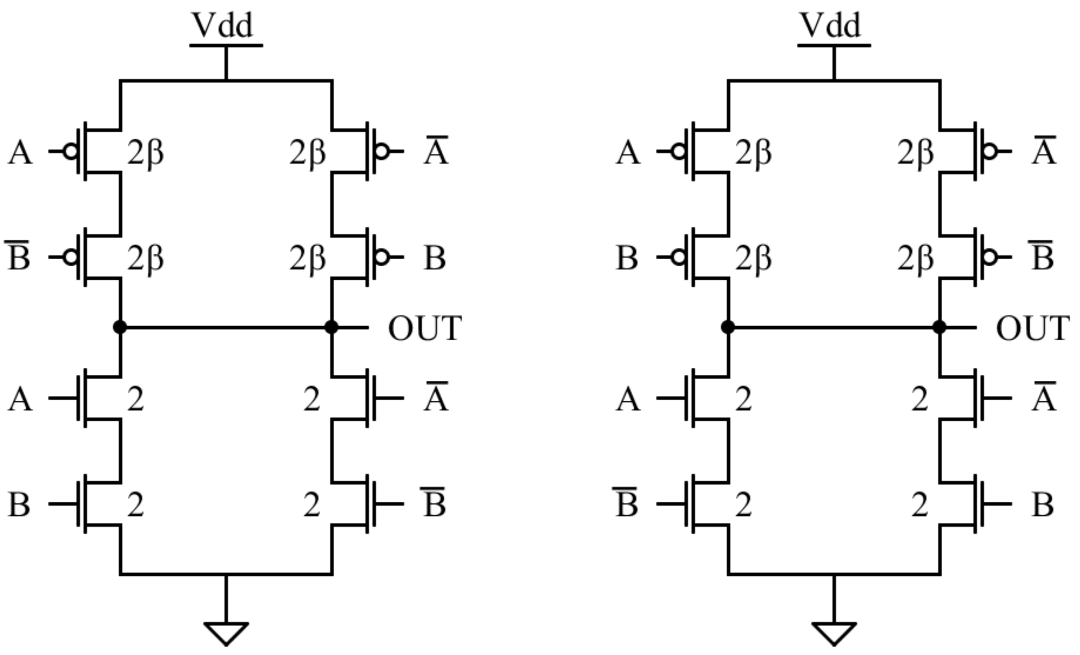


Figure 7.10: Xor/Xnor Core gate (left/right)

The transistor structure is the same, this implies that it can be used for the two gates. These are some of the simplest static structures. More typologies are reported in [16].
Now we proceed like the previous elementary gates, the fanin for these gates is:

$$fanin = 2 + 2 \cdot \gamma \tag{7.55}$$

The value of output parasitic capacitance is:

$$C_{OUT} = 2 \cdot C_{jn}(2W_n) + 2 \cdot C_{jp}(2\beta W_n) \tag{7.56}$$

The leakage current is computed like a gate with only two inputs because it is always used with the other two inputs with the inverted value, so we have only four possible combinations. The result value is:

$$I_{leak} = \frac{1}{4} \cdot \left[4 \cdot \left(I_{off,p} + I_{off,n}\right) + 6 \cdot \left(I_{gate,p} + I_{gate,n}\right)\right] \tag{7.57}$$

The resulting area is:

$$A = L_{MOS} \cdot 4 \cdot W_n \cdot (2 + 2 \cdot \beta) \tag{7.58}$$

7.8.5 Three state inverter

The model that we describe has three inputs, where we can connect the logic input and the two clock logic values. Notice that the inverter for the clock inversion is not included in the model. This choice is preferred to construct complex components starting from this with the highest degree of freedom. The circuit is shown in Figure 7.11 and shows a three-state inverter in C2MOS technology.

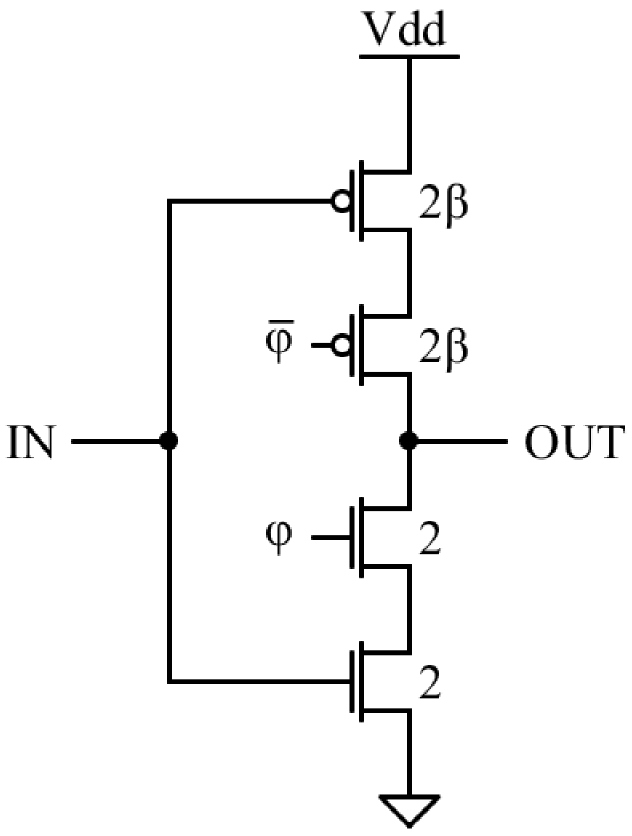


Figure 7.11: Three state inverter C2MOS

In this case we distinguish three different fanins of the inputs:

$$fanin_{IN} = 2 + 2 \cdot \gamma \tag{7.59}$$

$$fanin_{\varphi} = 2 \tag{7.60}$$

$$fanin_{\bar{\varphi}} = 2 \cdot \gamma \tag{7.61}$$

The output parasitic capacitance is:

$$C_{OUT} = C_{jn}(2W_n) + C_{jp}(2\beta W_n) \tag{7.62}$$

Also in this case the φ input is considered the same input with two different logic values. So the number of combinations is four, the result is:

$$I_{leak} = \frac{1}{4} \cdot \left[3 \cdot I_{gate,p} + 3 \cdot I_{gate,n} + \left(1 + \frac{1}{2^{\alpha}}\right) \cdot \left(I_{off,p} + I_{off,n}\right)\right] \tag{7.63}$$

And with area:

$$A = L_{MOS} \cdot 2 \cdot W_n \cdot (2 + 2 \cdot \beta) \tag{7.64}$$

The circuit can be used also as a Latch as shown in [18].

7.9 Composite Gate Models

In this section, we see all the models that are derived from the elementary gates. We don't rewrite all the single performance parameters for each gate but only the circuit and which components were used to create it. In some cases we specify only the path of interest inside the circuit, for all the other parameters the formulas used are straight-forward:

$$P_{static} = \sum_i P_{static,i} \tag{7.65}$$

$$Area = \sum_i A_i \tag{7.66}$$

$$E_{dynamic} = \sum_i E_{dynamic,i} \tag{7.67}$$

Where i represents the $i-th$ sub-components of the model. It is important to track the load of each sub-component inside the model, in order to correctly compute the dynamic energy and the delay.

7.9.1 And/Or Gate

The And/Or gate is obtained by simply connecting in chain a Nand/Nor gate and an Inverter like shown in Figure 7.12

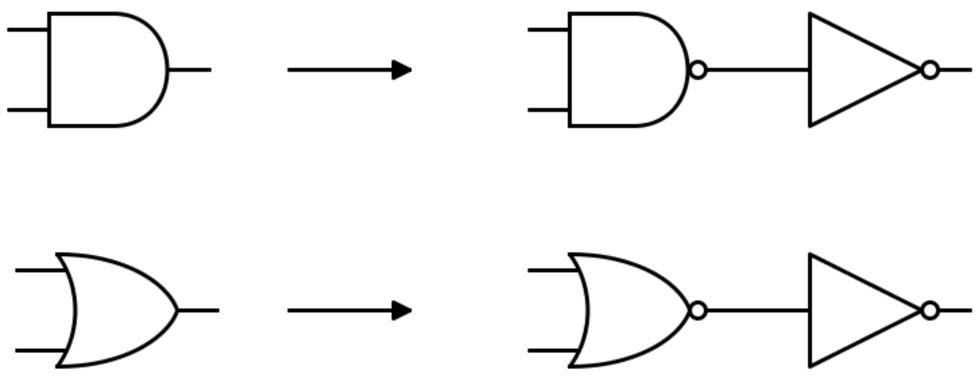


Figure 7.12: And/Or Gate composition

In multiple-input gates, we replace only the first input gate. For this model, the delay is the sum of the two gates, and the contamination delay is equivalent to the propagation delay.

7.9.2 Xor/Xnor Gate

To create our Xor/Xnor gate, we need three elements: the Xor/Xnor core and two inverters. The symbol of the Xor/Xnor core is represented as a box with four inputs, shown in Figure 7.13.

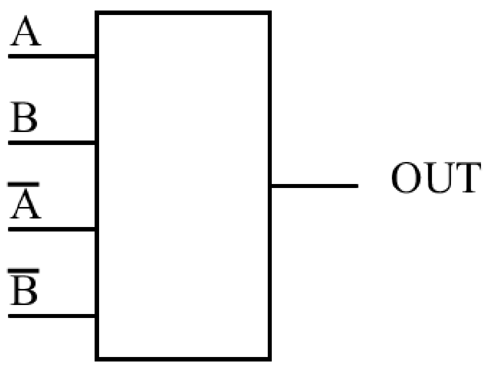


Figure 7.13: Xor/Xnor Core Symbol

Make a composition of this element and the Inverters we get the circuit in Figure 7.14.

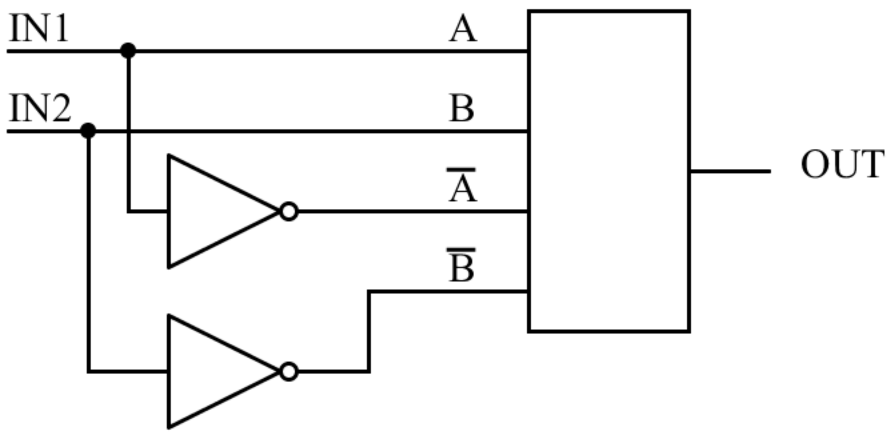


Figure 7.14: Xor/Xnor Gate composition

This is the Xor/Xnor with two inputs, which is reused to build a multiple-input gate. In this case, we have two different paths. The propagation delay is the path considering the Inverter and the Xor/Xnor core. The contamination path considers only the core component.
to create multiple inputs gates, we use the tree/chain configurations, since they are more optimized for that specific number of inputs. The practical example is shown in Figure 7.15. The number of stages in a multi input gate (worst case) is computed by the formula:

$$stages = log_2(inputs) \tag{7.68}$$

The critical path is obtained considering the upper nearest integer of stages (because in general is not an integer number), the contamination delays the lower nearest integer.

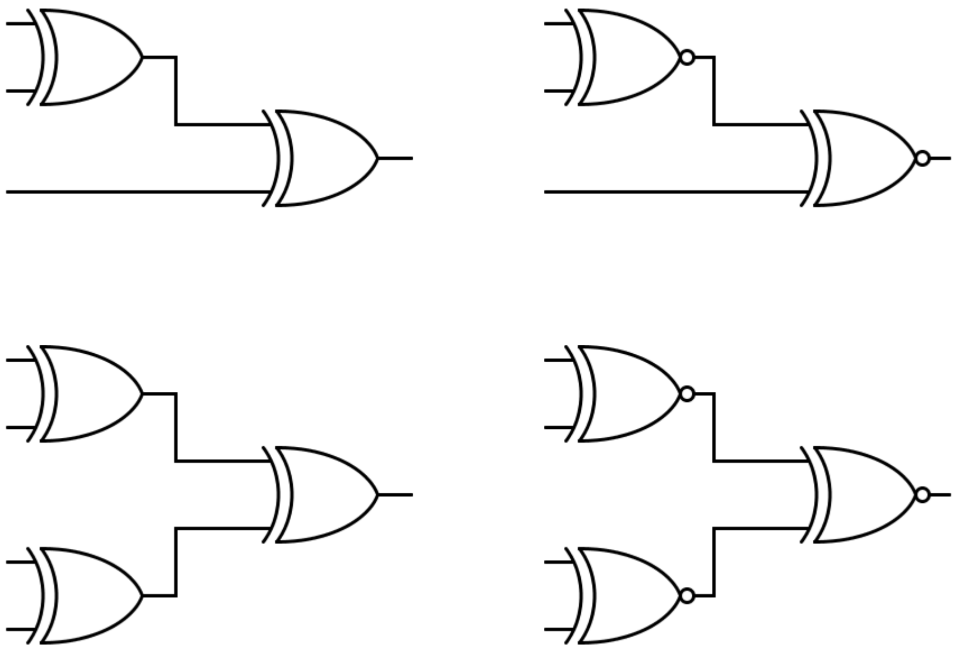


Figure 7.15: Xor/Xnor Multi Inputs

7.9.3 Half Adder

The half adder is the standard circuit shown in Figure 7.16.

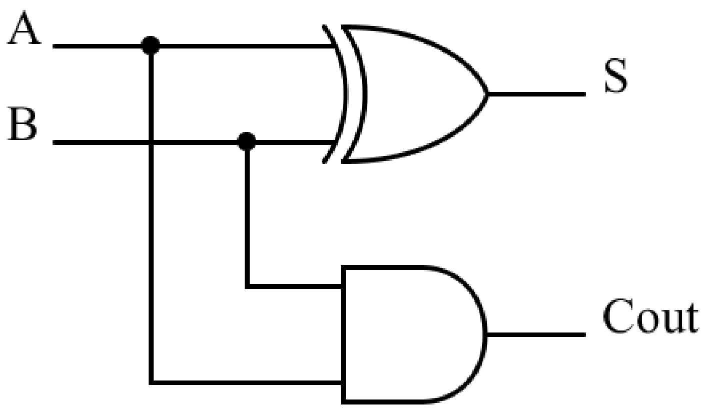


Figure 7.16: Half Adder Circuit

The critical path is the maximum between the two gates (because can have very different load) and the contamination delay the minimum.

7.9.4 Full Adder

The circuit is shown in Figure 7.17. The Full-Adder circuits have four main paths of interest:

- From A/B to S passing through two Xor gates
- From A/B to $Cout$ passing through one Xor and two Nand gates
- From Cin to S passing through one Xor
- From Cin to $Cout$ passing through two Nand

The computation of the propagation and contamination delay is obtained using the maximum and the minimum value between these paths. Notice that it is not possible to determine the maximum and the minimum value a priori because we need to know the load applied to each output, which changes the total delays. In [17] it can be checked how to explore more efficient circuits of Full-Adders using Xor gates.

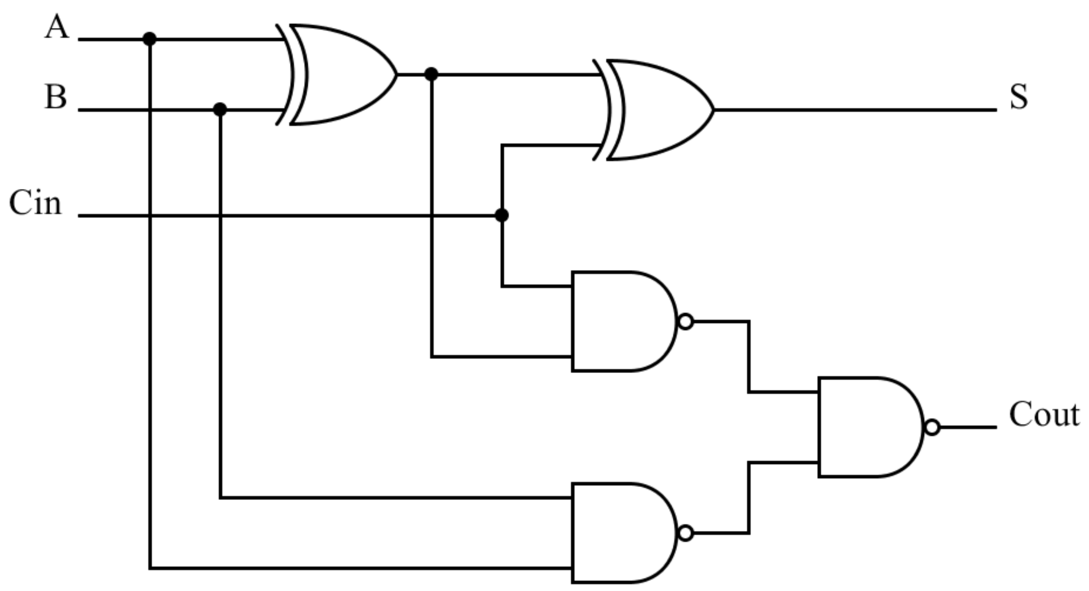


Figure 7.17: Full Adder Circuit

7.9.5 Multiplexer

Multiplexers are built hierarchically, starting from the 2-way component. The circuit of the 2-way multiplexer is shown in Figure 7.18.

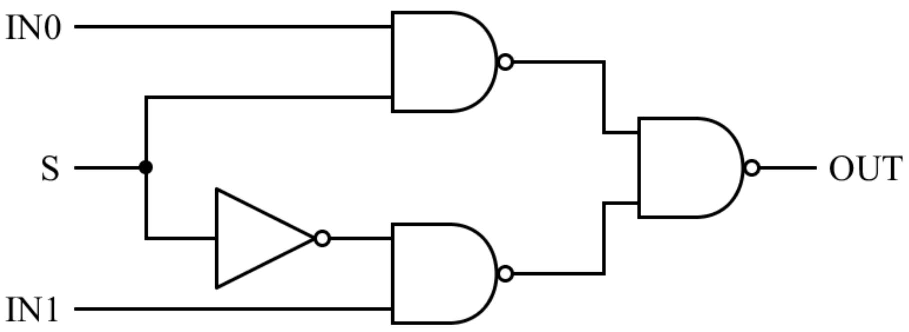


Figure 7.18: Multiplexer two ways circuit

The critical path of the circuit is represented by the selector *S*, because it has an additional Inverter than the other paths. The contamination delay is due to the two Nand gates. To create a more inputs multiplexer, it is

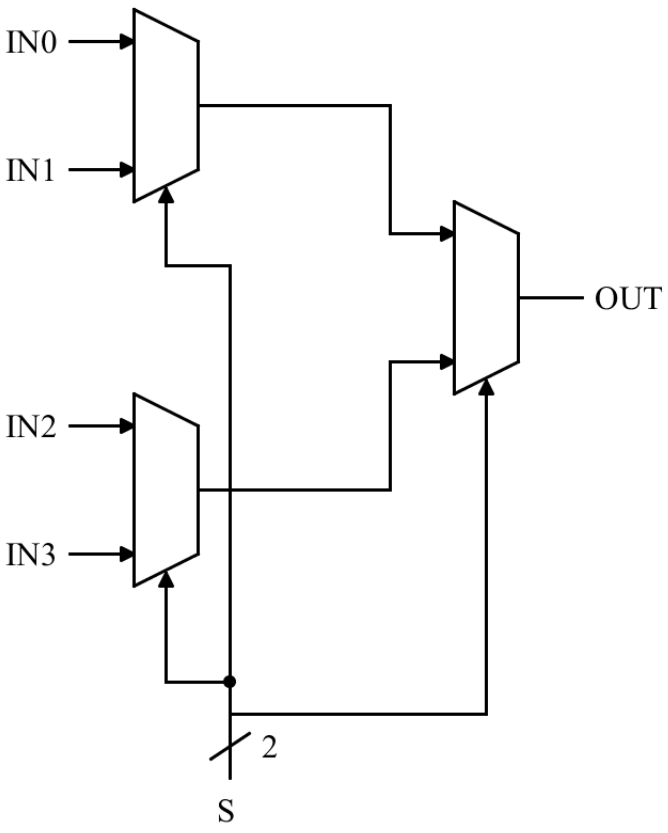


Figure 7.19: Multiplexer composed by two ways multiplexers

more convenient to use more than 2-way multiplexers because using only the standard logical structure, we need gates with high fanin values that are slower than multi-stage configurations. The configuration is shown in Figure 7.19. The computation of the propagation delay and contamination delay is similar to the case of Xor/Xnor with multiple inputs. We use the same equation for the number of stages in the equation 7.68.

7.9.6 Decoder

The decoder, compared to the Multiplexer, is more difficult to be synthesized automatically, the version created is very simple but inefficient. It needs improvements in the future using multiple stages with predecoders. The circuit is the classic like it is shown in Figure 7.20.

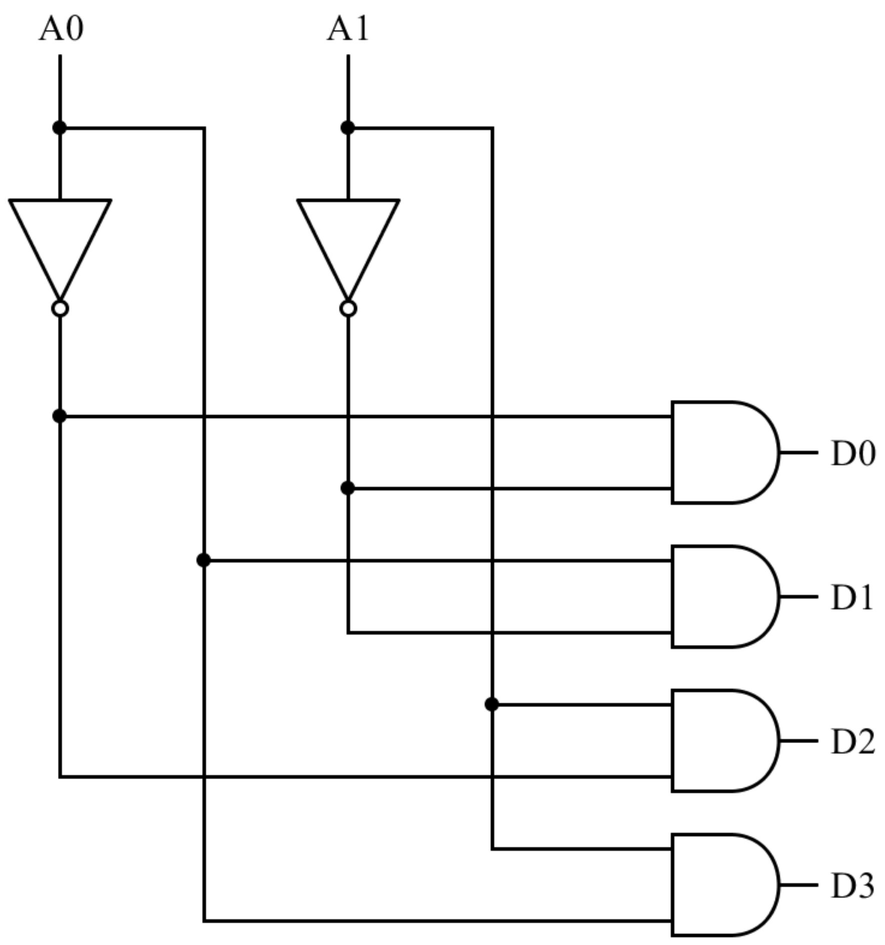


Figure 7.20: Decoder 2 to 4 circuit

The problem with this configuration is that the And gate has a number of inputs equal to the decoder inputs. This implies that the area and fanin grow very fast with the inputs. The propagation delay is due to the sum of one Inverter and a And gate chain. The contamination delay considers only to the And gate.

7.9.7 Driver

The Driver is simply a chain of inverter that grows the drive strength to each stage. The reference circuit is shown in Figure 7.21.

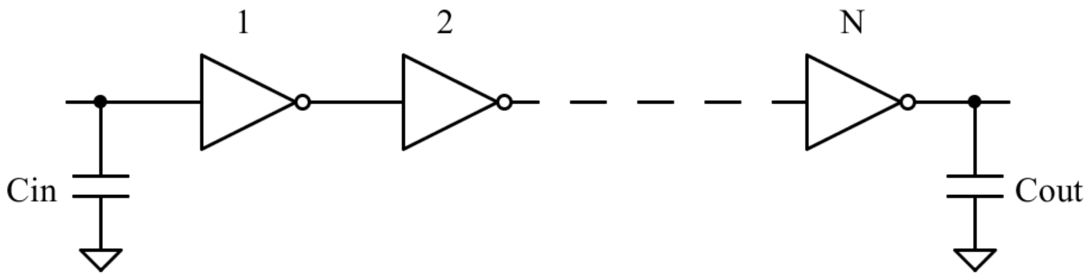


Figure 7.21: Inverter chain Driver

To choose the optimum number of stages we use the method of logical effort using the equations:

$$F = \frac{C_{out}}{C_{in}} \tag{7.69}$$

with effort f:

$$f = \sqrt[N]{F} \tag{7.70}$$

this implies that the number of stages are computed by:

$$N = \log_f F \tag{7.71}$$

The dimension of the transistor of the next stage is f times greater than the previous. As explained in the chapter on the language, the Driver can have to input the parameter f and the mode of operation. The mode of operation modifies only the value N, in fact, it is a fractional value, the rounding is chosen in function of the mode. The main difference is that for the auto and inverter mode we introduce at least one stage (inverter of dimension 1) and for the buffer mode two stages (inverter of dimension 1 + inverter f times greater).

7.9.8 Latch SR

Now we start with the sequential components: the simplest is the Latch SR, in particular with enable. But firstly we analyze the simple latch cell with feedback Nand gates. The timing of the cell is more complex rather than the previous gates due to the presence of the feedback. The delay of the latch cell is described in Figure 7.22.

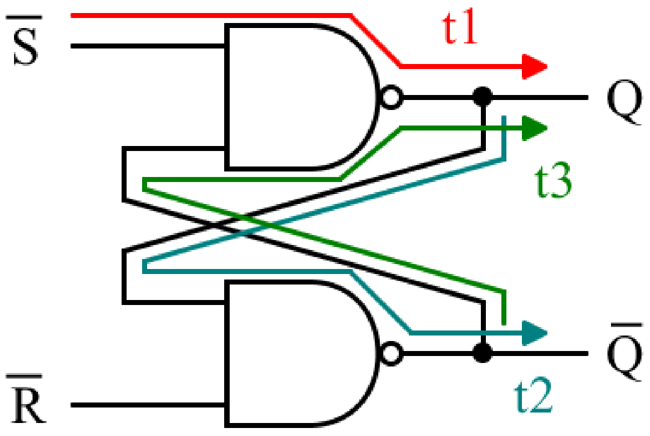


Figure 7.22: Latch cell delay

The delay obtained is:

$$t_{latch} = t_1 + t_2 + t_3 \tag{7.72}$$

The complete circuit with the enable part is shown in Figure 7.23.
For the propagation delay, of the complete cell, we add the delay of the latch cell with the delay of one Nand of the enabling part.

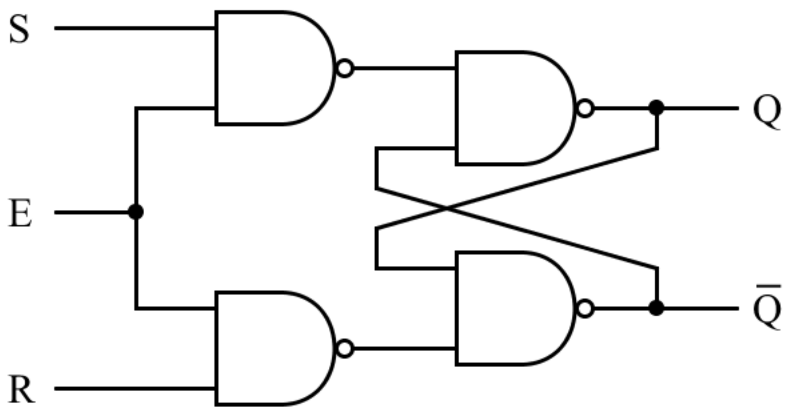


Figure 7.23: Latch SR circuit

7.9.9 Flip Flop Nand

The Flip Flop is more complex than the other components because we need to compute additional timing parameters. In DExIMA two different types of Flip Flops are present: we start with the simplest one made by Nand gates. The structure is a Master-Slave type, but to make what we are doing clearer, we put the whole circuit below in the Figure 7.24.

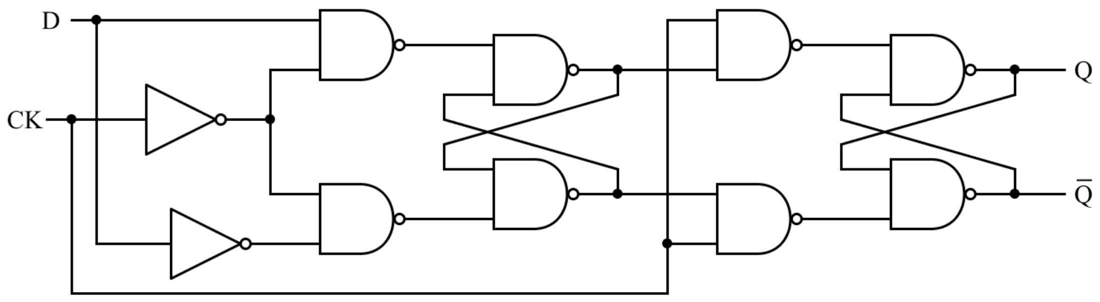


Figure 7.24: Flip Flop D with Nand gate

Basically in DExIMA, we don't use the single Nand gate, but we use two Latch SR and two Inverters. The structure of the Latch presented before is very easy to recognize. This is not the best way to implement a Flip Flop but it is useful to understand the structure and compare it with other structures. Now we analyze the timing part of the circuit. We need to extract the three timing parameters of a Flip Flop, we recall quickly:

- **Clock to output**: It is the delay between the transition of the clock in input and the variation of the logic value in the output.

- **Setup**: It is defined as the minimum amount of time before the transition of the clock in which the data in the input must be stable to be sampled correctly.

- **Hold**: It is defined as the minimum amount of time after the transition of the clock in which the data in the input must be stable.

Now, having the definitions, we add to the previous figure the path of these "delays". The Figure 7.25 shows the highlighted paths.

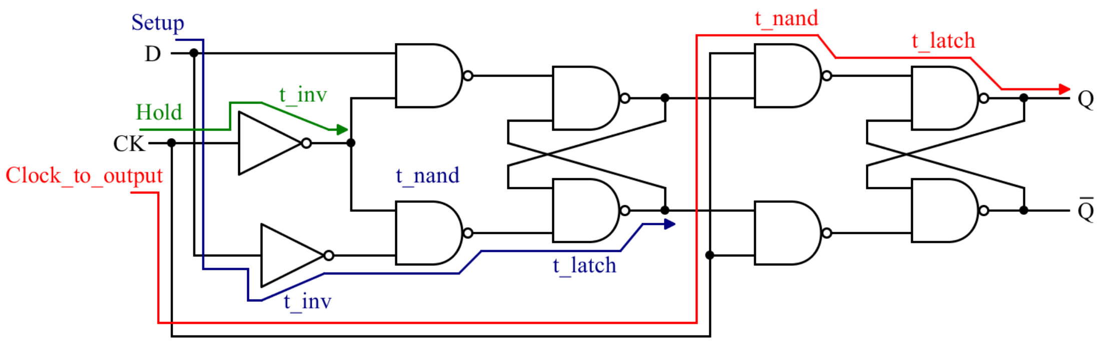


Figure 7.25: Flip Flop D Nand Delays

The red path is the path concerning the clock to output delay, in fact, when we have a transition of the clock, it enables the Slave Latch. In this case, the Slave is enabled and the Master is off. So the responsibility of the clock to output delay is only the slave stage. For the setup time, we need to have a stable input all the time before the enable of the Slave stage. This implies that the time needed is at least the propagation delay of the Master stage. So the component that needs to be considered is the same for the clock to output delay. But we need to pay attention to the fact that, the Setup time is constant because the Master stage is always loaded with the Slave stage that does not change. The clock to output time can change because it is a function of what we connect at the output of our Flip Flop. The hold time is computed as the delay of the inverter connected to the clock because when the clock toggles the Master stage, it turns off only after the inverter propagation. After this time the Master stage does not propagate the input so the output is unaffected. The hold path is shown in green in Figure 7.25, also in this case the hold is constant and does not depend on the output load.

7.9.10 Flip Flop C2MOS

This is the other version of Flip Flop D implemented in DExIMA. With this configuration, we can use fewer transistors. This is reflected in the performances because we get: lower area, lower delays, and lower power consumption in terms of static power and dynamic energy. The base component used is the three-state inverted explained before, we use again a Mater-Slave configuration. The complete circuit is shown in Figure 7.26.

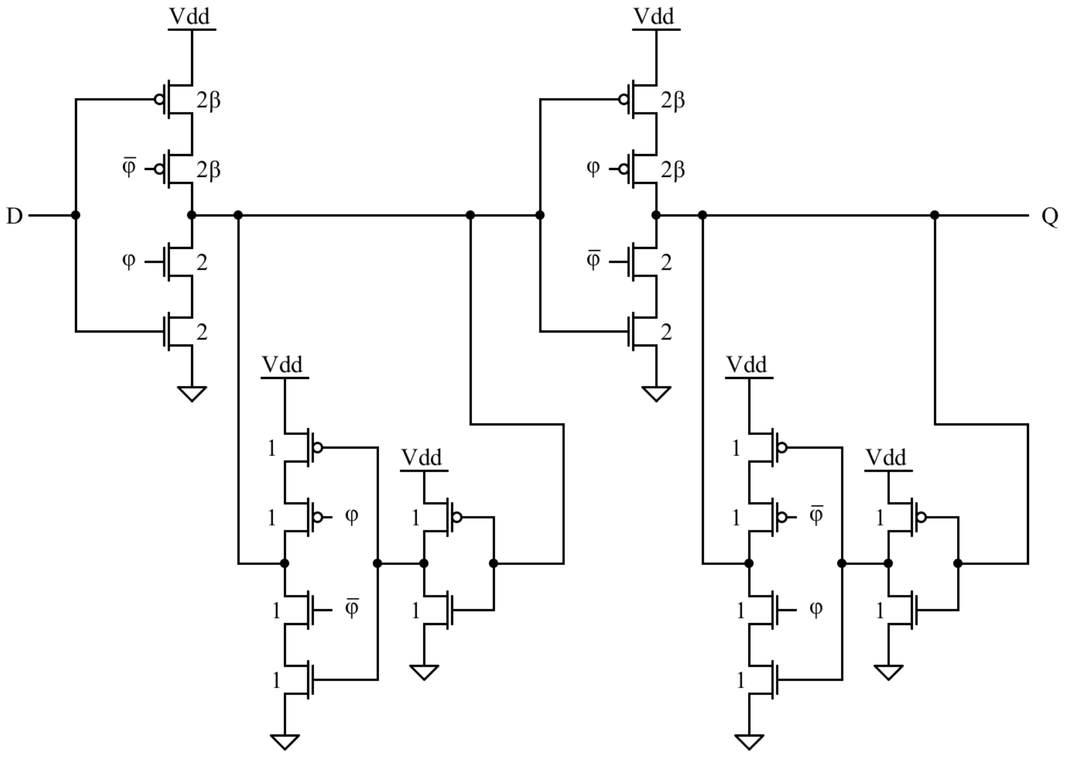


Figure 7.26: C2MOS Flip Flop D

There are a lot of configurations to create a Flip Flop using C2MOS gates like in [19], but most of all use transmission gates that are not implemented in DExIMA, so we use a more simple structure like in [20]. The circuit is composed in the directed chain by two, three state inverters, the clock of the two stages has opposite logic values to have the Master-Slave behavior. Theoretically, only the direct chain is needed to have the correct behavior. The problem is that we store logic values in the output capacitance of the three state stages, so if we wait some time, we lost the charge due to all the parasitic currents and other effects that discharge the capacitors, causing the loss of the data. To avoid this, we add a feedback chain that has only the scope to maintain the data memorized. Notice also that the dimension of the transistors of the feedback chain is the minimum possible, because we do not have to constrain about this part that is responsible only for the refresh of the logic value stored in the internal capacitance. In Figure 7.26 only one component is omitted, the inverter that is used to have the negated value of the clock. Similar consideration done for the Nand Flip Flop can be used here for the timing parameters. Also in this case the responsibility of the hold time is the clock inverter. The time paths of the three main times values are shown in Figure 7.27.

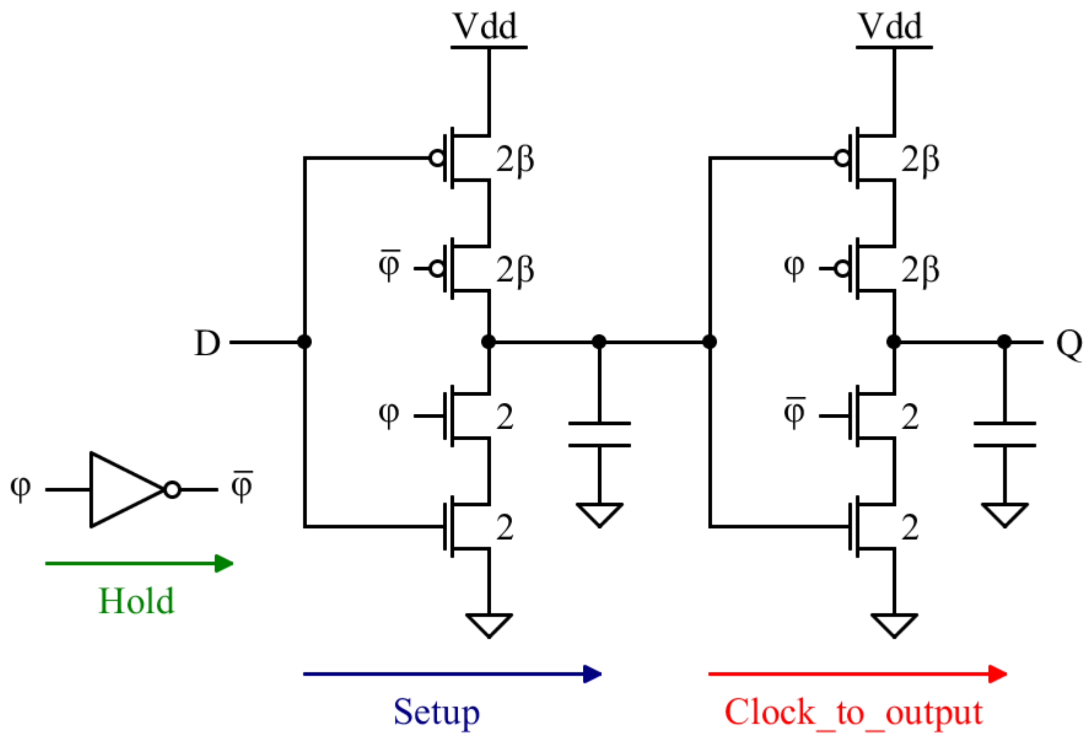


Figure 7.27: C2MOS Flip Flop D delays

We omitted the feedback part for better visualization, but it is included in the load capacitance of the stages. Like the previous case, the Slave stage is responsible for the clock to output delay, and the Master stage is responsible for the Setup time.

7.9.11 Ripple Carry Adder

The Ripple Carry Adder, as we know, is an adder created using a combination of Full-Adders that we have analyzed previously. In particular, the circuit implemented has also the part related to the Xor gates to create the inversion to do also the subtraction operation. The circuit is shown in Figure 7.28.

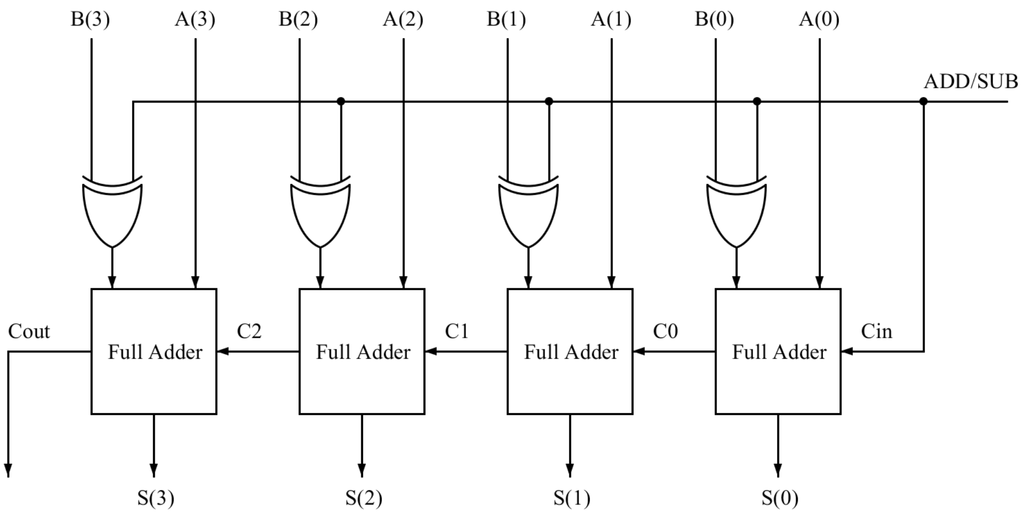


Figure 7.28: Ripple Carry Adder

The area and power parameters are easy to compute, they are the sum of these figures of merit done with all the composite models in the Figure 7.28. One important aspect is the timing. As we know, this type of adders are not so efficient in terms of timing because the limiting part is the chain of the carry that is usually the critical path. In Figure 7.29 two paths are highlighted, the carry path, which usually sets the critical path, and the direct path, which represents the worst path between input and output in the direct chain. To compute the critical path, DExIMA computes firstly the Carry Path, and after computes each direct path and sets to the critical path the maximum of it. This because we can find a situation where one or more wires in output are connected with a very large capacity load, this implies that charging the output can be slower than the path of the Carry. In conclusion, the Contamination delay is set equal to the fastest Full adder of the chain.

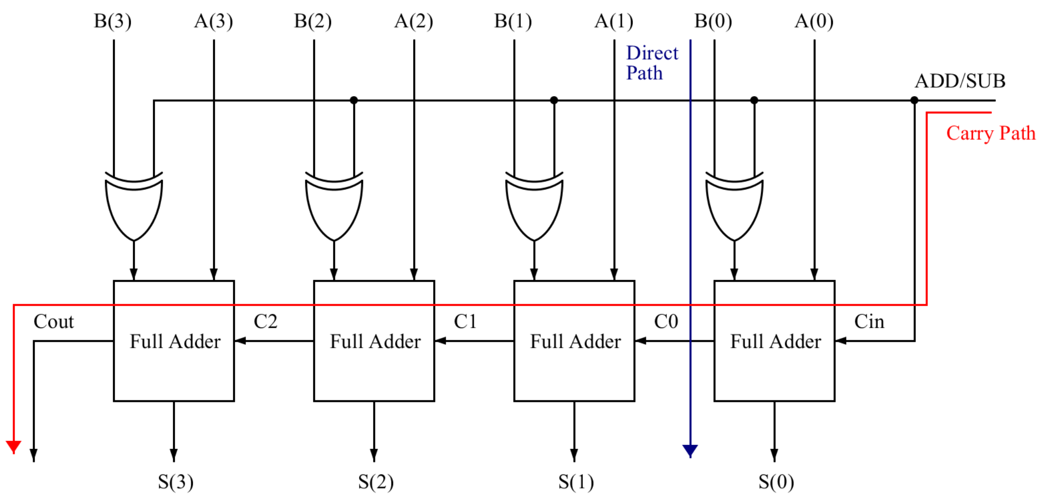


Figure 7.29: Ripple Carry Adder Delays

7.10 Memory Models

The memory models are the part related to the LiM. We will see in this section the cell used and the Architecture of the memory.

7.10.1 Memory Cell

Starting from now, it is possible to choose only one type of memory cell, the FLIPFLOP cell. It is composed of two components: a Flip Flop and a Multiplexer. The Flip Flop used is the C2MOS Flip Flop, because since it occupies less area and consumes less static power, this is more suitable for a huge memory where these two parameters are very important. The Flip Flop is used to store the information. The Multiplexer has another function, it is used to write inside the cell. Furthermore, we can distinguish between memory writing and output memory writing. The Flip Flop is provided also to enable the memory cell. To be usable by the external interface, the memory needs two inputs, one connected to the external and one connected using a component inside the memory. From In memory point of view we should use only two ports: the input called WR (Write) and the output called RD (Read), the memory cell circuit is shown in Figure 7.30. The custom LiM cell is composed of this elementary cell, plus all the logic we insert inside the cells section. How explained in the chapter about the language, when we call the keyword Memory, we reference this structure, if we insert the logic we

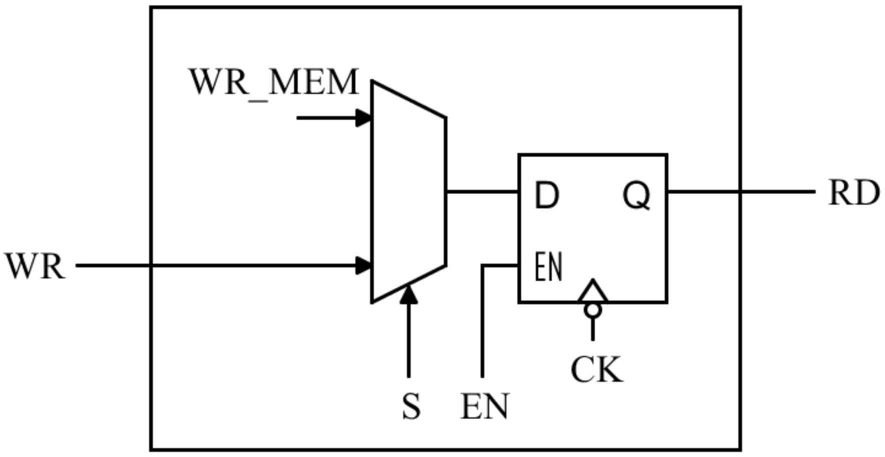


Figure 7.30: Memory cell

have more components inside the cell. The delay of the cell is related to the Read/Write operation. In the Read operation, in theory, we do not have to wait, since the data are available in the output. But this is not possible if, at the previous step, we do a Write operation and the data change. So the delay related to the Read operation inserted is simply the Clock to output delay of the Flip Flop. Instead, the Write delay is computed considering the delay of the Multiplexer in input plus the Setup time of the Flip Flop to have a stable input signal to sample. It is possible to access the other inputs of the cell, but it is not recommended because they are used in the composition of the architecture of the memory. Anyway, if used, it ensures a correct performance computation.

7.10.2 Flip Flop Memory Architecture

The memory architecture is made by the Flip Flop cell that emulates a standard memory. To explain all the components, we create a figure with a small memory that shows all of its components. The architecture is shown in Figure 7.33. The example memory has 4 rows and 4 columns (4x4) and with a bit address of 3 bits. The Input/Output parallelism is 2 bits, and the word length is 2 bits also. The structure has two decoders: one is the row decoder and the other one is the column decoder. The use of the two decoders has the advantage to have a smaller and faster decoder instead of using a big and slow one. Using two decoders needs an And stage, one input is used to choose the line and the other to choose the word. Another important component

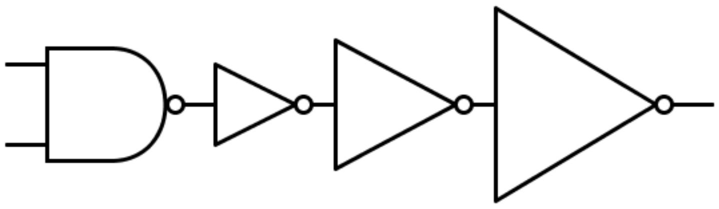


Figure 7.31: And Driver

is the column multiplexer used to choose in the output the wanted word. Considering only these components, the memory can be very slow because it needs to drive high capacities loads. In the upper part of the memory, we can see the inputs IN0, IN1 that are called in the real model WR port. In fact, it is the port used to write inside the memory. In the input, the buffer driver stages are placed to drive all the capacity of the connected multiplexers. Other buffers are present at the output of the decoders. In the figure it is not displayed but the And gates, in reality, are not simple And gates. The first stage is a Nand gate followed by an Inverter Driver, the circuit is shown in Figure 7.31. The cell clocks are all connected to the external pin CK. The And gates are used to enable the memory cells of the word to do a Read/Write operation of the wanted word. Another external pin is OUT/IN, and it is used to choose the way of internal multiplexers, used to choose if do an operation inside the memory or outside the memory. This input doesn't

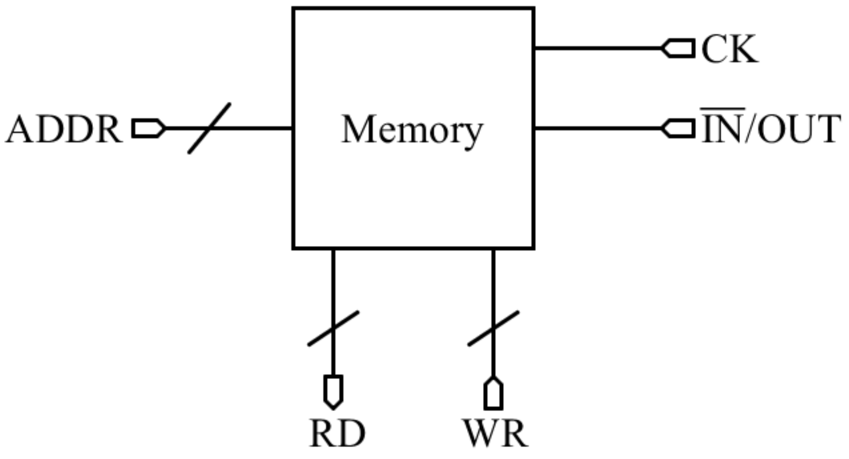


Figure 7.32: Memory Interface

have a driver because it can be used better with external drivers. The CK is usually connected to the external clock driver, the OUT/IN input can be used too much or not used completely. It is better to use an external driver to compute the energy dissipation depending on the algorithm. From the external point of view, the memory is represented by the interface shown in Figure 7.32.

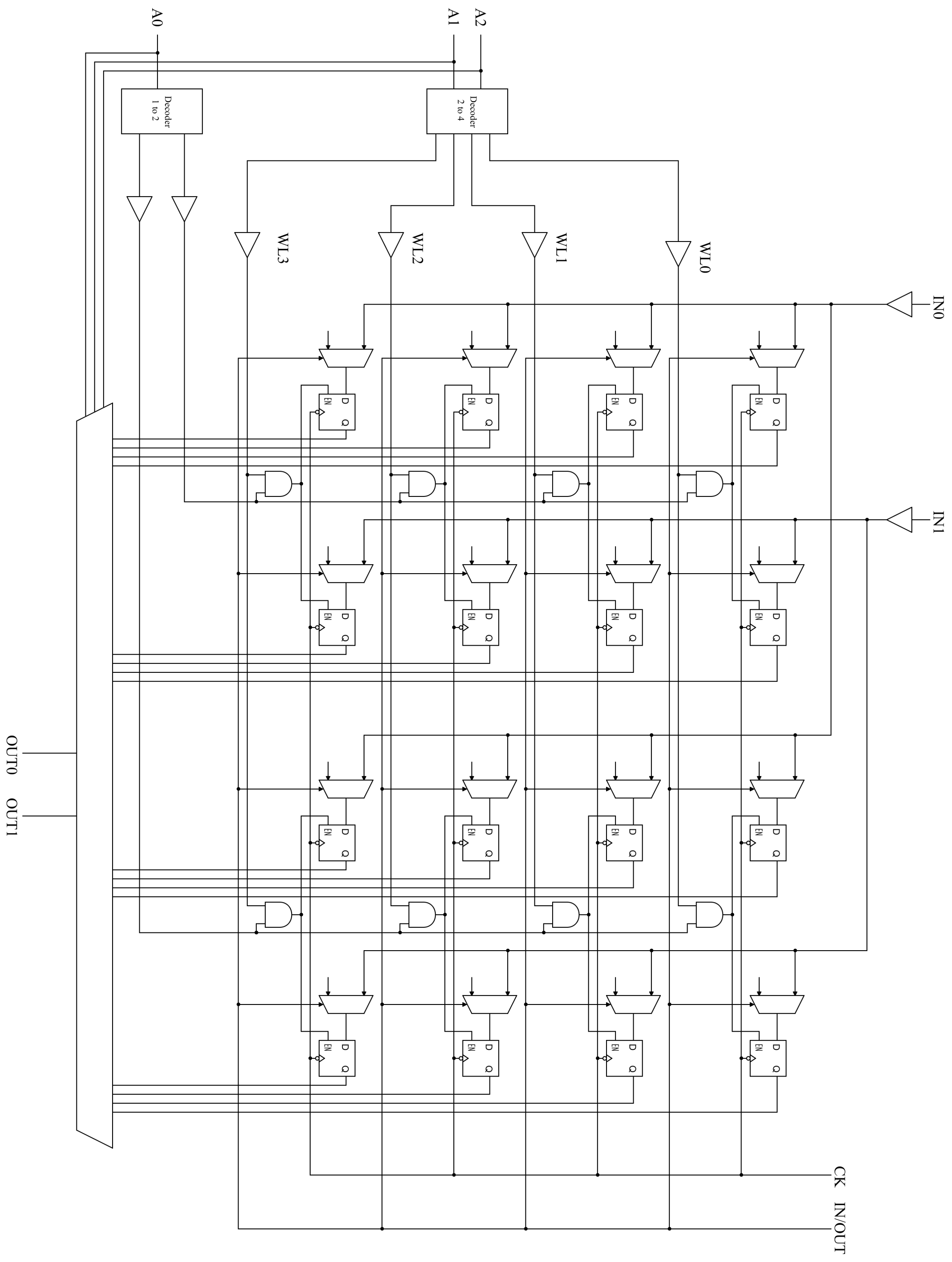


Figure 7.33: Memory Architecture

Chapter 8

DExIMA Data Structure

In this chapter, we describe each data structure organization and the hierarchy of DExIMA. It is important to know how the program is organized in order to understand how DExIMA stores and simulates our circuits. We will start with the main class of the program and go down with the hierarchy in a top-down approach. Before starting, we want to explain how the data structure is displayed. The main containers used are the vector and map object of C++. The vector and map containers are displayed simply like in Figure 8.1. The figures do not specify the exact object like the notation vector<string>, but only the category of object. For example, the keys of a map that contains the name of the object are shown simply by the word "Name", and not by the data structure "string", because it is more useful to display the behavior of the object differs from the exact type. The exact type of data is described in the specific section. The single objects are displayed like a rectangle with the name of the object inside.

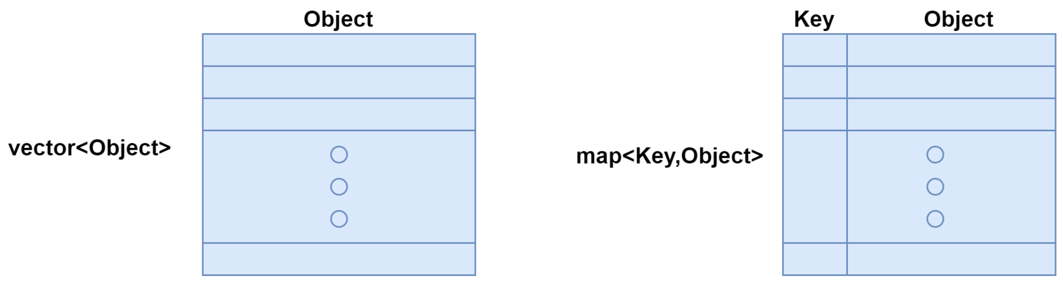


Figure 8.1: Containers Visualization

8.1 Architecture Class

The Architecture class as previously pointed out is the core of the whole program. It is a container of almost all the objects needed and has a lot of methods used to manage the objects and process some parameters.

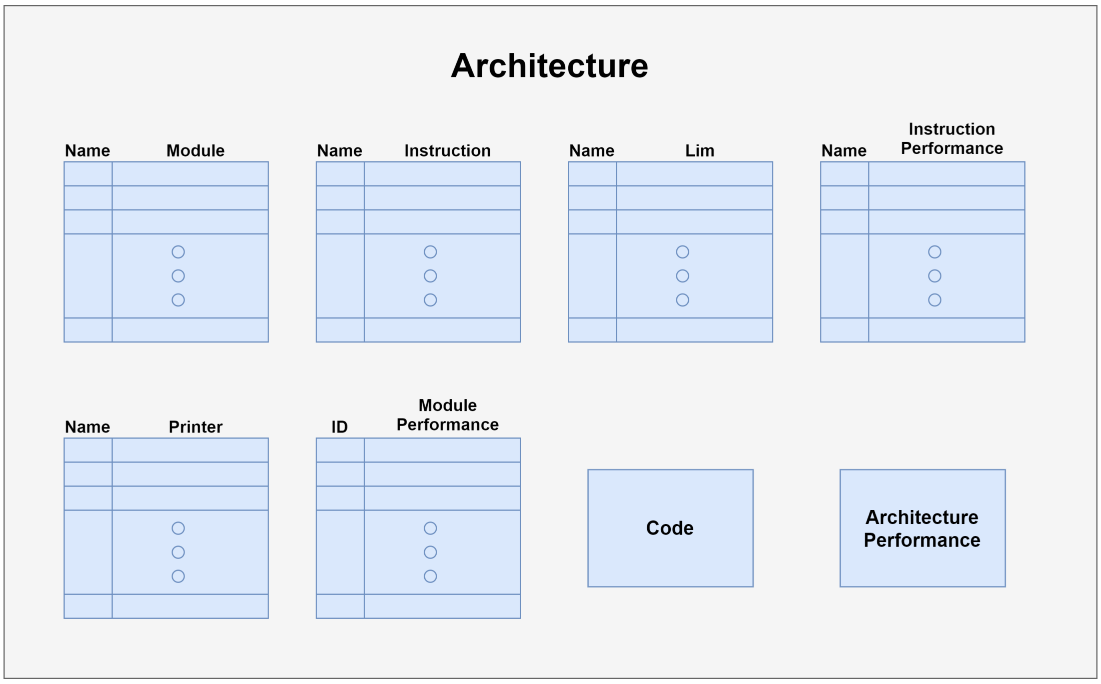


Figure 8.2: Architecture Class

Most data structures are map containers using a string as Key and contain different types of objects. The class contains these object maps:

- **Module**: It contains all the information about a component of the circuit.
- **Lim**: It contains the information about the specific memory
- **Instruction**: It contains the information about the instruction of the instruction set.
- **Code**: It contains the algorithm code of the circuit
- **Printer**: Object used to handle and compute the performance of the specific model.

- **Module Performance**: It contains the performance parameters of a module object.
- **Instruction Performance**: It contains the performance parameters related to instruction.
- **Architecture Performance**: It contains the performance of the whole architecture.

8.2 Module Class

DExIMA can be defined as a *Module* based program because each component of the architecture is defined by a Module object. The Module Class is represented in Figure 8.3, in the figure, there are all the components, the only omitted part is a string called *Code* that has special behavior that will be later described.

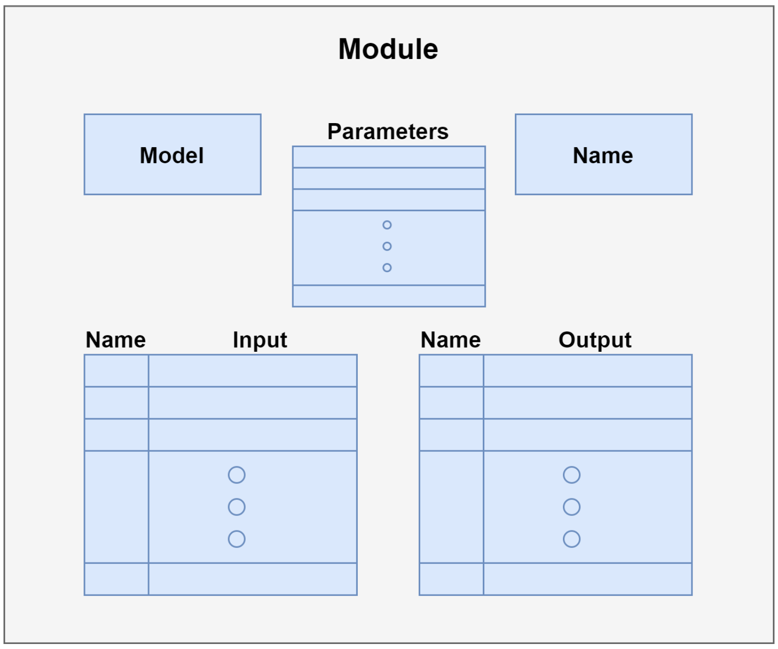


Figure 8.3: Module Class

The class contains two strings that are the Name of the Instance and the Model of the instance (And, Or, Xor, etc..). There is also a vector of strings, where the parameters of the gate are stored inside. It is important to remember that each model can have a different number of parameters. For example, the Nand gate has only one, the number of inputs, while the Inverter has none. These three fields are filled in the Init section where we specify: model, name, and list of parameters. The Module class has also two maps containing the Input and Output objects. These two objects are used to connect the Module object with other objects of the same type. The Key of the maps is the string containing the name of the Input/Output port, for example, IN0 or OUT.

8.3 Input/Output Class

The Input and Output are classes used to connect the components between them. It inherits classes from class *Port*. The port class has two variables inside it, the Name and the Parallelism. The first one saves the string Name of the port and the latter contains an unsigned int value that stores the value of the port parallelism.

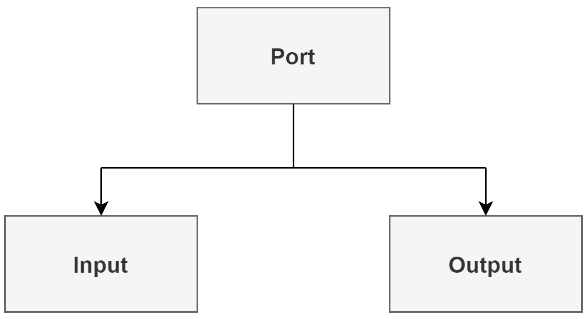


Figure 8.4: Port inherit scheme

The Input/Output ports have additional parameters depending of their nature. The two classes schemes are shown in Figure 8.5. Starting from the Output class, it has only a vector of float that contains the fanout of each wire of the port parallelism. These parameters are used to compute the load capacitance connected to the wire. On the contrary, the input object has two vectors, one containing the fanin of each port wire, and one contains a bool value that represents the state of the wire. The state of a wire can be connected or not connected. By default, all the wires are not connected and when we connect it in the Map section, we change the value of the specific wire of the port.

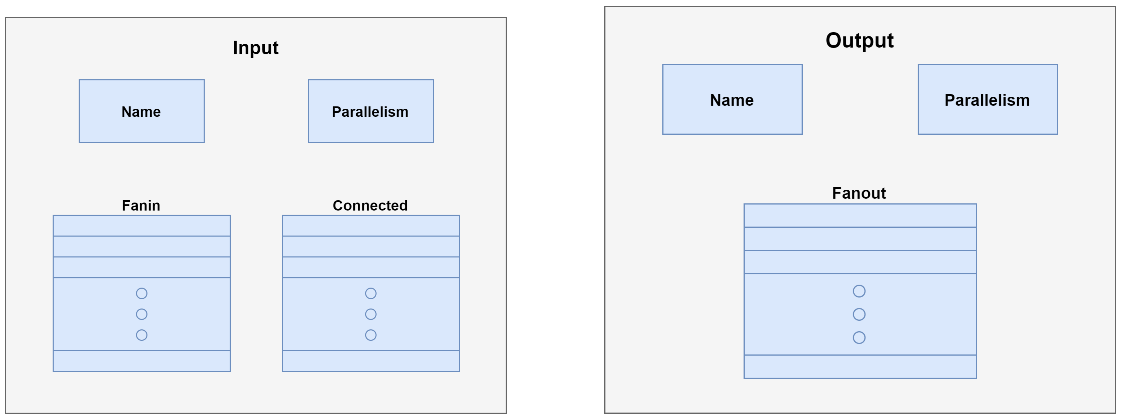


Figure 8.5: Input/Output Classes

8.4 Lim Class

The Lim class is used to store and manage all the information about the memory. The object saves the strings of the Lim Name and Type (ex. Sram, FLIPFLOP). The other parameters are used to specify the memory dimensions. The Address/Read/Write parallelism and the number of rows and columns are stored in an unsigned int variable.

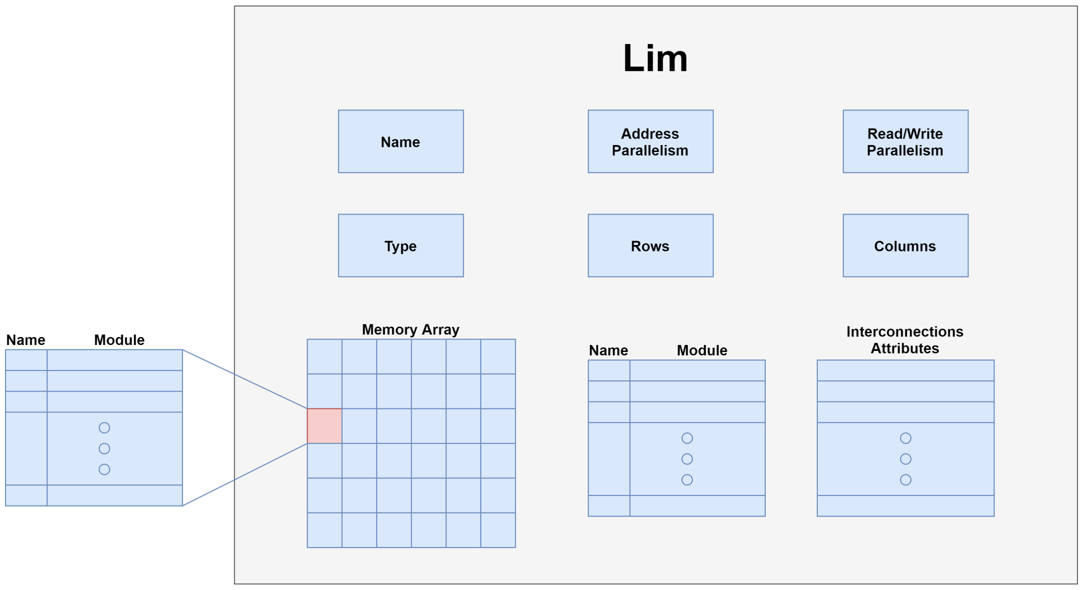


Figure 8.6: Lim Class

There is also a map of Modules used to store the out-of-cell component. The memory is represented in a 2D map with the same rows and columns. Each cell has inside another map used to store all the Modules pushed inside the memory cell (In a different point of view is a 3D array). By default, we have only one Module related to the memory cell depending on the type. There is also a vector containing the object called *InterconnectionAttribute* that stores the interconnection parameters of the memory array.

8.5 Instruction Class

The Instruction Class is used to store the information about the instructions of the instruction set. Similar to the previous classes, we find the Name and the Type that can be standard or LiM type (INSTRUCTION, LIM_INSTRUCTION). If it is a LiM instruction, we find also the name of the associated LiM.

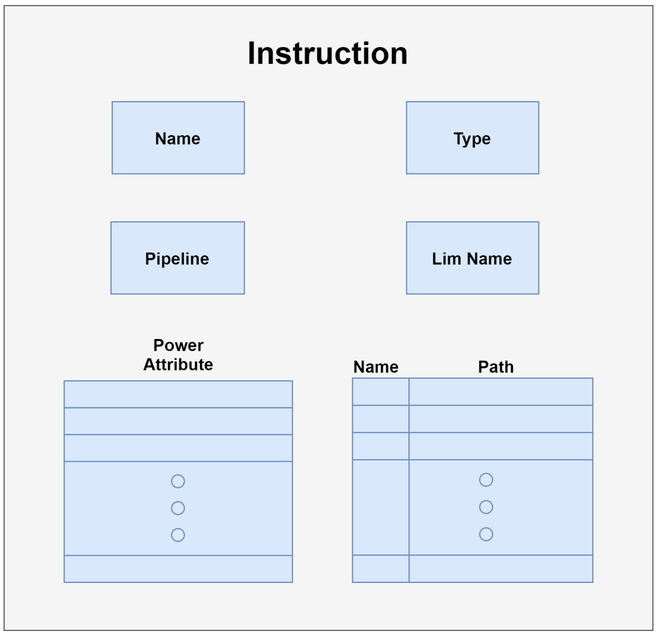


Figure 8.7: Instruction Class

Another important parameter is the Pipeline value, explained in the chapter about the language. In the class, there are two containers: a vector and a map. The vector is used to store an object called *PowerAttribute*. This class is composed of the instance and the attribute related to the power performances specified in the Power section. There is also a map containing a class called *Path*, that stores each path specified in the path section. The

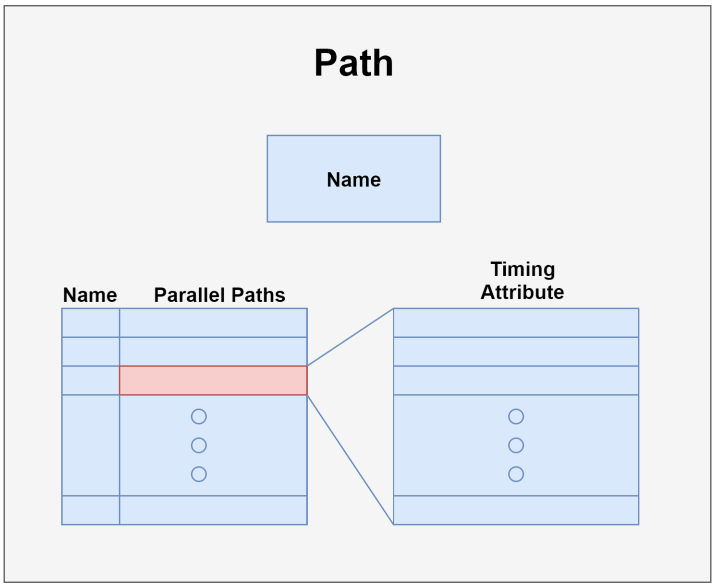


Figure 8.8: Path Class

number of objects inside the map is equal to Pipeline + 1.

8.6 Path Class

The Path class stores the components of each parallel path. The class contains the Name of the Path and a map with each subpath. Each element of the map is a vector containing objects of type *TimingAttribute*
This type of object stores the Instance to compute the timing performance of the associated attribute, similarly to the *PowerAttribute*.

8.7 Attribute Class

The Attribute class contains only a pointer of type Module and it is the base class of three other objects that share a pointer to Module.
The Power/Timing Attribute class adds only an enumeration that specifies attributes as the one of Power/Timing. The Interconnection attribute, in comparison to the others, has information about the interesting port and the index where we have connected a wire. The last info is the normalized wire length. The normalized wire length is defined as the distance in the number of cells between two connected Modules. This can be useful because if we change the type of cell in the memory, it usually has different dimensions.

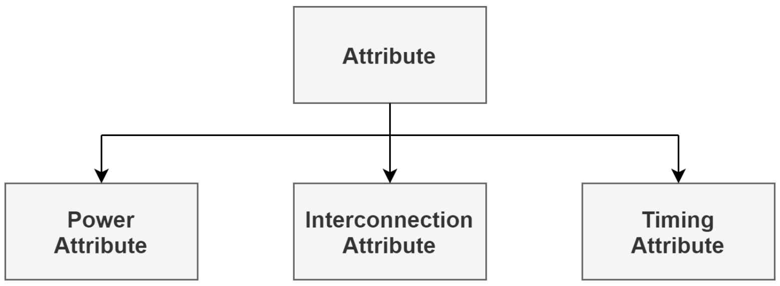


Figure 8.9: Attribute inherit scheme

To obtain the effective length, DExIMA multiplies it by the cell pitch of the memory.

8.8 Code Class

The code class contains the code of the algorithm. The class contains the number of clock steps computed in the code section. The map container stores how many times the instruction is called in the code section. The key on the map is the name of the instruction.

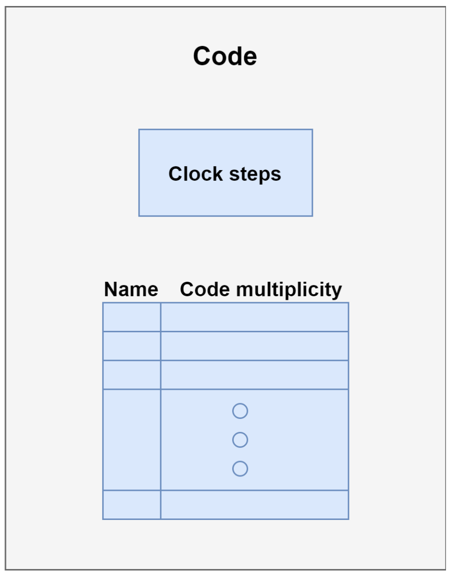


Figure 8.10: Code Class

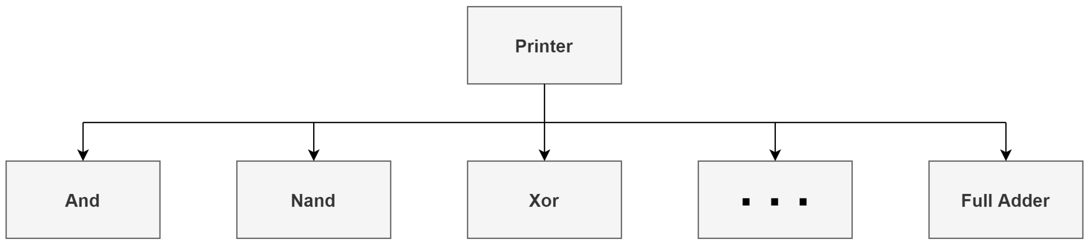


Figure 8.11: Printer inherit scheme

8.9 Printer Class

The printer class is the class responsible for the performance computation of a component. It has several inherit classes that use specialized methods to compute the performance of the specific gate. It uses also a method native of the Printer class to create the component's interface. All the gate types need a class that inherits the methods of the class printer. The printer inherits scheme is shown in Figure 8.11. The Printer object has all the data information about the specific gate, all these data structures are shown in Figure 8.12.
The base variable is a string that specifies the name of the model, for example, NAND. We have five different vectors of string that specify the interface of the component. The port names are the names used by the component in the map section; all the other vectors are based on this vector. The port type specifies if the port is an input or an output. The position of the parameters in the vector is related to the index in the port names vector that we are referencing. The port Parallelism is used to specify how many wires the port has. The port Multiplicity is used to specify how many ports of that type are usable. The Parameter vector can have how many parameters the component needs, and it is not directly linked to the dimensions of the other vectors. In the end, the Fanin vector is a vector of float that specifies the fanin of each port. All these vectors are created in the constructor of the specific printer, in 8.1 we can see a practical example for the NAND model.

```
Nand::Nand()
{
        m_printer_type = "NAND";
        m_parameters = {"INPUTS"};
        m_port_names = {"IN","OUT"};
        m_port_type = {"input","output"};
        m_port_parallelism = {"1","1"};
```

```
    m_port_multiplicity = {"INPUTS","1"};
}
```

Listing 8.1: Nand Constructor Example

The first line specifies the name of the model. The Nand gate has only one parameter called INPUT, which is used in the other vectors. The parameter names are used in the other vectors because, when the component interface is created, the value of the parameters inserted in the Init section are directly substituted in the other vector to properly construct the output interface of the component. The port types specified in the examples are the first an input type and the second an output type. In the specific case, we have for the two ports only parallelism, but it can be more than one, for example in the Flip Flop model. The parallelism is specified in the map section using the operator "[index]". In conclusion, the multiplicity vector is where we specify that the number of inputs defined by the parameter vector and the output is only one. The logic of the multiplicity is simple: the created ports are

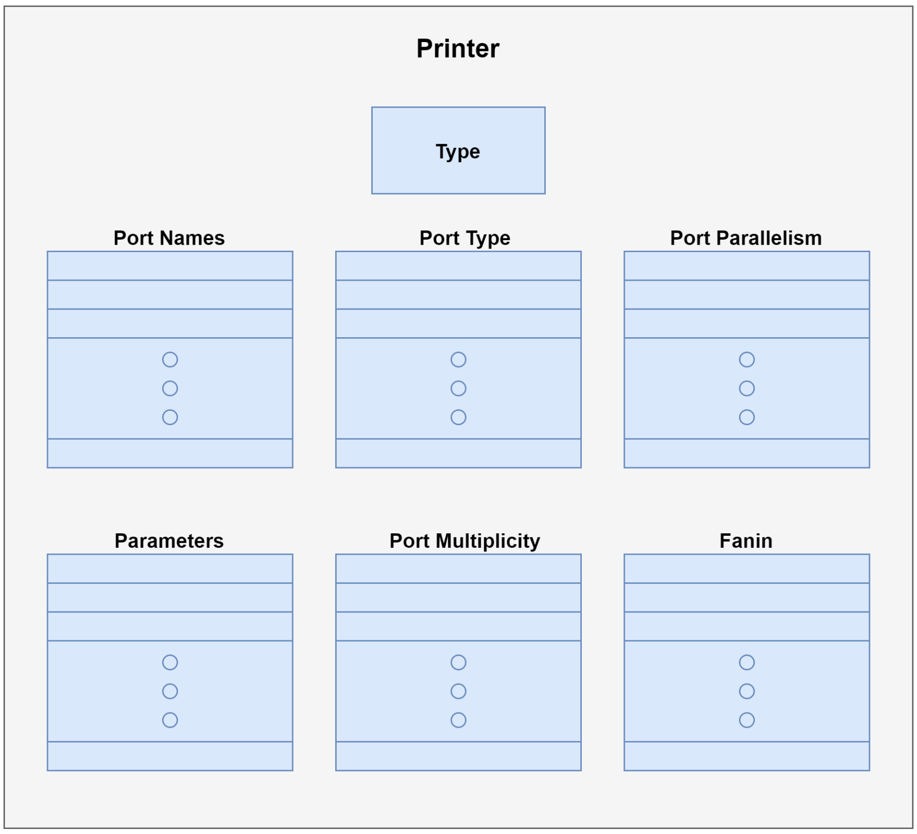


Figure 8.12: Printer Class

named by the name in the vector plus a number starting from zero to the multiplicity value minus one. For example, with a multiplicity of 2, it creates IN0 and IN1. This simple syntax is very useful to create the interfaces of the components in a very fast and automated way. The only problem related to this approach it is that is not possible to use mathematical operations inside it. For example, when we create a Decoder component, we need only the inputs, the outputs will be the power of two of the inputs, but in this way, we need two parameters to specify also the number of outputs to create it. The only parameter not specified in the constructor is the fanin vector because there is a method that is used to create it. This is due to the fact that it depends on the technology, and on the model, it is more complex compared to the previous parameters. This longer explanation highlights how this class is important to the correct work of the program.

8.10 Performance Class

The Performance Class is the Base class used to inherit from the specific Performance classes. The Performance class has the main four performance parameters:

- Dynamic Energy
- Static Power
- Area
- Delay

The inherit scheme is shown in Figure 8.13.
The inherit classes are used to store the main parameters, at different hierarchical levels.

8.10.1 ModulePerformance Class

The *ModulePerformance* class has the four main parameters, but has also an additional map. The two maps have as Keys two enumerations that represent the attribute for the Power and the Timing.
The values inside the maps are float values that represent the values of attributes. For example, all the components have the attribute of the Contamination delay, which can be the same as propagation delay. In that case,

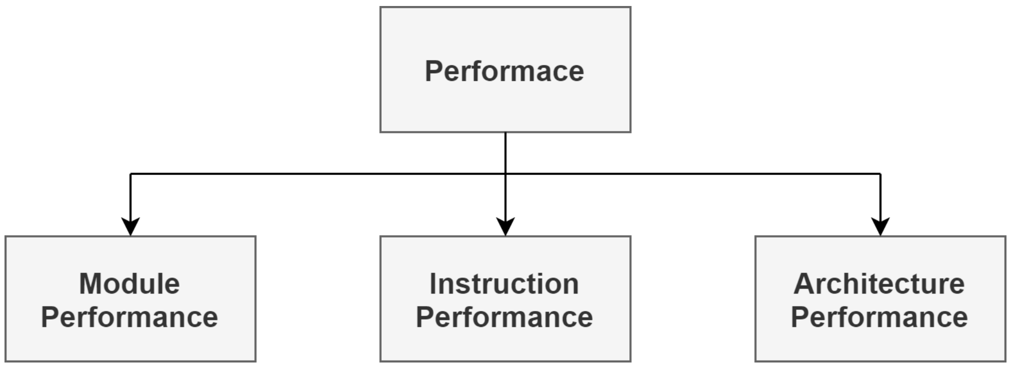


Figure 8.13: Performance inherit scheme

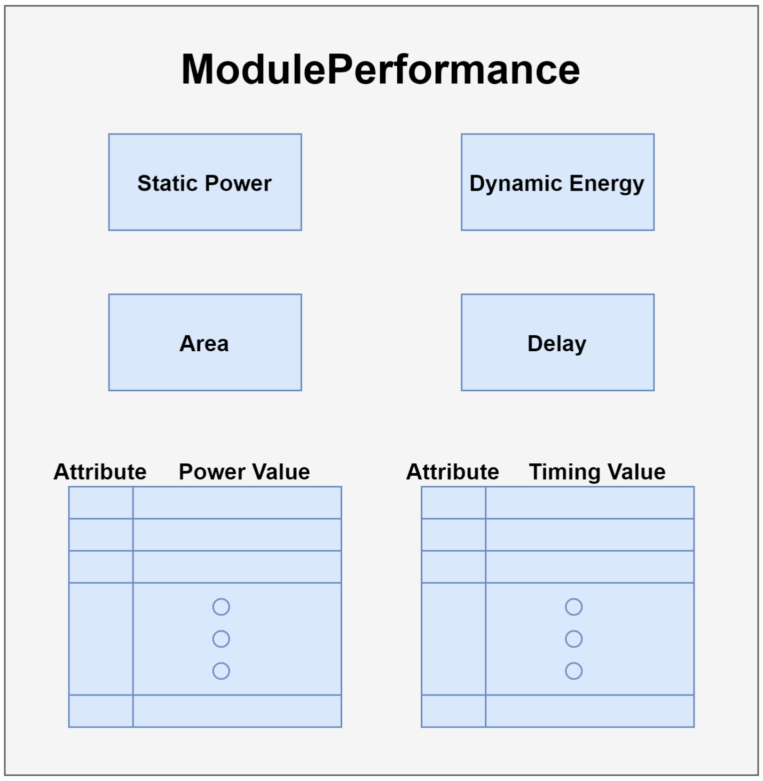


Figure 8.14: ModulePerformance Class

we can extract the contamination delay using this map. All the components can have an arbitrary number of attributes, depending only on the type of model.

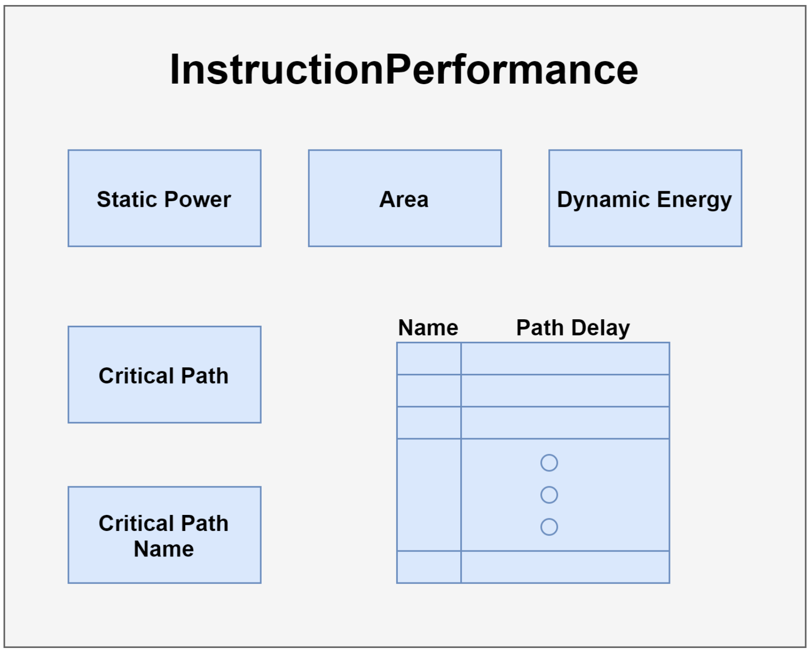


Figure 8.15: InstructionPerformance Class

8.10.2 InstructionPerformance Class

The *InstructionPerformance* class is used in the performance related to one instruction. The delay, in this case, is interpreted as the critical path and has added a string that specifies the name. It is used to highlight the responsibility of the critical path. We have also a map, that, for each path of the instruction, contains the value of the maximum delay of each subpath.

8.10.3 ArchitecturePerformance Class

The structure of *ArchitecturePerformance* class is shown in Figure 8.16. The common four parameters inherited from the Performance class are omitted to highlight only the added data structures. The class has two maps that store the information about Clock Drivers and Memories. Each object inside this map is a *Performance* class type, so we have the four main info about each component. For the clock Driver, this works perfectly, but the memory cannot have information about delay and energy that are intrinsic to memory. These parameters depend on the links inside the memory and the algorithm they are executed and it is not useful to store them. Different considerations can be made for the area and static power. These are fixed for each memory and can be used. Other useful information that can be known, is area and

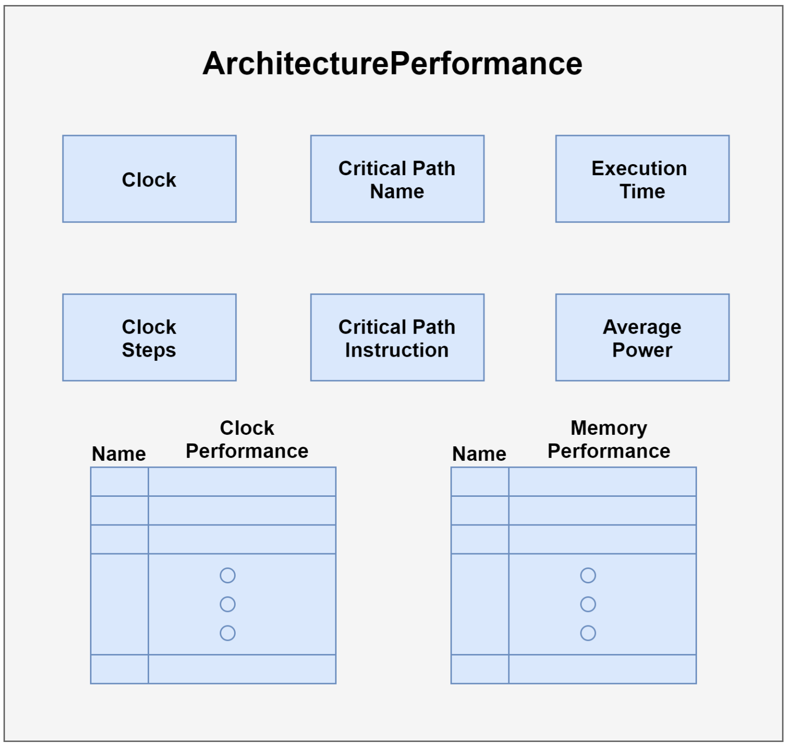


Figure 8.16: ArchitecturePerformance Class

static power related to all the circuitry of the memory like a row and a column decoder. The other unused variables of delay and dynamic energy are used to store these two parameters. The other six parameters are floats and strings that store general information about the circuit like the global critical path and the info to find with instruction and path are responsible for it. This class is the class that prints directly in the dof file, so all the info, except for the information, related to the instructions and technology are stored in this class.

8.10.4 Technology Class

The main purpose of the Technology class is to store the parameters that are frequently used in the computation of the performance. The Technology class is provided with several getter methods that are used by the printer classes when they compute the performance of gates. We did not display this class in an image because it has only float variables inside it. It is not useful to describe these parameters because they are already listed in the dof file. What we visualize in the dof file is simply the content of the Technology class. The class has two different scope methods: the first part consists of the interest of the methods in the computation of the derived parameters from the technology file, and the others are used for printer use. For example, the computation of the effective gate length from the TechFile parameters is a method of the first part, instead, we have very useful methods used to Normalize/Denormalize the capacity values, this belongs to the second part.

Chapter 9

Compilation Process

9.1 Sections stack

The compiler is the class that reads the dex file and creates all the objects in the Architecture class. As we already know, the dex file is organized in sections that use the keywords begin/end to open/close it. The compiler uses a stack to manage these sections. When the compiler reads a begin line, it checks if the next section is correct. If what it expects, is in the correct order it pushes the keyword of the specific section inside the stack. When the compiler encounters another begin keyword, it checks the admissible stack deep, because there are sections that do not admit sub-sections. If the stack is not "full" for that section, it makes another push.

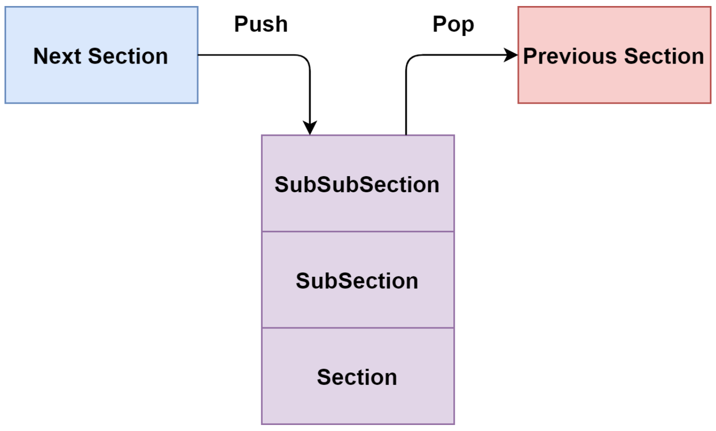


Figure 9.1: Compiler Stack

If the compiler encounters an end keyword it checks if the section keyword is the same at the top of the stack, if it is, do a pop operation. The schematic

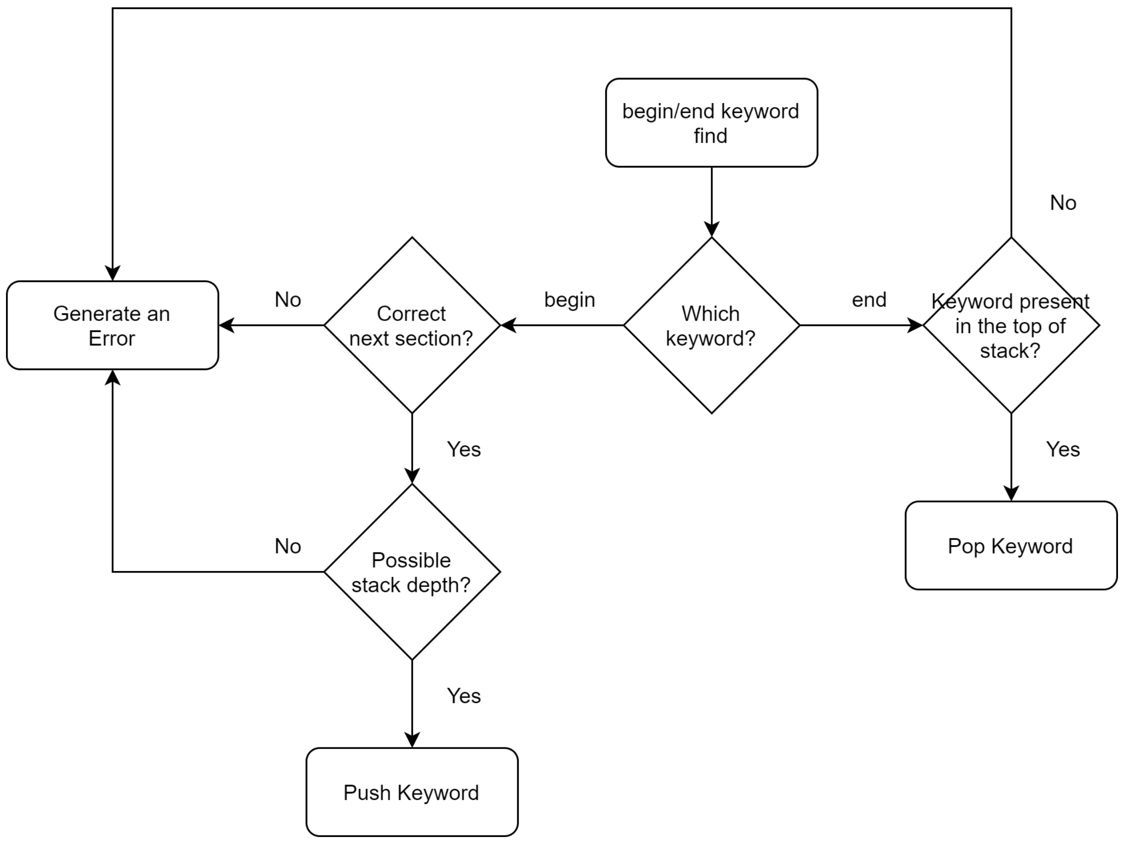


Figure 9.2: Sections Stack Algorithm

operation and algorithm are shown in Figure 9.1 and 9.2.

9.2 Sections State Machine

The section parser is organized using a software state machine of Moore type. Each line of the code means one operation of the state machine. The begin/end keyword are the triggers to the transitions inside the state machine. The main state machine has one state for each section. If one section has sub-sections inside, there is a nested state machine that controls the sub-sections. There is a very useful function inside the Compiler class called *transition_logic* that is used to manage directly the Section stack and the transitions between states. The function needs in input:

- **Line string**: It is the string of the line that we are analyzing in the dex file.

- **Future Keyword**: It is the keyword of the next state we need to transit.
- **Stack depth**: It is the allowed depth in the Stack means the nesting level of the next subsection.
- **Transition State**: It is the enum type related to the state we want to transit. This can be used with different types of enum because is a template function.

The function returns a bool variable, the value results true if at the line it starts with a begin/end keyword, otherwise, it results false. This is very useful not to treat the open/close section like the line in the specific section. Inside each section's state, there is a specific function to the section to parse. The power of this approach is that the code is modular, because in the future, in order to add or to remove a section, we simply have to add or remove the state and create the function to parse the new section. The function helps to transit inside it without knowing the implementation details about it and add the new section with few lines of code.

9.3 Compiler Error

The *CompilerError* class is responsible for the generation of the errors in our program. The object is created only one time in the *Compiler* class, and we exchange the reference in the *Parser* classes. The object is very simple, it contains the variables:

- **`m_error_occur`**: a *bool* variable that stores a flag to indicate if an error occurs or not, using True/False. It is used a lot of times inside the compiler routines because it indicated if continue or stop the program in case of error.
- **`m_error`**: an *enumerate* called *error_type* that contains the type of error that occurs during the compilation process.
- **`m_error_line_number`**: an *unsigned int* that stores the line number where the error occurs.
- **`m_error_line`**: a *string* that stores the code line responsible of the error inside the dex file.
- **`m_problem`**: a *string* containing the error message displayed by DEx-IMA.

- **`m_suggestion`**: a *string* containing the suggestion message displayed by DExIMA.

The use of the object is very simple: where an error occurs we use a method called `set_error`, where we specify the type of error and eventually the specific message, like error and suggestion. Normally, each error type has associated with a message stored in a file. If we need a specific error that includes information about a component or other information, it is possible to do. When we use this method, the error is set to true and it stores all the information needed to print in the terminal. If we call again the set error method, the information remains the same because we need to consider only the first error that occurs, since the following can be a function of the previous one.

9.4 Parser Organizations

The most used classes inside the compiler are the derived classes of *Parser*, which have the most used and useful functions to manipulate the strings. All the classes related to the parsing of the code lines use the *regex* expressions instead of using the built-in C++ methods of class string. There is a specific parser for each section, that is specialized to understand and parse its section.

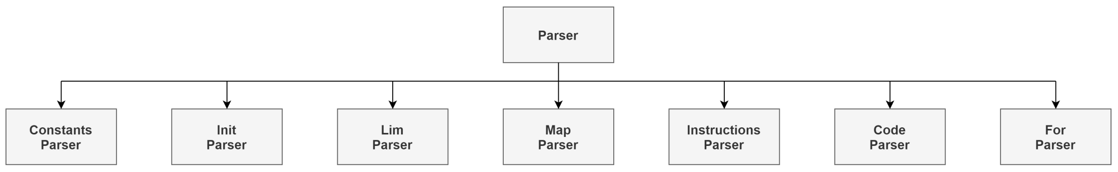


Figure 9.3: Parser Inherit Scheme

Each state of the compiler takes the wanted parser and uses its methods inside it. In each parser, there are regex variables used to match the most used expressions. Now we present some useful features of some of these derived classes starting from the *ForParser*.

9.4.1 For Parser

The *ForParser* is the object that manages the for instruction but it is used also to store some variables:

- **Iterator**: a string variable used to store the name of the iterator of the for loop.
- **Start value**: an int variable that stores the start iterator value of the loop.
- **Step value**: an int variable that stores the iterator increment value of the loop.
- **Stop value**: an int variable that stores the stop iterator value of the loop.

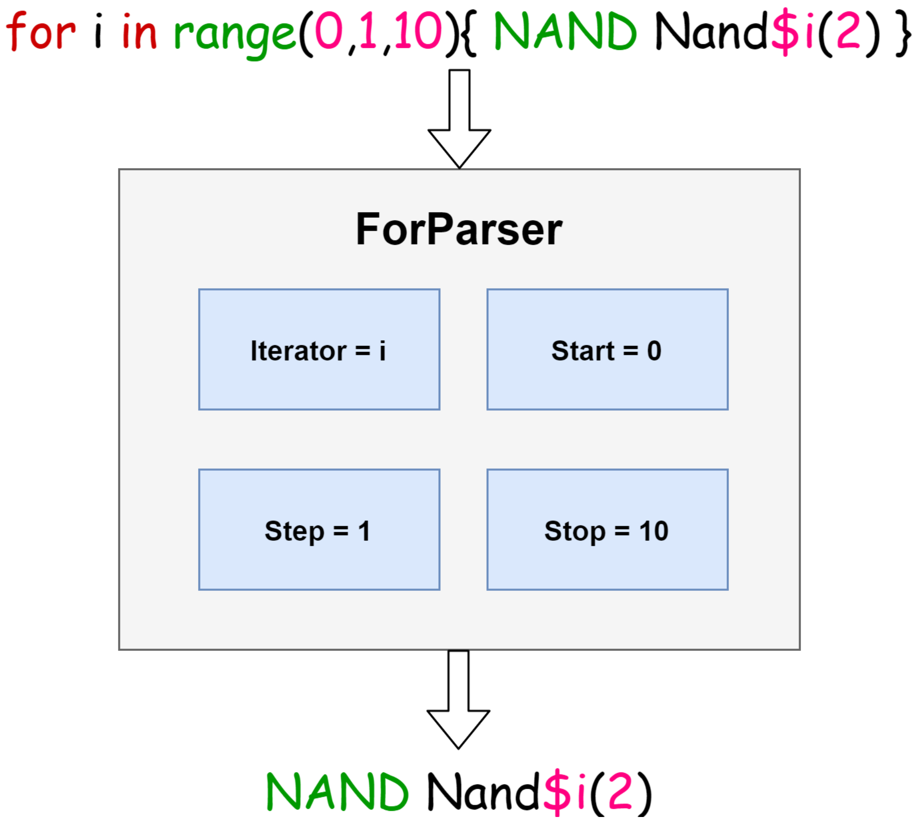


Figure 9.4: ForParser Operation

The scope of this parser is to get in input the string that uses the For construct. From the string, it extracts the four parameters needed and stores them inside the object. Finally, the process returns only the string related to the internal code part. After this process, we use the info extracted to do a C++ loop substituting each step with the value of the iterator inside the output string. When the For parser does this operation it checks the validity of all the syntax and the parameters. For example, if we insert values that create an infinite loop, the ForParser generates an error before starting the infinite loop. The Figure 9.4 shows the process just described. In the compiler, there is a function that recursively applies this operation because we can have nested For loops.

9.4.2 Constants Parser

The *ConstantsParser* class stores the constants inside the code and substitutes them in the string lines. This class has two maps: one that contains all the couple of the constant's names and the value, and another map that is used only to contain a couple of built-in constants and their values. This second map container is what is sent to the Technology class after the constants section. After the storage of a constant, each time the compiler parses a line, first of all, it checks if the expansion of one or more constants is present. This check is done for all the constants inside the map. This means that, if we have a lot of constants, we spend some time on each line to check the presence of all the constants, so it is useful to use the math environment when possible.

9.4.3 Init Parser

The *InitParser* is used to create the Module objects. It can be used in three different sections in order to have three dedicated methods that use common parsing operations. The three interesting sections are: init, logic, and cells sections because in these three sections we create the modules in the three namespaces of DExIMA. The operation that the InitParser does, is to check if the Model name exists and if the Instance name is already used by another Module. The last part checked is related to the values and the number of parameters, but this operation is done by the specific printer of the Model. If the model is used for the first time, is created also the printer object related to the model, and is inserted inside the map of printers. When the Module is created it has only the parameters, the Name, and the Model. The creation of the Input and the Output port is related to the generic printer methods as shown in Figure 9.5. The information used in the creation is the info in the constructor of the printer and the parameters inserted by the users. There is also information got by the Technology to create the fanin of the Inputs objects. The Init parser stores also a vector with names of the LiMs

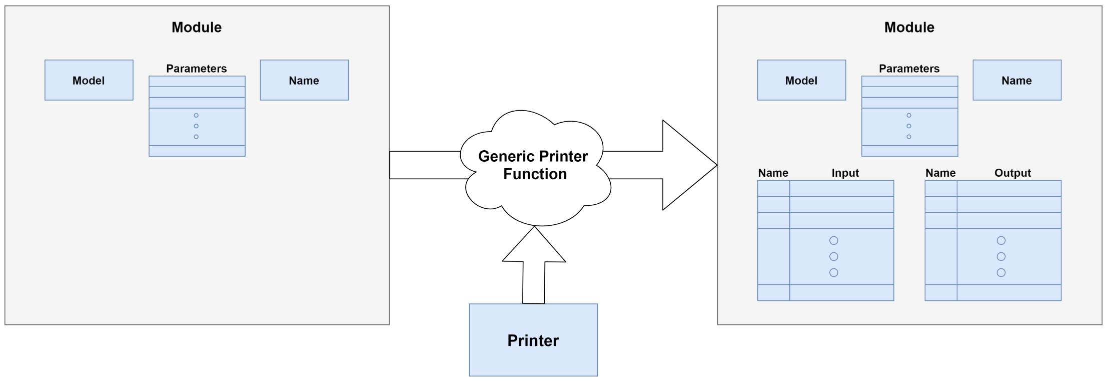


Figure 9.5: Module Interface Construction

to manage the transition to the next memory states.

9.4.4 Lim Parser

The Lim Parser constructs the memory geometry specified in the memdef section and does the computation to check if the Lim Module built is coherent with the number of rows and columns. The compiler generates an error if the values are not coherent and also makes suggestions to correct the problem with the suggested values. This parser creates the Lim object with all this information and fills the cells with a memory cell each.

9.4.5 Map Parser

The Map Parser is used to parse the map sections, both internal and external of Memory. It is specialized and has three different methods, one for each namespace. The map parser is the parser that has the highest computational cost, it needs to check several parameters. Now we explain these steps considering the worst case, the case of the LiM parser. The order of the operations is:

1) **Check the link operator**: In the first step, we identify the link operator and divide the instruction into two parts. The left is part related to the output port and the right one is related to the input port.

2) **Check the dot operator**: For each part, we check the presence of the dot operator and divide it into other two parts: the left represents the name of the instance and the right represents the port name.

3) **Matrix coordinates division**: Each part related to the instance name can specify the coordinates in the LiM array. First of all the string is already divided into other two pieces: the piece related to the only instance name and the other with only the coordinates of the cell.

4) **Instance name evaluation**: In this step, if the coordinates are specified, they are extracted and we check if the values are admissible coordinates, and the indexes do not exceed the memory array. After the coordinates, the name of the instance is checked. In particular, if the coordinates are specified, we check that the gate is present inside the memory cell. If we do not have coordinates, we check if the gate is present in the output cell gate container.

5) **Port index extraction**: The parts related to the port names are scanned to check the presence of the index operator. If present, the string is divided into two sub-strings with the name of the port and the index of the port.

6) **Port checking**: The first operation of the port checking is related to the existence of it. In fact, we check if the Instance with name extracted before having a port with that name. If the answer is positive, we check if it is an input or an output and, after having verified that, if it is in the correct position to be an input/output.

7) **Index checking**: The specified index is checked if it is a possible value for that port and does not exceed the maximum possible parallelism of the port. If all ends correctly, we set the parallelism of connection, if the index is specified we want to connect only one wire, otherwise, the whole bus is connected to the other port.

8) **Connection checking**: Finally, we need to check if the two ports can be connected. We can have two negative situations: the first happens when the two instances have a different number of wires involved in the connector, the second is when the considered input is already connected to another port.

9) **Fanout update**: The last step is to increment the Fanout of the output port with the value of Fanin of the input port and set the connected flag of the input port.

In all these steps we have omitted all the checking sequences of the spaces that can occur when paring the line, and we didn't say anything about the

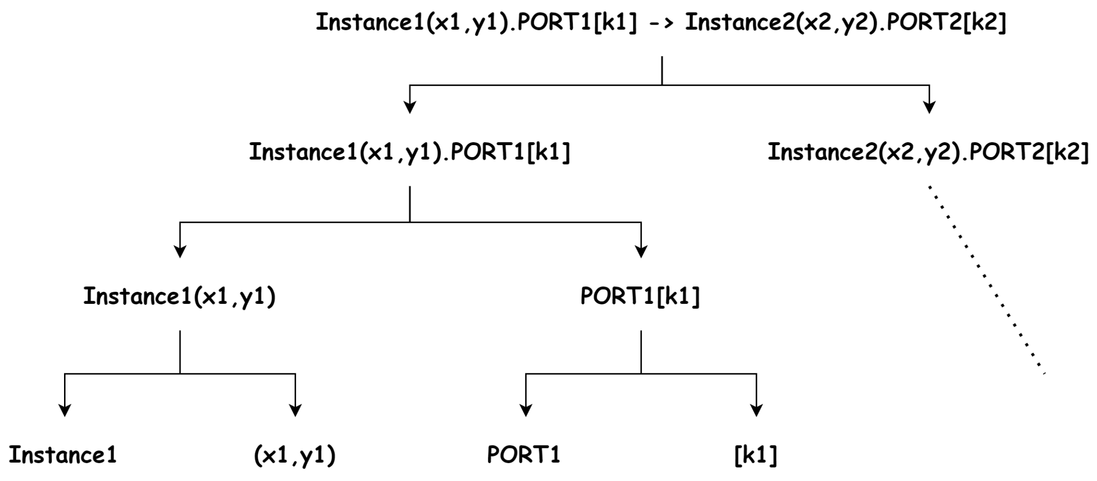

Instance1(x1,y1).PORT1[k1] -> Instance2(x2,y2).PORT2[k2]
Instance1(x1,y1).PORT1[k1]
Instance2(x2,y2).PORT2[k2]
Instance1(x1,y1)
PORT1[k1]
Instance1
(x1,y1)
PORT1
[k1]

Figure 9.6: Map section parsing pyramid

searching of the unwanted word in the black spaces of the line. Figure 9.6 is showing the process that is like a pyramid where at the top we find the line to parse and at the bottom all the data needed to know by the instruction and to be checked.

9.4.6 Math Parser

The Math parser, how as mentioned before uses the Shunting Yard algorithm to parse the line and compute the result of the maths operation. The Shunting Yard algorithm consists of two steps to compute the math operation. All the mathematical expressions we write are called *infix* notation. The first step to compute the expression is to translate the *infix* notation to the so called *reverse polish notation* (RPN) or *postfix* notation. In the postfix notation, the operators follow their operands, the advantage of this notation is that we do not need parenthesis to represent the operation and it is easier to compute in comparison to the infix notation. To make the infix to postfix transformation we need two data structures: a queue (FIFO) and a stack. The queue is used for the output of our computation and the stack is the operator's stack where we put the operators during the computation. We read our expression left to the right and we divide the expression into tokens (operands and operators) and put them inside an array. After doing this we follow the algorithm:

- If a number is found, we push directly to the output queue

- If an operator is found, it is pushed inside the operator's stack
- If the operator has precedence lower or equal than the operators at the top of the stack, the operator is popped off the stack and added to the output
- At the end, the remaining operators to the stack are popped out and pushed to the output

To clarify it, we make an example, we have the infix expression:

$$Expression(\text{infix}) = 2 \times 3/(7 - 5) \tag{9.1}$$

First of all we divide it into a token list:

$$tokens = \begin{bmatrix} 2 \\ \times \\ 3 \\ / \\ (\\ 7 \\ - \\ 5 \\ ) \end{bmatrix} \tag{9.2}$$

The operations we do are shown in table 9.1. The result of the operations is our RPN expression:

$$Expression(\text{postfix}) = \text{2 3} \times \text{7 5 - /} \tag{9.3}$$

Having our postfix expression is possible to evaluate easily the computation of the whole expression. To compute the result of the postfix equivalent we need only a stack. Also in this case we use the tokens and proceed from left to right. The algorithm is the following:

- If a number is found, we push in the stack
- If an operator is found we pop one value of the stack and use like right operand. Pop another value and use it as the left operand. Finally, we evaluate the operation and the result is pushed into the stack.
- At the end we pop out the result contained in the stack

Related to the previous example, the operations to do are shown in table 9.2. And we have our final result of 3. These are all the processes done by the *MathParser* and replace the expression with the computed value.

Token	**Action**	**Output Queue**	**Operators Stack**
2	Add number to output	2	
×	Push operator to stack	2	×
3	Add number to output	2 3	×
/	Pop operator to stack	2 3 ×	
	Push operator to stack	2 3 ×	/
(	Push operator to stack	2 3 ×	(/
7	Add number to output	2 3 × 7	(/
-	Push operator to stack	2 3 × 7	- (/
5	Add number to output	2 3 × 7 5	- (/
)	Pop operators until "("	2 3 × 7 5 -	(/
	Pop "(" operator to stack	2 3 × 7 5 -	/
	Pop entire stack	2 3 × 7 5 - /	

Table 9.1: Infix to postfix transformation

Token	**Action**	**Output Stack**	**Expression**
2	Push number to stack	2	
3	Push number to stack	3 2	
$\times$	Pop Right operand	2	$\times$ 3
	Pop Left operand		2 $\times$ 3
	Evaluate Expression		2 $\times$ 3 = 6
	Push Result	6	
7	Push number to stack	7 6	
5	Push number to stack	5 7 6	
-	Pop Right operand	7 6	- 5
	Pop Left operand	6	7 - 5
	Evaluate Expression	6	7 - 5 = 2
	Push Result	2 6	
/	Pop Right operand	6	/ 2
	Pop Left operand		6 / 2
	Evaluate Expression		6 / 2 = 3
	Push Result	3	
	Pop Final Result		

Table 9.2: Postfix expression evaluation

Chapter 10

Performance Computation

The Process of performance computation uses an approach to minimize the number of computations in the circuit. The philosophy is to compute the performance of some gates only if necessary. If we have two or more gates with the same performance, it is not necessary to compute again. Two components are defined identically if they have three parts in common:

- Model
- Parameters
- Fanout

The complete process is described in 7 steps. This chapter explores this process step by step. All the process is managed by the Simulator Class using the methods provided by the Architecture class.

10.1 Step 1: Module Encoding

The first step is the Encoding step. As said in the previous chapter, the Module class has a field called Code that is a string used to identify uniquely a Module with a simple string. The process is represented in Figure 10.1, where we have our Module class, and in output, we obtain the Code string from it. Each Module has a method called `encoding()` that creates the equivalent Code represented by the specific object. The whole process is called by the Architecture class that makes two main loops. One loop interests the architecture modules, the other is a loop of LiM memories that have a method to encode the Modules inside each memory. Each LiM has

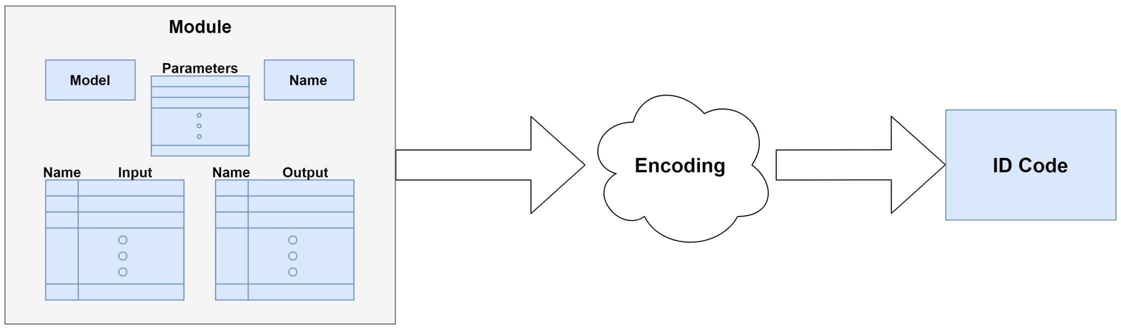


Figure 10.1: Encoding Phase

hierarchically two loops to encode the memory cells Modules and the out of cell Modules. The code has three fields, the three specified in the intro of the chapter. The first field of code contains the name of the model, for example, FF. After, the code is followed by an underscore with the list of the parameters of the model. The list separator is the char "+". The generic string part of the second field is $par_1+par_2+par_N$. The last field is the field related to the fanout and also in this case the fields are separated by the underscore. The fanout is written dividing the port with the char ":" and each wire with the "+" char. The third field is constructed by the example $wire_1+wire_2+wire_N:wire_1:wire_1+wire_2+wire_N$. For example, a simple Flip Flop code can be *FF_3_3.12+2.3+4*.

10.2 Step 2: Code Presence Verification

The second step is a verification step. There is a map inside the Architecture class that stores all the Module Performance of the circuit. The map has, as Key, the Code of the Model extracted at Step 1, and the value is an object of ModulePerformance related to the Code. Having this data structure, we associate a ModulePerformance to each Code. Figure 10.2 represents the process, which is a simple searching process inside the map structure. If the result of the verification shows to have the ModulePerformance inside it, we go directly on Step 4 and skip Step 3. If the ModulePerformance associated with that code is not present inside it, we go on Step 3. This step is done for all the Modules Encoded in the Architecture Class, including all the Modules inside the Lim objects. All the Modules refer to this map, including the nested objects.

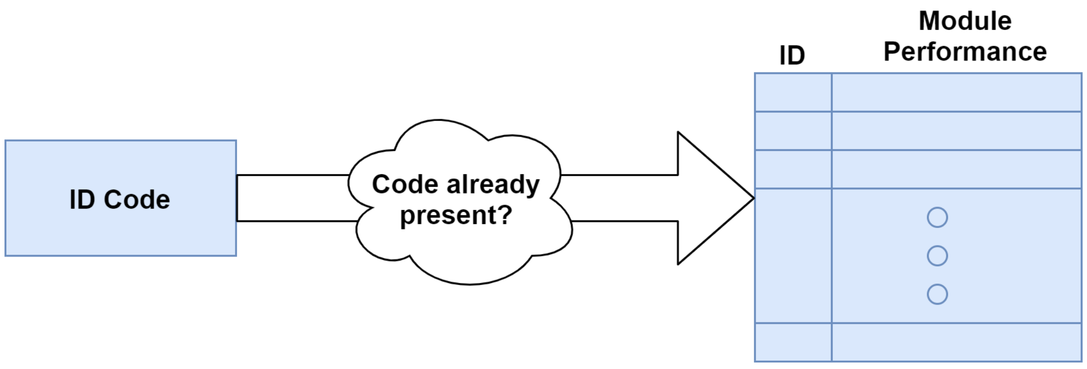


Figure 10.2: Verification Phase

10.3 Step 3: Performance Computation

Step 3 is evaluated when, at Step 2, a negative answer to the presence inside the map structure is taken. When happens, we take from the Printers maps the specific Printer of the module to compute the Performance. The process uses a method of the Module to take the Model type, the Model type is also the Key of the map containing the Printers.

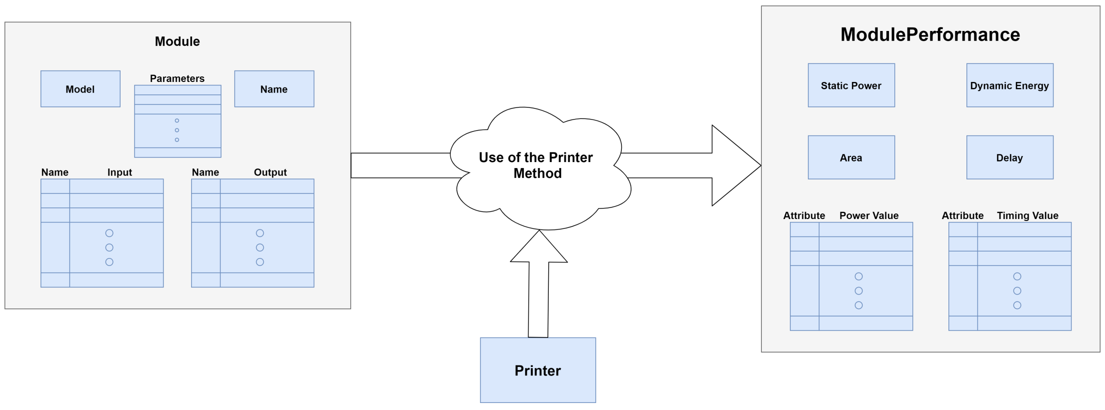


Figure 10.3: Performance Computation Phase

Each printer has a method that gets in input an object of type Module and applies all the specific methods to construct and fill an object of type ModulePerformance. In the end, the method returns this object.

10.4 Step 4: Performance Insertion

The Performance Insertion is a simple operation, where the Code computed by the encoding phase and the ModulePerformance are associated together in the map of all the ModulePerformance objects. Now, when we apply again Step 2, we already find the Computed ModulePerformance object.

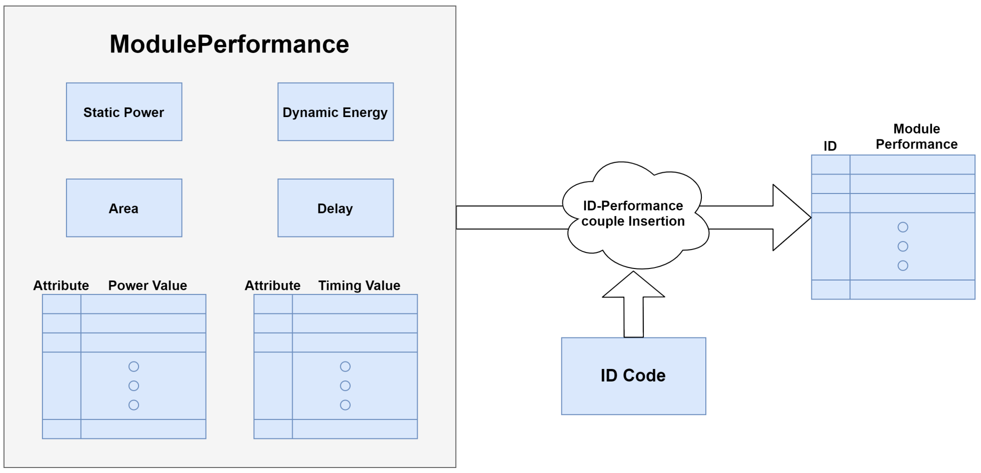


Figure 10.4: Code Insertion Phase

10.5 Step 5: Extraction Chain

The extraction Chain is the main operation used to get the computed Parameters in the previous steps.

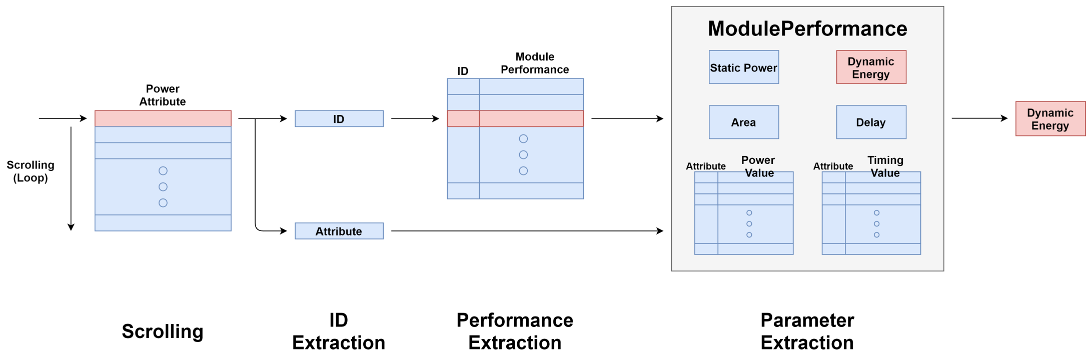


Figure 10.5: Extraction Chain Phase

The complete Extraction chain is shown in Figure 10.5. Now we look at all the steps of the extraction chain.

10.5.1 Scrolling

The first step is what we have called *Scrolling*. It is a loop over some data structures, and it depends on what type of performance parameter we need. To compute the whole area and static power, the loop is done with all the Modules objects of the architecture. The loop process is identical to what we've done during the encoding phase. This happens because all the components contribute to the area and static power. A different loop is done when we need delay and dynamic energy, the data structure used in these cases are the vectors inside the Instructions. For the dynamic power computation, it is used the PowerAttribute vectors and the TimingAttribute vectors for the delays computation.

10.5.2 ID Extraction

The ID Extraction is simply what we get from the Scrolling phase. Usually all the data structures in the Scrolling phase point to a Module object, more correctly all these Modules are in reality pointers to Module object. In fact, the "original" pointers are stored in the main structures inside the Architecture class. The pointers inside the Instruction objects are pointers to the same object creates previously inside the Architecture class. If we scroll inside a simple Module container, we extract only the ID Code inside the Module object. If we are using the Attributes objects inside the Instruction we extract also the attribute associated with the Module, like shown in Figure 10.5.

10.5.3 Performance Extraction

The Performance extraction phase is the phase where we use the ID Code to get the associated ModulePerformance object. The ID point to the container stored inside the Architecture object. The Scrolling is usually done by the Architecture and hierarchically to the nested object using some methods. For this reason, not all objects can access this container. All the methods get the reference of the container and get the desired ModulePerformance contained inside it.

10.5.4 Parameter Extraction

Now we have our ModulePerformance object and from this we can choose which parameters getting from it, depending on which type of Scrolling loop we are in. If we are computing the area and static power, we can directly get both at the same step. In the case of Instruction Scrolling, we need also the Attribute associated, if the attribute is not specified, the performance is directly extracted from the four main fields. On the contrary, if one attribute is present, we accede to the two maps.

10.6 Step 6: Instructions Accumulation

The Instructions Accumulation step is a loop to all the Instruction objects in the Architecture. For each Instruction, we do a series of Extraction chains in the data inside it. We mainly do a Scrolling to the PowerAttribute vector and a Scrolling to the TimingAttribute vectors inside the Path objects.

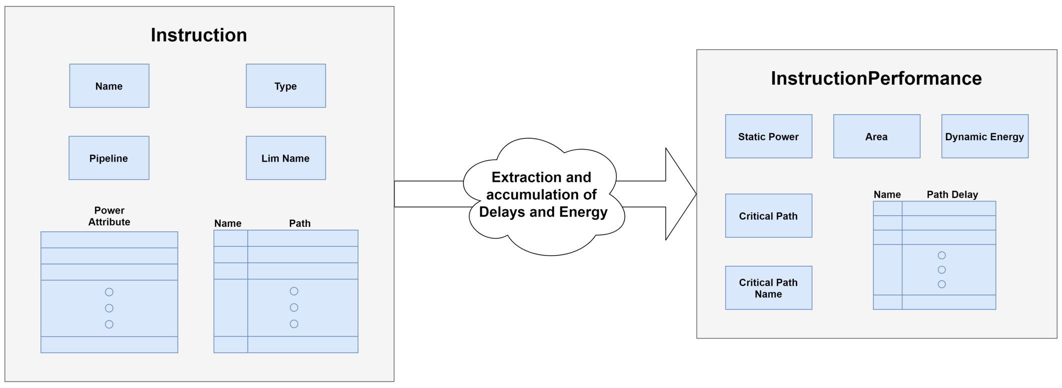


Figure 10.6: Instructions Accumulation Phase

The result, like shown in Figure 10.6 is an InstructionPerformance object that is the object called where we use that instruction.

10.7 Step 7: Total Performance Computation

At the final step, we have all the ingredients to compute the Performance of our Architecture. We simply need the InstructionPerformance container and the Code of our algorithm. The result is stored inside the ArchitecturePerformance object, and after it is printed in the dof file. The process is managed by the code that uses the multiplicity of each Instruction to accumulate the dynamic energy and, in the meantime to compare the delays of each instruction to find the worst. Finally, we have the total energy and the critical path.

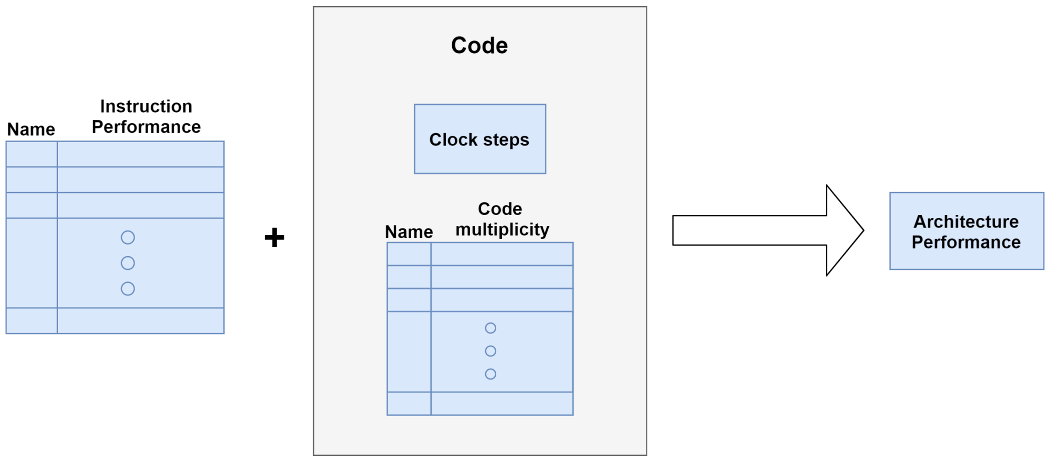


Figure 10.7: Total Performance Computation Phase

At this point, DExIMA chooses the clock period, which is the critical path if the user does not specify it. On the contrary, if the user sets it in the constants section, this is the value considered. To compute the dynamic power, we first compute the execution time of the algorithm:

$$t_{algorithm} = ClockSteps \cdot t_{period} \tag{10.1}$$

and heaving the total energy we can compute simply the average dynamic power:

$$P_{average} = \frac{E_{dynamic}}{t_{algorithm}} \tag{10.2}$$

The other parameters are simply the accumulations of all the Extraction chain operations for the other parameters.

10.8 Algorithm

Each step has usually sub-steps to complete the operations, the total algorithm execution is shown in Figure 10.8 where only the main operations used are presented. There are more loops inside that are omitted to have a better representation of the whole process.

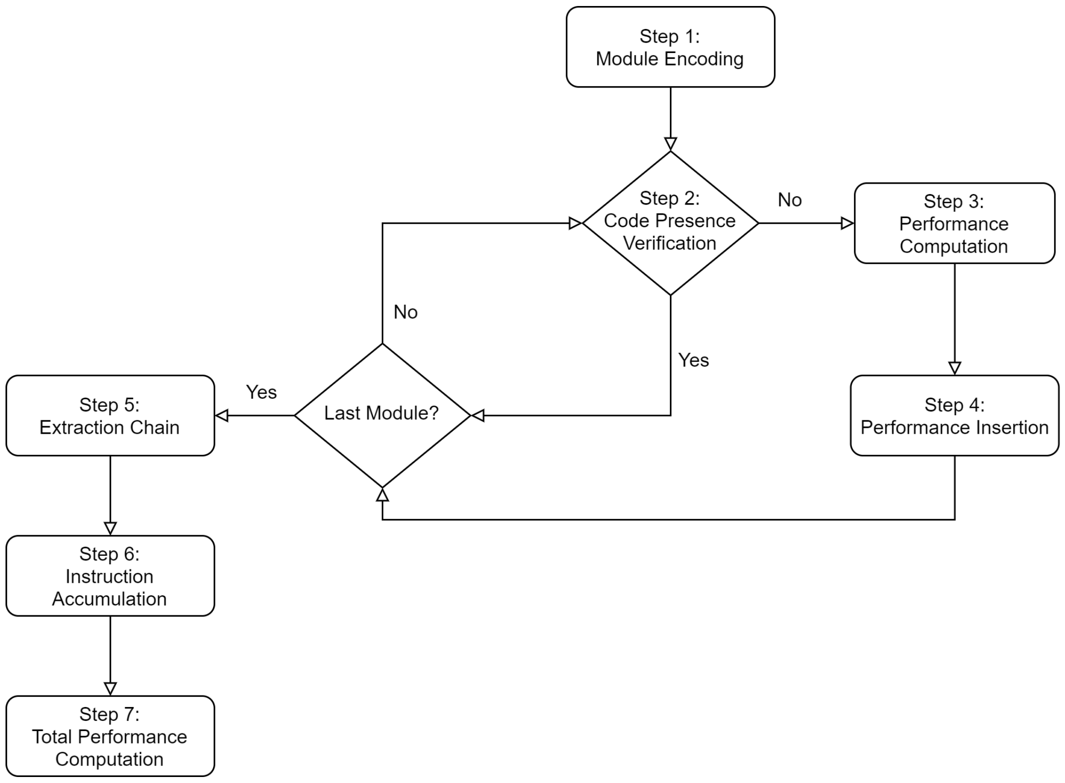


Figure 10.8: Simulation Algorithm graph

Chapter 11

Binary Neural Network Simulation

In this chapter, we will see the implementation of a Binary Neural Network in hardware and look at the performance obtained using different approaches and simulations. We will see the comparison between the new version and the previous version of DExIMA comparing it using modern tools like Synopsys Design Vision. The Binary Neural Network (BNN) is a type of Neural Network that uses a binary approximation to be reproduced in hardware in a more suitable way. We start by explaining how a Neural Network is made and how it works. At the end of this chapter, we will see some simulation results of this specific Architecture and the related algorithm.

11.1 Introduction

The Neural Networks are structures composed of elements called Neurons, which are interconnected between them by links called synapses. This name is inspired by the way in which the brain performs a particular learning task [21]. The Neuron is the base computational unit, as it is shown in Figure 11.1. The mathematical expression of a neuron can be written as:

$$net = \sum_{i=0}^{N} X_i \cdot W_i + Bias \tag{11.1}$$

The terms of expression are:

- X_i: the inputs of our Neuron, they can be a floating-point number. For the BNN, they are binary values ± 1.

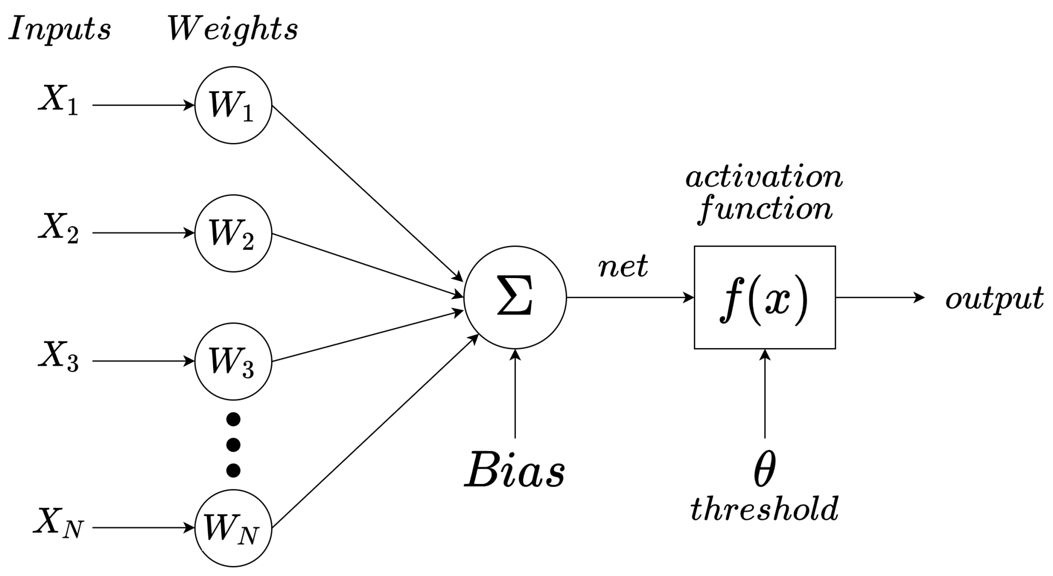


Figure 11.1: Artificial Neuron.

- W_i: the weight associate with the link called *synapses*. The most used type of weights is the floating-point values having the full precision. In the specific case of a BNN, we use a binarized weight that can assume the values of ± 1.
- *Bias* Is an additive term

The output of the net is passed between a function called *activation function* that it is used to insert a non linear behaviour using particular functions [23]. This function can be tuned inserting a threshold. The non linearity is necessary when we work with classes that are not linearly separable. The most used activation functions are:

- **Sigmoid Function**:
$$f(x) = \frac{1}{1+e^{-x}} \tag{11.2}$$
- **Hyperbolic Tangent**:
$$f(x) = \tanh(x) \tag{11.3}$$
- **ReLU** (Rectified Linear Unit):
$$f(x) = \max(0, x) \tag{11.4}$$

11.2 Neural Network

The neural networks are a composition of multiple neurons and they can be grouped by layers. Each layer is connected with the previous one and with the successive layer. The scope of the neural network is to recognize some objects of a specific class. This process is called classification, in which we insert in input an object and the response of the network corresponds to the type of class the object belongs to. The preliminary operation allowing our network to recognize the objects is called training. During training we insert, in input, an object with an associated label and the network responds with some results. The result obtained can duffer for the wanted class, so, the output error used to back-propagate the signal is computed. The back-propagated signal is used to update the weights. The updated weights now allow the network to recognize the correct classes. When the training is complete, if we have done good training with a good Neural Network Architecture, we get in the output the correct class of the input images with a certain accuracy rate. The Neural network can have a different number of layers, but they can be divided into three different categories:

- **Inputs Layer**: It is a layer that simply sets the input data to the Network, the number of the neuron is equal to the number of input data we have.
- **Hidden Layers**: The hidden layers are used for the computation and propagation of the information inside the Network.
- **Output Layer**: The output layer is responsible for the classification, the number of output neurons represents the number of classes that can be classified.

The simplest architecture is a single layer Perceptron shown in Figure 11.2 sub-figure a). But the single layer is able only to recognize linearly separable classes. By increasing the complexity of the model, we introduce the hidden layers, more layers imply more complex models. A neural network has the peculiarity that each neuron of a layer is connected to all the neurons of the next layer. This means that a large Neural network can have too many weight parameters that can be trained. In the Neural Network, there are two problems related to the complexity of the model. If we insert too many layers, the model begins too complex and this means that the network is capable of recognizing very well the samples used for the training, but has a poor

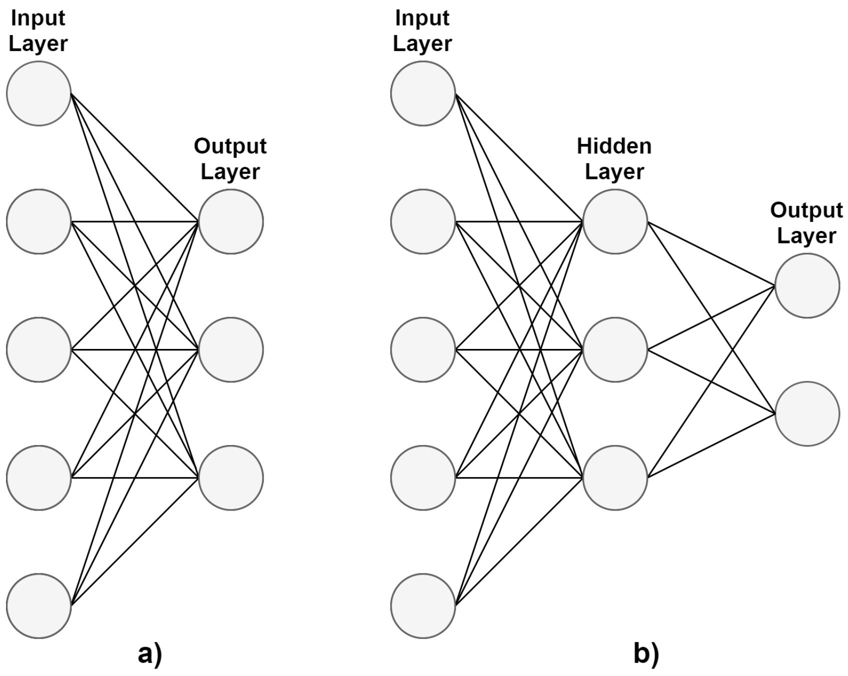


Figure 11.2: Neural Network Examples, a) Single Layer Neural Network, b) Multi Layer Neural Network.

performance when we try to classify new input samples. This phenomenon is called **overfitting**, and it occurs when we dispose of more parameters than needed. It loses its generality and it can recognize well only the training set. On the opposite side, we have the **underfitting**, this happens when we have not enough parameters to classify our samples and we need a bigger and more complex network to recognize them with suitable accuracy.

11.3 Convolutional Neural Network

The Convolutional Neural Network is a type of neural network used to classify and recognize complex data like images. From now on we reference always images to explain better what we need to focus on. The Layers of this type of network can divide into three categories:

- **Convolutional Layers**: Used to perform the convolution operation between the input image and a filter.
- **Pooling Layers**: Used to reduce the size and the complexity of the CNN.

- **Fully Connected Layers**: Layers used to perform the classification operation.

For more details about it [24]. From this point, we follow the same explanation and hardware implementations shown in [25].

11.3.1 Convolutional Layers

The convolutional layers take into account two objects: the input image called Input feature map **IFMAP** and the filter called **Kernel**. The result of the convolution is called Output feature map **OFMAP**. The Kernel is simply a matrix of the weight values, in the case of an RGB image the matrices are three, one for each color. In the Figure 11.3 we see an example of the Kernel.

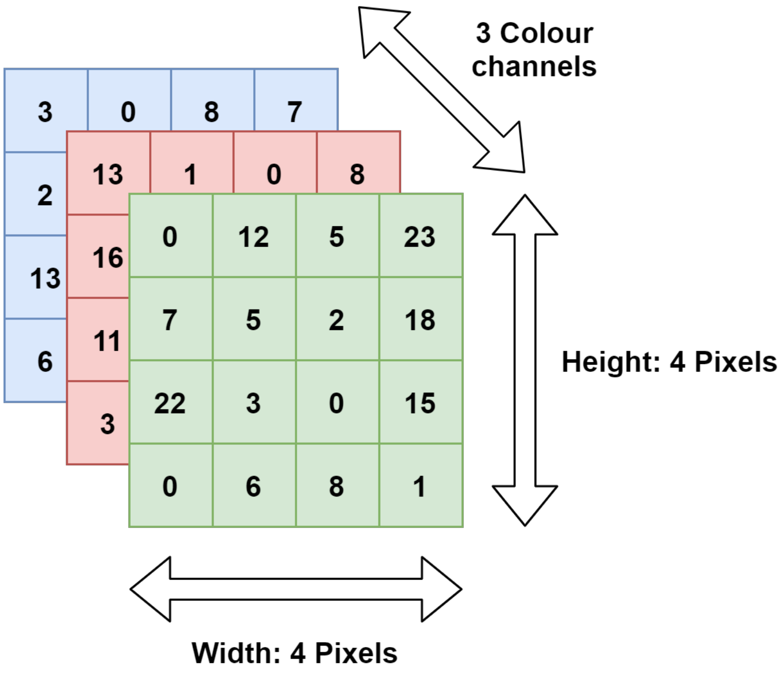


Figure 11.3: RGB Kernel Example.

The operation that performs the output feature map computation is described by the equation [25] derived from [26]:

$$y_0(j,i) = Bias_0 + \sum_{c_{in}=0}^{\#C_{in}-1} \sum_{k=0}^{W_y-1} \sum_{p=0}^{W_x-1} k_{0,c_{in}}(k,p) \cdot X_{0,c_{in}}(j \cdot stride + k, i \cdot stride + p) \tag{11.5}$$

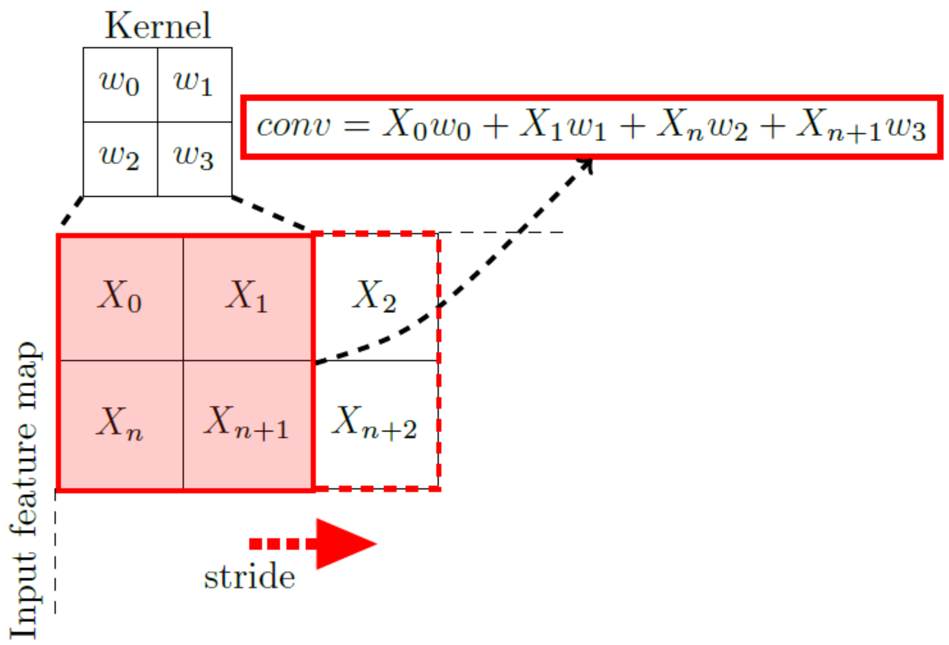


Figure 11.4: Convolution Process Example. Source:[25]

where i, j are the index of the output feature map corresponding pixel, c_{in} is the input channel index, $\#C_{in}$ are total input channels. W_x, W_y is the kernel's matrix size indicating the number of rows and columns respectively, 0 is referred to as the OFMAP considered. $k_{0,c_{in}}$ is the kernel weight and p, k are the kernel's indexes. The convolution process described by the equation 11.5 is shown in a graphical way in the Figure 11.4. The stride is the step of how we move the kernel in the input feature map. The Convolutional layer network doesn't connect each neuron of the previous layer with the next one. This has the advantage to reduce the number of parameters inside the model. The scope of the convolutional layers is to extract a high-level feature from images, like shapes. This feature extraction and reduction allows to have an easier and more efficient classification process.

11.3.2 Pooling Layers

The Pooling operations are different, the most used are: the max pooling and the average pooling [27]. The Pooling operation is similar to the Convolution operation, we have a window that shifts in the IFMAP with a certain stride. It usually uses a value of stride to not overlap the window in the different steps. Inside the window, we do the max operation between the values, in

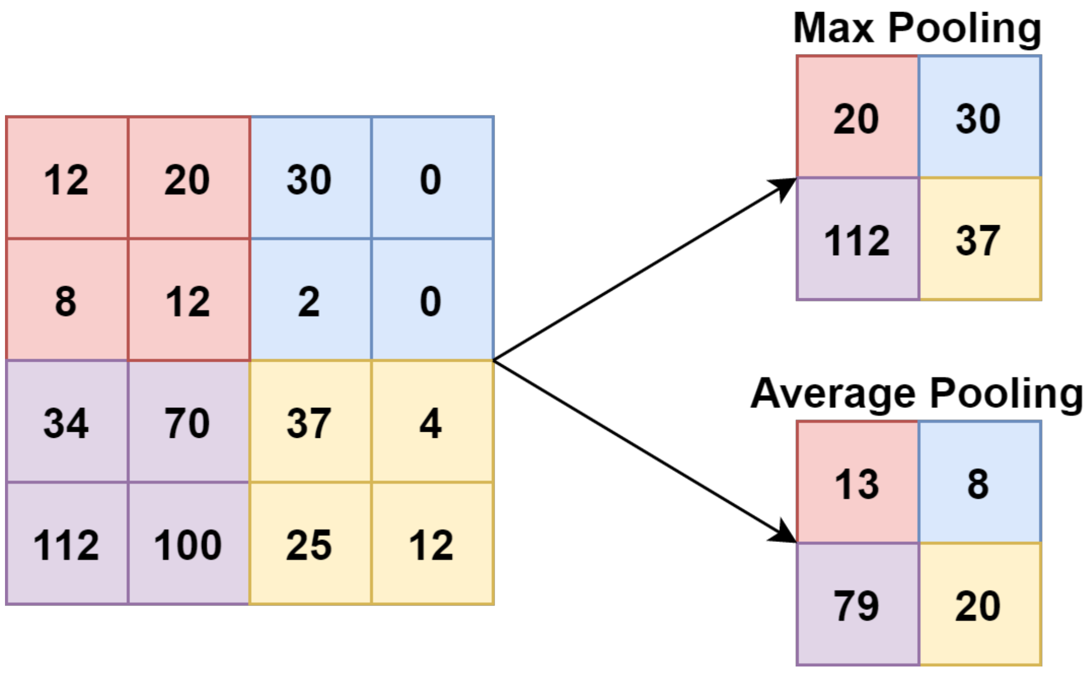


Figure 11.5: Pooling Operation Examples.

case of max pooling, or the average in the case of average pooling. In Figure 11.5 there is an example of these two types of pooling.

11.3.3 Fully Connected Layers

The fully connected layers are the most important part of our CNN. This part the responsible for the classification of the images after the feature extraction and reduction by the convolutional and pooling layers. As explained before, the neurons of this part are connected to all the neurons of the next layer, differently from the convolutional layers. In the output of the convolutional part, we have a matrix of data, that will be rearranged in a single vector to take in input the fully connected part; this operation is called **Flattening**. In Figure 11.6 we can see an example of CNN highlighting the three types of layers explained before.

11.3.4 Batch Normalization

A technique called **Batch Normalization** is used to reduce the problems coming from the training (such as slow convergence), in particular in very deep networks such as CNN. The technique is based on the normalization of

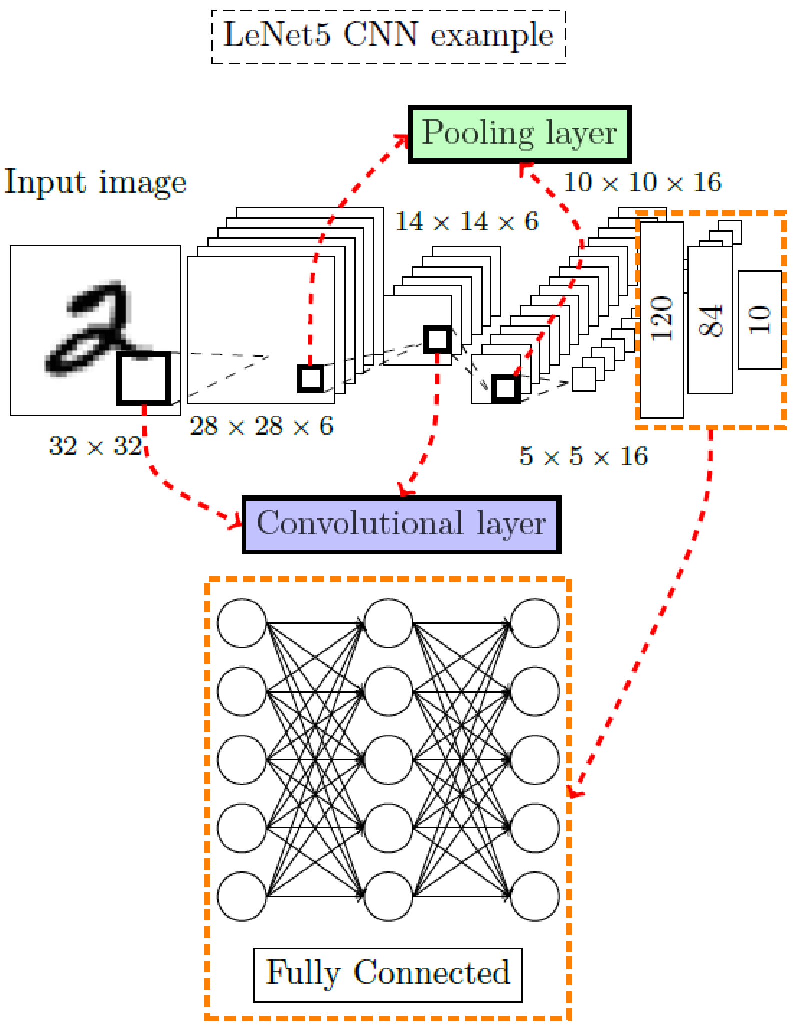


Figure 11.6: LeNet5 Netwok example. Source:[25]

the inputs of each layer, to shift the mean output activation of value 0 and a standard deviation of 1. This permits to have a faster training and more complex models without losing precision. This technique is used also for the implementation of the Binary Neural Network [25].

11.4 Binary Neural Network

The Binary Neural Network (BNN), is a CNN where we apply the binarization operation of the weights and the inputs. The operation is simply performed using the sign operation where we recall the relation:

$$sign(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \tag{11.6}$$

The binarization operation of, for example, the IFMAP is shown in Figure 11.7

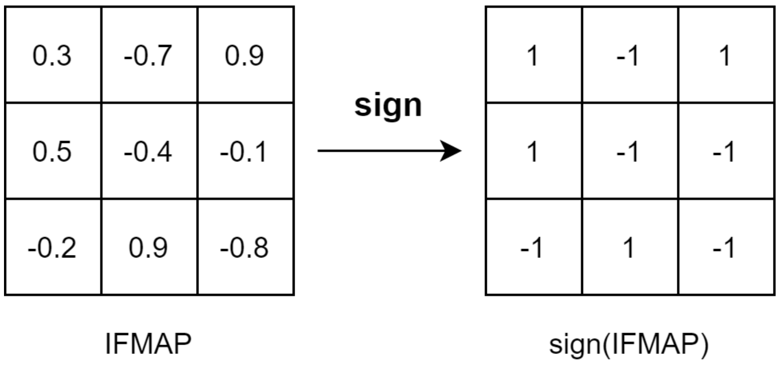


Figure 11.7: Binarization Process Example

From now we show how do the convolution operation is made, having the binarized weights of our network, as described in [28]. The first operation we do is to consider our binary values $\{-1,1\}$ into the couple $\{0,1\}$ because in the second case we can implement and elaborate using our more familiar binary notation with our logic circuits. The convolution is performed doing the XNOR logic operation and a pop-count operation. The XNOR operation is done using the truth table 11.1. The pop-count operation is simply the difference between the number of ones and the number of zeros in a word, like in the equation 11.7.

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

Table 11.1: XNOR truth table

Pop-count relation:

$$pop\text{-}count = \#1s - \#0s \tag{11.7}$$

We see an example in Figure 11.8 of the convolution operation using binary coefficients.

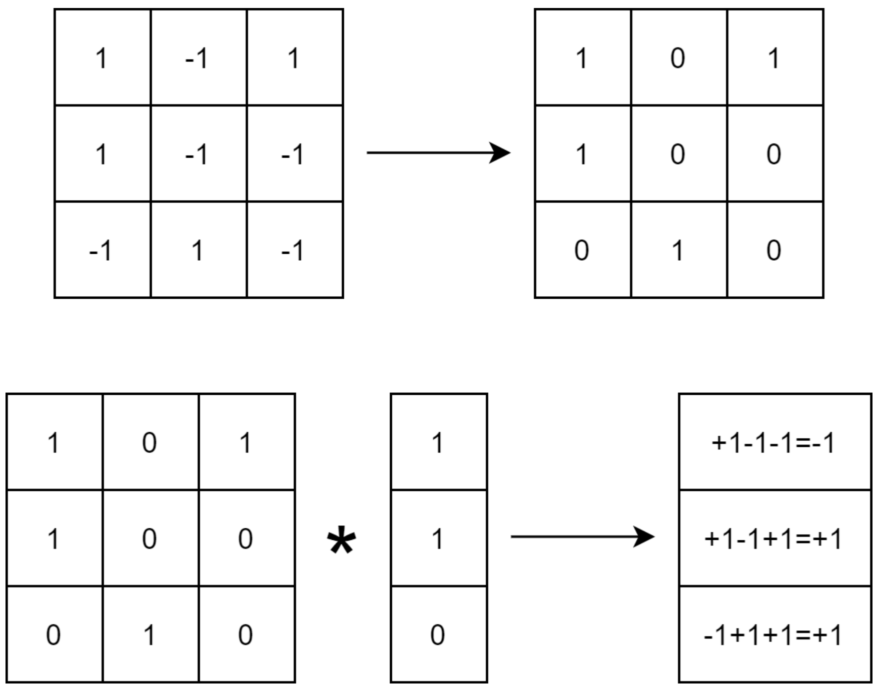


Figure 11.8: Binary Convolution Example

How we can see, we firstly do the XNOR operation and the pop count of the output like in equation 11.8.

$$\begin{cases} y(0) = \textit{pop-count}(\textit{xnor}(101{,}110)) = \textit{pop-count}(100) = -1 \\ y(1) = \textit{pop-count}(\textit{xnor}(100{,}110)) = \textit{pop-count}(101) = +1 \\ y(2) = \textit{pop-count}(\textit{xnor}(010{,}110)) = \textit{pop-count}(011) = +1 \end{cases} \tag{11.8}$$

Now that we have understood how the operation works, we can realize the simplest circuit to do this operation. As shown in Figure 11.9 there is the circuit used to make a simple convolution with a 2x2 kernel. The problem with this procedure is that the value after binarization is not equal to the original one, this introduce an error in the computation due to this approximation. To compensate the precision losses, we introduce an extra factor in our operation. The relation of our net is described [29]:

$$Conv_{out,XNOR\text{-}-Net} \approx (X_b \circledast w_b) \cdot \mathbf{K} \times \alpha \tag{11.9}$$

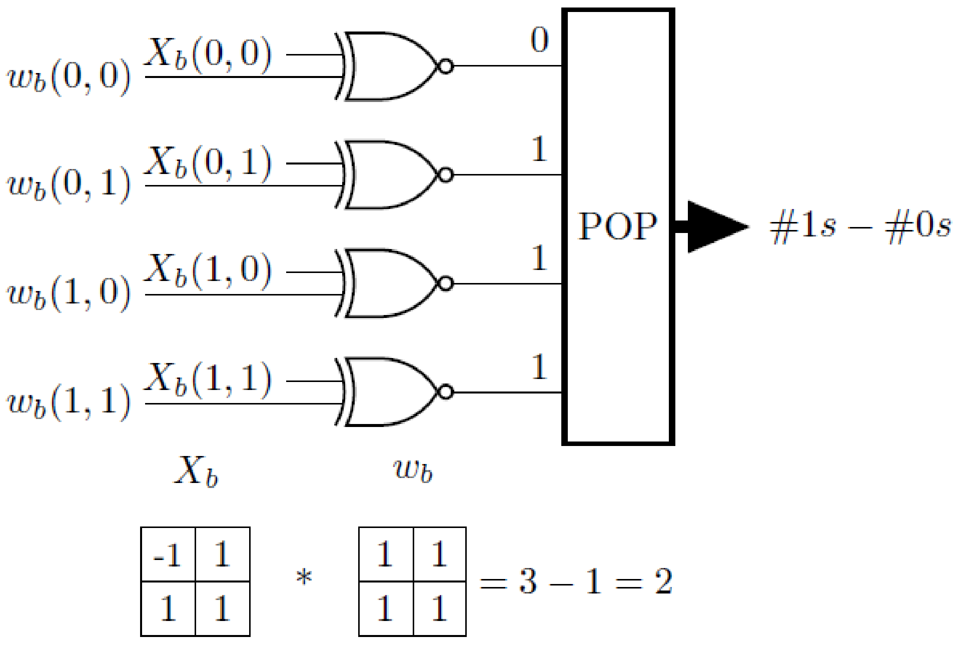


Figure 11.9: Binary Convolution Based on XNOR-Pop. Source:[25]

where X_b and w_b represent the input and the weight binarized, the $\circledast$ operator represents the binary convolution,"$\cdot$" the punctual operation,$\times$ the standard multiplication, the $\mathbf{K}$ and α are our correction factors [29].
The α factor is defined as:

$$\alpha = \frac{\sum_{i=0}^{N} \|w_i\|}{N} \tag{11.10}$$

where w_i is the weight considered at full precision and N the number of weights. The $\mathbf{K}$ factor is a matrix defined by the relation:

$$\mathbf{K} = \frac{\sum_{c_{in}=0}^{\#C_{in}-1} \left|X(:,:,c_{in})\right|}{\#C_{in}} * \begin{bmatrix} \frac{1}{W_xW_y} & \frac{1}{W_xW_y} & \cdots \\ \frac{1}{W_xW_y} & \frac{1}{W_xW_y} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{11.11}$$

The first term represents the absolute punctual sum of the multiple input feature maps divided by the number of the input channels. The second term is a matrix of size $W_x \times W_y$ which contains the same factor in all positions.

11.5 Hardware Implementation

The goal of implementing this algorithm in a LiM structure is the advantage to have a high-level of parallelism. The other advantage is that the transfer between processor and memory is reduced to the minimum, this leads us to do almost the computation inside the memory. The first operation we need to perform is the Xnor operation like described in equation 11.9. The idea is to put inside each memory cell an Xnor gate. Doing so, we can do the binary product between the content of the cell and an external input. The resultant architecture is shown in Figure 11.11. In this example, we are doing the convolution operation between a kernel of dimension 2x2 and a 4x4 IFMAP with a stride value of 1. As we can see, we use the multiplexer to switch the incoming bit that will be sent to the bit pop count operation. The convolution operation of the highlighted IFMAP portion is done through the equation:

$$Incoming\ bit_0 = pop\text{-}count(\overline{X_0 \oplus w_0}, \overline{X_1 \oplus w_1}, \overline{X_4 \oplus w_2}, \overline{X_5 \oplus w_3}) \quad (11.12)$$

To simplify the pop-count operation and reduce the complexity of the cell we can rearrange the operation in this way:

$$pop\text{-}count = \#1s - \#0s = 2\#1s - length(word) \quad (11.13)$$

where length(word) is the size of the array entering in the pop-counter.

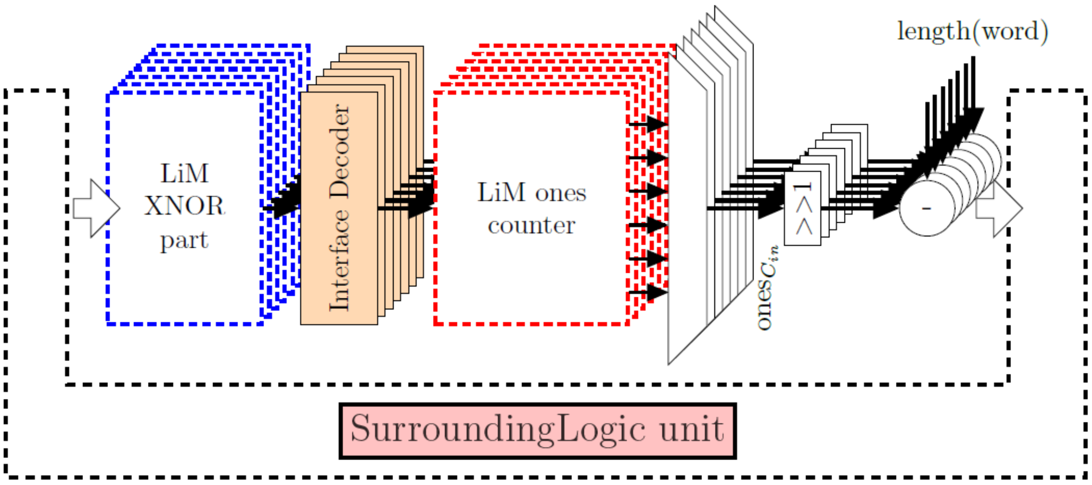


Figure 11.10: Lim complete Architecture. Source:[25]

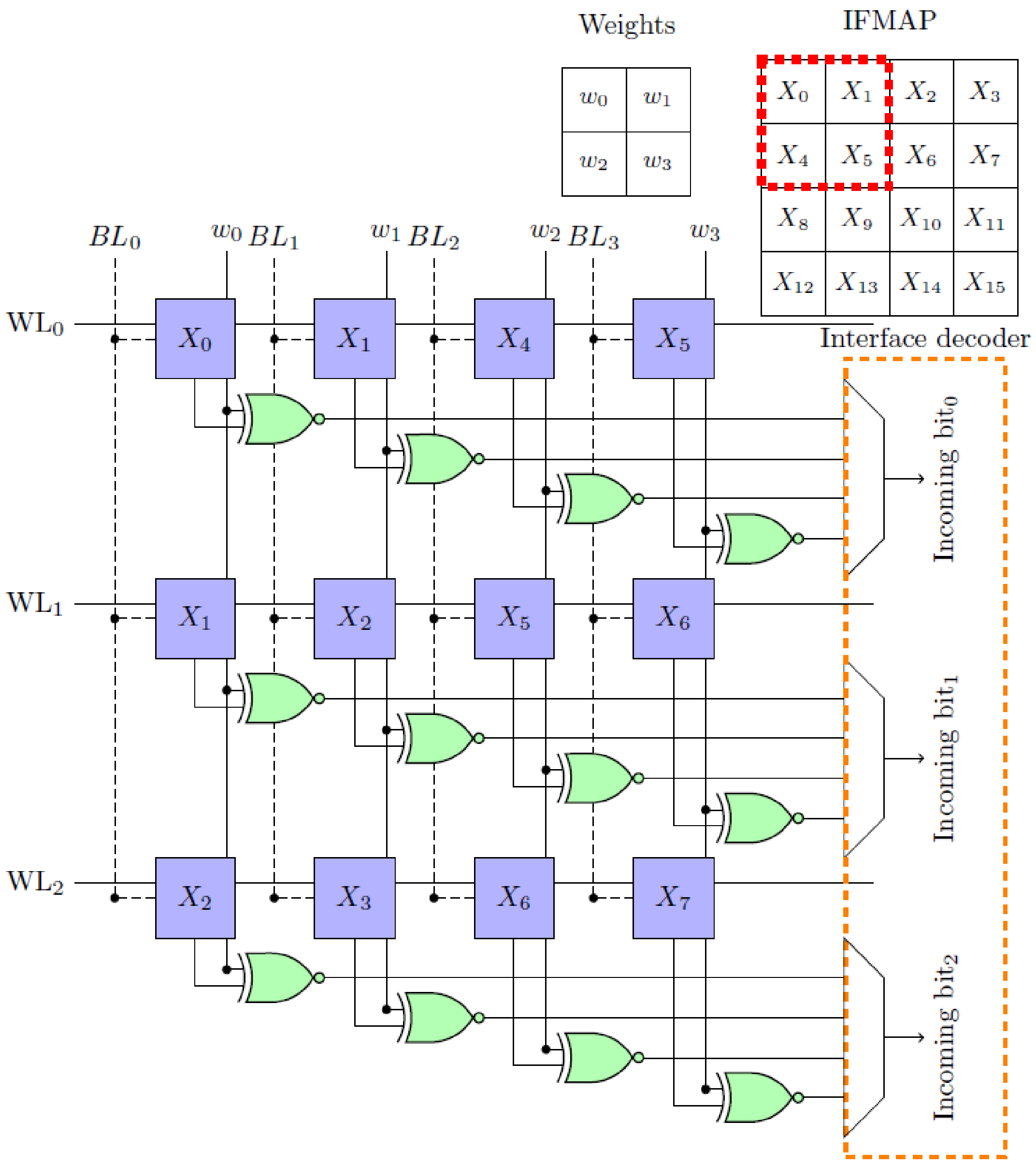


Figure 11.11: XNOR part of the XNOR-Pop Unit LiM implementation. Source:[25]

The one counter circuit is realized using Half Adders as shown in Figure 11.12. Connecting these two units we can get the complete architecture. If we have the number of output feature maps higher than one, we can simply duplicate this structure and do the operations in parallel. The whole architecture is shown in Figure 11.10, we can see all the blocks analyzed before and we

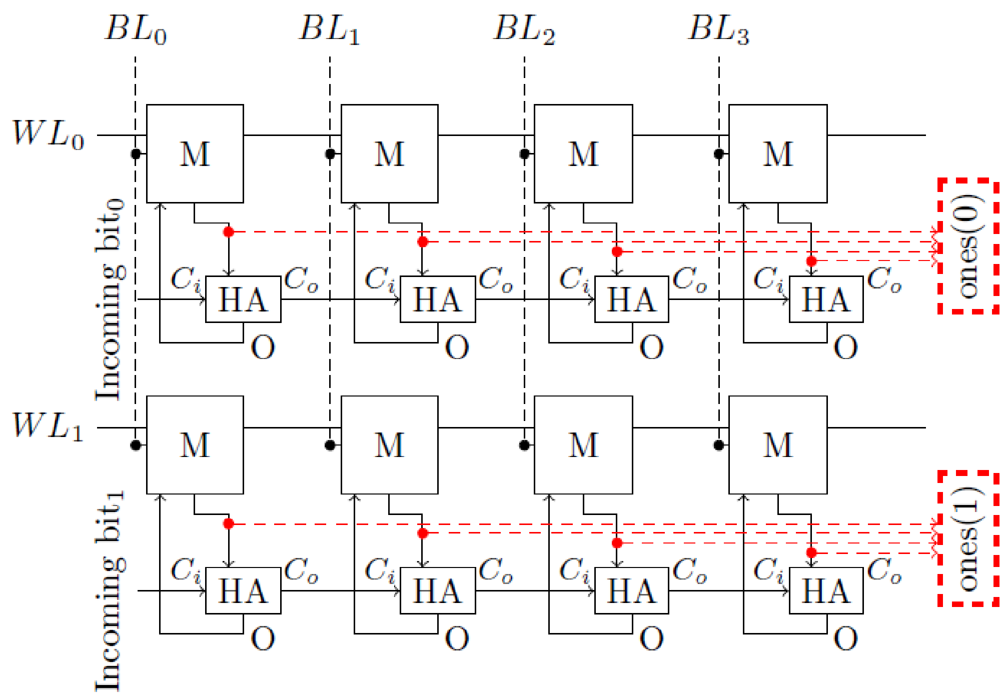


Figure 11.12: One counter Lim circuit. Source:[25]

have also a part called the surrounding logic unit. The surrounding logic unit contains all the circuits used to compute the corrective factors α and **K** that is in the equation 11.9 and other circuits used to compute the batch normalization. For more info about this part, everything is explained in detail in [25]. It is possible to use the same architecture to compute also the fully connected part inverting the role of the weights and the inputs [25]. The dimension of our memory depends on the parameters used for the convolution operation and also for the fully connected part. In fact, the dimension is defined by which part of the two requires more resources, if we consider only the Xor convolutional part:

$$\begin{cases} \text{Memory size}_x = D_{out}^2 \\ \text{Memory size}_y = W_x \times W_y \end{cases} \tag{11.14}$$

where D_{out} is the dimension of the output feature map obtained by the equation:

$$D_{out} = \frac{D_{in} - W_x}{stride} + 1 \tag{11.15}$$

this equation considers only the W_x dimension because the kernel is usually regular and has the same dimension of the y direction W_y. Moreover, that these dimensions are equal to the dimension of the memory of the one counter part.

11.6 DExIMA Implementation

First of all, what we need to know is which algorithm we implement using this architecture. The model chosen to implement is a neural network able to recognize images of the Fashion-MNIST dataset which contain clothes images on a grey scale. The CNN is shown in Figure 11.13.

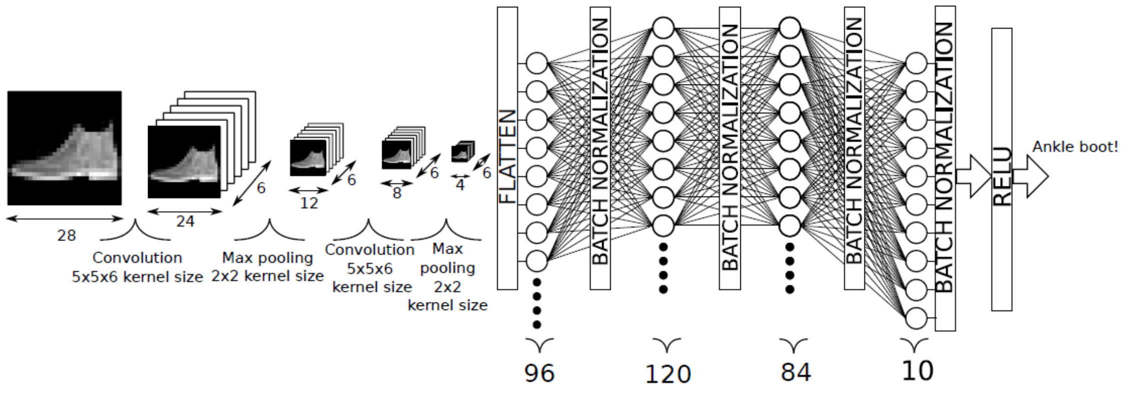


Figure 11.13: CNN used for the Fashion-MNIST dataset. Source:[30]

In the beginning, the intention was to implement the entire algorithm, but for some reasons that were previously explained only the first convolution operation is implemented. In this section, we will see how the architecture is implemented using the first version of DExIMA that we call DExIMA 1.0, and after we will see the same architecture implemented with the new version of the software that we call DExIMA 2.0. In the end, we make a comparison between the two versions and the architecture created in VHDL and we also do a synthesis using Synopsys Design Compiler. The VHDL codes are a courtesy of Andrea Coluccio that implements them for his Master Thesis [30].

11.6.1 DExIMA 1.0 Implementation

The reason why we make only the convolution operation is dependent on the fact that the first version of DExIMA had a big bottleneck, the speed. To do the first six convolution operations, the tool takes at least 45 minutes, and the speed trend has an exponential behavior. In particular, by adding one output feature map, the tool increases the computed delay of a factor of 3 and this does not permit the simulation of the whole algorithm. We start computing the dimension of the memory that will be able to do the first convolution operation. The input feature map is composed of an image of 28x28 pixels and uses a kernel of a size of 5x5 and a stride of 1. This means that the output feature map dimension will be:

$$D_{out} = \frac{D_{in} - W_x}{stride} + 1 = \frac{28 - 5}{1} + 1 = 24 \tag{11.16}$$

this reflects what it is shown in Figure 11.13, the total dimension of memory is:

$$\begin{cases} \text{Memory size}_x = D_{out}^2 = 576 \\ \text{Memory size}_y = W_x \times W_y = 25 \end{cases} \tag{11.17}$$

The first problem occurred while writing the code because we need a high number of components to create the whole architecture, but we did not have an instrument to create multiple instances in DExIMA, and we cannot parameterize the code using the constants. This needs an external script that creates the code that will be compiled by DExIMA 1.0. A Python code that automatically creates the architecture is written, and all the files are needed for the tool because the first version of the program tries to write four different files for different parts of the architecture. The input information needed for the Python script is shown in 11.1 and consists of the kernel size, the output feature map size, and the number of output feature maps.

```
# Memory variables
kernel = 5 # kernel dimension (square case)
of_map = 24  # OFMAP dimension (square case)
memory_parallelism = 64  # memory output parallelism
C_out = 6  # output number of feature maps

#Memory dimensions in x and y dimension
size_x = (of_map ** 2) * C_out
size_y = kernel ** 2
```

Listing 11.1: Header of CNN Python Script constructor

Another problem was the fact that it was not possible to create multiple memories but just one. To resolve this problem, we simply adapt the whole architecture to work only with one memory, this is shown in Figure 11.14.

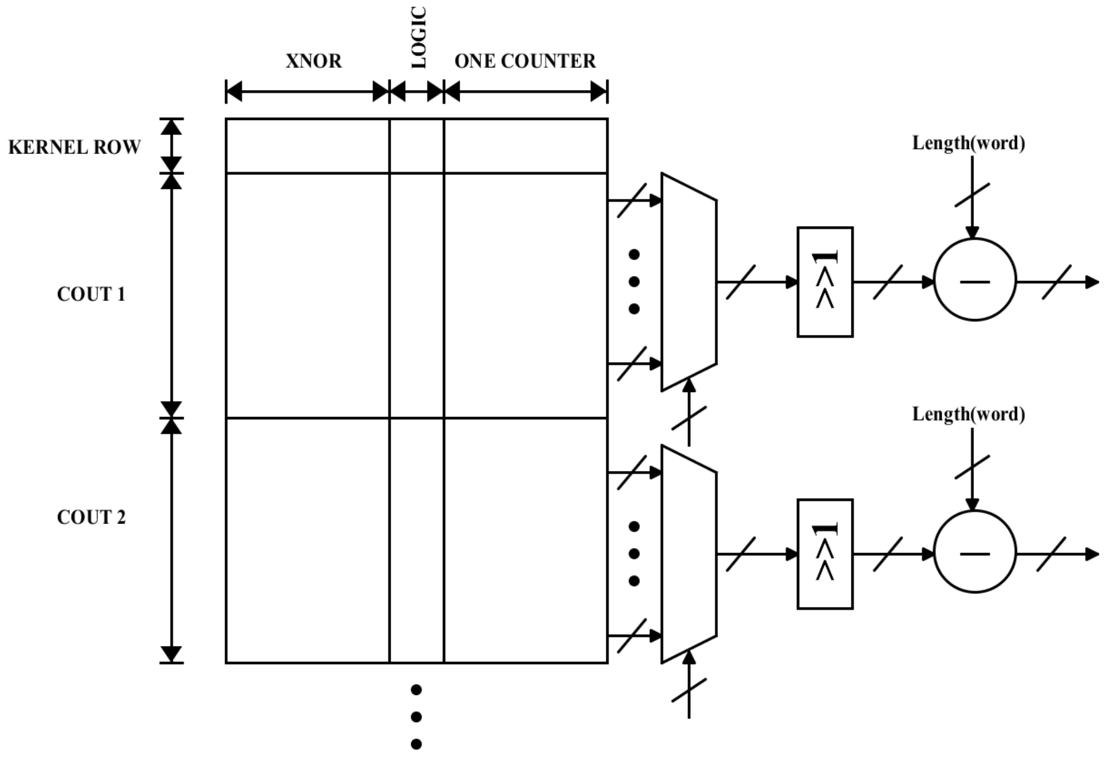


Figure 11.14: DExIMA 1.0 XNOR-Net Architecture

In this way, it is possible to avoid the problem of the parallelism growing the memory in vertical for each output feature map. The advantage of the previous version of DExIMA, that was possible to build a memory with the wanted dimension because the tool does not create directly the memory interface circuity. This process is demanding to CACTI a memory simulator designed by HP (Hewlett-Packard) [31]. With the adapted memory, in vertical we have the different output feature maps and we can use a single row to store the kernel weights. Each output feature map goes to the output multiplexer and the result is shifted by one to multiply it by two and subtract with the word length. In the horizontal dimension, we can recognize the Xnor part and the One counter part. Between them there is the logic needed to interface the two blocks, this logic isn't inside the cells so it does not waste memory cells.

11.6.2 DExIMA 2.0 Implementation

Using the new version of DExIMA we do not need an external script to write the code, but it can be done directly from hand. Different from the first version of DExIMA it is not possible to create a memory to the wanted dimension, but we need to have a dimension able to create the memory interface circuitry.

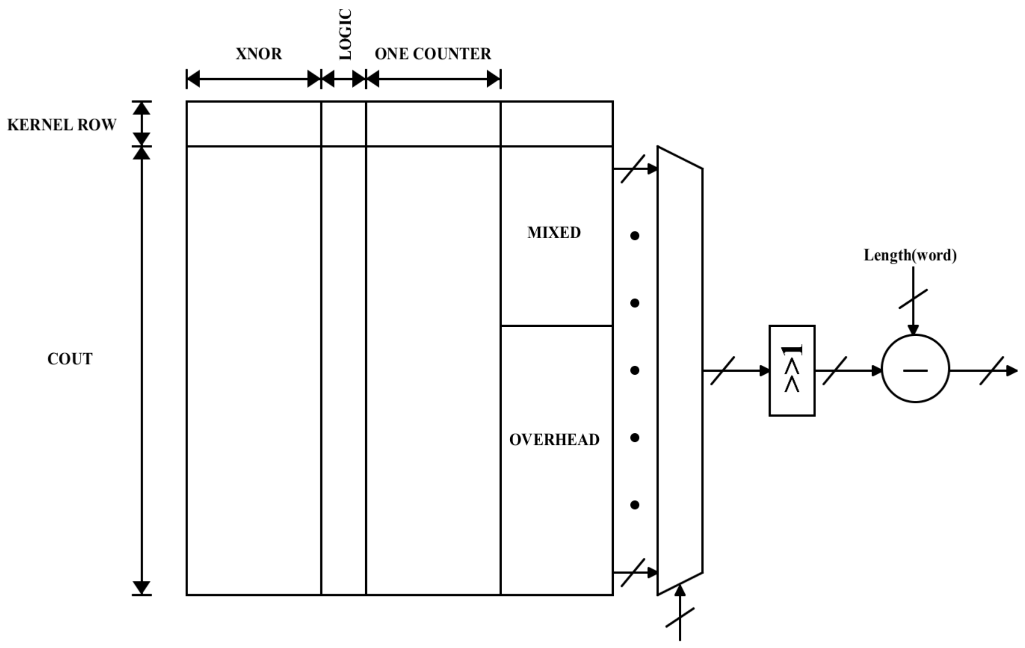


Figure 11.15: DExIMA 2.0 XNOR-Net Architecture

The dimensions should be the same computed before, but the dimension of the effective memory nearest to the wanted is obtained by constructing a memory of the size of 512x64. However, in this way we have more cells than needed, we can recognize in the results that the area and the static power have this overhead due to the implementation. The Architecture used is shown in Figure 11.15, the architecture is very similar to the previous one, but in this case, we have only 512 rows of the 576 needed, so the memory continues the XNOR and One counter part in the right part of the memory. Because in this case, it is larger on the x axis compared to the previous, we can use this part of memory. The upper part is filled with the remaining components, this is called MIXED because it contains the Xnor and the One counter part. The last part of the memory is called OVERHEAD, and it is not used due to the dimensions constrain. The needed cells for the algorithm are (not considering the kernel row) $Cell\_needed = 576 \cdot 50 = 28800$ but the memory created has $Cell\_created = 512 \cdot 64 = 32768$, this means that we have an excess of 3968 cells, which corresponds about to 13,8% more cells. This must be taken into account where the results are presented. In this version, we have the advantage of creating multiple memories that work in parallel so we don't need to change the memory shape varying the number of output feature maps and we can simply duplicate the memory doing copy and paste in the dex file.

11.7 Results and Comparisons

11.7.1 Simulation Time

The first parameter we analyze is the bottleneck of the previous version, the Simulation time. First of all, we show the simulation times obtained by DExIMA 2.0 for the convolution algorithm, the results are shown in Figure 11.16. The Figure shows how the simulation time increases by adding the output feature maps. There are two examples of how important the dex file organization is, in fact the performance got in output is equal, but with two different computational times. This is because if we have duplication

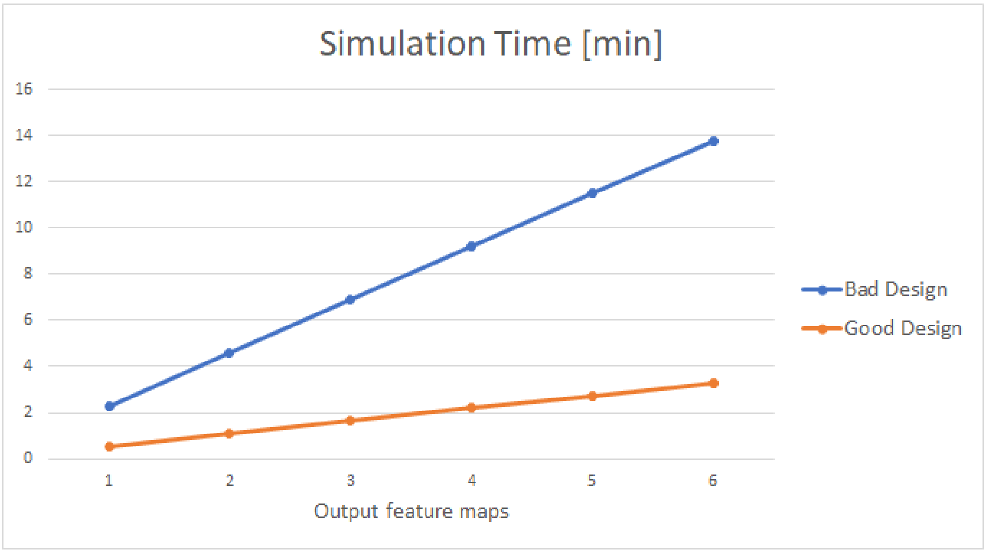


Figure 11.16: Comparison of the convolution algorithm using a Bad or Good design in DExIMA 2.0

of the same circuits it is unuseful to connect the duplicate ones and insert

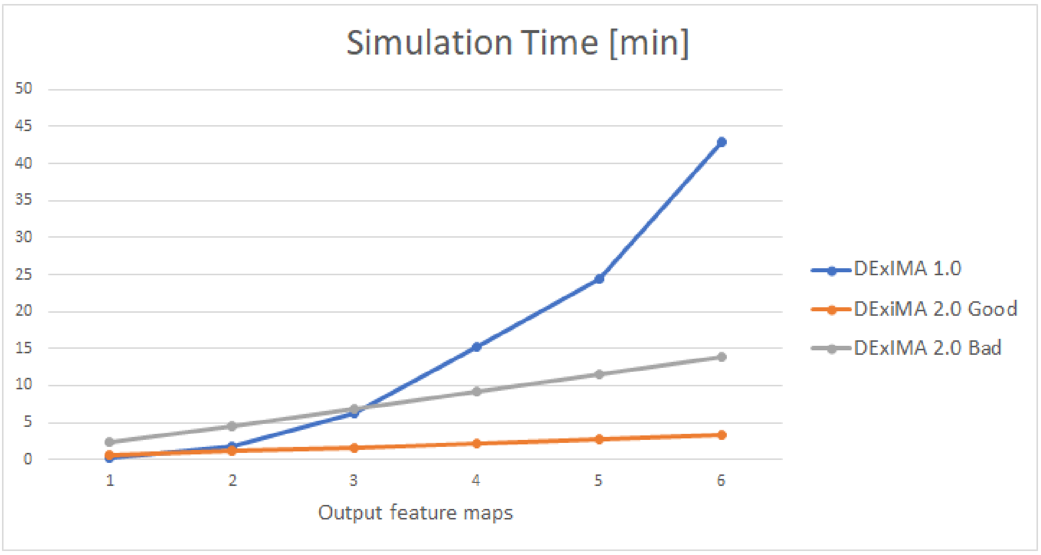


Figure 11.17: Comparison of the convolution algorithm between DExIMA 1.0 and DExIMA 2.0

them in the instruction part, we can use only one copy in the mapping part and instructions. Remind that the map section is very expensive in terms of computation time because we need to extract a lot of information. Now we can see the comparison between the computational time of the previous version of DExIMA and the new one. The results are shown in Figure 11.17. The previous version of DExIMA shows an exponential trend that increases very fast, but for small architectures, we have advantages only in the case of Bad Design. Since the new compiler is more simple than the previous one, this implies fewer operations to be checked. But the advantages of the new version are significant, we have linear behavior comparing to the exponential one. In the good design, we have much relevant performance improvement, with 6 output feature maps, we have about 45 minutes for the previous version and about 5 minutes for the new one. The last information about the simulation time is related to the difference between the compilation time and simulation time. The previous times are the sum of these two parameters. As we can see in Figure 11.18 they differ in orders of magnitude. The scale for the compilation is expressed in minutes and the scale for the simulation in milliseconds. This shows that the bottleneck of the new version is the Compiler because they need to do several operations to parse and check the dex file. However, the simulation process is very fast due to the algorithm explained in the previous chapters.

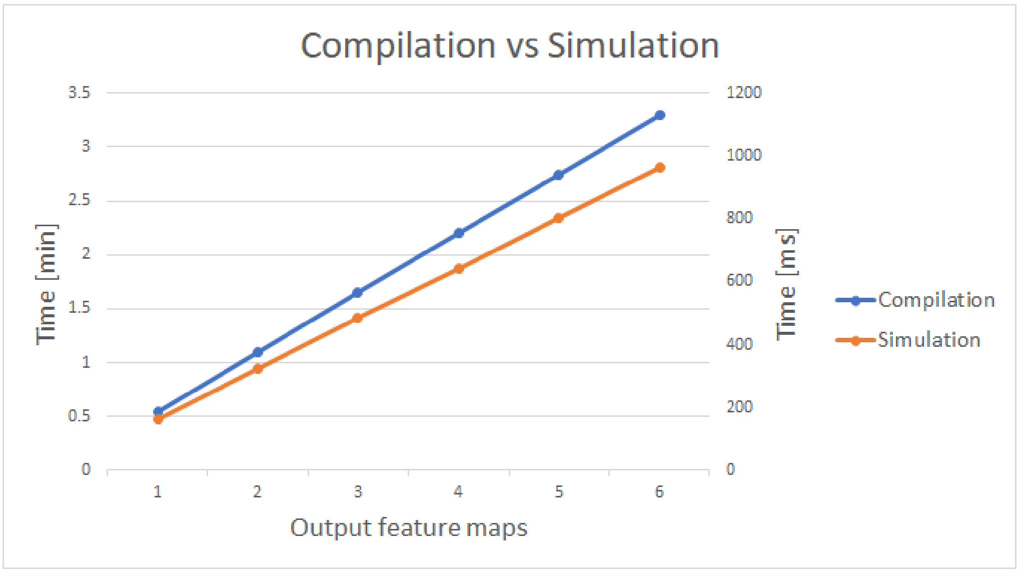


Figure 11.18: Comparison between the Compilation and Simulation time of the convolution algorithm

11.7.2 Writing Code Effort

Another important parameter is the effort when we write the code in DExIMA. As we said before, to create n components in the previous version of DExIMA we need to write n lines of code, in the new version we need only one line with a for loop construct.

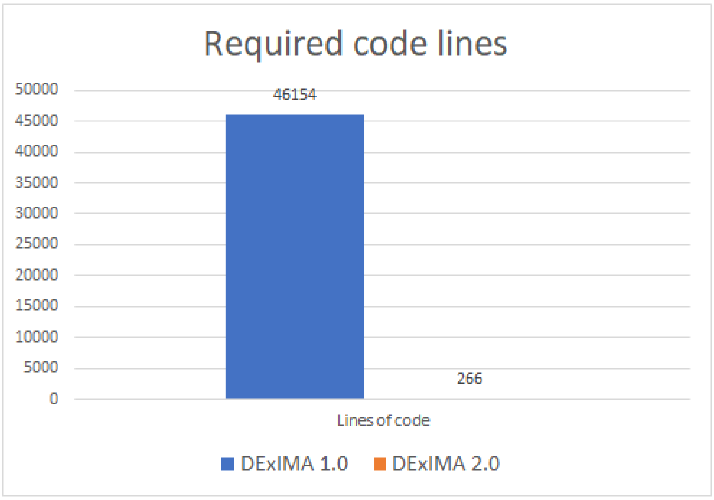


Figure 11.19: Line of code needed to describe convolution algorithm and architecture comparison

The Figure 11.19 shows an histogram with the two values of code lines. The difference is evident, we use only 266 lines of code versus 46154 lines. This shows how different the two compilers are.

11.7.3 Critical Path

Starting from now, we compare also the performance using the synthesized circuit by Synopsys. First of all, we see the values of the critical paths got from DExIMA in the two versions and the value obtained with Synopsys, the results are shown in Figure 11.20. We can see that the critical path for the new DExIMA and Synopsys is very similar instead of the value for the previous version of DExIMA, which is almost double the expected. The responsibility of this path lies in the part of the circuit that uses the ripple carry adder, which that is the most limiting component inside the architecture.

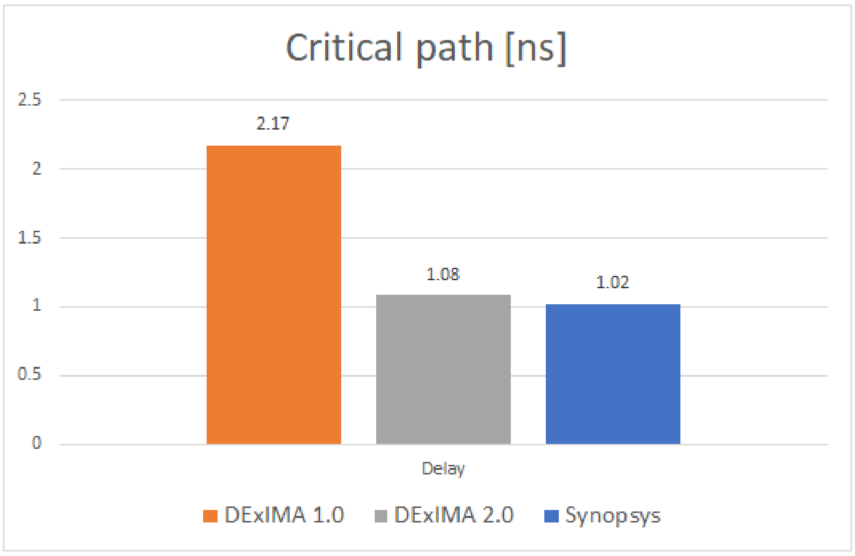


Figure 11.20: BNN Critical path comparison

11.7.4 Dynamic Power

Now we show the results in terms of dynamic power, which are shown in Figure 11.21. Also in this case the values obtained by the new version of DExIMA and Synopsys are very similar, the result obtained by DExIMA differs only for some mW. On the contrary, the value obtained from the first version is lower than expected. The value of dynamic power obtained is

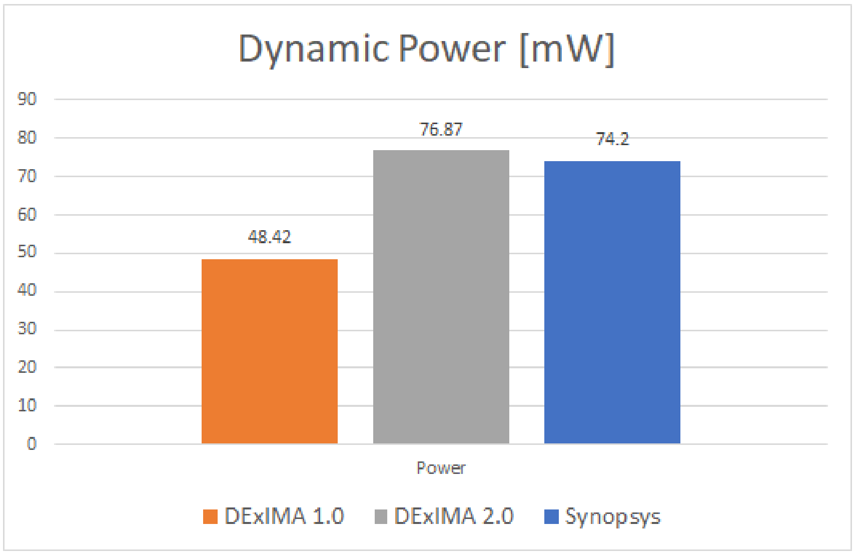


Figure 11.21: BNN Dynamic Power comparison

computed using the execution time that is function of the clock period. The clock period used for the second version of DExIMA and Synopsys is set at the value of 2.2 ns because it is the nearest to the value of the critical path obtained from the first version of DExIMA. By doing this, we have a fair

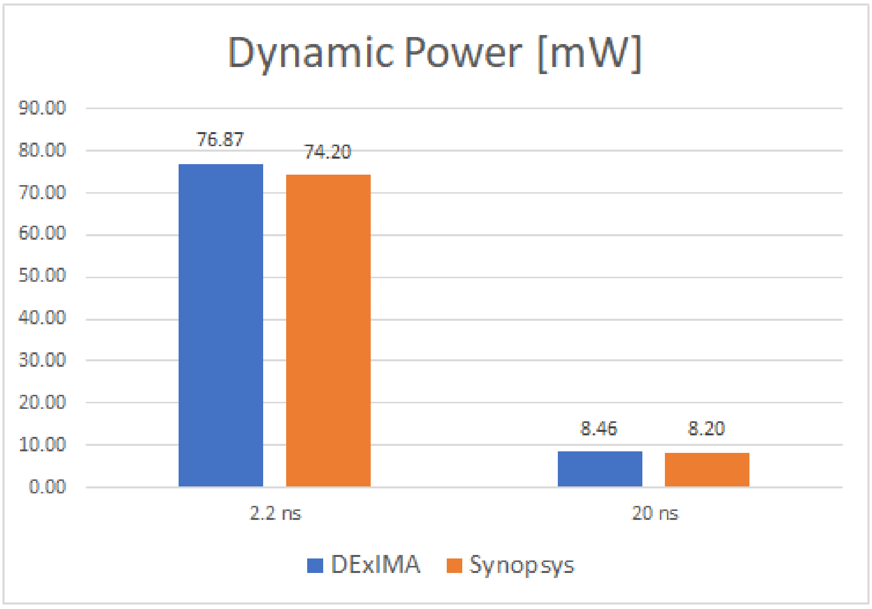


Figure 11.22: BNN Dynamic Power comparison using different clock period

comparison, because the clock period of DExIMA 1.0 is set to the critical path of 2.17 ns. For this reason, we can do only one comparison with the first version of DExIMA, but we can do with the new version because we can change the value of the clock period independently from the critical path. The Figure 11.22 shows two simulations varying the clock period. In the first simulation, we set a clock of 2.2 ns and the second a value of 20 ns and we can observe that the results are in line with the Synopsys results. All the computation considers the switching activity of the components and an input probability of 0.5. In the last Figure the 11.23, there are the results in the two clock periods of how the dynamic power varies if we do not consider the switching activity of the gates.

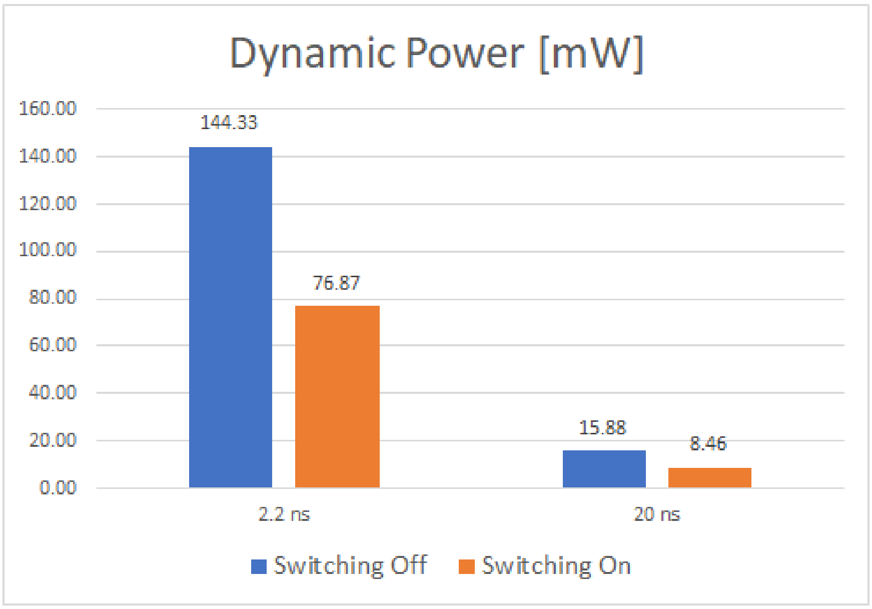


Figure 11.23: BNN Dynamic Power comparison enabling or disabling the switching activity computation

11.7.5 Area

In the area, we firstly show the absolute values and we specify all the sub-parameters. The Figure 11.24 shows the results, and we can notice that DExIMA 2.0 has a higher area compared to the two others, as expected. The cells in the new version of DExIMA have an additional component, the multiplexer. This increases the area occupied by the memory and the circuit synthetized by Synopsys uses a structure more similar to a register file than to real memory, this means that the circuitry of the interface is smaller than

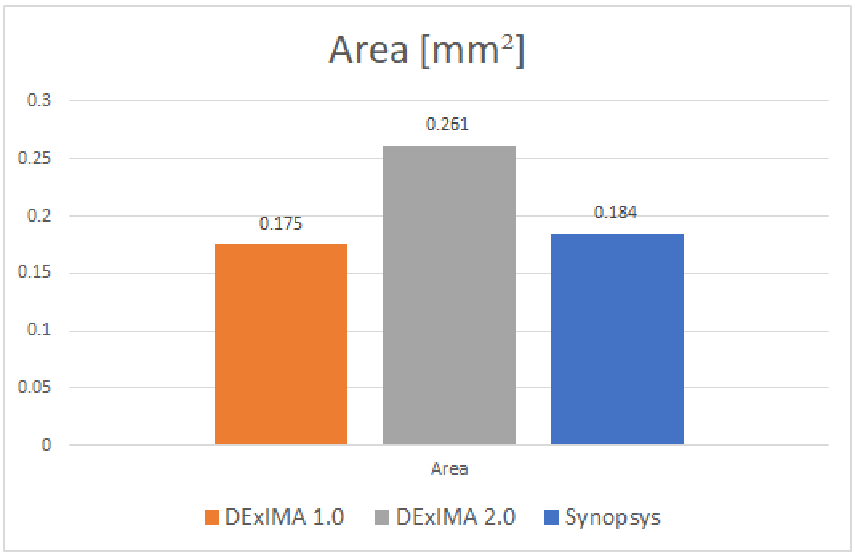


Figure 11.24: BNN Area comparison

the one implemented with DExIMA 2.0. Knowing each component of the

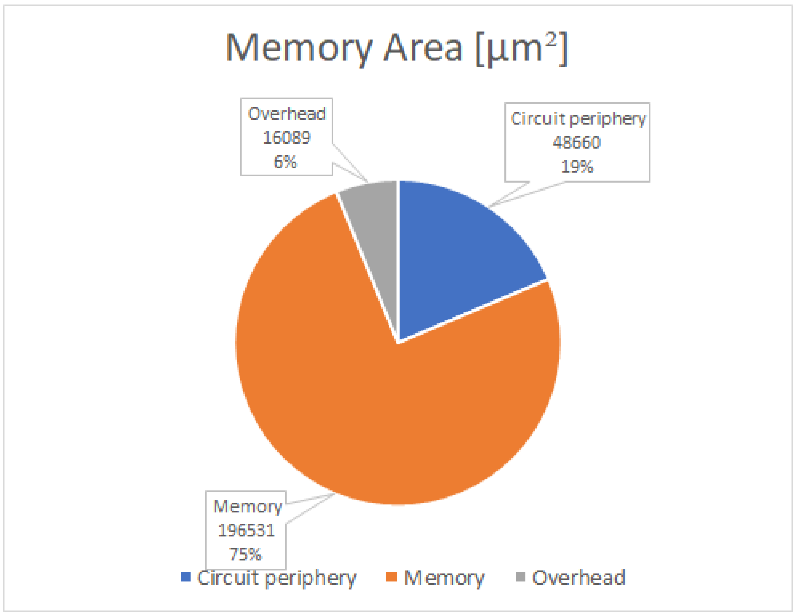


Figure 11.25: BNN Area components

memory we plot a pie chart. Figure 11.25 shows also the overhead part of the memory. There is also the part related to the circuitry used inside the memory, but it is not possible to isolate the circuitry overhead related to the additional cells. The Memory area is the sum of the area of the components inside the cells and the out of the cells components. In Figure 11.26 there is

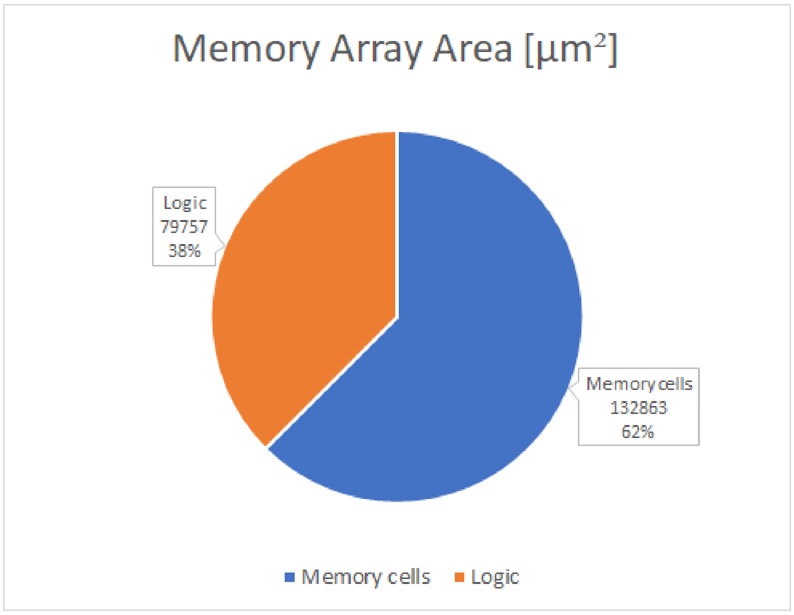


Figure 11.26: BNN Array Area Components

a more detailed division of the memory area, which shows the memory area due only to the cells, and the other piece is all the logic inserted inside the cells, and inside the memory. This graph includes also the value of overhead.

11.7.6 Static Power

Some of the considerations done with the area can be repeated for the static power. We start, as before, with the absolute results, that are shown in Figure 11.27. This result shows how what was obtained by DExIMA 1.0 is out of scale in comparison to what was obtained by Synopsys. The result of DExIMA 2.0 on the contrary has only three times more than the Synopsys result, but in this total, we need to dived in all its subparts. The result of DExIMA 2.0 is the sum of interface static power and the memory cells static power (without the logic and overhead). Finally, we can see in the Figures 11.29 and 11.28 the division of the static power components is organized in the same way it was organized for the area.

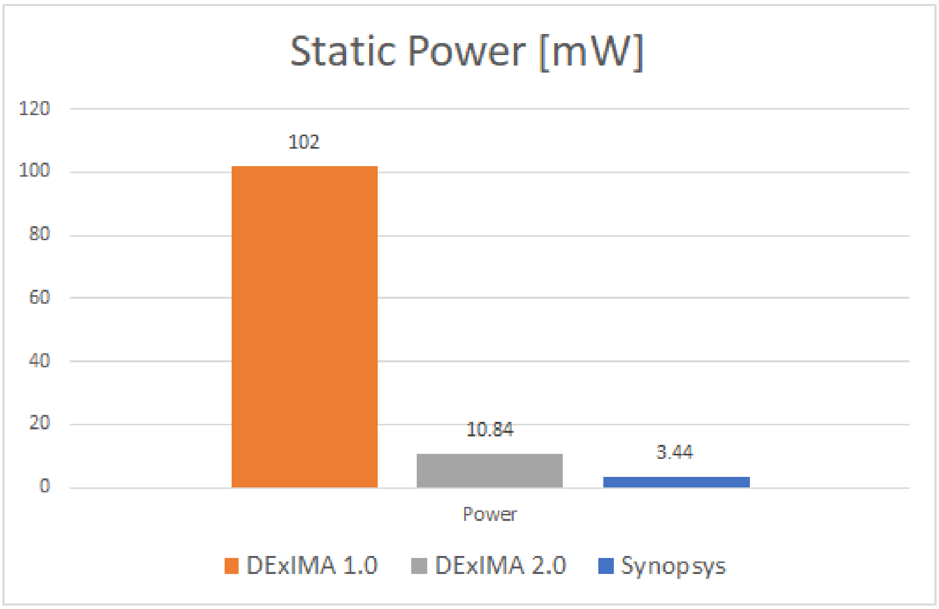


Figure 11.27: BNN Static Power Comparison

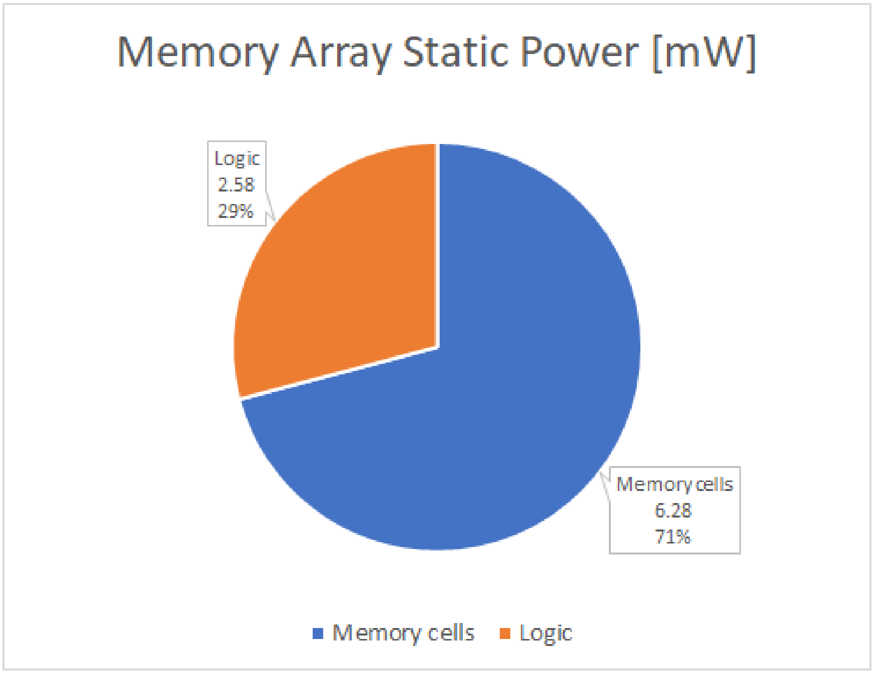


Figure 11.28: Memory Array Static Power Components

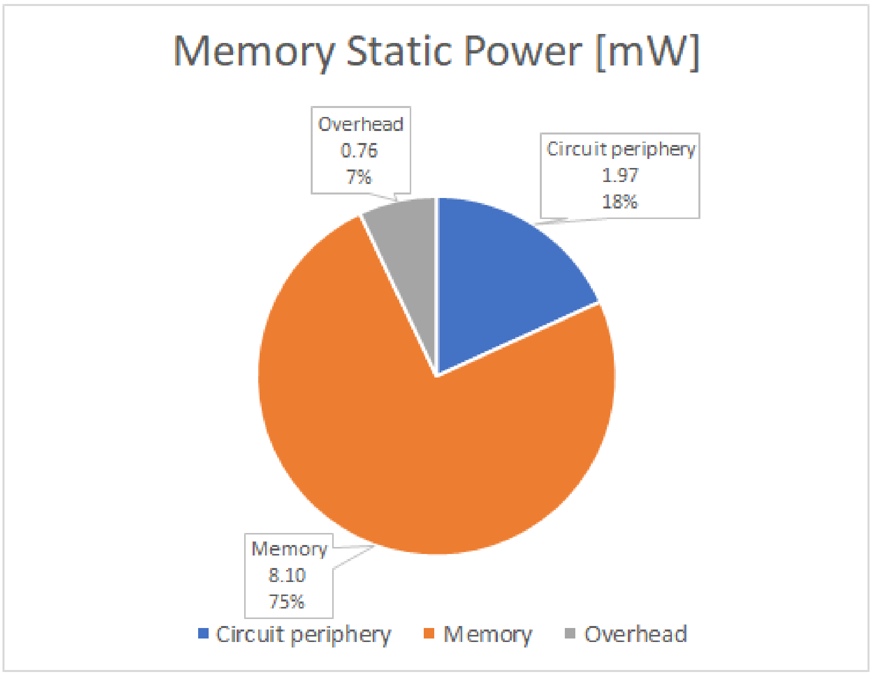


Figure 11.29: Memory Static Power Components

11.8 Cacti Comparison

In the last part of this chapter, we want to make some comparisons with the Result obtained using Cacti. The memory that we consider is with the same dimensions of the memory used previously for the BNN. The memory has 512 rows and 64 columns, with one port for reading and one for write, and uses the same configuration used in DExIMA 1.0. This can be useful to understand the contribution of the previous result of only the Cacti part.

11.8.1 Area

The area results are shown in Figure 11.30, and it is evident how the area in DExIMA is higher, this is due to the different memory cells definition. Cacti have an optimized 6T Static Ram cell for memory. On the contrary, DExIMA uses the Flip Flop and the cell having also more components for the correct work of the architecture. The ratio between the area of DExIMA and Cacti is about 6.5. If we do raft computation in terms of how many transistors does DExIMA has inside the cells versus the 6T cell we obtain that the DExIMA cell has 50 transistors inside it. The ratio about the number of transistors is about 8.33, so the transistor size is different inside the cell, but in this way we have an idea of the reason why the area got by DExIMA is much higher than the one got by Cacti.

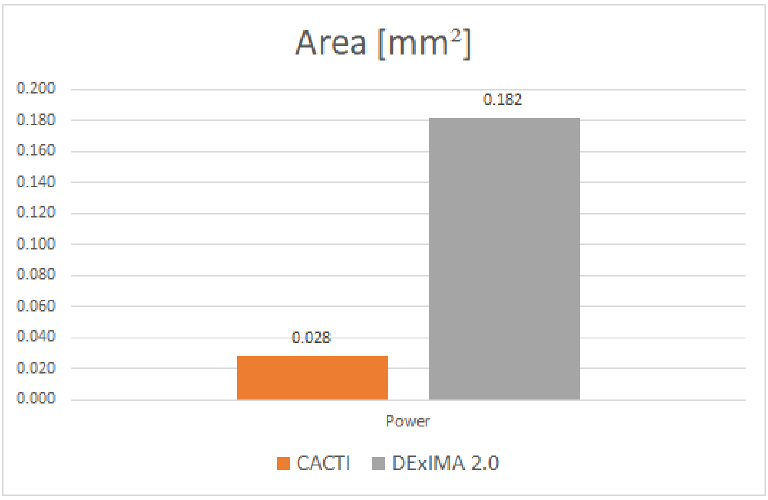


Figure 11.30: Cacti Area Results

11.8.2 Static Power

The static power also in this case is higher for DExIMA due to the different memory organizations.

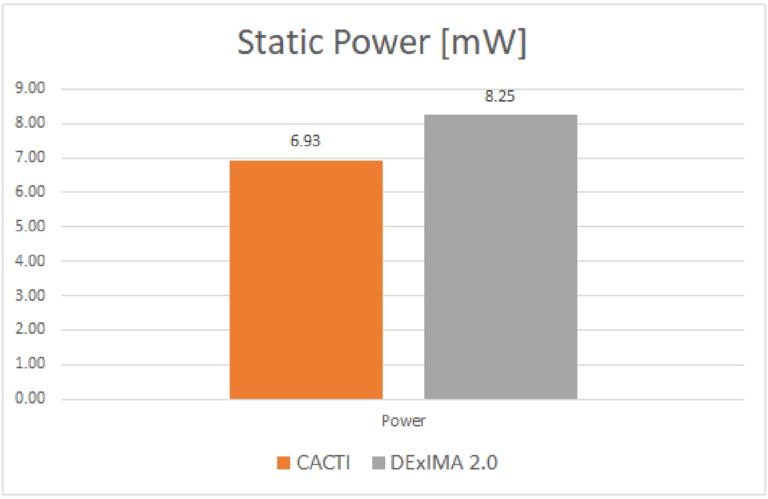


Figure 11.31: Cacti Static Power Results

11.8.3 Memory Access

The last characteristic given by Cacti is the Access Time of the memory, in Figure 11.32 there are the values given by Cacti and the values given by DExIMA. As we can see for DExIMA, both the Write and the Read times are inserted. The Read Time is longer than the Write time because the bottleneck for the reading is the output multiplexer that is big and slow. The Write process, instead, is faster because uses only adaptive drivers that are optimized for speed.

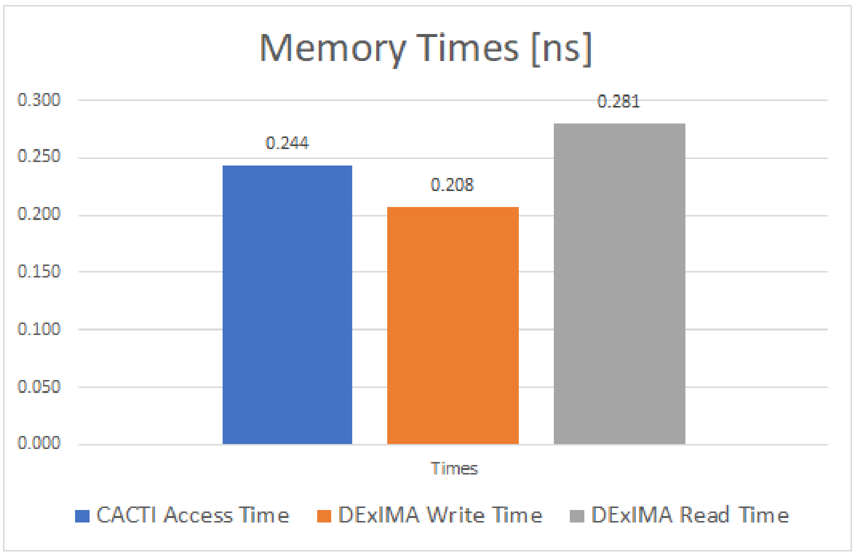


Figure 11.32: Cacti Memory Access Time Results

Chapter 12

DExIMA Results

In this chapter, we will see the general results and some comparisons between the features of the program and its previous version. Most of the main features are shown in the results of the BNN of the previous chapter but here we explore a more general perspective.

12.1 Previous Version Comparison

One important thing that we want to underline is that the previous version of DExIMA (1.0) and the new version (2.0) are completely different. The only common parts are related to the language and the Module class, which is very similar to the CompiledModule class of the previous version, all the other parts differ. The complexity of the program is different, as well. Figure 12.1 shows the difference measured in lines of code of the C++ program. The new version of DExIMA has 123% lines more than the previous version. Another important aspect that is very weak in the previous version is the number of handle errors of the compiler. In fact, in the previous, version it often happens that the compiler doesn't report the error, but the program starts. If we are lucky, we have an error in runtime like the segmentation fault, and we indicate that something went wrong. Otherwise, the simulation ends successfully but some parameters are wrong. Sometimes it happened in version 1.0 that we found a stochastic behavior each time we did the simulation obtaining wrong parameters. Figure 12.2 shows the difference in the number of manageable errors in the two versions, in fact, the previous version can to recognize only 7 errors against the 108 errors of the new one.

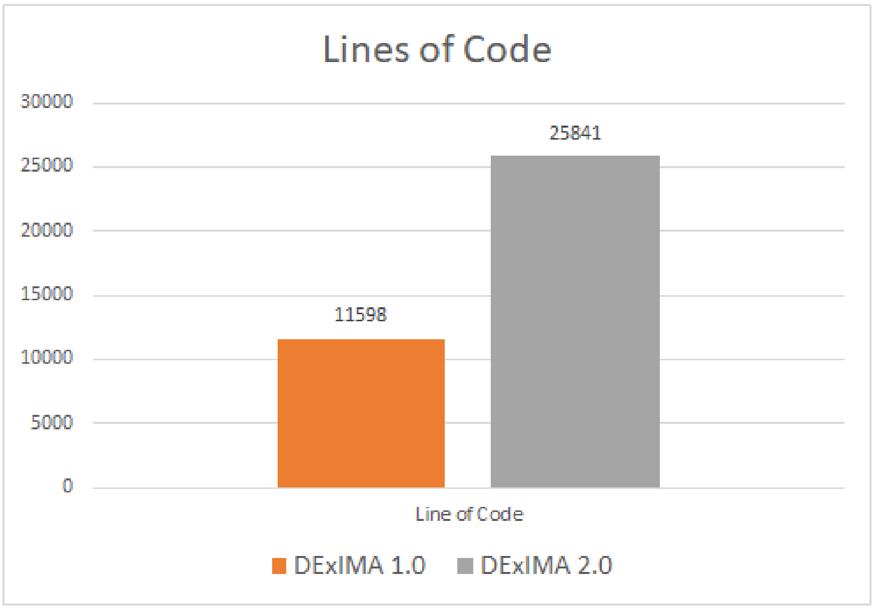


Figure 12.1: DExIMA versions Lines of code

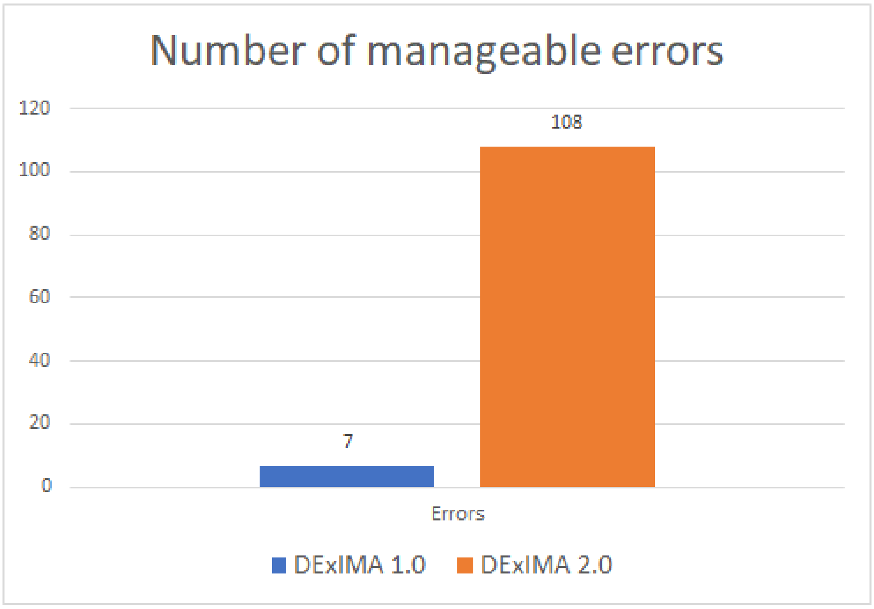
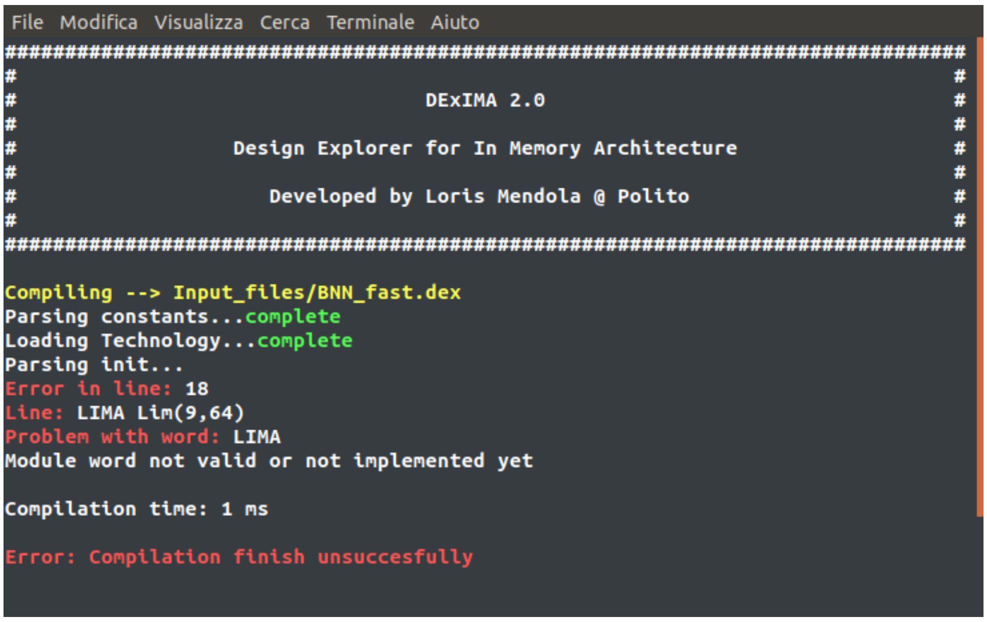


Figure 12.2: Number of Manageable Errors

12.2 Output program interface

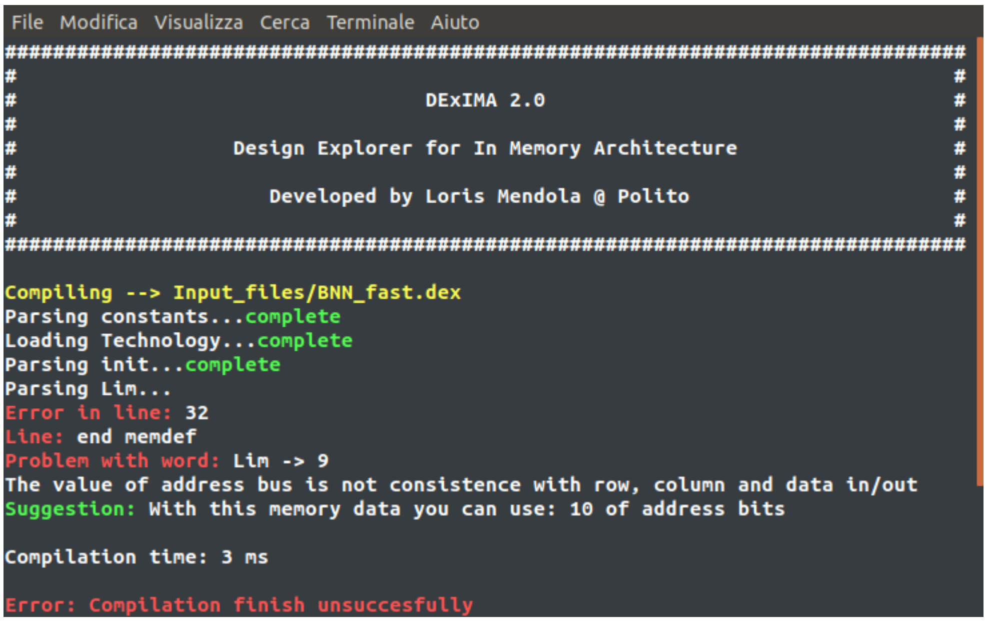
In this section, we first show how the errors are displayed, and after how the output of the program is when it is running. As we can see in Figure 12.3,



```
################################################################################
#                                                                              #
#                                   DExIMA 2.0                                 #
#                                                                              #
#                   Design Explorer for In Memory Architecture                 #
#                                                                              #
#                      Developed by Loris Mendola @ Polito                     #
#                                                                              #
################################################################################

Compiling --> Input_files/BNN_fast.dex
Parsing constants...complete
Loading Technology...complete
Parsing init...
Error in line: 18
Line: LIMA Lim(9,64)
Problem with word: LIMA
Module word not valid or not implemented yet

Compilation time: 1 ms

Error: Compilation finish unsuccesfully
```

Figure 12.3: Simple Error Generated by DExIMA

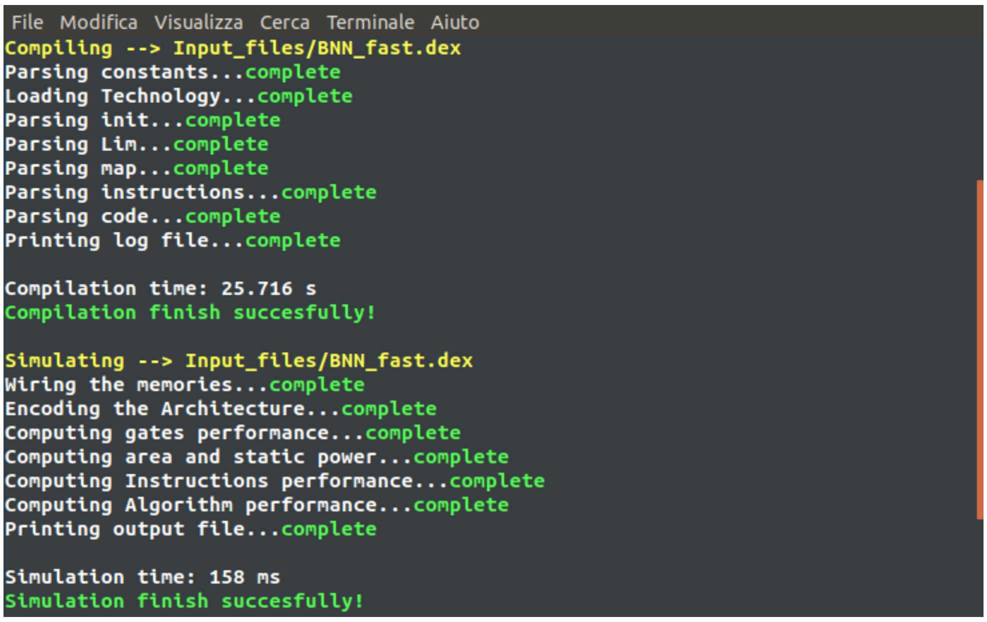
it is shown how a simple error is displayed.
As it is possible to see, the first info is the line in the dex file where the error occurs and after, the line of the error and the word of the parameters



```
################################################################################
#                                                                              #
#                                   DExIMA 2.0                                 #
#                                                                              #
#                   Design Explorer for In Memory Architecture                 #
#                                                                              #
#                      Developed by Loris Mendola @ Polito                     #
#                                                                              #
################################################################################

Compiling --> Input_files/BNN_fast.dex
Parsing constants...complete
Loading Technology...complete
Parsing init...complete
Parsing Lim...
Error in line: 32
Line: end memdef
Problem with word: Lim -> 9
The value of address bus is not consistence with row, column and data in/out
Suggestion: With this memory data you can use: 10 of address bits

Compilation time: 3 ms

Error: Compilation finish unsuccesfully
```

Figure 12.4: Error with Suggestion Generated by DExIMA

responsible for the error are displayed, at the end there is the error message. In some cases also another field called Suggestion is available, there is also a specified suggestion on how to resolve the occurred error. This is shown in Figure 12.4. The last screenshot that we analyze is where it is shown all the information displayed if no errors occur.



```
Compiling --> Input_files/BNN_fast.dex
Parsing constants...complete
Loading Technology...complete
Parsing init...complete
Parsing Lim...complete
Parsing map...complete
Parsing instructions...complete
Parsing code...complete
Printing log file...complete

Compilation time: 25.716 s
Compilation finish succesfully!

Simulating --> Input_files/BNN_fast.dex
Wiring the memories...complete
Encoding the Architecture...complete
Computing gates performance...complete
Computing area and static power...complete
Computing Instructions performance...complete
Computing Algorithm performance...complete
Printing output file...complete

Simulation time: 158 ms
Simulation finish succesfully!
```

Figure 12.5: Output Generated by DExIMA

In Figure 12.5 we see in the first line the name of dex File that we are compiling, each line after, shows the section that the program is parsed. When is complete, the word *compete* is displayed. As we can see after the parsed constants section, we load the Technology like explained in the previous chapters, at the end we print the log file. In the part related to the Simulation it is possible to recognize the steps explained in the simulation chapter. The first part computes the connections inside the memories to add the equivalent fanout. The other steps are well known from the simulation chapter. The output file is the dof file in the output. At the end of each section, the computation time spent is displayed. The compilation and successive simulation are done in series for all the dex files contained inside the folder specified in the input line of the program.

12.3 Models Comparison

To make a fair comparison, we stored the performance of the models implemented in DExIMA to compare with the performance of the Technology Library used by Synopsys. The Technology used is the FreePDK45nm [32, 33], which is a free library (more correct is a Process Design Kit PDK) that is possible to consult. The FreePDK45nm is developed by North Carolina State University and it is also possible to consult some information in [34]. First of all, we read the liberty files and other files related to the technology to extract the parameters of area, static power, delay, and dynamic energy. For the area and static power, it was simply because the value is directly written. For the delay and dynamic energy, we find that we have a different delay in dependency of the input, and have different values for the rising and falling time. When we incur in this situation we have simply done the average between the rising and falling time. Every model uses different load capacitance to compute the performance, but the lowest value of capacitance is in common for all the models of the library, it is $C_L = 0.365616\ fF$. Another important parameter used in the technology file is the input transition time, in DExIMA this parameter is omitted, it is like having a null transition time in input, for this reason, we consider the lowest value of transition time. Doing all these considerations we need to highlights another aspect: the two technologies are similar but not identical, so we have some variations in the parameters used (like I_{on}, I_{off}, etc...), and the dimension of transistor used for the models also this case is slithery different for some models. First of all, we showed the majority of gates together, and after we focus on the Flip Flop because it has more parameters to analyze, especially on the delays part.

12.3.1 Area Comparison

Figure 12.6 shows a histogram with almost all models exploited in DExIMA that are available in the FreePDK45nm technology. In general, we can see that the DExIMA models occupy less area, this is expected because the models in the library are "complete". This means that they have also a specific physical layout for each cell and use also a standard cell layout. In DExIMA this aspect is not directly exploited, the area of each model is simply the area of transistor plus a 15% overhead to the interconnects, so the occupation is always lower. But when we create a memory this is an advantage because it can be denser with respect to a standard cell layout. It is possible to insert also overhead to take into account a standard cell layout.

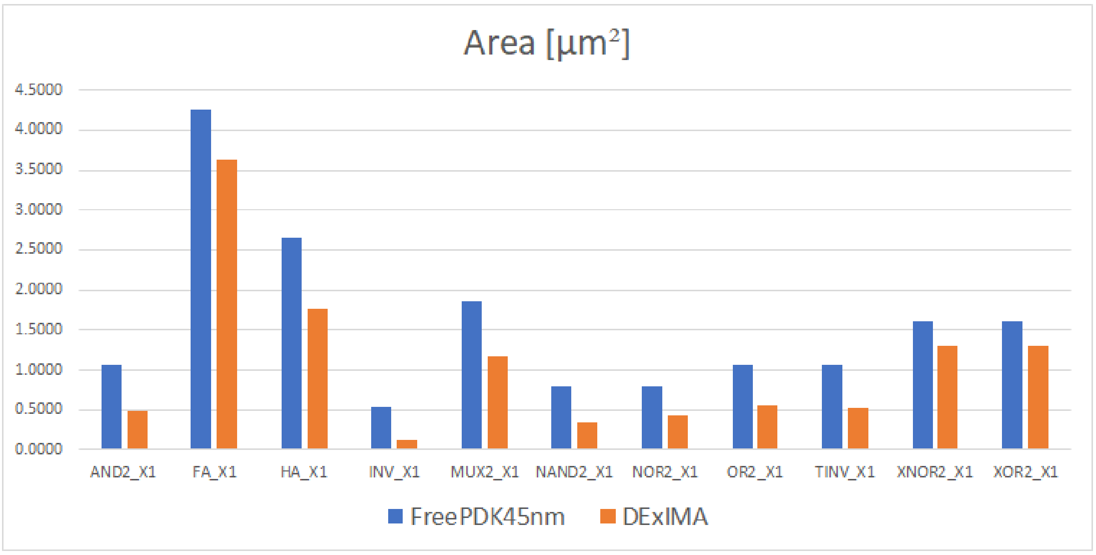


Figure 12.6: DExIMA Models Area Comparison with FreePDK45nm

This can be done by modifying a parameter in the TechFile that is normally set to zero. If we want, for example, an overhead of 50% we simply set this parameter to 0.5, it is possible to see this information in the dof file in the Technology part.
As disclosed before some models have different transistor dimensions of the gate. One of the models that are easy to recognize the difference compared to DExIMA, is the Inverter INV_X1 where X1 stays from the drive strength of the gate. As explained in the Models chapter the inverter has a dimension of 1 (we refer only to the n-mos) and the Nand has a dimension of 2. In the FreePDK45nm, we can observe that the inverter n-mos has the dimension compared with the n-mos of the Nand, this means that from the DExIMA point of view the inverter is like having drive strength X2 because it is double in comparison to the standard inverter in DExIMA. The same consideration can be done for the Three-state inverter but in the case of DExIMA, it is realized using C2MOS configuration. From the base models, the gate area is about 50% less than the FreePDK45nm.

12.3.2 Dynamic Energy Comparison

The dynamic energy is computed inserting for each gate the load specified before $C_L = 0.365616\ fF$, to have the same reference load. This can be done in DExIMA using the LOAD model and inserting the value of capacitance

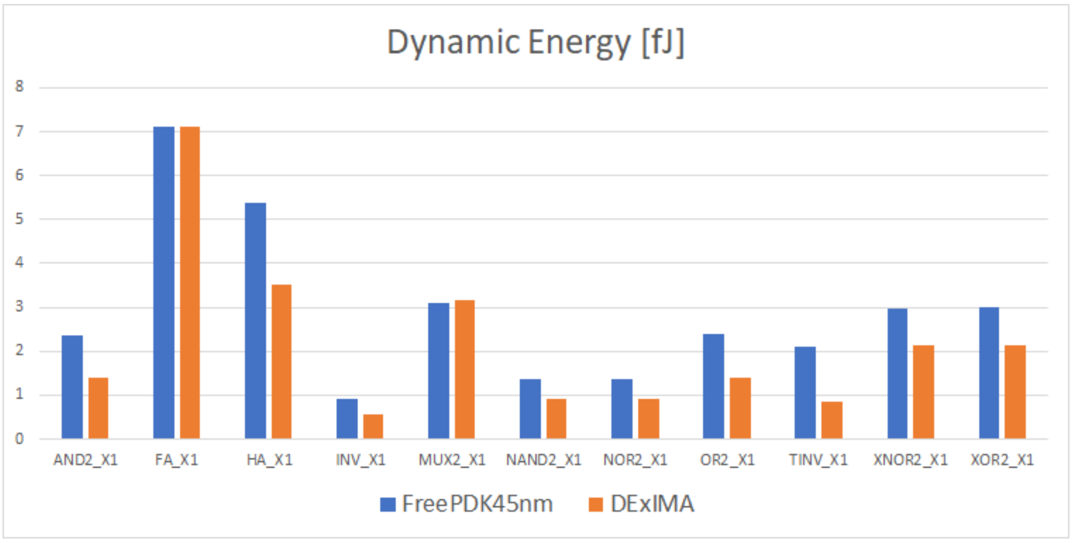


Figure 12.7: DExIMA Models Dynamic Energy Comparison with FreePDK45nm

(remember that the input expected value is in pF). In general, the values of dynamic energy are expressed in $\mu W/GHz$, but the results in DExIMA are directly in J so we use directly the value in fJ. In general, these values are very good considering the difference between the two models. Another parameter that in DExIMA is not computed is the energy dissipated by the internal nodes of the gate, this is why some models dissipate less energy than the FreePDK45nm values.

12.3.3 Static Power Comparison

Also, in this case, the data are good like shown in Figure 12.8, because the value is very similar except to the values obtained by the Xor/Xnor gates that for DExIMA are more efficient models. If we imagine the standard composite model of the Xor using three Nand and Two Inverter, we obtain a value of static power of:

$$Xor_{composite} = 3 \cdot Nand + 2 \cdot Not = 3 \cdot 17.23 + 2 \cdot 7.18 = 66.05\ nW \quad (12.1)$$

This exceeds the value of FreePDK45nm that is of 36.16 nW, it is about the 50%. This having also a perfect matching of the value of static power of Nand gate, it shows how the specific implementation is important (width of transistors, topology, etc...), and doing a perfect comparison is impossible.

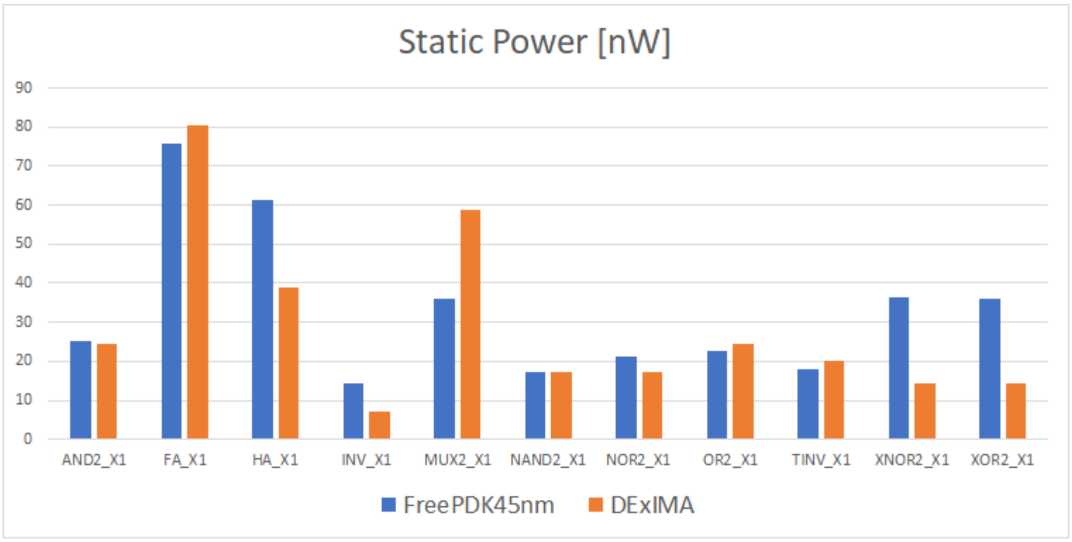


Figure 12.8: DExIMA Models Static Power Comparison with FreePDK45nm

Considering also that some models can use pass transistors to optimize the gate, in DExIMA this type of topology is not implemented.

12.3.4 Delays Comparison

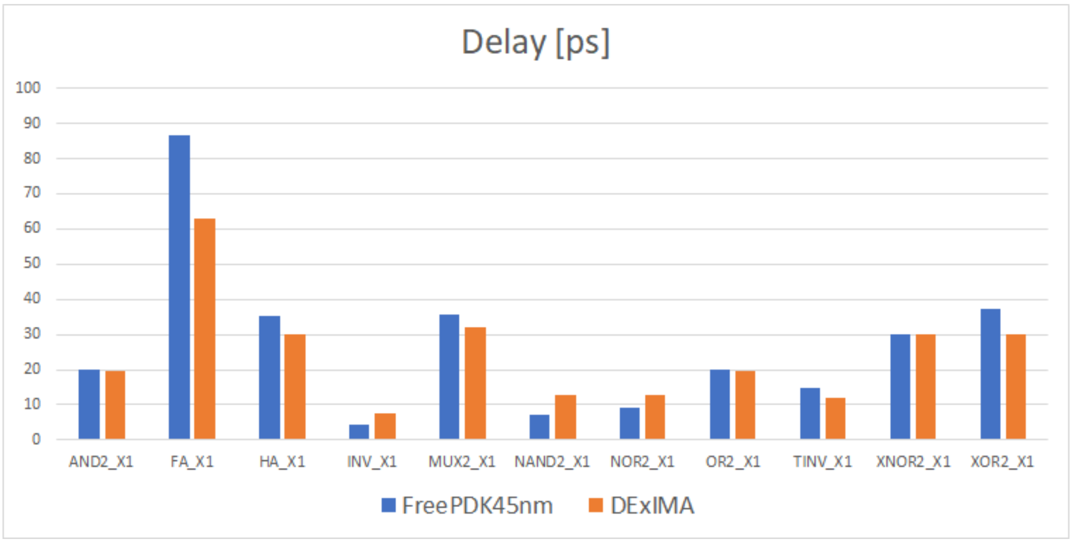


Figure 12.9: DExIMA Models Delays Comparison with FreePDK45nm

In Figure 12.9 you can see the result of the delays: the elementary gates models are a little slower than the corresponding value for the PDK45nm.

For example, if we take the inverter gate, the DExIMA model is slower as expected because the dimension of the transistor and the I_{on} are smaller, which means higher delays.

12.4 Flip Flop Models Comparison

Looking for the performance of the Flip Flop in DExIMA that implements two different types as explained in the Model chapter.

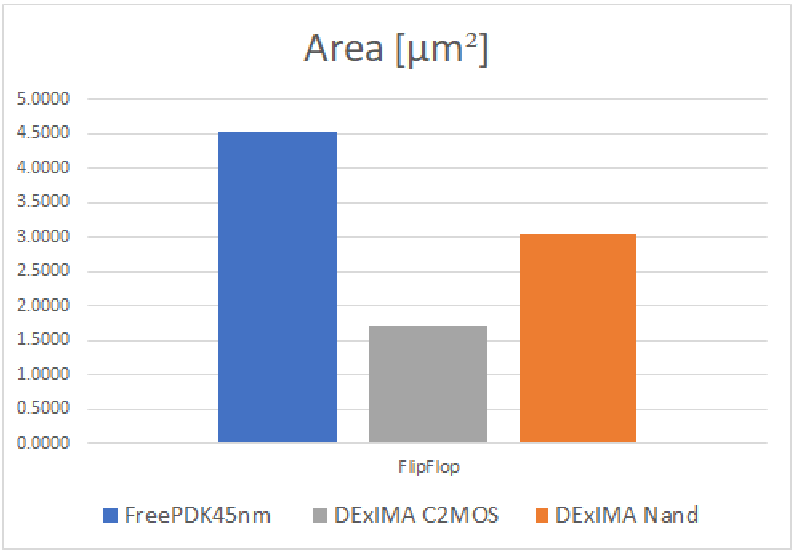


Figure 12.10: Flip Flop area comparison

If we look in Figure 12.10 we see the three models, the component of PDK45nm, the Nand version and, the C2MOS version of DExIMA. It is possible to see that the version of C2MOS is more compact in terms of area, for this reason it is used to create the memory cell of DExIMA. Instead, if we look at the static power in Figure 12.11 we find that the values of static power of C2MOS are very similar with the FreePDK45nm technology instead of the Nand, which has also double the value. The dynamic energy is similar for the three versions of the models. The last part is related to the timing performance shown in Figure 12.13, we divide the histogram into three sections related to the clock to output, setup, and hold time. The performance of timing is similar to the FreePDK45nm for the C2MOS model, in the case of Nand the Setup time result higher. Finally, we have the value of Hold time that is about 2 times the value obtained in the FreePDK45nm. But we expect this result because the responsible of hold times in DExIMA is due

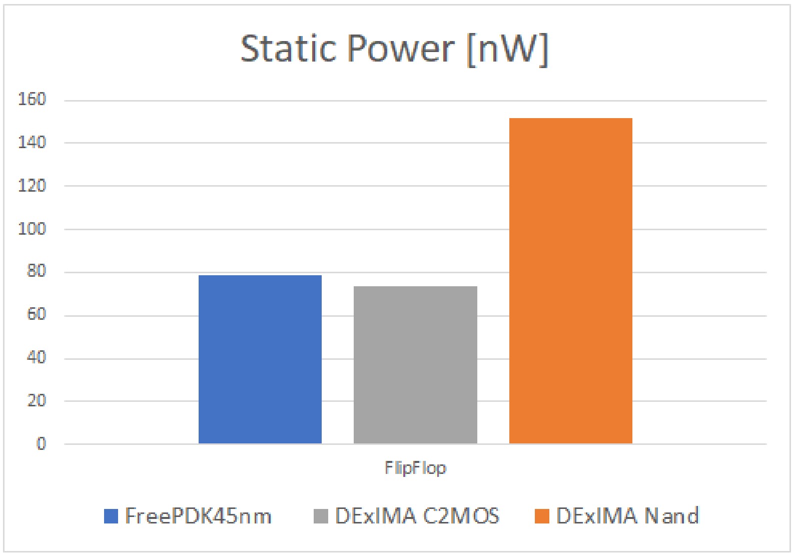


Figure 12.11: Flip Flop static power comparison

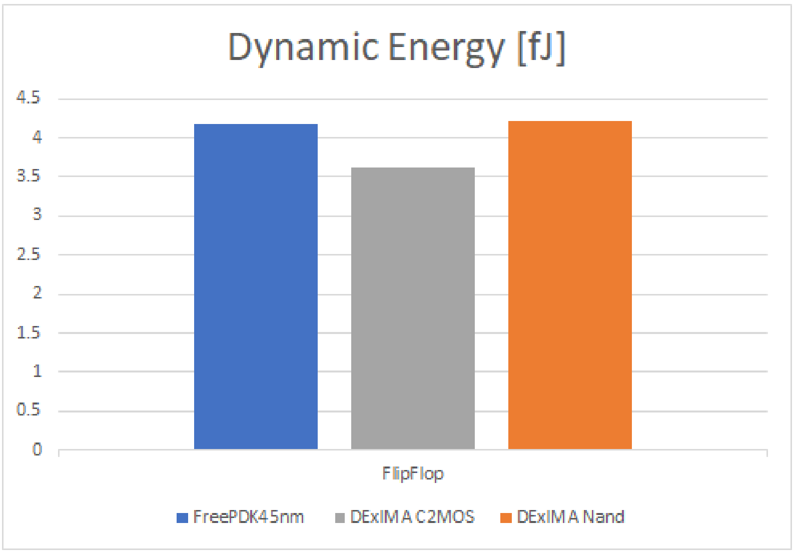


Figure 12.12: Flip Flop dynamic energy comparison

to an inverter. As said before, the dimension of the inverter in DExIMA is half that of the technology. This means that the inverter is 2 times slower this is verified with the result obtained in the hold times.

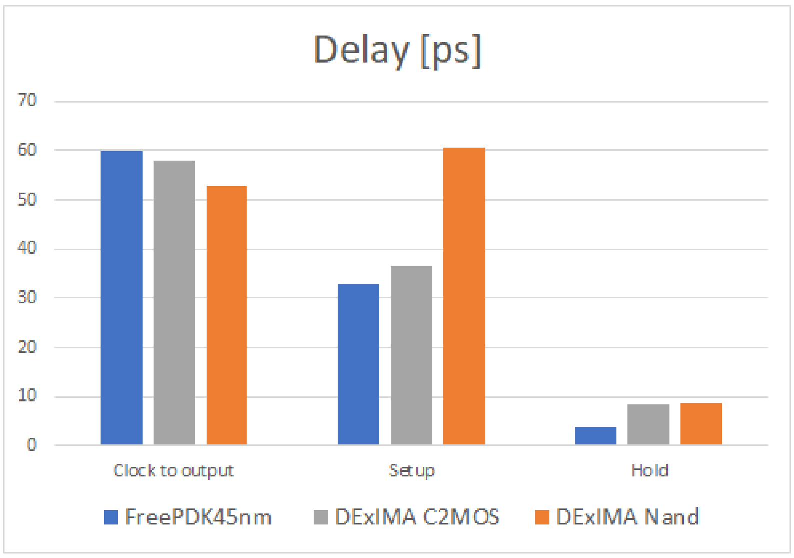


Figure 12.13: Flip Flop delay comparison

Chapter 13

Conclusions and future work

In this chapter we summarise all the upgrades done in the program and the newly added features, highlighting the problems with the previous version explained in the chapter of Motivations (Chapter 3). After this part, we explain the new problems and the upgrade needed for the future.

13.1 DExIMA 2.0 Features

The new DExIMA has several features embedded in it:

- **Configurability**: It is possible to set several simulation parameters like clock period, technology, etc. It is also possible to use the constants to configure the architecture. Adding new technology files is also possible.

- **Modularity**: All the components of the program, from the parsing sections that can be added and removed, to the printers, are modular.

- **Maintainability**: All the classes and functions are well separated and is possible to test each and easily add a new one.

- **Generality**: The program can integrate different types of models thanks to the printer's generality. The models are not restricted to CMOS technology but can be used also by emerging technologies like the pNML (perpendicular Nano Magnetic Logic) or other types of technology.

13.2 DExIMA 2.0 Improvements

Now there is the list of subsections that specify all the improvements and modifications.

13.2.1 Language improvements

In the new version are added several features to the language, which means an addition to a compiler level. The newly introduced features are:

- **Cycle For**: Now it is possible to use this smart construct to create multiple instances and connect them with only one line of code, without specifying one by one, and without using an external script to do it.
- **Constants**: The constants are constant variable that is possible to use in each part of the code, very useful to parameterize our architectures.
- **Built-In constants**: Thanks to these special constants it is possible to modify the behavior of the simulation.
- **Math Environment**: The math environment allows to do math operations directly in a DExIMA script and it is useful in combination with the constants to parameterize the code and the loops.
- **Comments**: Comments are also a useful tool to modify and manage the code efficiently and increase readability.
- **Linter**: It is also a useful tool to understand before the compilation process if in the code something is wrong.
- **Sections**: Now there are more sections and subsections to customize our circuit and options to customize the best possible. Before there were only 3 sections in the .arch file (init, map, operation), but now there are 6 sections and 7 different subsections.
- **Pipeline**: Now the instruction can have more than one path, moreover we can specify sub-paths inside its path.
- **Attributes**: The attributes permit to refine the simulation and specify the different characteristics of Flip Flops and the memory in terms of power and timing.

- **Log and Dof Files**: Thanks to these two output files the user can track the tool operation and store in a unique file a lot of information about the performance computed, and technology parameter. It is useful to identify the behavior of each piece of architecture.

13.2.2 Models Improvements

In this version, the models are changed and refined to have better precision in the performance computation. The additions and improvements have done are:

- **Specialized Models**: Now each base gate has a custom model and designed at the transistor level, and all the complex gates are a composition of these. Now not all the models are based on the Nand 2 gate.
- **Dynamic Inputs Variation**: Now all the base gates are sensible to the inputs variation and the performance of it changes. In the previous version if we change the number of inputs the performance does not change and it remains also a Nand 2 but with the possibility to connect more "virtual" inputs.
- **Computation of Secondary effects**: Now some important secondary effects are taken also into account like the Stack Effect.
- **Attributes differentiation**: The parameters of the performances are more than four because we take also different specifications into account like the clock to output, setup, and hold of Flip Flop, and differentiate the specification of power and timing.
- **Fanin/Fanout interaction**: The modules now can recognize the load and the dimension of it, without considering only the load of one Nand 2. Moreover, the memory cell has a real cell and we can connect with another component and recognize how much load is connected to it.
- **Switching activity evaluation**: Now it is possible to compute the switching activity of our gate choosing the input gates probability. Remember that the computation is worst the case using the "isolate gate" approach.
- **Easy Interface construction**: Creating a new model now, it is very simple to create the interface. We need to create a class derived from

the printer and set some vectors of strings, and in this way, DExIMA automatically creates all the routines to the component interface and manages the errors. On the contrary, in the previous version, we need to modify too many files to specify every single aspect of the component.

13.2.3 Fixed Problems

In this subsection we list the problems highlighted in Chapter 3 and how we resolved them with the new version of the tool:

- **Configuration Files Writing Effort**: Thanks to all the new features added and the improvements of the new compiler, the effort of code writing is reduced to the minimum.
- **Error checking problem**: Now the compiler can detect all types of errors. As shown before we have 108 different manageable errors versus the previous 7. Moreover, the compiler returns important suggestions to resolve the errors.
- **Configuration files separation**: Now we have only one file, the .dex file, instead of four different configuration files. With this approach, we can connect all the components between them, also the memories. Doing this it is possible to share part of the tool code of the compiler. It is now possible also to create multiple memories in the same architecture.
- **Random Behaviour**: Now that random behaviors are not present anymore, all the successive simulations of the same script get the same result.
- **Simulation Times**: Thanks to the new computation structure, the simulation is very fast, and the bottleneck is the compiler. But in this case, it is better because we have a linear behavior instead of exponential law.
- **Models Efficiency**: Now the models are more complex and specialized, meaning that they have performances, that are more close to reality, instead of using only Nand 2 gate.
- **Simulation parameters orientation**: Now it is possible to modify the simulation constrain thanks to the built-in constants, and it is also possible to change the technology used for the "synthesis".

- **Insertion of a new model**: Now the insertion of a new model needs only the creation of class associated with it. The interface is very easy to implement, and the performance equations are grouped all in the same class. If the model is derived from others, in this case, we can reuse the modules already present.
- **Absence of Documentation**: Now documentation generated by the Doxygen automated tool [35] is available in HTML to navigate in it, or PDF form but it is more diluted with its about 900 pages.

13.3 Further Improvements

Some features of the tool need to be optimized and refined for the next updates.

- **Compiler optimization**: Some constructs of the compiler can be optimized to reduce the computation cost and speed up more the compilation step. On the contrary, the simulation part is well optimized.
- **Switching activity propagation**: It can be very useful to add the computation of the switching activity considering also the propagation between the gates after the connections.
- **Stack Effect refinement**: From now on, the stack effect computation is the worst-case approach, that can be useful to create an optimized model to compute it more precisely.
- **New Memory architectures and models**: It is needed also to add a more realistic model for the memory like inserting the SRAM and DRAM cells and the related architecture.

Appendix A

Compiler Specifiers

A.1 Built-In Constants

Name	**Default**	**Scope**
VDD	TechFile	Set the voltage Supply in $[V]$
CLOCK	Critical Path	Set the Clock period in $[ns]$
AR	TechFile	Set the Aspect Ratio of unitary N-mos
SF	2	Set the stack factor
NODE	45	Set the technology node in $[nm]$
TECH	LOP	Set the technology type
SWITCHING	OFF	Enable the Switching computation
PROB	0.5	Set the inputs probability

Table A.1: Table Containing the available built in constants

A.2 TechFile Parameters

Name	**Scope**
Year	Technology Year
Lgate	Gate Length $[m]$
Xj	Source/Drain extensions length $[m]$
Gamma	Scaling factor for lateral diffusion
Inter_over	Interconnection Overhead
Cell_over	Standard Cell Overhead
Aspect_ratio	Minimal N-Mos Aspect Ratio
Beta	P-Mos/N-Mos Width Ratio
Vdd	Supply Voltage $[V]$
Cox	MOS capacitance $[F/m^2]$
Ion	Drain Saturation Current $[A/m]$
Ioff	Subthreshold Current $[A/m]$
Igate	Gate Current $[A/m]$
CJ0N	N-Mos junction capacitance $[F/m^2]$
CJ0P	P-Mos junction capacitance $[F/m^2]$
CJSWN	N-Mos sidewall junction capacitance $[F/m]$
CJSWP	P-Mos sidewall junction capacitance $[F/m]$
CGD0N	N-Mos overlap capacitance $[F/m]$
CGD0P	P-Mos overlap capacitance $[F/m]$
MJN	N-Mos Bottom Capacitance Parameter [%]
MJP	P-Mos Bottom Capacitance Parameter [%]
MSWN	N-Mos Sidewall Capacitance Parameter [%]
MSWP	P-Mos Sidewall Capacitance Parameter [%]
PBN	N-Mos Bottom Capacitance Parameter $[V]$
PBP	P-Mos Bottom Capacitance Parameter $[V]$
PBSWN	N-Mos Sidewall Capacitance Parameter $[V]$
PBSWP	P-Mos Sidewall Capacitance Parameter $[V]$
C_Interc	Interconnection Capacitance $[F/m]$

Table A.2: Technology Parameters description table

A.3 Available Technologies

Technology nodes		
HP	LOP	LSTP
14	18	22
16	20	25
18	22	28
20	25	32
22	28	37
25	32	45
28	37	53
32	45	65

Table A.3: Table of available technologies

Appendix B

DExIMA Models

Model	**Keyword**	**Parameters**	**Inputs**	**Outputs**
Inverter	NOT	Void	IN	OUT
Nand	NAND	#Inputs	IN0,IN1...	OUT
And	AND	#Input	IN0,IN1...	OUT
Nor	NOR	#Inputs	IN0,IN1...	OUT
Or	OR	#Inputs	IN0,IN1...	OUT
Xor	XOR	#Inputs	IN0,IN1...	OUT
Xnor	XNOR	#Inputs	IN0,IN1...	OUT
FlipFlop Nand	FF_NAND	Parallelism	D,CK	Q,Qn
FlipFlop C2MOS	FF	Parallelism	D,CK	Q
FlipFlop Enable	FF_EN	Parallelism	D, CK,EN	Q
Clock Driver	CK_DRIVER	Effort	IN	OUT
Driver	DRIVER	Model, Effort	IN	OUT
Decoder	DECODER	Input Par, Output Par	IN	OUT
FlipFlop Cell**	FLIPFLOP	Void	WR, WR_MEM, S,EN,CK	RD
FullAdder	FA	Void	A,B, CIN	S,COUT

HalfAdder	HA	Void	A,B	S,COUT
Latch SR	LATCH_SR	Void	S,R, EN	Q,Qn
LiM Memory	LIM	Address Par, Read/Write Par	ADDR,WR, CK, SEL	RD
Capacity Load	LOAD	Capacity Value [pF], Parallelism	IN	Void
Multiplexer	MUX	#Inputs, Parallelism, Ways	IN0,IN1... S	OUT
Ripple Carry Adder	RCA	Parallelism	A,B, ADD	S,COUT
Three State Inverter	TNOT	Void	IN CK,CKn	OUT
Xor Core*	XORCORE	Void	IN0,IN1, IN2,IN3	OUT

Table B.1: Models description table

* The model cannot be used
** The model can be specified only in the TYPE field of the memory section (memdef)
Par: Is the abbreviation of *Parallelism*

Bibliography

[1] Santoro G, Turvani G, Graziano M., *New Logic-In-Memory Paradigms: An Architectural and Technological Perspective.*, Micromachines. 2019; 10(6):368.

[2] S. Angizi, Z. He and D. Fan, *PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-Efficient Logic Computation*, 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, 2018, pp. 1-6, doi: 10.1109/DAC.2018.8465706.

[3] W. Huangfu, S. Li, X. Hu and Y. Xie, *RADAR: A 3D-ReRAM based DNA Alignment Accelerator Architecture*, 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, 2018, pp. 1-6, doi: 10.1109/DAC.2018.8465882.

[4] Cofano M., Vacca M., Santoro G., Causapruno G., Turvani G., Graziano M., *Exploiting the Logic-In-Memory paradigm for speeding-up data-intensive algorithms* Integration 2019
https://doi.org/10.1016/j.vlsi.2019.02.007

[5] G. Papandroulidakis, I. Vourkas, A. Abusleme, G. C. Sirakoulis and A. Rubio, *Crossbar-Based Memristive Logic-in-Memory Architecture*, in IEEE Transactions on Nanotechnology, vol. 16, no. 3, pp. 491-501, May 2017, doi: 10.1109/TNANO.2017.2691713.

[6] N. Piano, *DExIMA: a Design Explorer for In-Memory Architectures.* MA thesis. Politecnico di Torino, 2019.
url: https://webthesis.biblio.polito.it/12547/

[7] Vassili Kaplan, *A New Algorithm to Parse a Mathematical Expression and its Application to Create a Customizable Programming Language*, ICSEA 2016 : The Eleventh International Conference on Software Engineering Advances

[8] Nebi Caka, Milaim Zabeli, Myzafere Limani, Qamil Kabashi (2007), *Influence of MOSFET parameters in its parasitic capacitance and their impact in digital circuits*, WSEAS Transactions on Circuits and Systems. 6.

281-287.

[9] Fabrizio Riente, Izhar Hussain, Massimo Ruo Roch and Marco Vacca, *Understanding CMOS Technology through TAMTAMS Web*, 10.1109/ TETC.2015.2488899, IEEE Transactions on Emerging Topics in Computing

[10] T. Skotnicki, G. Merckel and C. Denat, *MASTAR - A Model For Analog Simulation Of Subthreshold, Saturation And Weak Avalanche Regions In MOSFETs*, [Proceedings] 1993 International Workshop on VLSI Process and Device Modeling (1993 VPAD), Nara, Japan, 1993, pp. 146-147, doi: 10.1109/VPAD.1993.724762.

[11] International Technology Roadmap for Semiconductors ITRS. (2020). url: http://www.itrs2.net/

[12] R. V. Menon, S. Chennupati, N. K. Samala, D. Radhakrishnan and B. Izadi, (2004). *Switching Activity Minimization in Combinational Logic Design*, 47-53.

[13] F. N. Najm, *Transition density: a new measure of activity in digital circuits*, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, no. 2, pp. 310-323, Feb. 1993, doi: 10.1109/43.205010

[14] Shengqi Yang, W. Wolf, N. Vijaykrishnan, Yuan Xie and Wenping Wang, *Accurate stacking effect macro-modeling of leakage power in sub-100 nm circuits*, 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design, Kolkata, India, 2005, pp. 165-170, doi: 10.1109/ICVD.2005.41.

[15] Sang-Pil Sim, Shoba Krishnan, Dusan M. Petranovic, Narain D. Arora, Kwyro Lee, Cary Y. Yang, *A Unified RLC Model for High-Speed On-Chip Interconnects*, IEEE Transactions on Electron Devices, Vol. 50, No. 6, June 2003

[16] Jagannath Samanta, Madhuresh Suman, Jaydeb Bhaumik and Soma Barman. (2016), *A Transistor Level Implementation of Reed Solomon Encoder in GF(2^8)*, J. of Active and Passive Electronic Devices. 00. 1-19.

[17] Subodh Wairya1, Rajendra Kumar Nagaria2 and Sudarshan Tiwari (2012), *Comparative performance analysis of XOR-XNOR function based high-speed CMOS full adder circuits for low voltage VLSI design*, International Journal of VLSI Design and Communication Systems (VLSICS). 3. 221-242.

[18] S. Barra N. Bouguechal A. Dendouga O. Manck (2008), *Design And Layout of Finite State Machine Using C2MOS Latch in CMOS 0.35μm technology*, International Conference on Electrical Systems Design and

Technologies, Hammamet Tunisia, Nov. 8-10, 2008

[19] Kunwar Singh, Satish Chandra Tiwari, andManeesha Gupta (2014), *A Modified Implementation of Tristate Inverter Based Static Master-Slave Flip-Flop with Improved Power-Delay-Area Product*, The Scientific World Journal. 2014. 453675. 10.1155/2014/453675.

[20] V. Stojanovic and V. G. Oklobdzija, *Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems*, in IEEE Journal of Solid-State Circuits, vol. 34, no. 4, pp. 536-548, April 1999, doi: 10.1109/4.753687.

[21] S Agatonovic-Kustrin and R Beresford, *Basic concepts of artifcial neural network (ann) modeling and its application in pharmaceutical research*, Journal of pharmaceutical and biomedical analysis, 22(5):717-727, 2000

[22] Camuñas-Mesa LA, Linares-Barranco B, Serrano-Gotarredona T., *Neuromorphic Spiking Neural Networks and Their Memristor-CMOS Hardware Implementations*, Materials. 2019; 12(17):2745.

[23] Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall, *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*, arXiv:1811.03378v1 8 Nov 2018

[24] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet, *Convolutional networks and applications in vision*, Proceedings of 2010 IEEE International Symposium on Circuits and Systems (2010), pp. 253–256.

[25] Andrea Coluccio, Marco Vacca and Giovanna Turvani, (2020), *Logic-in-Memory Computation: Is It Worth it? A Binary Neural Network Case Study. Journal of Low Power Electronics and Applications*, 10. 7. 10.3390/jlpea10010007.

[26] Wang, Y., J. Lin and Z. Wang, *An Energy-Efficient Architecture for Binary Weight Convolutional Neural Networks* IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26 (2018): 280-293.

[27] Scherer, D.; Müller, A.; Behnke, S., *Evaluation of pooling operations in convolutional architectures for object recognition* International Conference on Artificial Neural Networks; Springer: Berlin, Germany, 2010; pp. 92–101.

[28] Yash Akhauri, *Binary neural networks*, https://software.intel.com/content/www/us/en/develop/articles/binary-neural-networks.html, December 2020.

[29] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi, *Xnor-net: Imagenet classifcation using binary convolutional neural networks*, In European Conference on Computer Vision, pages 525-542. Springer, 2016.

[30] Andrea Coluccio, *In-Memory Binary Neural Networks.* MA thesis. Politecnico di Torino, 2019.
url: https://webthesis.biblio.polito.it/10988/

[31] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi, *Cacti 6.0: A tool to model large caches*, Technical report, Hewlett-Packard Laboratories - School of Computing, University of Utah.

[32] J. E. Stine et al., *FreePDK: An Open-Source Variation-Aware Design Kit*, 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07), San Diego, CA, 2007, pp. 173-174, doi: 10.1109/MSE.2007.44.

[33] J. E. Stine et al., *FreePDK v2.0: Transitioning VLSI education towards nanometer variation-aware designs*, 2009 IEEE International Conference on Microelectronic Systems Education, San Francisco, CA, 2009, pp. 100-103, doi: 10.1109/MSE.2009.5270820.

[34] North Carolina State University, Process Desing Kit FreePDK45nm https://www.eda.ncsu.edu/wiki/FreePDK45:Contents (2020)

[35] Doxygen documentation https://www.doxygen.nl/index.html