POLITECNICO DI TORINO

Department of Electronics and Telecommunications
Master degree in Electronic Engineering

Master Thesis

Molecular Field-Coupled Nanocomputing cell modelling and characterisation

Supervisors
Prof. Mariagrazia Graziano
Prof. Gianluca Piccinini
Dott. Yuri Ardesi
Dott. Giuliana Beretta

Candidate
Erik Lo Grasso

April 2022

Sommario

Le leggi di Moore hanno fin'ora predetto con estrema precisione lo sviluppo dei circuiti integrati in termini di transistor per chip, lunghezza di canale degli stessi e area. Sebbene furono formulate empiricamente all'inizio degli anni *'70*, negli anni nacque una vera e propria *corsa all'oro* da parte dei *manufacturers* affinché lo sviluppo negli anni ricalcasse la traiettoria immaginaria che queste leggi definirono, e per 50 anni ci riuscirono. Nell'ultima decade si è assistiti ad un rallentamento generale della crescita prevista da Moore, simbolo che si sta raggiungendo un limite nello spazio di miglioramento di un sistema realizzato secondo gli standard moderni.
Il CMOS è una tecnica circuitale di realizzazione delle porte logiche che negli anni *'90* rivoluzionò l'elettronica grazie agli inobiettabili vantaggi rispetto alle tecniche usate in precedenza.
Fissata la tecnica di realizzazione a livello circuitale, per ottenere migliorare un sistema è possibile percorrere due linee di pensiero distinte:

- La prima a cui si fa riferimento come "More than Moore" è rivolta al livello architetturale, riorganizzando ciò che si ha in maniera quanto più efficiente possibile e integrando sempre più sistemi in maniera eterogenea per la creazione di cosiddetti System-On-Chip (SoC).

- La seconda a cui si fa riferimento come "More Moore" riguarda il quanto di una porta logica, il transistor, con l'idea di migliorarne l'implementazione rendendolo più piccolo ed efficiente garantendo prestazioni, area e potenza richiesta per il funzionamento ottimizzate.

Raggiunto però il limite precedentemente citato riguardo i miglioramenti ottenibili sul CMOS, si passa a considerare tecnologie alternative che possano sostituirlo efficacemente. A queste tecnologie alternative ci si riferisce in generale come "Beyond CMOS" e una delle più promettenti prende il nome di *Field-Coupled Nanocomputing* (FCN).
Il FCN abbandona il concetto di transistor basando la propagazione dell'informazione sui campi elettrostatici. Questo paradigma tecnologico consente un significativo risparmio energetico in quanto non si basa sulla migrazione di portatori di carica come nel caso del CMOS e dei transistors in generale.
Una delle possibili implementazioni del FCN interessa i *Quantum-Dot Cellular Automata* (QCA). In funzione di come sono fisicamente implementati i QCA, si ottengono dimensioni ultra scalate e frequenze operative ordini di grandezza più importanti rispetto all'attuale

stato dell'arte CMOS. Una delle implementazioni più promettenti è quella molecolare, tale per cui ogni *QCA* viene composto da due molecole complesse contenteni tre Quantum-Dot ciascuna.
Grazie a *SCERPA*, algoritmo in grado di simulare la propagazione dell'informazione dovuta all'accoppiamento elettrostatico delle molecole, la creazione e validazione di layouts in questa tecnologia risulta approcciabile, tuttavia può richiedere molto tempo trattandosi di un algoritmo iterativo che risolve un sistema non lineare dipendente dalla/e tensione/i di ingresso e dalle configurazioni di carica assunte da ogni molecola. Lo scopo di questa tesi è stato sviluppare uno strumento che lavori in concomitanza con *SCERPA* per la caratterizzazione ingresso-uscita delle celle, che permetta la validazione delle stesse in maniera semplice e sia in grado di creare una libreria sfruttabile nella simulazione di sistemi troppo estesi per *SCERPA*.
Caratterizzare una cella in maniera efficace prevedendo una percentuale ragionevole di casistiche di utilizzo ha richiesto l'integrazione di *add-on* specifici che trattino ad esempio un *fan-out* maggiore di 1, cosi come la gestione di terminazioni in cascata e ancora la generazione di combinazioni di ingressi adattative e misurate nel contesto *SCERPA*, dove il tempo di simulazione richiesto dipende fortemente dal numero di queste. Tutto ciò è stato inquadrato in un contesto di spinta automatizzazione con il fine ultimo di aiutare l'utilizzatore.
Dopo un'analisi introduttiva della tecnologia focalizzata sui suoi aspetti chiave utili per comprendere lo strumento di caratterizzazione e riportata nel capitolo 1, si passerà, nel capitolo 2, all'implementazione e alle possibilità che offre, discutendo infine i risultati relativi la caratterizzazione di layouts fondamentali, nel capitolo 3, e l'applicazione degli stessi per la valutazione di un circuito più complesso quale la *XOR* nei capitoli 4 e 5.

Abstract

Moore's laws have predicted with extreme accuracy the *IC* development in terms of transistors per chip, channel length and area. Although these were formulated empirically in the early '70s, the manufacturers worked hard to stay on the laws' traced tracks with outstanding results.
In the last decade, the development curve has slowed down marking the reach of a possible technology limit.
Nowadays, the CMOS is still the most used circuital technology: introduced in the early '90s, it revolutionised modern electronics with amazing improvements but, now, the enhancement margins are going to saturate.
Actually, there are two main ideas on which designers rely to build more valuable systems: the "More than Moore" approach deals with a high level of abstraction, looking at the systems as heterogeneous (*SoCs*) and trying to reorganize them from the architectural point of view. The other possibility is the "More Moore" one which is interested in the low level of abstraction, working on the transistor implementation.
If there are no more margins to improve the technology following the discussed approaches, a new and worthful solution that outperform the precedent one must be considered. There are a lot of new emerging "Beyond CMOS" technologies, but one of the most promising is the *Field-Coupled Nanocomputing* (*FCN*).
The *FCN* exploits blocks (*QCA*) coupled with electrostatic fields to ensure information propagation. This approach avoids charge transport's defined logic states, enabling the technology for ultra low power, high operating frequency and strongly scaled devices depending on the *FCN*'s implementation technology. The molecular implementation is considered one of the greatest solutions: each *QCA* is composed of two complex molecules with three Quantum-Dots each able to aggregate the charge and work at room temperature.
Through *SCERPA*, the algorithm implemented in *MATLAB* that can simulate information propagation in layouts due to electrostatic fields and externally applied voltages, nowadays creation and evaluation of basic cells are approachable. However, *SCERPA* can be particularly time demanding depending on the layout under test and the input combinations chosen for the simulation, so that too complex layouts could be untreatable.
This thesis project aims to develop a *Characterisation tool* that can work with *SCERPA* to extract the $V_{in} - V_{out}$ behaviour of generic layouts. The tool would be a valid instrument to both validate and characterise a cell permitting the eventual design *fine-tuning* and the stamp of a library file exploitable in the simulation of complex devices.

The role of the mentioned *Characterisation tool* is everything but trivial: the necessity to prevent the work condition of a circuit in a more complex layout pretend the use of dedicated *add-ons* to treat with, for instance, the need for a *bi-stable* simulation got with a kind of termination, a higher *fan-out* or the adaptative input combinations measured in the *SCERPA* context, where the time overhead is hardly dependent from these. The entire tool has been developed with a strong bias towards automation with a precise focus on ease of use.
After a brief technology overview with attention to fundamental notions used in the tool implementation, this thesis project would analyze the implementation in *MATLAB* of the *Characterisation tool* showing the results on basic layouts at first and applying them on a more complex circuit like an exclusive OR.

If you cannot understand my
argument, and declare
it's Greek to me
you are quoting Shakespeare.

[B. Levin, Quoting Shakespeare]

Contents

List of Acronyms

CMOS Complementary MOS.

csv Comma Separeted Values.

DUT Device Under Test.

FCN Field-Coupled Nanocomputing.

IC Integrated Circuit.

INV Inverter.

MolFCN Molecular Field Coupled Nanocomputing.

MV Majority Voter.

QCA Quantum-Dot Cellula Automata.

QD Quantum-Dot.

SCERPA Self-Consistent ElectRostatic Potential Algorithm.

SOA Self Operating Area.

SoC System-on-Chip.

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Technology overview

1.1 Moore's law and Beyond CMOS

The development of new technologies in modern electronics is related to a few main factors like the capability to improve the integration density, the power consumption and the performance.
One of the most known *Moore's laws* deals with the number of transistors in the IC's versus time: Gordon Moore predicted transistors to double on chips every two years at first [1], and every *1.5 years* later, following the rule:

$$\frac{N_{tr}}{IC}(t) = \frac{N_{tr}}{IC}(t_0) \cdot 2^{\frac{t-t_0}{1,5}} \tag{1.1}$$

Gordon Moore's laws also refer to the IC's area and the transistors' channel length. Following its previsions, the IC's dimension would increase by 50% every three years while the channel length gets decreased by 50% over the same amount of time:

$$A_{IC}(t) = A_{IC}(t_0) \cdot 1{,}5^{\frac{t-t_0}{3}} \tag{1.2}$$

$$L_{CH}(t) = L_{CH}(t_0) \cdot 2^{-\frac{t-t_0}{6}} \tag{1.3}$$

Although these rules got formulated in the early '70s, they are still valid:

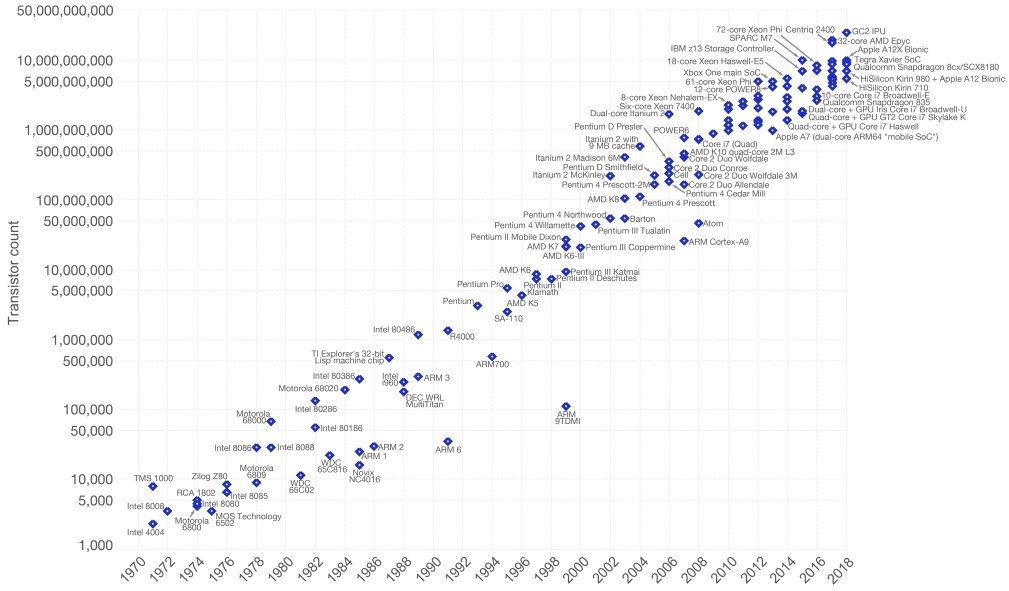


Figure 1.1: Transistor count versus years, 1970-2018 [2]

Today, to obtain more valuable systems, two main trends can be considered:

- **More than Moore (MtM)**: the exploitation of the current technology to implement SoC through heterogeneous integration, 3D integration and new architectures, relying on bigger systems or better organized instead of looking at the low level;
- **Beyond CMOS**: leave the CMOS standard for new and distinguished technologies with which realize the logic.

The combination of these two approaches defines tomorrow's electronics in terms of ICs. This thesis project is focused on the Beyond CMOS approach, concentrating in particular on *Field-Coupled Nanocomputing* (FCN) based on *Quantum-Dot Cellular Automata* (QCA) and its most promising *molecular implementation*.

1.2 Field-Coupled Nanocomputing (FCN)

The idea on which FCN is based considers to leave the transistor concept, giving space to other technologies that encode the binary information in charge-transport-free ways. The standard logic gates realized with transistors, encode the binary information following a current-based approach. For instance, it is a current from the output to the ground that pulls down the output voltage as it is a current from the power supply toward the output that carries the voltage output to the high level.
On the other hand, FCN uses electrostatic fields to transport the binary information avoiding useless power loss. The results deal in an ultra-low-power implementation [3], really high operating frequencies (up to *THz*) [4, 5, 6] with an incredible integration density depending on the implementation technology. A possible implementation for the FCN is due to QCAs among the several options.[4, 7, 8]
QCAs are blocks realized through Quantum-Dots (QDs) that promise to capture the charges in a certain configuration based on the externally applied forces in general.

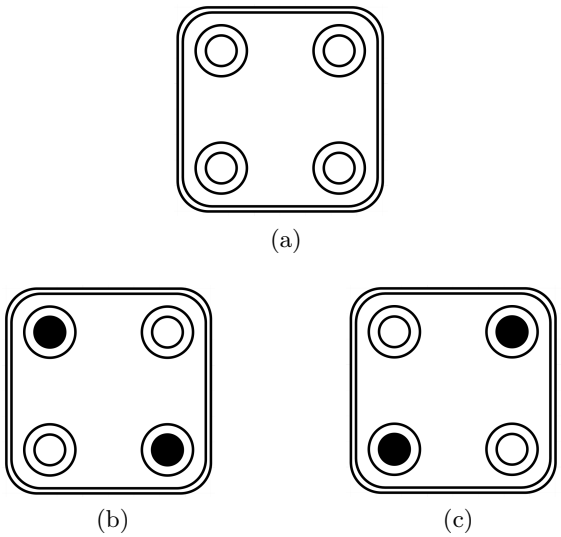


Figure 1.2: Four QDs QCA blocks: (a) Blank QCA block, (b) '0'-logic state QCA [9], (c) '1'-logic state QCA [9]

The logic states referred into figures 1.2b and 1.2c are obtained through the natural tendency of charges to configure in a minimum potential energy state.
The QDs at the edges of the squared container represent low potential zones where the charges are more likely to stay. The repulsion driven behaviour of identical charges discriminates the two diagonal as a possibility to identify a logic state, considering the two QDs interested as the farthest couple each respect to the other.
The two discussed diagonals are identical in terms of probability to obtain one or the other if no externally applied forces are considered. The main idea on which this technology relies, deals with putting these blocks side by side: the electrostatic field induced by charges configuration can work as a *charge influencer*, deciding how the charges of the

next QCA will configure, so which diagonal they will intercept.

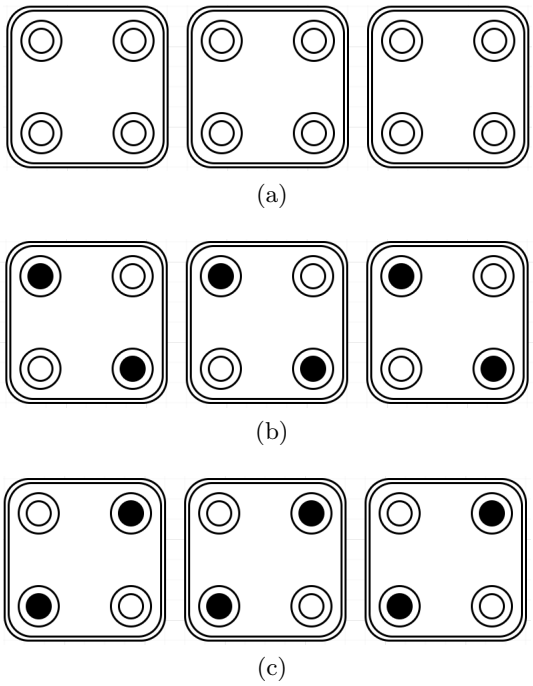


Figure 1.3: QCA wires: (a) Blank QCA wire, (b) '0'-logic state induction in QCA wire, (c) '1'-logic state induction in QCA wire

Ideally, by making chains of QCAs side by side, the information can propagate from a starting point to a different part of the layout in a domino-like way as shown in figure 1.3b and 1.3c.
The concept of making logic through this as other technologies, considers building different basic logic gates like the AND, OR, NOT and the complexes NAND, NOR, XOR, XNOR. Basically, the complex gates can be derived starting with the simplest ones but, at least, is needed something more than a wire to implement whichever logic function in a system. The FCN paradigm considers three main elements working as the fundaments for more complex systems:

- Wires
- Majority Voters (MVs)
- Inverters (INVs)

Wires are straight-forward structure as discussed before and can have different forms like *L-connectors*, or straight ones. The MVs are more interesting looking at theirs flexibility:

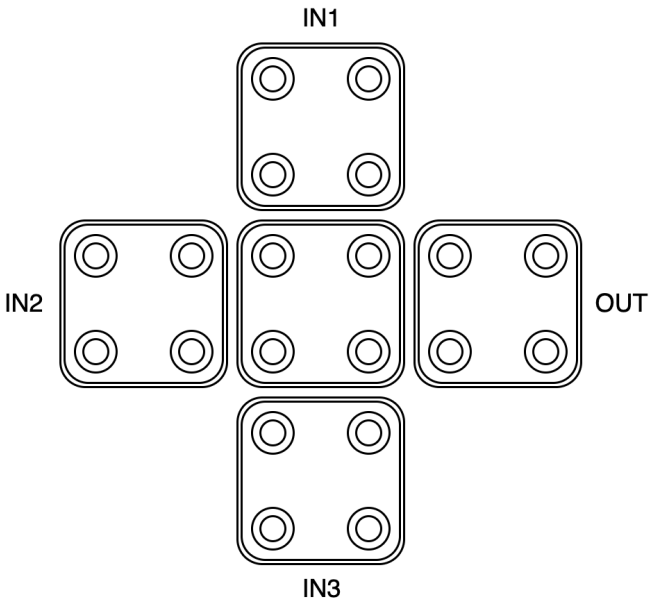


Figure 1.4: Blank MV with explicited inputs and output [9, 10]

The MV realized in FCN is not different from the one realized in CMOS technology. The output will configure in the same state of the central QCA that works like a *conveyer* for the inputs. The logic state of the central QCA is the one assumed by two of the inputs at a time.

Table 1.1: MV complete truth table with logic states

IN1	IN2	IN3	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The truth table shows a particular behaviour if one of the inputs is fixed to a logic state. If the table is split into two subtables taking, for instance, *IN1* as discriminant, an equivalent AND-gate and OR-gate will be obtained respectively.

Table 1.2: AND equivalent truth table as subtable of the MV

IN1	IN2	IN3	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1

Table 1.3: OR equivalent truth table as subtable of the MV

IN1	IN2	IN3	OUT
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Having an AND-gate and an OR-gate available through the use of MVs, it is possible to build NAND-gates and NOR-gates through the use of INVs. The NAND-gates (as the NOR ones) enable the logic synthesis of whichever boolean function would be realized following the *De Morgan's Theorems*.

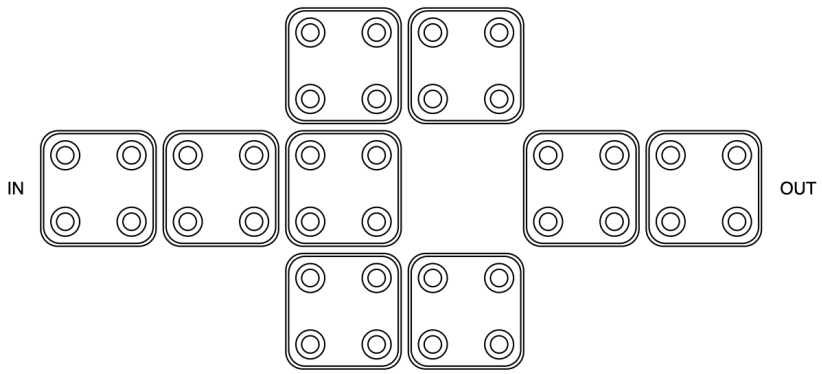


Figure 1.5: Blank INV-gate with explicited input and output [9, 10]

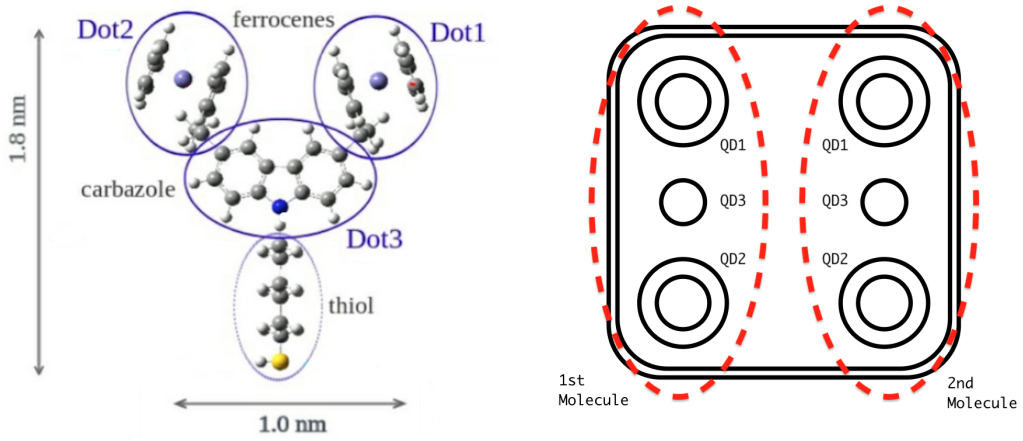
The INV-gate together with the MV, can reproduce the behaviour of a standard CMOS NAND-gate or NOR-gate.
Despite of standard CMOS INVs where it represents the simplest logic gate in terms of transistors number, the MolFCN ones rely on a complex layout, comparable with the MV one in terms of QCA number[1].

[1]The layouts shown are good in principle. The real ones get more complicated in order to work properly and they will be analyzed in details in the following chapters.

1.2.1 Molecular Implementation and clocked systems

The QDs in the QCAs can be implemented in several ways as mentioned before [4, 7, 8]. One of the most attractive is the molecular one, where every couple of QDs is implemented employing a molecule like the *bis-ferrocene*[2]. The *bis-ferrocene* name hides a more complex structure composed by two *ferrocenes* $Fe(C_5H_5)_2$, a *Carbazole group* $C_{12}H_9N$ and a *Thiol group*.
Each of these molecules works as a redox centre [4] and it is seen by charges as a low potential zone where they can aggregate.

- **Ferrocene**: each of them works as the responsible QD for the logic state encoding. In the QCA representation shown in figure 1.2a, each planar circle refers to one of the ferrocenes for each *bis-ferrocene* molecule;
- **Carbazole group**: it permits the link among the two ferrocenes and the *Thiol group*. Also it works as a redox centre like the ferrocenes, representing the fundamental third QD of the *bis-ferrocene* molecule;
- **Thiol group**: it permits the anchoring of the *carbazole* with guest surface usually realized in gold. Also that molecule represents a redox centre for the *bis-ferrocene*.



(a) (b)

Figure 1.6: Molecule integration in QCAs: (a) *Bis-ferrocene* molecule[3] [11], (b) *Bis-ferrocene* molecules in a QCA

[2] A lot of different molecules exist for this scope like the *Diallyl-butane* or the *decatriene*. Some are more appropriate than others and the bis-ferrocene is one of the greatest thanks to its quasi-ideal symmetry.

In figures 1.6a and 1.6b can be appreciated a third QD not described so far but with a fundamental role.
The electrostatic field induced by a QCA to the next one can move the charges giving a bias for the QDs they will occupy. In a chain of QCAs, the information propagates from the input to the output with a certain delay. This delay cannot be zero and, although the Columbian interaction is really fast, the information will clash if it tries to propagate along with a too big layout.

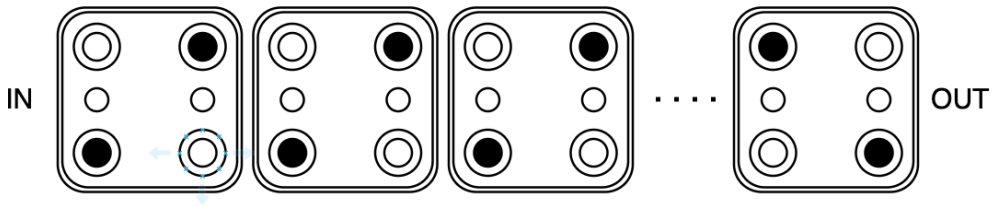


Figure 1.7: Generic length wire with the information clash far from input

The third QDs of QCAs can be exploited with a *vertical clock system* to avoid this criticality, dividing the layout into clock phases and permitting the information to cover relatively long distances.

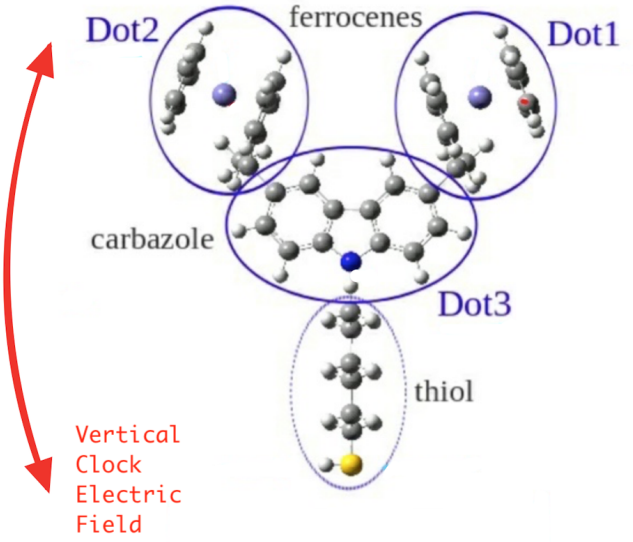


Figure 1.8: Vertical clock electric field on the *bis-ferrocene* molecule

[3]The other fundamental advantage of this technology is the integration density. The dimension of a molecule is comparable with to $\frac{1}{10} \cdot L_{CH}$ of a modern state-of-art transistor [12]

The vertical clock can work as an enable signal for the molecules. It can attract the charges toward the *QD3* designating a *NULL or RESET logic state* for the molecule:

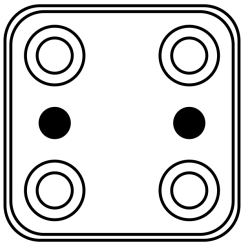
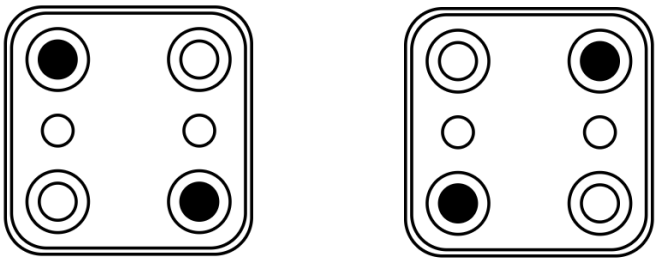


Figure 1.9: QCA in RESET state

On the contrary, it can push the charges towards *QD1 and QD2* enabling almost the totality of charges to aggregate on the designated *QD*. In this case, the QCA is in the *APPLIED state*.



(a) (b)

Figure 1.10: Six QDs QCA blocks in *APPLIED state*: (a) '0'-logic state QCA, (b) '1'-logic state QCA

The idea of dividing the layout into phases exploits a multiple-phase vertical clock so that the release of QCAs follows the information propagation. It is fundamental the timing with which this operation is performed: QCAs that have to switch due to the interaction with the previous phase, need to stay in RESET state until the information is ready at their input.
This timing rule ensures the information propagation and the switching in a stable logic state of the discussed phase.

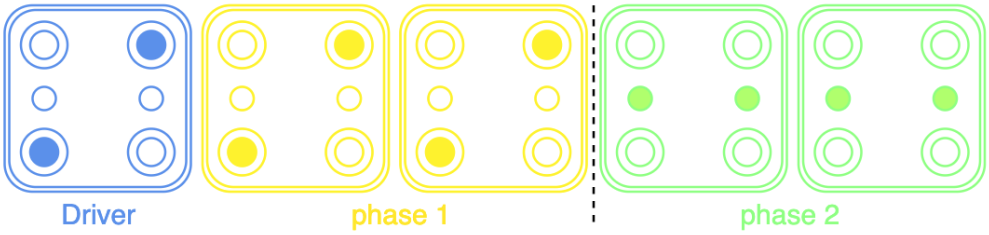


Figure 1.11: Example of information propagation from the first phase to the second one in a eight molecule wire

The clock scheme versus time can be summarized in four steps repeated for the number of information to propagate:

- **Switch**: from a RESET state, the vertical clock is applied and the charges are pushed toward *QD1 and QD2*;
- **Hold**: the clock stays applied until the QCAs have assumed a stable logic state;
- **Release**: from the APPLIED state, the clock is released toward the RESET state and the charges can move from *QD1/QD2* to *QD3*;
- **Reset**: the clock maintains the RESET state until another information to propagate arrives from the pipe;

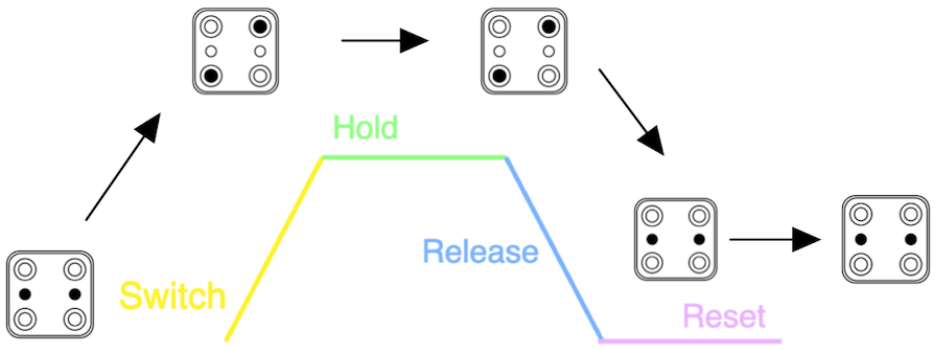


Figure 1.12: Visual scheme of a complete clock cycle with focus on QCAs states

In a multiple-phase system the information propagate following the high states of the vertical clock in a p*ipeline-like* behaviour:

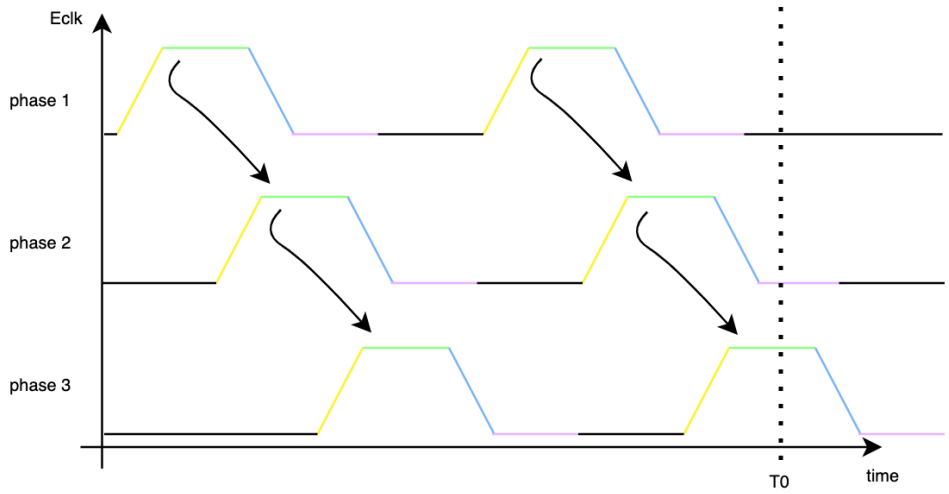


Figure 1.13: Example of a three-phase layout's clock scheme. For each repetition of the three phases, an information propagates along the complete layout

The phases do not need to cover the entire cell layout. If *n-phases* are enough to have a correct information propagation, these can be repeated *m-times* creating the mentioned *pipeline-like* system. Let's assume that the scheme in figure 1.13 refers to a layout like:

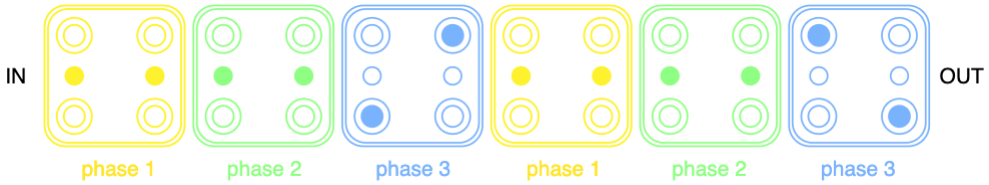


Figure 1.14: Phase repetition in a three-phase wire cell: the first input value propagated is a *'0'-logic* followed by a *'1'-logic*. The layout is a charge configuration snapshot at instant *T0*[4].

The example in figure 1.14 shows how is possible to exploit the repetition of clock phases to cover bigger layouts. For solution like these, the classical theory of pipelined systems is valid: the layout latency[5] can be defined $T = 2$, as the throughput[6] ($\frac{1}{T} = \frac{1}{2}$).

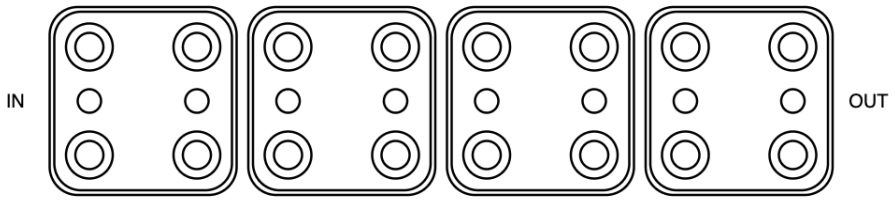
[4]Notice the colours used in figure 1.13 that are not correlated with the ones in figure 1.14. In the first case they discriminates the steps in a clock cycle (*switch*, *hold*, *release*, *reset*) while in the second one they are used to distinguish the clock phases.

[5]The latency of a system is defined as the time required for an input to become an output in terms of clock cycles.

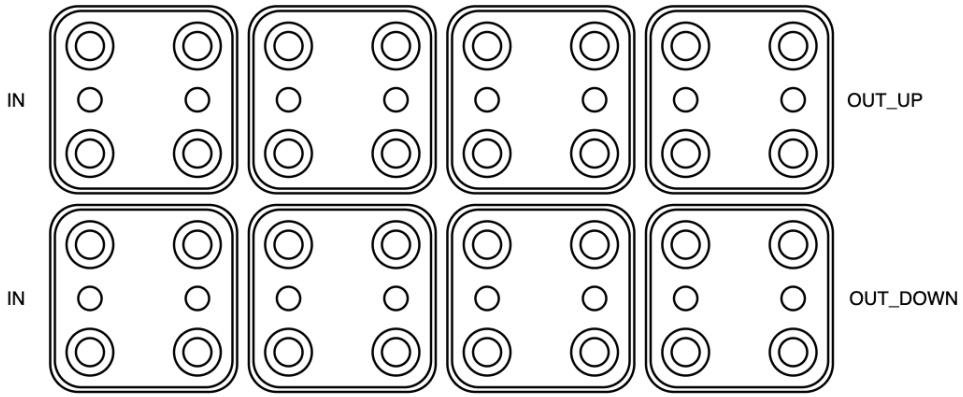
[6]The throughput of a system is defined as the number of value that is capable to compute for unit of time (clock cycle).

1.2.2 Layout improvements exploiting *bi-stability*

The layouts already presented are good in principle. To obtain a significant charge separation in molecules, it is necessary to limit the border effects noticeable in molecules that have no more than one adjacent molecule [10, 13]. For instance, the output molecule in a layout is usually weaker than a centre one: while the first one is interacting just with the previous one and has no molecule in front, the second is placed among two stable state QCAs providing good sustenance.
To achieve *bi-stability* is possible to implement the layouts using two adjacent lines instead of one:



(a)



(b)

Figure 1.15: *Mono-Phase Eight Molecules Wires*: (a) *Single-line* wire, (b) *Two-lines* or *bus* wire.

Conceptually the two wires work in the same way. Having doubled the QCAs in the second solution, the molecules are seeing neighbours on three edges and that provides the wanted charge separation and reliability.
It is also true that an adjacent QCA along the vertical axis produces an interaction and a voltage level shift not observed in single-line layouts: this effect does not invalidate the circuit behaviour and it will be discussed withing *SCERPA* simulations.

1.3 Self-Consistent ElectRostatic Potential Algorithm (*SCERPA*)

A layout realized in *MolFCN* can be seen as a certain number of points in the space on whose the charge aggregates. These points correspond to the molecule's QDs. Assuming the charges per QD as accumulated in an infinitesimal point of the space, the problem related to the charge interaction can be simplified by looking at the aggregated charge per QD of the *j-th* molecule as:

$$Q_n^j = \sum_{i=0}^{N} q_{n,i}^j \tag{1.4}$$

where n identifies the molecule QD, and i represents the point charge composing the aggregate one.
Given the set $\{Q_1^j, Q_2^j, \ldots, Q_{N_{AC}}^j\}$ for each *j-th* molecule and associating a position in the space to each of them, the voltage in a generic point r of the space can be evaluated:

$$V_j(r) = \frac{1}{4\pi\epsilon_0} \sum_{n=1}^{N_{AC}} \frac{Q_n^j}{d(r_n^j, r)} \tag{1.5}$$

The voltage generated by any aggregated charge in the layout, together with the *transcharacteristic* of the complex molecule used in the construction, enables for an information propagation model:

$$V_{in,i} = V_{D,i} + \sum_{j=1, j\neq i}^{N_{AC}} V_{j,i}(V_{in,j}, E_{clk}) \tag{1.6}$$

where:

- $V_{in,i}$ is the input voltage of the *i-th* molecule;
- $V_{D,i}$ is the voltage imposed by external drivers;
- $V_{j,i}$ is the voltage generated by all the other molecule in the layout and depends on their inputs voltages and the vertical clock known a *priori* and specified as input.

SCERPA implementation solves this self-consistent problem in an iterative way driven by the search for a compliant charge configuration on each QD of the layout. Each iteration of the algorithm is known as *SCERPA step* and depends on the previous one, following the rule:

$$V_{in,i}^k = F_i(V_{in,1}^{k-1}, \ldots, V_{in,i-1}^{k-1}, V_{in,i+1}^{k-1}, \ldots, V_{in,N}^{k-1}) \tag{1.7}$$

where F_i denotes the function 1.6.

SCERPA implementation in *MATLAB* provides a lot of functionalities for the layout simulation. It is a rich tool capable to evaluate both graphically and numerically the behaviour of a *MolFCN* circuit but it can be decisive time demanding.
The simulation time overhead of a layout depends on the number of molecules and the several options available for *SCERPA*.
In general, the simulation time increases with the *SCERPA* steps and at any of these correspond a vertical clock value.
Thus, it straight-forward the direct dependence between the number of clock cycles to be simulated and the computational time.
In figure 1.12, the clock states in a cycle have been described visually. The implementation of these in *SCERPA* is discretized in *little steps* to simulate a behaviour closer to the reality where the switch from a state to the next one cannot be instantaneous.

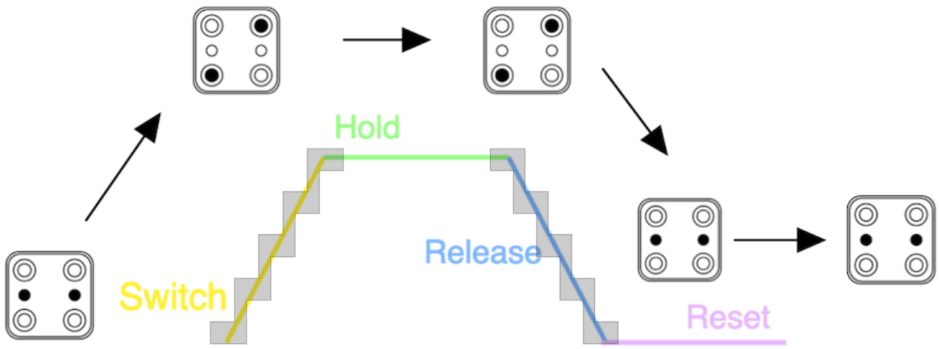


Figure 1.16: Discretization of clock states transition[1]

The use of these *intermediate values* means the necessity for the algorithm to simulate them with additive *SCERPA* steps in order to obtain a good simulation accuracy. Finer is the sweep discretization, better will be the simulation but it will require clearly a higher computational time. To mitigate the time overhead, some computational cost reduction feature as the set of custom *damping factor*, the *Interaction Radious (IR)* set and the *Active Region (AR) mode* could be considered [10].

[1]The discretization must be considered also for constant value states like the *Hold* and the *Reset* because the number of steps for each clock state must be the same for *SCERPA*.

Part II

Characterisation Tool

Chapter 2

Characterisation tool implementation

2.1 Objectives and definitions

2.1.1 A step towards VLSI

SCERPA implementation permits quasi-physical analysis of small generic input layouts as discussed before, due to the high time overhead. Looking at the possible implementation of complex systems following the modern digital design flow, *SCERPA* could be not adapt. Today, the synthesis of an *RTL*'s described system relies on libraries of gates completely characterized from the electrical and the physical point of view.
The aim of this thesis project is the creation of an additive *SCERPA*'s module for the $V_{in} - V_{out}$ characterization of generic cell layouts and the creation of libraries useful to simulate more complex architectures without paying the *SCERPA* overhead.
Knowing the behaviour of a certain basic cell, is useless to simulate it again every time a new layout is implemented. From this point of view, the use of libraries, working as fetchable tables, improves significantly the simulation efficiency[1].
The discussed additive module (*characterisation tool* from now on) is created to help the user in both the design and the library creation process of a cell layout with a development particularly aimed at the automation of every needed procedure.
The introduction of this tool would move the actual technology a step forward in the *MolFCN*'s circuit development giving the opportunity to users to have a fast look at complex implementation and work with them easily.

[1] A *SCERPA* simulation can take hours depending on how big is the cell layout and on the number of input combinations to simulate. On the other hand, the use of library would reduce that time to milliseconds

2.1.2 V_{in} and V_{out} definition

The characterisation of a generic layout requires, at first, a definition of *input voltage* (V_{in}) and *output voltage* (V_{out}) that remains consistent also where a similar layout is connected in cascade to the one under test.

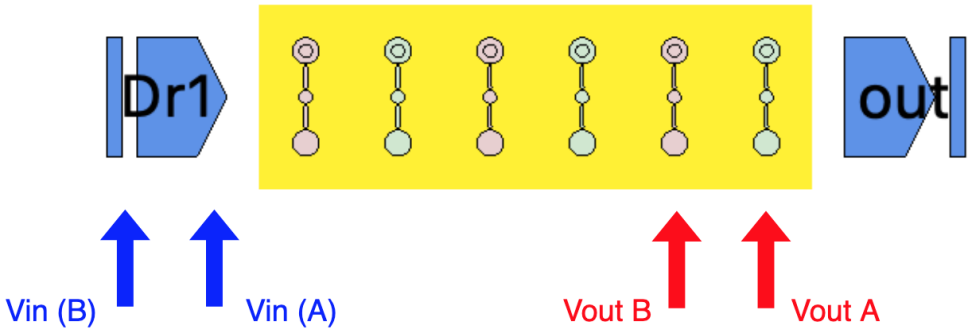


Figure 2.1: *Mono-phase Six Molecule Wire* with explicit V_{in} and V_{out} interested molecules[2]

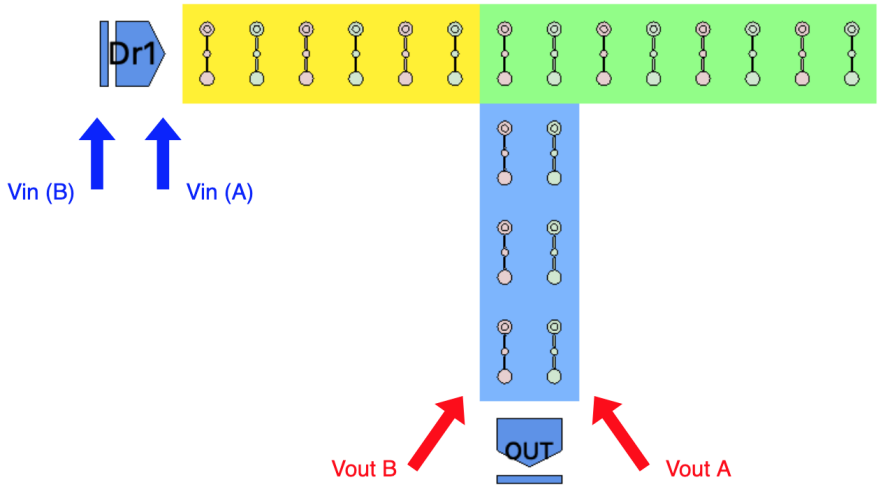
The layout represented contains three *kind of objects* that must be defined:

- **DrX labels**: driver *X-th*. The externally applied voltages to a layout are modelled through drivers with the same shape of a standard QCA. Looking at the *driver mode*:
  - **SingleMolDriverMode**: the driver label is equivalent to a unique molecule or half a QCA;
  - **DoubleMolDriverMode**: the driver label is equivalent to two molecules as it is a whole QCA.
- **QCAs**: layout body. The QCAs represent the layout and are distinguished with colours considering possible the use of different clock phases in the design.
- **OUT labels**: layout outputs. To identify where a layout finishes, there is the possibility to introduce a named label as in figure 2.1. The output labels must be always inserted during the cell layout design because these identify the output direction as well as the output themselves. *SCERPA* let's correspond a dummy molecule for each output label useful to calculate the output voltage in the proximity of the last molecule where the initial molecule of another cell could stay.

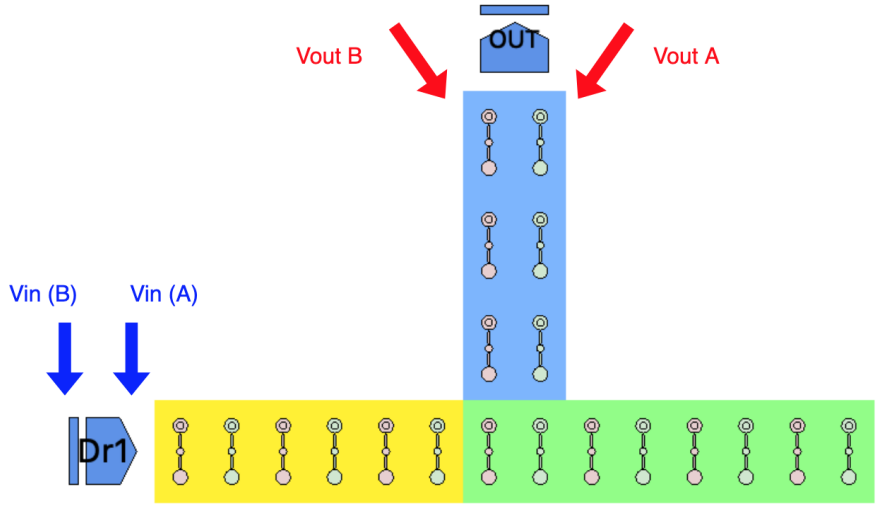
In particular, the *DoubleMolDriverMode* paradigm is the one used in the development of the characterisation tool for library files derivation. If a generic input is considered as

[2]Despite the QCAs representation seen in the *Introduction part* of this thesis work, here and so on the layouts will be represented using the software *MagCAD* [14, 15]. *MagCAD* permits to design the cell layouts graphically, providing a *"*.qll"* description file compatible with *SCERPA*.

built by two molecules in general, each layout output must rely on the same number of output molecules to ensure the definition compliance. A generic output can face three different directions basically due to the planar-based technology which is *MolFCN* [16, 17]. For this reason, a standard identification of the output molecules must be introduced to ensure consistency between the created library files and the cell layout design.



(a)



(b)

Figure 2.2: *Three-phase L-connector* with output along the *y-axis*: (a) Downward output, (b) Upward output.

As a rule to identify the molecules: the ***A*** molecule (voltage) is always the rightest one

while the ***B*** is always on his left.

As explained in section 1.2.2, there is the necessity to work on bus layouts for bi-stability. The *single-line* layouts are good in principle to get into the problem but are not robust enough to be used in real implementations.
The V_{in} and V_{out} definitions can be easily extended to bus cases:

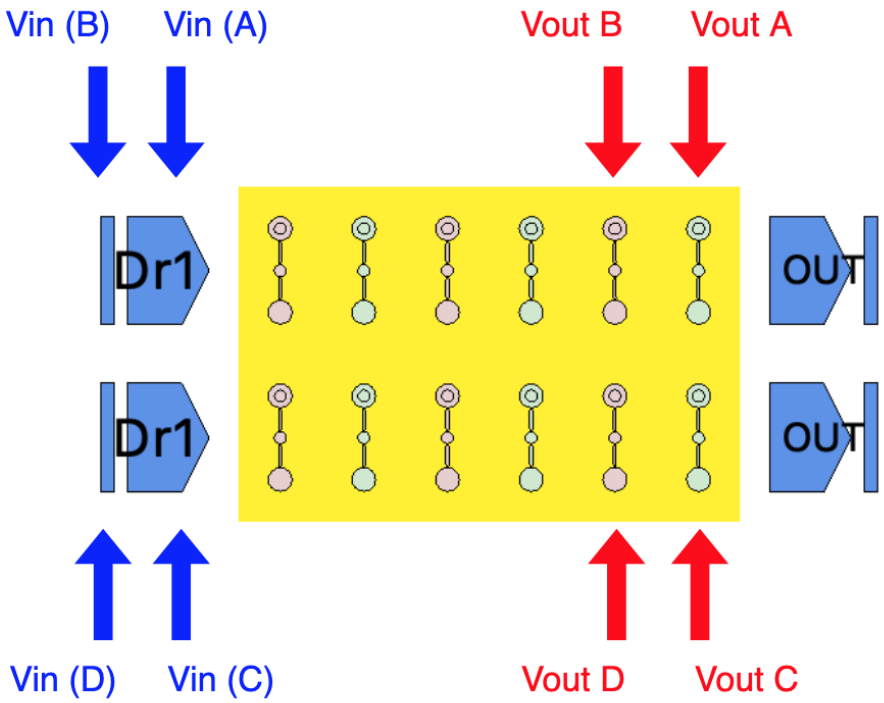
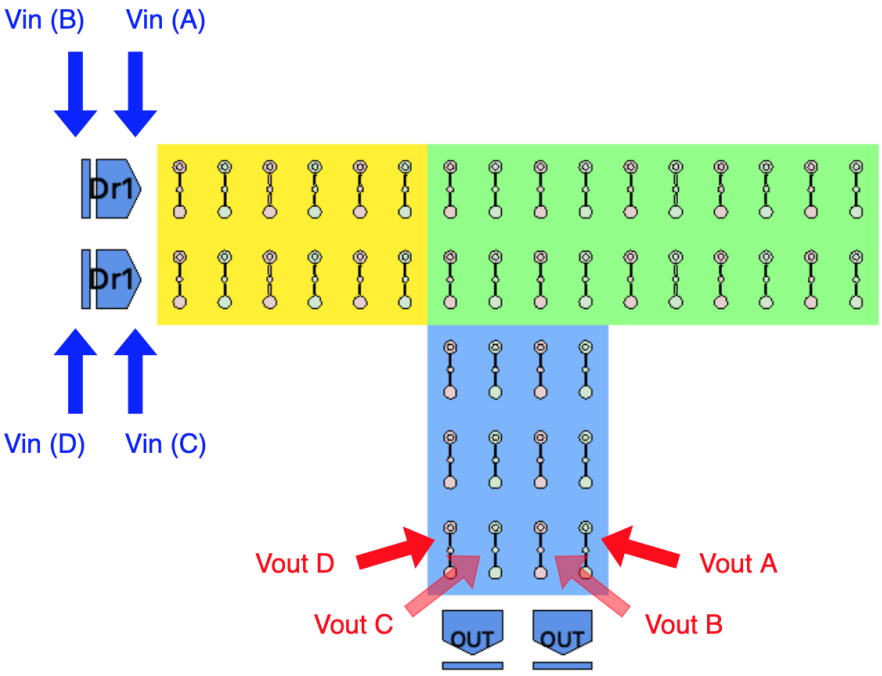


Figure 2.3: *Mono-phase Six Molecule Wire@bus* with explicit V_{in} and V_{out} interested molecules

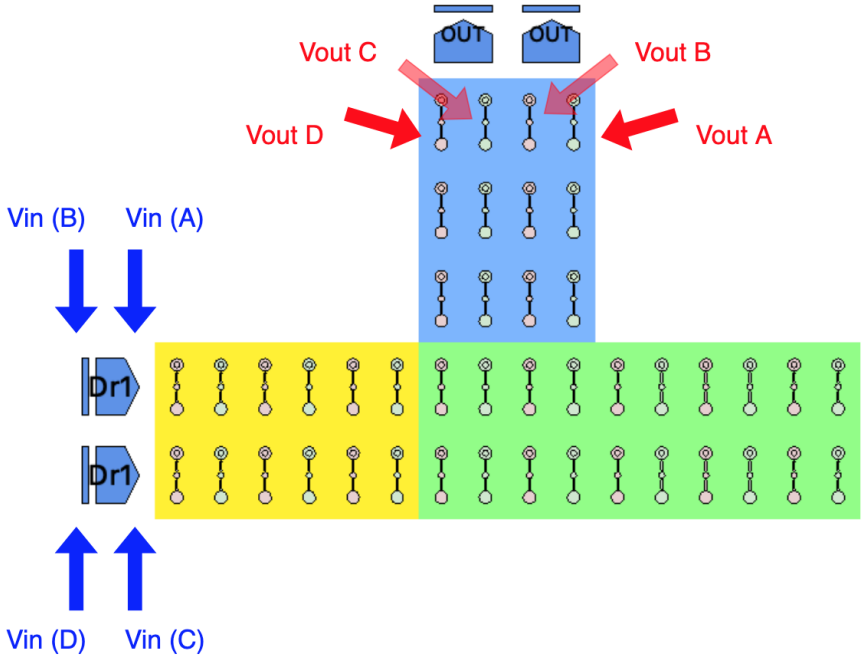
The bus paradigm doubles the molecules as the driver and the output labels. While the adjacent drivers can be the same if the layout represents a wire or different if the simulation wants to consider them as independent, the output label must be equal in name, because these represent the same and unique independent output[3].

The output molecules for cells like the ones shown in figure 2.2a and 2.2b but bus based:

[3]The correct outputs naming is one of the rules on which the *characterisation tool* is based. The definition and concepts discussed here have to be considered mandatory to have the tool working properly.



(a)



(b)

Figure 2.4: *Three-phase L-connector@bus* with output along the *y-axis*: (a) Downward output, (b) Upward output.

Also here, as a rule to identify the molecules: the ***A*** molecule (voltage) is always the rightest one preceded by the ***B*** one. On the second line ***C*** is found preceded by ***D***.
If the output molecules are adjacent like in figure 2.4a and 2.4b, the molecules ***D*** and ***C*** precede ***B*** and ***A*** and this is always true no matter about the output direction along the *y-axis*.

2.1.3 *Debug Mode* and *User Mode*

Among the available features of the *characterisation tool*, the possibility to choose between *Debug Mode* and *User Mode* is the most interesting. As discussed in section 2.1.1, this tool would help the designer in the creation and validation of the layout. Once the layout is frozen out, it can be characterized creating a library file.
The *Debug Mode* is focused on the part of creation and validation of the layout while the *User Mode* creates the library file.
In the *MATLAB* implementation that will be deeply analyzed in the next sections, the *Debug Mode* and the *User Mode* are developed as a parallel flow that exploits the same data structure and scripts.
What changes for the user is the simulation output:

- **Debug Mode**: the best way to evaluate the behaviour of a layout is the graphical one. The *Debug Mode* is designed to plot the $V_{in} - V_{out}$ characteristic given the input voltages that must be specified in the launch script[4].
  Using the Debug Mode, the definition of V_{in} and V_{out} is slightly different from what has been seen in section 2.1.2.
  To ensure good observability of the characteristics and avoid confusion, the plots must be cleaned up from clutter:

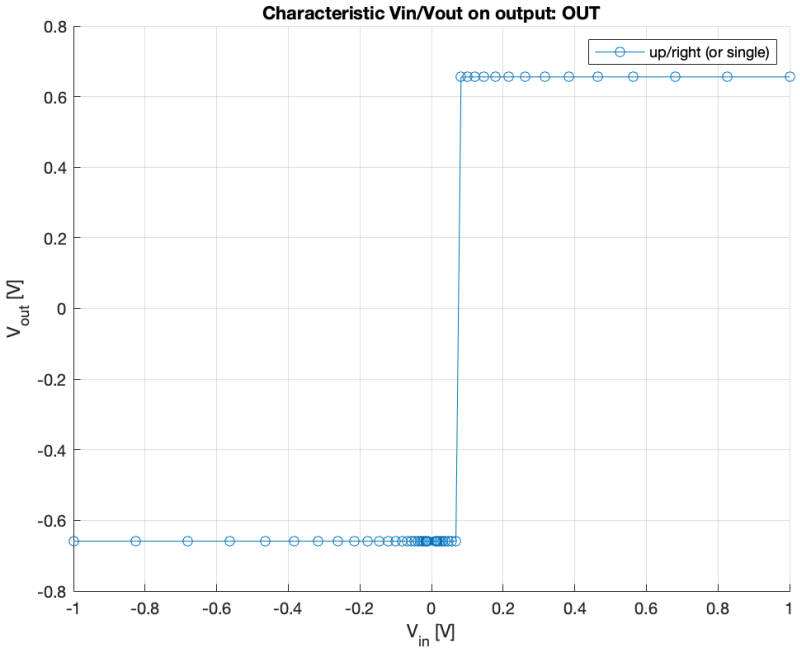
  - **Single-Line layout**: using figure 2.1 as a reference, each driver molecule corresponds to an output one. For each *input-output* couple can be introduced a V_{in}-V_{out} characteristic, meaning that plotting the voltages derived from these two couples, the output graph will contain two specular characteristics[5]. Keeping two characteristics in the same plot cannot help the user in any way so only one of these is maintained as shown in figure 2.5a.

  - **Bus layout**: the couples of molecules interested in the characterisation are now four (figure 2.3).
    Also in this case, as for *single-line* layouts, a de-cluttering operation is needed to improve the observability. Considering the doubled lines, it is interesting to evaluate two couples of $V_{in} - V_{out}$ voltages: one for the above edge and one for the down edge[6].

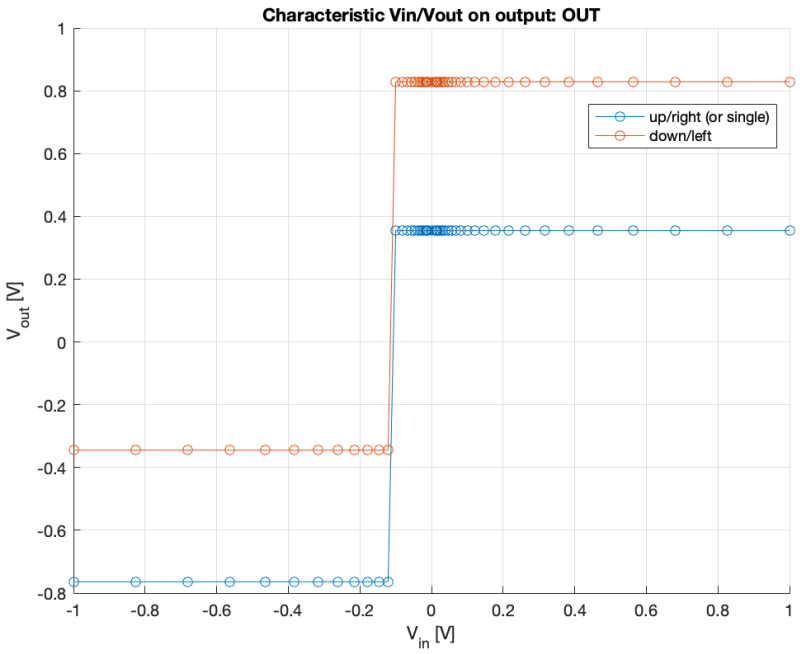
[4]The launch script is a *MATLAB* script that prepares and describes every input needed from *SCERPA* to start the simulation of a cell layout.

[5]The charges in molecules of the same QCA move in diagonal following their natural behaviour to find a minimum potential energy configuration. If the charges occupy the QDs in a specular way, also the voltages produced by them must be opposite in sign but almost equal in absolute value. This behaviour has been discussed in detail in section 1.2.

[6]The bus paradigm permits to obtain a strong bi-stability but introduces also a bias in the characteristics given by the vertical coupling as discussed in section 1.2.2. Considering the two couples $V_{in} - V_{out}$, the discussed bias can be noticed graphically as its dependence on the layout shape.



(a)



(b)

Figure 2.5: *Debug Mode* $V_{in} - V_{out}$ characteristic: (a) three-phase twenty-four molecules *single-line* wire, (b) three-phase twenty-four molecules wire@bus

The values of V_{in} to simulate must be chosen correctly in order to obtain the cell behaviour ensuring a low overhead with *SCERPA*.
One-input devices like the ones characterised above are simulated with a straightforward *sweep* in input, setting a minimum value, a maximum value and a step both for *Debug Mode* and *User Mode*.
On the other hand, *multi-input* devices need specified inputs in *Debug Mode* while every combination of the inputs must be tested in *User Mode*: the features relative to the input settings will be analyzed in detail with the implementation of the tool.

- **User Mode**: the complete characterisation of a cell requires the test of almost every possible input combination to create a library with a high coverage percentage. On the contrary of *Debug Mode*, the definitions of V_{in} and V_{out} discussed in section 2.1.2 are valid, meaning that single-line layouts need for two couples of V_{in} and V_{out} while four voltages couples are needed for bus layouts[7]. The results computed by the *characterisation tool* are stored in a *csv* file in a tabular form, making it easy to fetch for future use.
  Considering the possibility to work on multi-phase layouts where the clock phases may repeat as reported in figure 1.14, every *csv* comes with an information file reporting the output phase as the layout latency. This file still considers a little information but is designed with the idea of being more complete in future.
  The discussed table of values is created dynamically with as column as input/output molecules have to be considered and as rows as input combinations have been tested on the cell.

Table 2.1: Head of a *csv* library file related to a mono-phase six molecules wire as the one shown in figure 2.1

$Dr1$	$Dr1_c$	V_{outB}	V_{outA}
-1	1	0.65521	-0.6587
-0.8254	0.8254	0.65506	-0.65884
-0.68129	0.68129	0.65504	-0.65885
-0.56234	0.56234	0.65504	-0.65885
-0.46416	0.46416	0.65516	-0.65873
-0.38312	0.38312	0.65539	-0.65852
-0.31623	0.31623	0.65556	-0.65837
-0.26102	0.26102	0.65569	-0.65825
-0.21544	0.21544	0.65535	-0.65826
⋮			

[7] Assuming a *one-input* - *one-output* device as driving example

The table shown above has been derived using a logarithmic sweep from $-1\ V$ to $1\ V$ [8]. $Dr1$ is the closest driver molecule to the first molecule of the layout while $Dr1_c$ is complementary to it due to the *DoubleMolDriverMode* discussed in section 2.1.2.
The characterisation of the equivalent bus layout in *User Mode* provides four output voltages instead of two:

Table 2.2: Head of a *csv* library file related to a mono-phase six molecules wire@bus as the one shown in figure 2.3

$Dr1$	$Dr1_c$	V_{outB}	V_{outA}	V_{outD}	V_{outC}
-1	1	0.35428	-0.76366	0.82863	-0.34483
-0.8254	0.8254	0.35392	-0.76368	0.82861	-0.34508
-0.68129	0.68129	0.35394	-0.76369	0.82858	-0.34508
-0.56234	0.56234	0.35411	-0.7637	0.8285	-0.34502
-0.46416	0.46416	0.35454	-0.76372	0.82836	-0.34483
-0.38312	0.38312	0.35471	-0.76374	0.82832	-0.34479
-0.31623	0.31623	0.35487	-0.76377	0.82836	-0.34476
-0.26102	0.26102	0.355	-0.7638	0.82839	-0.34473
-0.21544	0.21544	0.35511	-0.76382	0.82841	-0.34471
-0.17783	0.17783	0.3552	-0.76383	0.82844	-0.34469
			⋮		

The inputs in table 2.1 remain two because the bus is driven by two identical drivers. In these cases is useless to store two others identical columns considering equal the datasets applied as input.
If the bus layout is driven at the same physical input with two different drivers, they will be treated correctly with the stamp of other inputs columns highlighting the complete input combination for each row.

The user has to choose how the simulation must be performed through flags in the *launch script*. Although the number of flags is minimal to reduce the user effort in the tool usage, the choice for *Debug Mode* or *User Mode* must be set manually.

[8] The possibility to choose between a logarithmic sweep or a linear one will be described in the next sections. The logarithmic sweep permits the input values accumulation where the transaction is more likely to be, improving the characteristic accuracy without paying the feature in terms of time overhead.

2.1.4 Basic Methodology

The *characterisation tool* provides numerous features like the different running modes presented in the previous section. Both these two can be exploited to introduce a standard methodology for cell developing, layout validation and characterisation:

1. Prepare a preliminary layout for the DUT;
2. Launch *SCERPA* within *Characterisation tool@Debug Mode* and evaluate the behaviour;
3. Layout fine-tuning;
4. Launch *SCERPA* within *Characterisation tool@User Mode* to create the library files;
5. Use the library with a *netlist-like* method.

The proposed scheme provides good results regarding the necessity for a layout to be validated before starting with a long simulation for the characterisation.
While the *Debug Mode* uses just a sweep as V_{in} giving an outcome in a relatively short time, the *User Mode* is designed to test every input combination resulting certainly in a longer simulation.
The cell layouts proposed from now on have been designed and simulated using the discussed scheme.

2.2 Function logic, features and implementation

The *characterisation tool* that will be discussed, is implemented in *MATLAB* following a modular approach. The description of every choice, feature and function will be introduced following a driving example like the characterisation of a *Three-Phase Majority Voter@bus*.

2.2.1 The *launch script* standard

The *launch script* concept was introduced before as the one involved in the simulation preparation for *SCERPA* and now, also for the *characterisation tool*.
For each cell layout to simulate, a launch script must be designed also if there are only few differences from one to the others. However, the skeleton is common to every launch script and can be summarized in principle:

- **Layout Definition**: the file *qll* and the parameters for *SCERPA* to treat with the layout are indicated here;
- **Absolute Paths Definition**: to ensure multi-platform compatibility, the automatic definition of the absolute paths is considered as a standard;
- **Driver values Definition**: the simulation runs on certain input values that are defined here;
- **Clock Definition**: the specified input values must be treated correctly with a strict correspondence with the clock phases. The clock phases and their synchronization are introduced here;
- **Layout Termination**: because of *bi-stability*, a custom termination can be automatically connected at the output of the layout. This optimization is considered here;
- **SCERPA**: when everything is ready for *SCERPA*, it can start with the simulation;
- **Characterisation**: after that *SCERPA* have concluded with the simulation, the characterisation can start using the computed results and the input parameters.

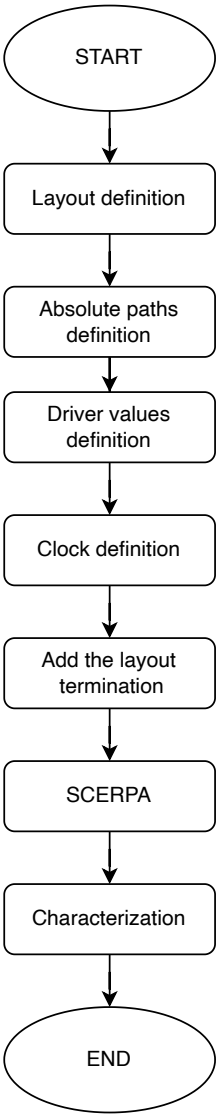


Figure 2.6: Launch script flowchart

2.2.2 Layout Definition

The first important task of the launch script is the layout definition.

Listing 2.1: Layout Definition

```
1  %% Layout settings
2
3  %molecule
4  circuit.molecule = 'bisfe_4';
5
6  %layout
7  circuit.magcadImporter = 1;
8  file = 'majority_voter_large.qll';
9  circuit.qllFile = sprintf('%s/%s',pwd,file);
10 circuit.doubleMolDriverMode = 1;
11 circuit.phases_repetition = 1; % The layout is given by the
       repetition of n-phases per how many times?
12 circuit.debugMode = 1; % - characteristic visually plotted instead
        of tabled
13 circuit.LibEvaluation = 0; % Evaluate the behaviour starting from
       the library
14
15 %debugMode LibEvaluation
16 %   0           0          --> Characterize the layout creating
       the .csv
17 %   0           1          --> Use the library instead of SCERPA
       to eval Vout
18 %   1           0          --> Evaluate the layout correctness (
       plot the characteristic)
19 %   1           1          --> Test every input combination with
       libraries (InOut_eval.m)
```

The modular nature of *SCERPA* as the *characterisation tool* need for data structure valid for sharing purposes. *MATLAB* provides the *struct* object type as a heterogeneous structure on which the fundamental elements for the processing can be stored.
The launch scripts rely on many of these structures as *circuit* that contains the device information.
With the listing 2.1 as a reference, in the *layout definition* section are defined:

- The molecule used to synthesize the layout $\longrightarrow$ *line* 4;
- The name and the absolute path of the *qll* surrounded by the *magcadImporter* flag, indicating whenever the layout is described through the *qll* file or directly into *MATLAB* $\longrightarrow$ *lines* $7 \div 9$;
- The driver paradigm to be used. It must be the double one for characterisation $\longrightarrow$ *line* 10;

- The latency of the layout (*phase_repetition*) $\longrightarrow$ *line* 11;
- The running mode (*Debug Mode* or *User Mode*) $\longrightarrow$ *line* 12;
- The behaviour evaluation through the previously extracted layout library instead of *SCERPA* (for debugging purposes only) $\longrightarrow$ *line* 13.

The last three discussed items have been introduced to work with the characterisation with respect to the original launch script for *SCERPA*.
In particular, the latency cannot be computed looking at the *qll* file because of the lack of information.
The choice for the *Debug* or *User Mode* must be provided by the user following the scheme discussed in section 2.1.4 while the *LibEvaluation* would not be used in characterisation.
How can be noticed from lines $15 \div 19$, the *debugMode* flag and the *LibEvaluation* are used to identify four different uses of *SCERPA* within the *characterisation tool*:

Table 2.3: *debugMode* and *LibEvaluation* flags operation modes

DebugMode	*LibEvaluation*	*Behaviour*
0	0	Layout characterisation in *User Mode*
0	1	Evaluate the output through the library instead of *SCERPA*
1	0	Layout characterisation in *Debug Mode*
1	1	Library debug: test every input combination fetch on the library

2.2.3 Absolute Paths definition

MATLAB is a multi-platform software and a script in this programming language would run on every operating system (Microsoft Windows, UNIX).
The modular structure of *SCERPA* as the one of the *characterisation tool* are deployed as a main directory and a lot of subdirectories including the ones for the algorithm, the information plot, the simulations and, from now on, the library together with the characterisation.
The launch script and *SCERPA* itself move inside the main directory to fetch every function needed and the absolute paths make it feasible. This paradigm gives the flexibility looked up by the user to manage with files and the directories in is own system without worring about path compliance.

Listing 2.2: Absolute Paths definition

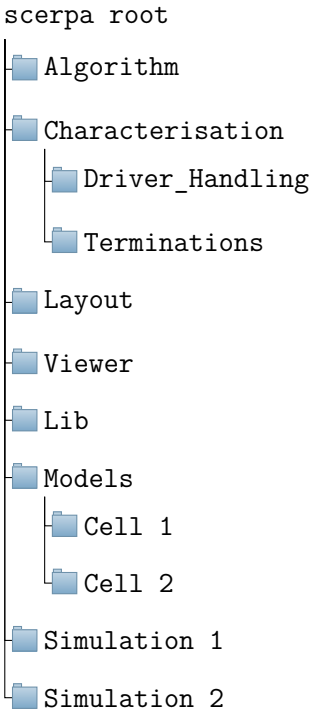
```
layout_path = pwd;

cd('../../')
scerpa_path = pwd;

characterization_path = strcat(scerpa_path, '/Characterization');
driver_handling_path = strcat(characterization_path, '/
    Driver_handling');
termination_path = strcat(characterization_path, '/Terminations');
Library_path = strcat(scerpa_path, '/Lib');
```

The basic directory tree used for simulation is shown on the right. The absolute paths are referring to a structure organized in that way:

- **Algorithm**: *SCERPA* implementation scripts;
- **Layout**: *SCERPA* scripts for the layout import and handling;
- **Viewer**: *SCERPA* scripts for the layout import, handling and data structures;
- **Characterisation**: scripts of the *characterisation tool*, driver handling and termination handling;
- **Lib**: cells libraries extracted;
- **Models**: cells layouts and relative launch scripts.



2.2.4 Driver values and clock definition

Having introduced the cell layout under test, the next step considers the input values and the clock definition. *SCERPA* handles the input combinations through a cell-matrix (*Values_Dr*) defined in the launch script. It must be compliant with the clock definition because of the strict relationship between inputs propagation and clock phases, as discussed in section 1.2.1.
A clock cycle is defined by the states *Switch*, *Hold*, *Release* and *Reset* and each of them is discretized in steps in order to implement them on *MATLAB*.
The step granularity can be defined through the *clock_step* parameter in the launch script:

Listing 2.3: Clock states structure definition

```
%% Input signals and clock settings

%definitions
circuit.clock_low = -2;
circuit.clock_high = +2;
circuit.clock_step = 5; %how many values for the 'p'?

%Step simulation implementation
circuit.pSwitch =  linspace(circuit.clock_low, circuit.clock_high,
    circuit.clock_step); % if step = 1 -> [-2 -1 0 1 2]
circuit.pHold =   linspace(circuit.clock_high, circuit.clock_high,
    circuit.clock_step); % if step = 1 -> [2 2 2 2 2]
circuit.pRelease =  linspace(circuit.clock_high, circuit.clock_low
   , circuit.clock_step); % if step = 1 -> [2 1 0 -1 -2]
circuit.pReset =   linspace(circuit.clock_low, circuit.clock_low,
   circuit.clock_step); % if step = 1 -> [-2 -2 -2 -2 -2]

%Cycle to simulate
circuit.pCycle = [circuit.pSwitch circuit.pHold circuit.pRelease
   circuit.pReset]; % if step = 1 -> [-2 -1 0 1 2 -> 2 2 2 2 2 ->
    2 1 0 -1 -2 -> -2 -2 -2 -2 -2 ]
```

A finer grain improves the simulation accuracy increasing the time overhead because of the correspondence between a value in the *pCycle* structure and a *SCERPA step*. For each *pCycle*, an input dataset propagates correctly along a clock phase of the cell.
Once the structure of a clock cycle is defined, the combination of the inputs can be introduced considering that each of them must be stable at the cell's input for the entire duration of a clock cycle (20 *SCERPA* steps in the code snippet).

Assuming to set to 2 V the input of a *mono-phase six molecules wire* like the one in figure 2.1, the *Values_Dr** structure would be:

Listing 2.4: *SingleMolDriverMode* input values definition

```
Dr2V = num2cell( 2 * ones(1, length(circuit.pCycle) ) );
circuit.Values_Dr = {
        'Dr1' Dr2V{:} 'end'
};
```

If the *DoubleMolDriverMode* is active, also the values for the complementary driver must be defined:

Listing 2.5: *DoubleMolDriverMode* input values definition

```
Dr2V = num2cell( 2 * ones(1, length(circuit.pCycle) ) );
Dr2V_c = num2cell( -2 * ones(1, length(circuit.pCycle) ) );
circuit.Values_Dr = {
        'Dr1' Dr2V{:} 'end'
        'Dr1_c' Dr2V_c{:} 'end'
};
```

So far, the structure of a complete clock cycle and the input values have been introduced. At least, the strategy and the clock phases for the layout to evolve among the clock states need to be defined. *SCERPA* uses a matrix (*stack_phase*) as the structure to share the information about the layout clock phases:

- ***stack_phase* rows**: each row describes the behaviour of the relative clock phase:
  - first row $\longrightarrow$ first clock phase;
  - second row $\longrightarrow$ second clock phase;
  - $\vdots$
  - *n-th* row $\longrightarrow$ *n-th* clock phase.
- ***stack_phase* column**: instant clock value for each phase in the layout[9].

*In *Values_Dr* are defined the drivers' values per molecule. The names need to be compliant with the labels used in the *qll* file that describes the cell layout, considering the underscore followed by the character 'c' as the method to define the complementary molecule values.

[9] A *SCERPA* step corresponds to each *stack_phase* column.

The *stack_phase* for the *mono-phase six molecules wire* considered as example would be:

Listing 2.6: *stack_phase* definition for the *mono-phase six molecules wire* with a single input value

```
1    circuit.stack_phase(1,:) = [circuit.pCycle];
```

The examined example represents the simplest setup configuration and it is easy to make it complex. Considering the driving example chosen for this presentation, the *three-phase majority voter@bus* with a layout:

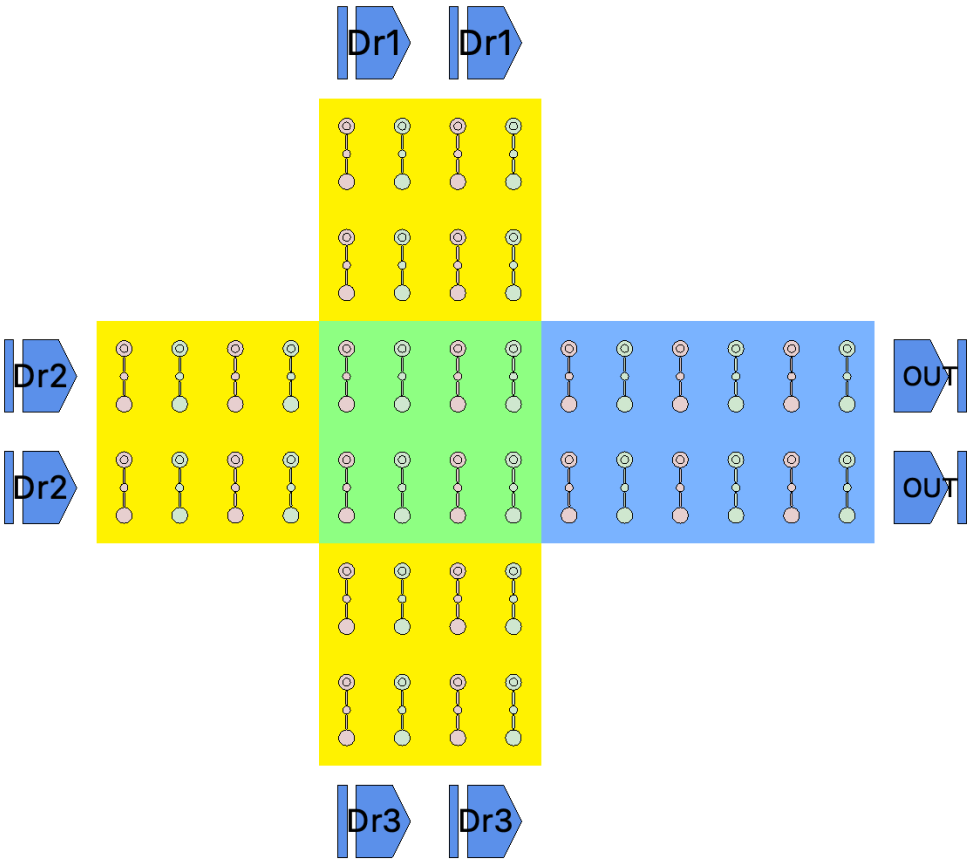


Figure 2.7: *Three-phase Majority Voter@bus* layout

The three independent inputs and clock phases require particular attention in the starting setup configuration. While the clock states structure shown in listing 2.3 can remain the same also in this case, both the *Values_Dr* and the *stack_phase* would be re-designed.

Assuming to set every cell input to 2 V, considering the *DoubleMolDriverMode* active:

Listing 2.7: *DoubleMolDriverMode* input values definition

```
Dr2V = num2cell( 2 * ones(1, length(circuit.pCycle) ) );
Dr2V_c = num2cell( -2 * ones(1, length(circuit.pCycle) ) );
circuit.Values_Dr = {
        'Dr1' Dr2V{:} 'end'
        'Dr1_c' Dr2V_c{:} 'end'
        'Dr2' Dr2V{:} 'end'
        'Dr2_c' Dr2V_c{:} 'end'
        'Dr3' Dr2V{:} 'end'
        'Dr3_c' Dr2V_c{:} 'end'
};
```

The *stack_phase* has to consider the three clock phases used in the layout design and also the order of them to ensure the propagation:

Listing 2.8: *stack_phase* definition for the *three-phase majority voter@bus* with a single input value

```
circuit.stack_phase(1,:) = [circuit.pCycle circuit.pReset
    circuit.pReset];
circuit.stack_phase(2,:) = [circuit.pReset circuit.pCycle
    circuit.pReset];
circuit.stack_phase(3,:) = [circuit.pReset circuit.pReset
    circuit.pCycle];
```

With the shown *stack_phase* configuration, the input can move at first along the *first phase* while *phase two* and *phase three* are in *Reset state*. After the first clock cycle, the information propagates on the *second phase* (the *MV* core that decides for the output), and only on the last clock cycle it will be available at the output while the other phases are maintained in *Reset state*.
The use of *SCERPA* to simulate an input value at a time can be an understatement requiring a discrete overhead in time for the setup manipulation. Whereas the aim of this work, a great scenario would consider the set of many input values in a pipe and the simulation of them in a *one-shot* run.
The *stack_phase* configuration gets moderately complex in this case: it has to consider as many clock cycles as the input combination that would be simulated in the pipe, so a *pCycle* repetition in line with them:

Listing 2.9: *stack_phase* definition for the *three-phase majority voter@bus* with several input values

```
1    circuit.CompleteCycle = repmat(circuit.pCycle, 1,
        driver_length);
2    circuit.stack_phase(1,:) = [circuit.CompleteCycle circuit.
        pReset circuit.pReset];
3    circuit.stack_phase(2,:) = [circuit.pReset circuit.
        CompleteCycle circuit.pReset];
4    circuit.stack_phase(3,:) = [circuit.pReset circuit.pReset
        circuit.CompleteCycle];
```

The *Values_Dr* matrix can be designed as shown below, creating a custom structure for each driver molecule and adding them in the matrix modifying the values for compliance with the clock. Considering the necessity to create new custom structures every time a new cell would be simulated and the difficulties in the synchronization between driver and clock values[10], especially when sweep intervals are considered as input, the creation of a script that manages these criticalities was fundamental.
The mentioned script named as *driver_comb_creator()* moves the difficulties of setting up the input combination to a completely automated function that creates the *Values_Dr* matrix relying on the input parameters inserted by the user.
Among the numerous features, our spotlight focuses:

- The *Debug Mode* and *User Mode* handling;
- The *SingleMolDriverMode* and *DoubleMolDriverMode* handling;
- Customisable number of independent input;
- Customisable sweep steps;
- Customisable sweep type (linear or logarithmic);
- Computational cost reduction strategies related to customisable dummy molecules and dummy inputs;
- Four possible input value can be manually forced in *Debug Mode*: *'1'-logic*, *'0'-logic*, *'sweep'*, *'not_sweep'*;
- Automatic generation of all the input combinations in *User Mode*;
- Automatic pipeline length calculation for clock definition[11].

[10]For instance, if the driver values number does not match the length of the *stack_phase* rows exactly, *SCERPA* will generate an error of inconsistency but only on the interested *SCERPA* step, meaning that hours of simulation could be thrown away because of inattention.

[11]The discussed *CompleteCycle* structure is treated by the *driver_comb_creator()* script because of data availability, getting into simplified launch scripts that see the functions as black boxes.

The function uses the structure *circuit* as input and the dedicated structure *driver_parameters* on which are defined its customisable options. The results overwrite the already created *circuit* structure considering that the new elements processed by the script need to stay inside it.

Listing 2.10: Driver value definition through *driver_comb_creator()* function

```
%% Driver definition
%Number of value with which control the variation
driver_parameters.Nsteps = 9; %Number of steps on the input sweep
driver_parameters.Ninputs = 3; %Number of physical input of the
    layout
driver_parameters.bothDriversActive = 1; %Use Dr_c as dummy for
    each input(fixed to 0)
driver_parameters.Nactive_inputs = 3; %Number of input to consider
     active (useful to debug long simulation)
driver_parameters.Nphases = 3; %original number of phases with
    which the layout was designed
driver_parameters.sweepKind = 'lin'; %sweep creation following a
    linspace ('lin') or a logspace ('log') -> Nstep adviced 50

%Definition of inputs to use in debug mode -> '1'        -> driver
    value fixed to '1'-logic;
%                                             '0'        -> driver
    value fixed to '0'-logic;
%                                           'sweep'      -> the
    driver sweep from -1 ('0'-logic) to 1 ('1'-logic);
%                                        'not_sweep'     -> the
    driver sweep from 1 ('1'-logic) to -1 ('0'-logic);
%

% In debugMode are not considered dummy inputs and dummy molecule
    in doubleMolDrivers because its
% not necessary to reduce the computational effort
driver_parameters.Dr1 = '1';
driver_parameters.Dr2 = 'not_sweep';
driver_parameters.Dr3 = '0';

cd(driver_handling_path)
circuit = driver_comb_creator(driver_parameters, circuit);
```

driver_comb_creator function

The logic behind this function is simple and can be summarized in a few steps:

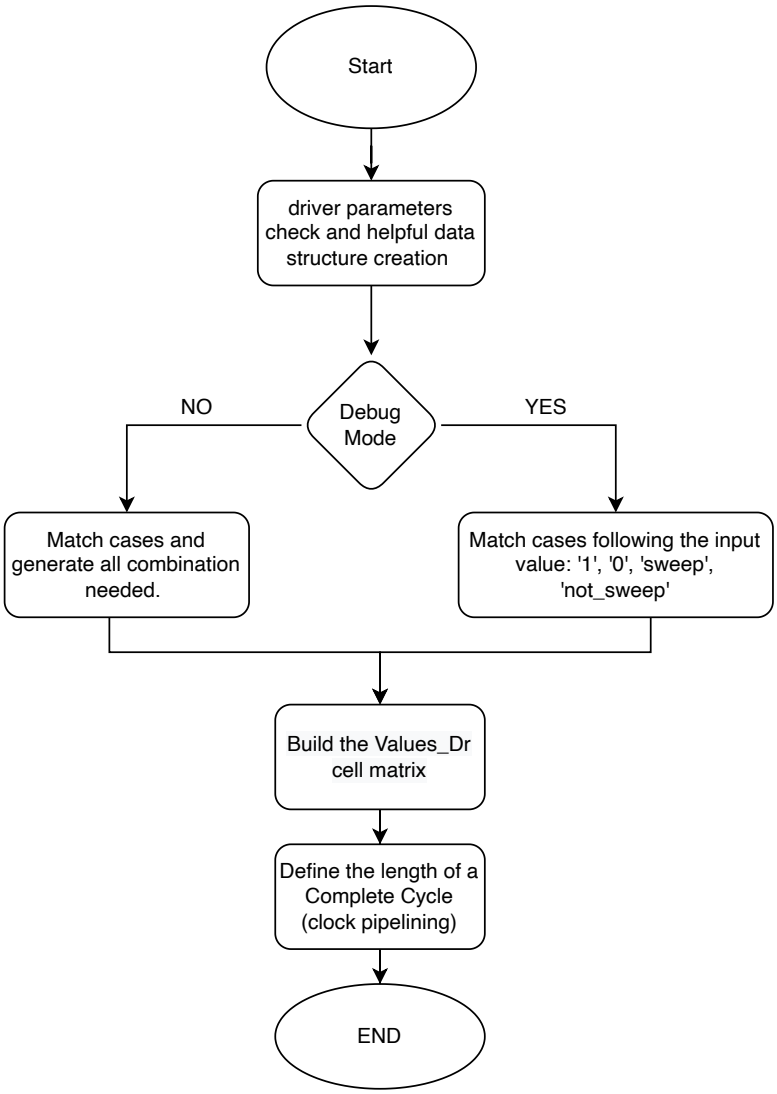


Figure 2.8: Flowchart for the *driver_comb_creator* function

The first operation considered by the function is a conformity check on the input parameters, followed by the creation of starting structures for the composition of the *Values_Dr* matrix in *Debug Mode*.
As discussed before, the length of the input values depends on the clock states granularity as on the clock phases involved in the layout design and on the latency of the layout.
This part is strictly correlated with the input parameters and it prepares the field for the next processing.
The mentioned starting structures include:

- **Sweeps**: creation of a linear or logarithmic direct *sweep* (from $-1\ V$ to $1\ V$[12] and the complementary *not_sweep*). The granularity of these vectors depends on the number of steps chosen as input.
- **Fixed values**: creation of fixed value arrays with the same length of the sweeps. At least is needed a *'1'-logic*, a *'0'-logic* and an inactive values array ($0\ V$) to implement dummy inputs and/or dummy molecules.

While in *Debug Mode* is needed a parsing with the defined value per independent input, the *User Mode* works differently.
In this case, every input values combination must be simulated over the whole cell physical inputs. The combinations are not computed randomly but rely on the input parameters, like the starting structures, through the sweep type and the number of steps chosen.
Let's assume to have a three inputs cell layout like the MV where each of them is completely active[13], considering a number of steps equal to *three* on a linear sweep, the generated direct sweep would be:

$$[-1 \quad 0 \quad 1]$$

[12] These voltages values have been derived empirically by looking at simulations, considering that overcoming these, the output voltages saturate to a stable value and is useless to increase the simulation effort for no more information.

[13] Meaning that no computational cost reduction features are active.

Considering the three inputs, the generation of every combination between the sweep vectors like the one already shown, provides:

Table 2.4: Every combination generated for a three input cell with $Nsteps = 3$ and a linear sweep

Dr1	Dr2	Dr3
-1	-1	-1
-1	-1	0
-1	-1	1
-1	0	-1
-1	0	0
-1	0	1
-1	1	-1
-1	1	0
-1	1	1
0	-1	-1
0	-1	0
0	-1	1
0	0	-1
0	0	0
0	0	1
0	1	-1
0	1	0
0	1	1
1	-1	-1
1	-1	0
1	-1	1
1	0	-1
1	0	0
1	0	1
1	1	-1
1	1	0
1	1	1

The number of combinations should be calculated before starting with the simulation, considering how is easy to make it computationally unfeasible:

$$\#Combination = Nsteps^{\#independent\_inputs} \tag{2.1}$$

The equation can be easily verified by looking at the example already shown where $3^3 = 27$ combinations have been listed.

The exponential relationship makes the number of combinations explode fastly:

Table 2.5: Combinations number in front of to cell inputs and the sweep number of step: in green[15] the conditions on whose the simulation takes no more than *0.5* hours, in yellow[15] the conditions on whose the simulation takes up to *2.5* hours, in red[14] the critical conditions on whose the simulation becomes unfeasible

Nsteps \ #input	1	2	3	4
1	1	1	1	1
2	2	4	8	16
3	3	9	27	81
4	4	16	64	256
5	5	25	125	625
6	6	36	216	1296
7	7	49	343	2401
8	8	64	512	4096
9	9	81	729	6561
10	10	100	1000	10000

The table has been filled with colours in order to identify feasible simulations that can be completed in a reasonable time[14].

- **Green**: less than *0.5* hours required by *SCERPA* to complete the simulation[15];
- **Yellow**: up to *2.5* hours required by *SCERPA* to complete the simulation[15];
- **Red**: more than *2.5* hours required by *SCERPA* to complete the simulation[15].

The time required by *SCERPA* for the simulation can be attenuated reducing the computational cost in such a way. A few features related to computational cost reduction have been mentioned in the list 2.2.4.
The flag *bothDriversActive* disables the complementary driver molecule of each driver fixing it to 0 V if the *DoubleMolDriverMode* is active. This condition permits maintaining the double molecule driver paradigm emulating the *SingleMolDriverMode*: it can be useful for debugging purposes and/or to reduce the complexity if both the molecule of each driver are considered as independent.
The best option in terms of accuracy needs, in fact, to consider the inputs as independent

[14]The characterisation in *User Mode* creates the library file needed for other simulations. The run of *SCERPA* on a certain cell layout considering several combinations must be taken into account just one time. For this reason, also the red highlighted conditions could be considered in order to achieve the requested characterisation accuracy.

[15]The simulation times considered were derived empirically on a *Three-Phase Majority Voter@bus* on a consumer mobile CPU: Intel® Core-i5 5257U 2.7GHz in single core.

with a granularity pushed to the single molecule. This approach is unfeasible with the actual *SCERPA* implementation: considering this level of granularity, each input of a bus layout would count as four independent inputs.
For instance, the characterisation in *User Mode* of a three inputs cell like the *Majority Voter* would require the combination of *twelve sweeps* to create the input dataset needed for simulation:

$$3^{12} = 531441 \;\; combinations\ with\ Nsteps = 3$$

The set of $Nsteps = 3$ is a strong understatement but also in this condition, the combinations are absolutely oversized. Setting the flag *bothDriversActive* to active in these conditions is likely to move the granularity from the single molecule to the single driver (a couple of molecules) achieving less accuracy but $3^6 = 729$ combinations[16].

The other strategy to reduce the computational cost is related to the inputs considered as a couple of molecules. For a multi-input cell, is possible to consider one or more inputs as dummies. The *driver_parameters* structure specifies the number of physical inputs for the cell (*Ninputs*) and the number of active inputs (*Nactive_inputs*). If these two numbers are different, ($Ninputs - Nactive\_inputs$) drivers will be considered dummies (fixed to 0 V). As for the flag *bothDriversActive*, in this case, the combinations will reduce drastically and with them the computational cost. In both cases, the strategies work on the exponent of equation 2.1.
Clearly, considering an input as a dummy one, does not permit the complete cell characterisation, but the strategy can help the debug with the idea of starting with the whole characterisation later.

Following whichever of the examined strategies, so far the input structures are ready to fill the *Values_Dr* matrix considering the flowchart in figure 2.8.
The next step considers the instantiation of this and the length definition of a *CompleteCycle* as anticipated in footnote 11. The *CompleteCycle* length can vary due to the characterisation running mode and the combination number to simulate: the definition inside the script solves useless data passing through the structure, making the launch script cleaner.
The *Values_Dr* matrix and the *CompleteCycle* array gets stored in the *circuit* structure that is returned by the function in order to be ready for the next processing.

[16]Fixing the complementary driver molecules to 0 V will impact on the number of combinations as the molecules are no more there.

2.2.5 Termination process

At this point of the launch script, the layout has been introduced as the driver and clock values. *SCERPA* could start without any relevant problem, however, the result of the simulation would be slightly different from what is expected.
The cell output is so far floating, giving rise to border effects as discussed in section 1.2.2. The characterisation should not be performed with the floating output because this condition is far away from its real use and would affect the behaviour.

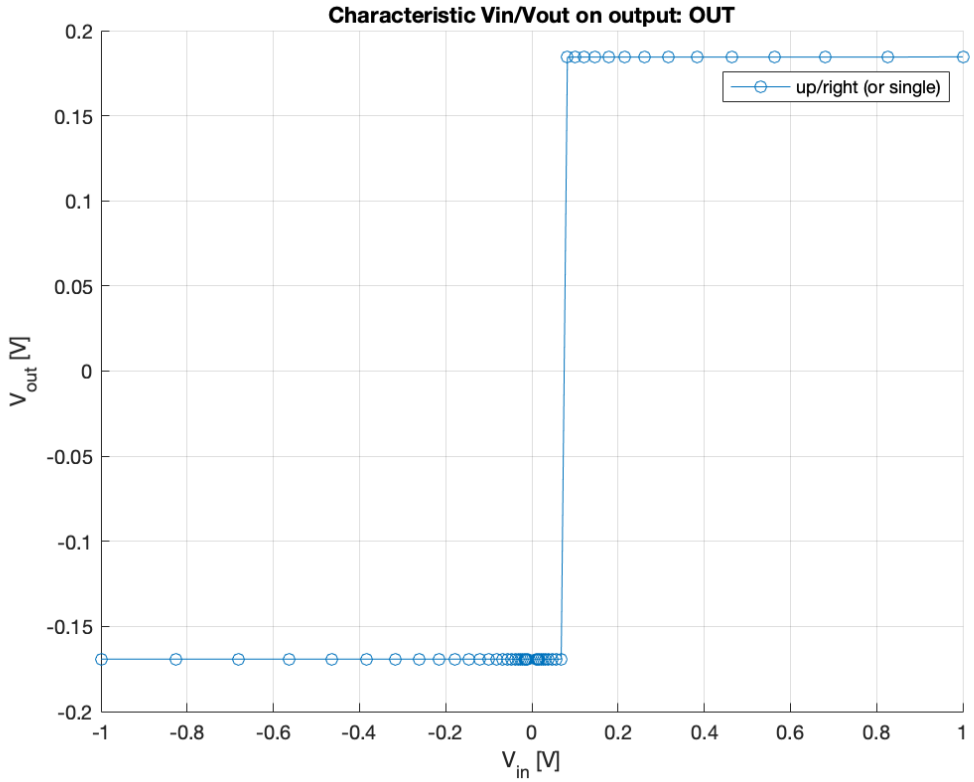


Figure 2.9: *Debug Mode* $V_{in} - V_{out}$ characteristic of a three-phase wire of twenty-four molecules with floating output

As shown in figure 2.9, the output voltage is weak while the characteristic behaviour is as expected.
Considering that the cell is designed to be implemented in complex architectures with several surrounding logic, the layout has to be simulated with something that models the outputs load like a termination.
The use of a termination permits the achievement of *bi-stability*, raising the output voltages greatly, emulating a configuration closer to the real one:

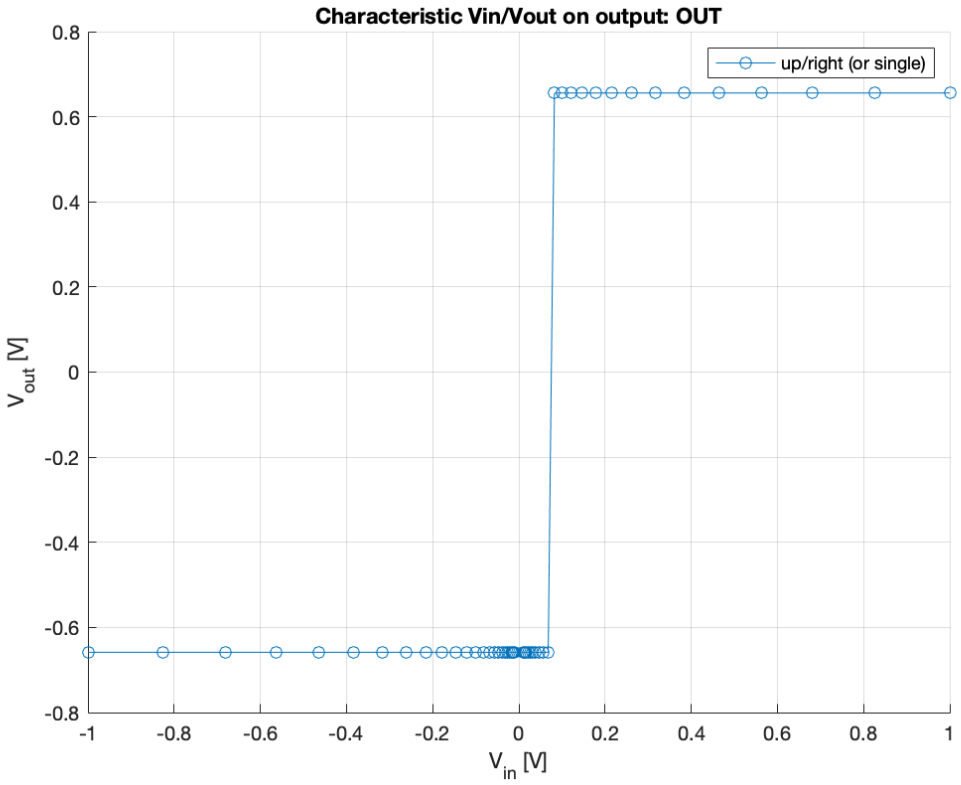


Figure 2.10: *Debug Mode* $V_{in} - V_{out}$ characteristic of a three-phase wire of twenty-four molecules terminated

Figure 2.10 has been obtained considering the layout terminated with a wire as long as its last phase[17]:

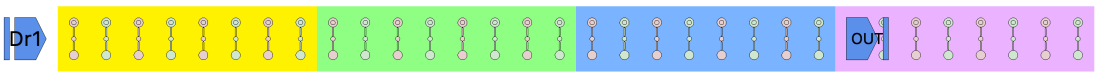


Figure 2.11: *Three-phase wire of twenty-four molecules* terminated with an *eight molecules wire* in fourth phase

[17]The termination can be customised in the launch script using parameters similar to the ones seen for the function *driver_comb_creator()*. They will be analyzed with the other function features in the next sections

The function that handles terminations is concerned in the run-time modification of the *qll* file, providing the result in a new file created from scratch: *layout_with_termination.qll*[18]. The new *qll* file is always saved in the same directory of the original one, avoiding troubles given by the use of a fixed name.
As for the *driver_comb_creator()* function, let's introduce a code snippet regarding the implementation of the termination function in the launch script and the available features:

Listing 2.11: Layout termination implementation through dedicated function

```
%% Termination settings
circuit.termination = 1; %set to '1' if you want to add a
    termination to the layout for bistability
circuit.use_custom_termination = 0; %set to '1' if you want to use
     a custom layout file to choose the termination.
                                    %If set up to '0', a number of
                                         molecule equal to the
                                        ones of the last phase
                                        will be used for the
                                    %termination. (TO DO)

circuit.use_custom_length = 0; %set to '1' if you want to use a
    custom number of mols to realize each the termination.
                                        %The number is red from
                                            the file_termination

cd(termination_path)

if circuit.termination == 1
    file_termination = '8x2_mol_termination.qll';
    circuit.qllFile_termination = sprintf('%s/%s',
        termination_path, file_termination);
end

[circuit, multiout_parameters] = add_termination(circuit); %It
    will handle internally the case circuit.termination == 0
```

[18] Avoiding the modification of the original file, a trace of the work remains in the directories with the possibility to see through *magCAD* [14, 15] the termination result in any moment.

The termination function is designed to handle whichever layout in a fully automatic way. In fact, the flags used to control the flow are just three although there are countless cases to take into account:

- The possibility to disable the termination addition;
- Design rule check for layout correctness;
- Fan-out greater than one handling;
- Single-line and bus layout handling;
- Termination addition along whichever direction (also mixed) when the fan-out is greater than one:
  - Horizontal ($angle = 0°$);
  - Vertical upward ($angle = 90°$);
  - Vertical downward ($angle = 270°$);
- Automatic or customisable termination length (read from file);
- Script ready for future customisable termination shape implementation.

add_termination function

The function involved in the layout termination is *add_termination()* that actually works together with a pre-processing sub-function for the fan-out handling.

The main function follows the logic scheme below:

- **Pre-processing for fan-out handling**: the layouts can have more than one output and each of them must be terminated in these cases. This task is not trivial and is assigned to a sub-function that will be explored later in this section;
- **Original *qll* read and design rule check**: the layout file format is an *xml* although it has is own extension. At this point, the function extracts the listed information about *QCAs*, looking for the outputs, their position, direction and labels. Contemporary, a conformity check is performed to be sure the layout is compliant with termination rules;
- **New *qll* stamp**: having the outputs information, the function is ready to re-design the layout adding the entries regarding the termination. The entries are treated as strings and printed inside a new *qll* merging them with the original ones;
- **Drivers values and clock values update**: the termination is placed at the output with a clock phase chosen to ensure *bi-stability*. This phase must be equal to the $output\_phase + 1$ as shown in figure 2.11 with the pink piece of wire overlapped to the *OUT* label. Adding a new phase to the original layout, also the clock and the driver values must be updated to permit *SCERPA* to work correctly;
- **Add the Layout Termination**: if the layout has a single output, the function has concluded its work. On the contrary, the process reiterates with the pre-processing sub-function that prepares the mid-layout to continue until the full process is finished.

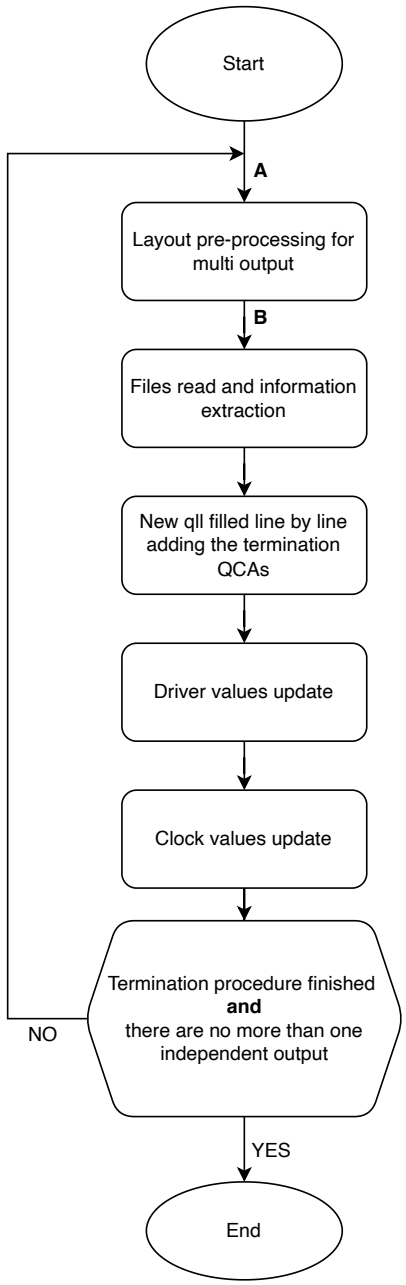


Figure 2.12: Flowchart for the *add_termination()* function

The *add_termination* function can treat an independent output at a time, single-line or bus. For this reason, the iterative method provides the termination of each output starting with the last modified *qll* to obtain a fully terminated layout. The mid-layouts are stored in temporary files named as *layout_in_process.qll* but these should not be seen by the user considering they are removed after the creation of the last *qll* file[19].
The mid-layout handling is a task of the sub-function *multiout_pre_termination_proc()* that works as a feeder for the *add_termination* function:

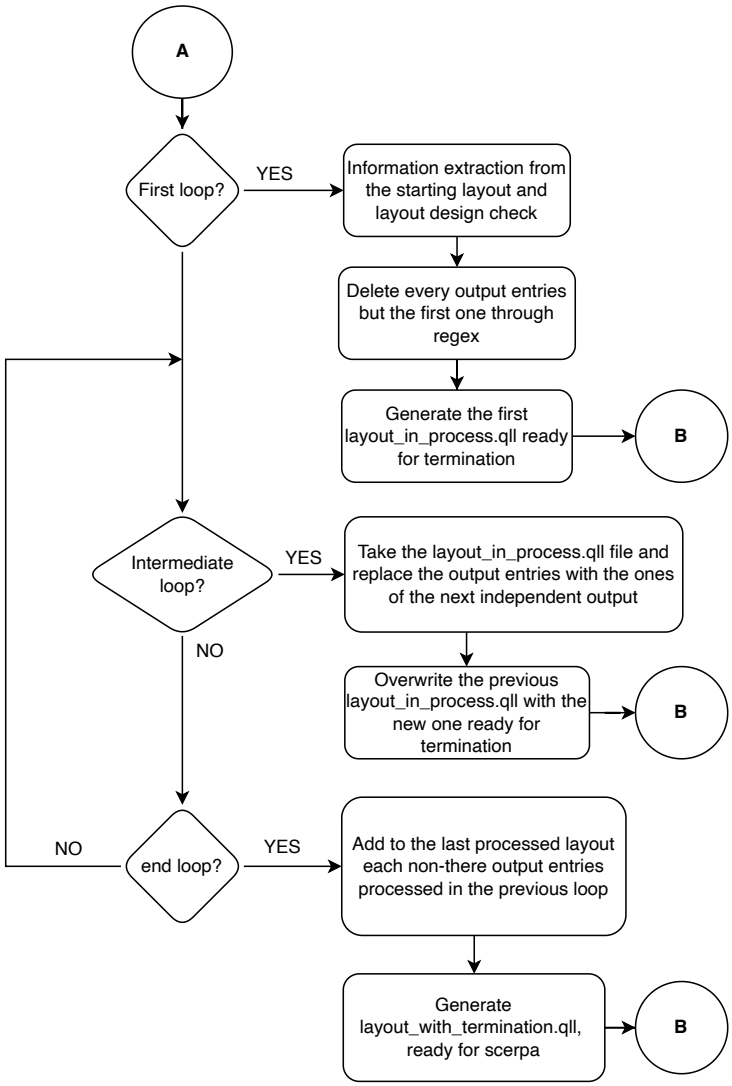


Figure 2.13: Flowchart for the *multiout_pre_termination_proc()* function

[19]The last *qll* file will be the one used by *SCERPA* for the simulation and it is named *layout_with_termination.qll*

As the flowchart shows, the sub-function *multiout_pre_termination_proc()* discriminates the actual iteration loop in three types:

- **First loop:** when the launch script calls the *add_termination()* function, the first pre-processing operation is performed and it corresponds to the *first loop* branch inside *multiout_pre_termination_proc()*.
  Considering it is the first approach with the layout, the information must be extracted to understand how many iterations are needed to complete the termination process, preparing patterns and structures to handle the *qll* file efficiently.
  Once the sub-function knows how many independent outputs has to treat, it deletes every entry referred to them but one[20] from the *qll* file, creating the first mid-layout file ready for post-processing.
  The *add_termination()* function sees the layout as a single-output one due to the pre-processing, being able to terminate it in a standard way. In the end, the process starts with another iteration that will be recognized as an *intermidiate loop*;

- **Intermidiate loop:** when the process re-iterates, the pre-processing function takes the last mid-layout created to continue with the substitution operation related to the outputs. Once the last processed output has been terminated, the sub-function can delete its label in the mid-layout file, inserting the next one in order, letting to *add_termination()* function free for another ride; This operation iterates until every output has is own termination and the mid-layout is ready to be filled again with all the labels related to the output processed;

- **End loop:** the intermediate loops have already concluded with the pre-processing and the terminations are on their position[21].
  However, the trick on the output label substitution has deleted every label so far that *add_termination()* iterates. The end loop branch of *multiout_pre_termination_proc()* will insert the mentioned missing output labels, composing the final layout file ready for *SCERPA*: *layout_with_termination.qll*.

On each *add_termination()* iteration, the new phase to be adopted for termination is calculated as $output\_phase + 1$. If the fan-out is greater than one, the output phase could be different depending on the output as the termination phase. If the termination phase is greater than the maximum one concerned in the layout, the clock matrix must be enlarged for compliance with *SCERPA* and with the clock values also the driver ones need to be stretched out as discussed in section 2.2.4.

[20] The delete and the writing operation are performed efficiently exploiting file pointers and regular expressions. One of the first operations performed by *multiout_pre_termination_proc()* is, in fact, the creation of patterns for regular expressions that permit the run of dedicated commands for matching and substitution. This operation is ensured by the regularity of *qll* files that maintain always the same structure, considering they are formatted as *xml* files.

[21] The information about the multiple outputs fill the data structure *multiout_parameters* shown in listing 2.11. This data structure is initialized inside the *add_termination()* function and is fundamental for the characterisation tool that must know how to treat the *SCERPA* extracted data.

Let's assume to work with a *stack_phase* structure equal to the one shown in the listed 2.9 where the layout is designed using three phases. The insertion of a termination in the fourth phase would consider a new row in the *stack_phase* to propagate the information also in the new layout appendix:

Listing 2.12: *stack_phase* definition for a generic *three-phase* layout terminated on phase four. In red the modifications inserted to treat with the fourth phase

```
1       circuit.CompleteCycle = repmat(circuit.pCycle, 1, driver_length);
2       circuit.stack_phase(1,:)  = [circuit.CompleteCycle circuit.pReset
            circuit.pReset circuit.pReset];
3       circuit.stack_phase(2,:)  = [circuit.pReset circuit.CompleteCycle
            circuit.pReset circuit.pReset];
4       circuit.stack_phase(3,:)  = [circuit.pReset circuit.pReset
            circuit.CompleteCycle circuit.pReset];
5       circuit.stack_phase(4,:)  = [circuit.pReset circuit.pReset circuit.pReset
            circuit.CompleteCycle];
```

The red parts identify the modification required to pass from a three-phase layout to a four-phase one: at first, a copy of the last *stack_phase* row is attached to the bottom of the matrix with a *reset state cycle* added as a prefix, then the same *reset state cycle* is copied at the end of the other rows ensuring the matrix dimensional compliance.
The *stack_phase* update is not enough because of the rows enlargement: the driver values must be stretched out to ensure the consistency required by *SCERPA* and discussed in section 2.2.4 to start with the simulation.
The stretch operation is performed by copying the last generated values in the *Values_Dr* matrix for each row to preserve the last status assumed by the layout.
Both the discussed modification are handled automatically by the *add_termination()* function that is capable to recognize the new phases used adapting the input structure for the *SCERPA* simulation.
Once the termination process concludes, the new file *layout_with_termination.qll* is pointed in the *circuit* data structure, replacing the original one instantiated in the first rows of the launch script. *SCERPA* has now everything it needs to start with the simulation, considering it is completely blind to the operation performed before his call.

2.2.6 Characterisation process

The data structures, the input parameters and the files needed by *SCERPA* for the simulation are now ready and located. *SCERPA* has is own parameters to control the simulation flow described briefly in the articles [9, 10, 18] and in detail in the dedicated documentation. From the launch script:

Listing 2.13: *SCERPA* launch

```
%% SCERPA launch
if circuit.LibEvaluation == 1
    cd(characterization_path)
    Vout = InOut_eval([-0.52 0.52 -0.52 0.52 -0.12 0.12], circuit,
        charactSettings);

else
    cd(scerpa_path)
    generation_status = SCERPA('topoLaunch', circuit, settings);
                        SCERPA('plotSteps', plotSettings);
end
```

The code snippet shows a two-way branch through an if-statement on the *LibEvaluation* flag. The functionalities of this flag were discussed in section 2.2.2: it permits to test the libraries previously extracted[22] in *User Mode*, bypassing *SCERPA*. The *InOut_eval()* function works as a table reader and extracts the interested rows from the library file, looking at the input dataset. The layout simulation through *SCERPA* relies on the function:

$$SCERPA(command,\ option1,\ option2);$$

where the command *'topoLaunch'* makes *SCERPA* simulates the layout descripted in *circuit* applying the algorithm as defined in *settings*. On the contrary, the command *'plotSteps'* makes *SCERPA* plots the information extracted by means of the previous simulation in a format specified by the *plotSettings* structure.
Once the layout simulation concludes, the information needed for characterisation are ready to be processed. This process is handled in a function named *characterization()*, used in a similar way to what has already been shown in the launch script.

[22] The function *InOut_eval()* has not to stay into the launch script if not for debugging purposes. The use of this function can be exploited in the simulation of complex architectures through the union of several libraries as will be shown later with the XOR application.

At first, the definition of some input parameters is needed to customise the function behaviour with the available features:

Listing 2.14: Characterisation function parameters for the *three-phase MV@bus*

```
%% Characterization settings
charactSettings.enableCharacterization = 1;
charactSettings.LibPath = Library_path;
charactSettings.LibDeviceName = "majority_voter_bus_3phase";
charactSettings.out_path = settings.out_path;
charactSettings.AllHoldValues = 0; %set to '1' if you want to
   plot every Vout when the output is in the Hold state. '0'
   means just the last one
```

The options are reduced to the essential considering the always true idea to have a high automation grade:

- Possibility to disable the characterisation process;
- Customisable library name, dinamically updated for multi-output layouts;
- *Debug Mode* and *User Mode* handling;
- Voltages selection when the interested molecules are in the last step of the *hold state cycle* or take all the hold state correspondent values.

The characterization function has to be launched after *SCERPA*:

Listing 2.15: *SCERPA* launch

```
if charactSettings.enableCharacterization == 1 && circuit.
   LibEvaluation == 0
    cd(characterization_path);

    characterization( circuit, charactSettings,
       multiout_parameters );
end
```

The code snippet shows the data structures needed for the characterisation process. In particular, the *multiout_parameters* structure built up during the termination process is useful for the function to understand how many independent outputs were involved in the design. Any cell can be characterised among each output, meaning that to each of them a result (a plot in *Debug Mode* or a table in *User Mode*) have to be computed by the function concerning every input[23].

[23] If many inputs are involved in the layout design, at least a plot is printed out for each independent output. The reason why the minimum amount of plots can be one is due to a control on the V_{in} variance during the plot preparation: considering the possibility to choose a constant value as input, the trivial inputs are so far skipped.

One of the options shown in listing 2.14 permits to set the library name (*LibDeviceName*): in particular, in *User Mode* this name is automatically updated attaching at the end the independent output label declared during the layout design stage. Working with multi-outputs raise the necessity to produce more than one plot or library table at each run and the strategy to avoid criticalities is the one discussed so far.
The other flags shown are used to set the library path on which save the produced data tables, the simulation path where the *SCERPA* processed data can be fetched out[24], and the extraction conditions.

Every *MolFCN* layout needs the deeply analyzed vertical clock to ensure propagation and the reasons why have been discussed in section 1.2.1. Considering the structure of a single clock cycle, the voltage on each molecule is strictly dependent on the forced clock: when the molecule is in *RESET state*, the voltage across the principal *QDs* is almost 0 V while when the clock is in *APPLIED state*, the voltage across the principal *QDs* is well defined and depends on the external applied electric field.
In this last condition also known as *HOLD state*, the information are available and can be read out but exists time steps on whose that is not true.
The characterisation process must select automatically the right time steps to derive the correct cell behaviour. These correspond to the instants on whose the interested output molecules are in *HOLD state*.

The *HOLD state* as the other states of a clock cycle is discretized, as discussed in section 2.2.4. With this in mind, there are several time steps for each clock cycle on whose the characterisation function has the green lights to read the values: through the *AllHold-Values* flag, the function behaviour can be modified to consider just the last *HOLD state* time step for each clock cycle or it can consider all the *HOLD state* time steps, increasing the number of values in the plots as in the library tables.
Having clarified the strategy on which the data collection is based, the V_{in} and V_{out} structures can be filled considering the *running mode*: in section 2.1.2 the definitions of the input and output voltages have been introduced with attention on the differences between *Debug Mode* and *User Mode*. In both cases, the input voltages maintain the same definition while the outputs voltages are taken on twice the molecules for *User Mode*. With the data structures completed, the plot of figures as the table creation is straight-forward and will be discussed, together with the logic and other details, in the next section.

[24] *SCERPA* simulation produces a table where are stored the voltages of every molecule involved in the layout for each time step. This table is accompanied by a *MATLAB* workspace with the data structures processed and created during the simulation.

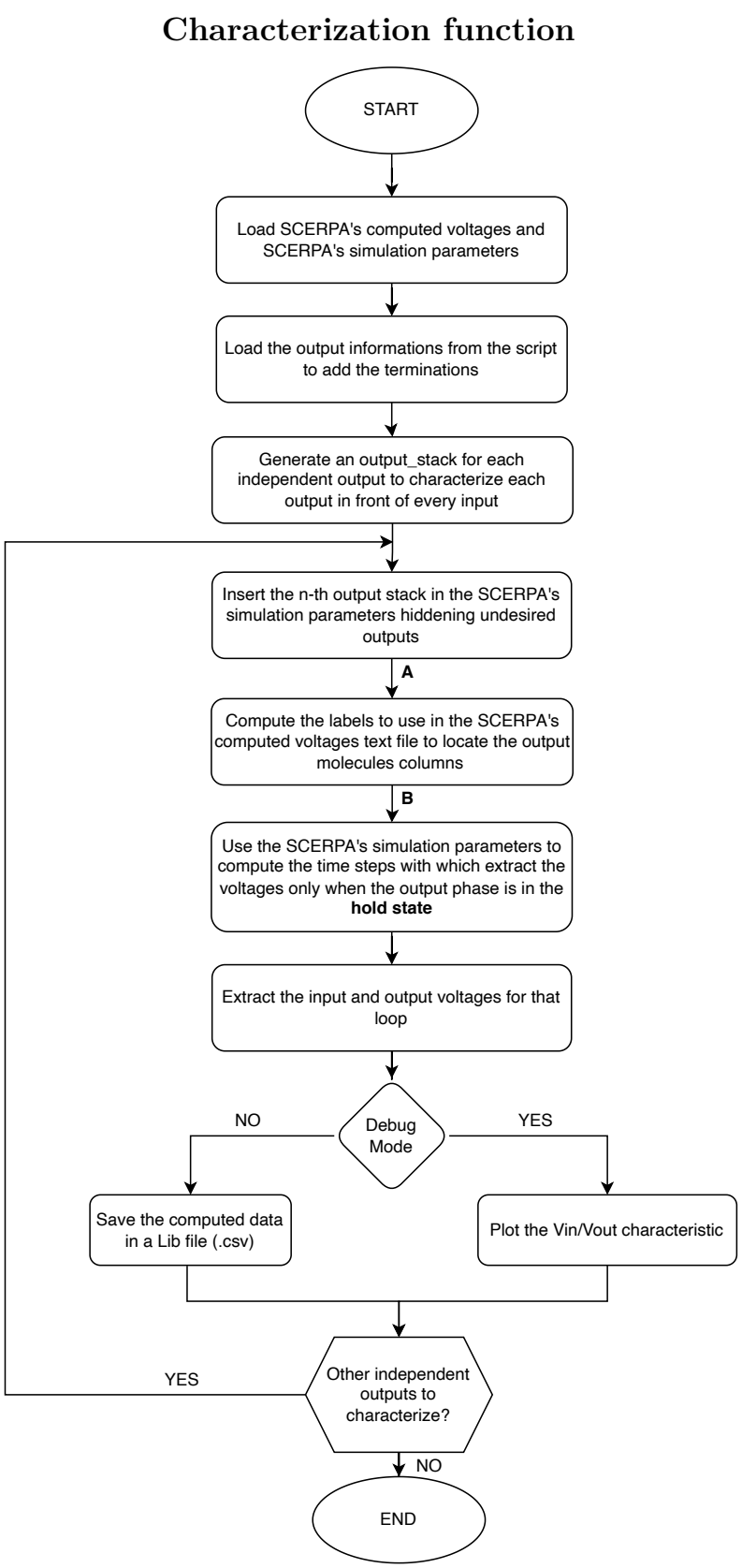


Figure 2.14: Flowchart for the *characterization* function

The function *characterization()* represents the core of the *characterisation tool*. It has to process the information extracted by *SCERPA* to obtain reliable $V_{in} - V_{out}$ graphics as tables that can be exploited in the simulation of complex architectures.
The flowchart shown in figure 2.14 is related to the *characterization()* function that works with a basic idea similar to the one seen with the *add_termination()* function.
Instead of trying to process data in a *one-shot* run, the function works on them one independent output at a time and iterates the procedure among every output until it is fully characterised[25]. For this reason, firstly is needed an initialization part of the simulation data and a preparation of the layout information for the characterisation process, then the data collection can start.
The flowchart entries can be summarized:

- **Initialization and structures preparation**: *SCERPA* permits the setting of a directory for each launch script where the simulation output is saved. On each of these directory at least a *simulation_output.mat* and an *additional_information.txt* files can be found. The first contains every data structure created and filled for and by *SCERPA* during the simulation with information about the layout, the input values simulated, the clock, the charge distribution and so on. The second is a table of output voltages organized per molecule and time step.
  The function *characterization()* loads both the files in order to extract the layout information needed and the voltages during the iterations;

- **Dummy *output_stack***: as discussed initially, the function *characterization()* processes one independent output at a time like the *add_termination()* function. At this point, the function is not aware of the layout and in particular about its outputs. Considering that *SCERPA* needs for the entire layout to pay the simulation overhead just one time, the information listed in *simulation_output.mat* refer to the full layout. The mentioned *MATLAB* workspace contains several stacks where the information about molecules are stored:

  - ***stack_driver***: contains each molecule identified as a driver molecule with labels, positions, phases and other information about their status;
  - ***stack_mol***: contains each molecule composing the layout body with labels, positions, phases and other information about their status;
  - ***stack_output***: contains the dummy output molecules that correspond to the output label inserted during the layout design with labels, positions, phases and other information about their status;

  The function *characterization()* refers to the *stack_output* to know where the output to characterise is positioned. If the layout concerns multiple outputs, the function would find them all. The idea is to modify dynamically the *stack_output* on each

[25]The meaning of "Fully characterised" would denote that a plot or a table for each independent output has been printed out.

iteration with a dummy copy that contains just one independent output at a time, permitting the extraction of every plot or table as the layout is designed with a single output. This approach needs the definition of a complete *stack_output* saved on the function start and useful to fetch the next output to process, then the characterisation can go on with a number of iterations equal to the number of independent outputs.

- **Output pattern extraction**: once the characterisation starts, the first operation required is the labels extraction to identify the output molecules inside *additional_information.txt*. This file is, in fact, organized with a label for each column in the form:

$$\begin{aligned} Drivers &\implies driver\_(ID)(a|b); \\ Body\ Molecules &\implies Vout\_(ID)(a|b); \\ Dummy\ Outputs &\implies out\_(LABEL); \end{aligned}$$

where the *(ID)* is a four-digit number. The molecules of the same *QCA* are identified by the use of the same *(ID)* followed by the character *a* or *b*. On the other hand, the *(LABEL)* field for the dummy outputs molecules correspond to the one chosen during the layout design.
To identify the correct columns to consider in the data fetch, the patterns precomputation is needed starting from the stacks discussed before and the molecules' positions. Considering the several layout configurations to predict, the operation is not trivial and it is assigned to a dedicated sub-function named as *golden_outMol_finder()* discussed later;

- **Index computation**: having identified the columns to fetch inside the voltages table, the indexes to use in the time steps selection must be computed. At least is needed the starting index and the step in order to extract every value. Both depend on several parameters like the clock states length, the clock phases involved in the design and the latency. The formula used inside the function to compute the starting index is the following:

$$\begin{aligned} index = \quad & 1 + \\ & + \ clock\_step \cdot (output\_phase - 1) + \\ & + \ clock\_step + 1 + \\ & + \ (latency - 1) \cdot clk\_cycle\_length; \end{aligned} \tag{2.2}$$

  - Starting offset for alignment with the *stack_phase*;

- If the layout is designed with more than a clock phase, the *stack_phase* composition is similar to what has been shown in listing 2.9. The output has to be on the last phase, so its available after $(output\_phase - 1)$ clock states, where each clock state has a discretized duration of *clock_step* time steps. The multiplier refers to the length of a clock state and not to the length of a clock cycle because the information propagation starts with $(output\_phase - 1)$ *RESET states* until the data is available at the output;
- With the previous terms, the clock cycles corresponding to the output phase has been selected. However, the interest is on the *HOLD state* or rather the second clock state for each clock cycle. The offset $(clock\_step + 1)$ skips the first *SWITCH state* and moves the pointer to the first value of the *HOLD state*;
- The last equation term refers to the possibility to have a bigger layout where the clock phases could repeat as shown in figure 1.14. A layout realized with this approach would have a latency equal to the number of phases repetitions from the input to the output, meaning that the first fetchable output will be available after the latency. The latency in terms of time steps is managed as a offset of $(latency - 1)$ clock cycles where each clock cycle has a discretized duration of $(4 \cdot clock\_step)$ time steps.

The starting index calculated points to the first voltage assumed by the output when it is valid and in *HOLD state*. The data fetch is provided employing an iterative strategy that read the correct voltages with a fixed step of $(4 \cdot clock\_step)$. Once the pipe is filled, the next values would be available after every clock cycle, re-marking the natural pipeline behaviour discussed in section 1.2.1.

- **Data fetch**: After the starting index computation, data can be fetched out reading each *HOLD state* values or just the last one considering the *AllHoldValues* flag. The two cases are managed differently due to the distinct necessities on the index handling. The starting index is designed to point at the first *HOLD state* value with the idea to let to the fetch structures the task to manage it:
  - **Last Hold Value**: in this case, the starting index is moved at first to the last hold value adding the offset $(clock\_step - 1)$. Then, after every fetch it gets repositioned adding the offset $(4 \cdot clock\_step)$ until the end;
  - **All Hold Values**: in this case, the starting index is not modified before the data fetch, it gets increased $(clock\_step - 1)$ times on each iteration and then updated with the offset $[(4 \cdot clock\_step) - clock\_step]$ where the decrement works as a reset for the inner counter in order to restart the process on the next iteration.

The output voltages are extracted following the indexes with the discussed strategy. They are saved with respect to rows in a dedicated matrix value with an order that depends on the library file creation, avoiding the necessities for any post-processing of the matrix. On the contrary, the input voltages are extracted more efficiently from the *SCERPA* workspace considering the definition discussed in section 2.1.2.

- **Store operation**: Once the data are ready in dedicated structures, the function decides to plot them or list them in a *csv* file.

  - ***Debug Mode***: every V_{out} of the layout is plotted in front of the V_{in}, after a de-cluttering operation. As discussed in section 2.1.2, the only molecules considered are the rightest ones whichever is the output direction;
  - ***User Mode***: the V_{in} and V_{out} matrix are merged to compose an ordered table, printed as a *csv* file with the name chosen in the launch script. The *csv* is not the only file printed. Considering the necessity to provide further information with the characterised cell, a *text* file with the same name and the same *csv* formatting is included on the characterisation output.
    This file so far contains the latency and the output phase for the layout but is designed to include whichever information could be necessary for the future.

  Once the computation has concluded, the function can iterate on the characterisation of the next independent output or it can exit.

During the steps description for the function *characterization()*, a fundamental sub-function for the output pattern extraction has been mentioned: *golden_outMol_finder()*. This sub-function aims to fetch the labels correspondent to each output molecule in order to fetch the correct voltages later into the *additional_information.txt* file.
The idea on which is based the sub-function is the use of the output labels inserted during the layout design to extract the position of the outputs.
One of the design rule checks done in this sub-function controls the conformity of the output labels. These must be inserted properly to have a working *characterisation tool*. In particular, are fundamental the direction of each independent output and the names that must be used for both the labels with bus layouts.
From the output label's angle, the sub-function recognizes the output direction and the output molecules' position. On the contrary, from the name the *characterisation tool* discriminates independent outputs from the dependent ones.
Once the output label is available, the sub-function looks for the output molecules considering the position coordinates[26] and the direction.
The output can be directed in three directions:

- Horizontal ($angle = 0°$): the adjacent output molecules must be searched varying the x-coordinate;
- Vertical upward ($angle = 90°$): the adjacent output molecules must be searched varying the y-coordinate and in particular, increasing it;
- Vertical downward ($angle = 270°$): the adjacent output molecules must be searched varying the y-coordinate and in particular, decreasing it;

[26] In the design stage with *MagCAD*, a position in terms of coordinates is associated to each placed molecule. The coordinates are indicated as a dataset [*x-coord*, *y-coord*, *layer*] and are imported by *SCERPA* from the *qll* file in the workspace.

Once the outer output molecule is found, the other ones are searched with the same strategy, taking the already found molecule position as a reference and testing the others in the stack. If the layout follows the bus paradigm, the sub-function handles the output labels as a couple without modifying the strategy.
The logical behaviour of this sub-function is shown in the next flowchart:

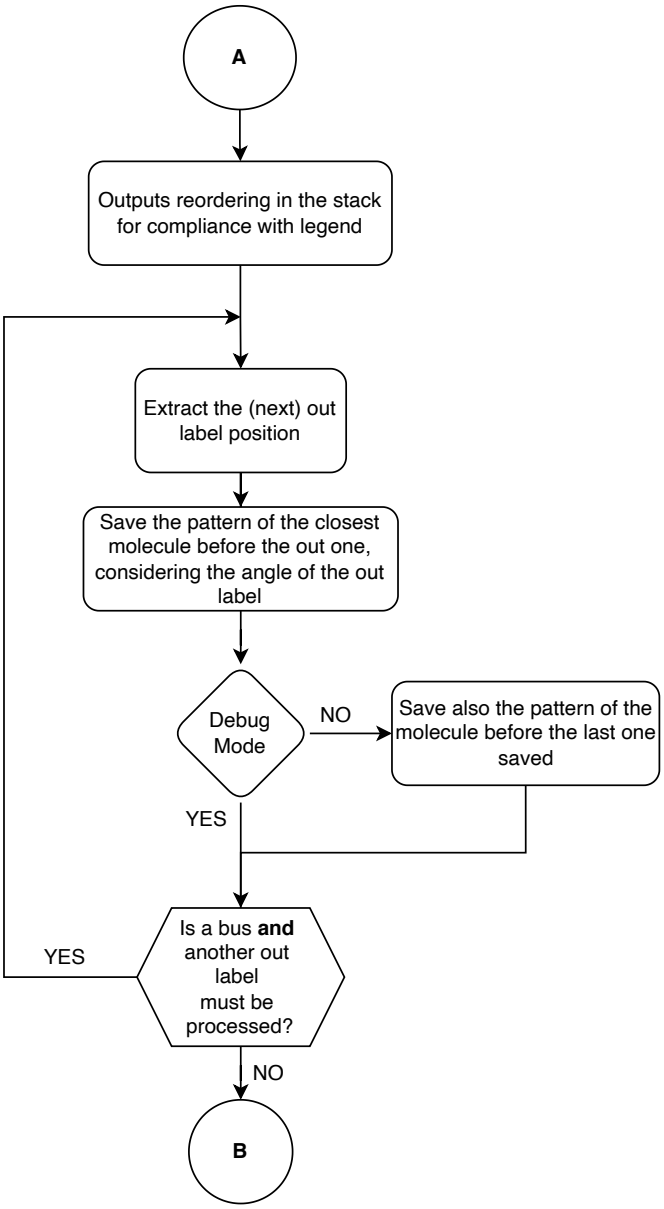


Figure 2.15: Flowchart for the *golden_outMol_finder()* sub-function

Chapter 3

Characterisation tool results

The results chapter will show the characterisation of some cells used for the tool debug as for the application on a complex architecture discussed later. Each section discusses a cell where the layout, the termination strategy and the *characterisation tool* output are analyzed starting from the most simple solution to the complex ones. *SCERPA*'s parameters are maintained the same for every simulation and are the following:

Listing 3.1: *SCERPA* parameters

```
%% SCERPA settings
settings.out_path = '../MVBus_QLL_LAYOUT';
settings.dumpDriver = 1;
settings.dumpOutput = 1;
settings.dumpClock = 0;
settings.dumpVout = 1;
settings.driverSaturation = 0;

settings.plot_voltage = 0;
settings.plot_chargeFig = 0;
settings.plot_molnum = 1;
settings.solver = 'E';

settings.immediateUpdate = 0;
settings.pauseStep = 0;
settings.damping = 0.6;
settings.activeRegionThreshold = 0.005;
settings.verbosity = 0;
settings.conv_threshold_HP = 0.005;
settings.enableRefining = 0;
settings.enableActiveRegion = 1;
settings.plotIntermediateSteps = 0;
settings.plotActiveRegionWindow = 0;
```

3.1 Mono-phase Six Molecule Wire

The first and simplest solution is a single-line wire, composed of six molecules in a single phase:

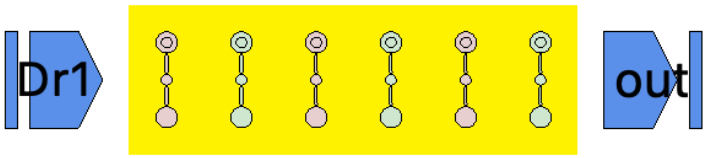


Figure 3.1: *Mono-phase Six Molecule Wire* layout

The layout has been simulated with an automatic termination: considering the mono-phase design, the termination is given by a copy of the wire itself:

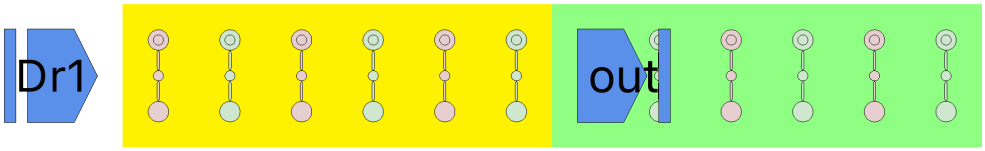


Figure 3.2: *Mono-phase Six Molecule Wire* layout terminated

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\,V$ to $1\,V$:

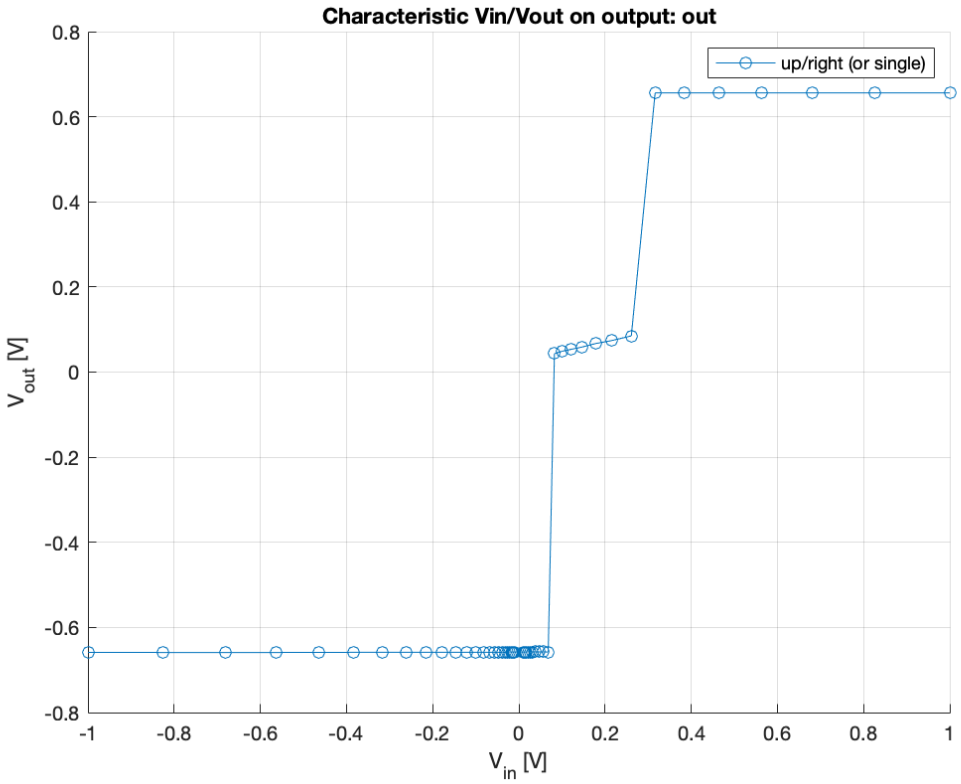


Figure 3.3: *Mono-phase Six Molecule Wire* $V_{in} - V_{out}$ characteristic

The characteristic behaviour is strictly dependent on the layout choices, the clock phases involved and the number of molecules. For instance, the use of just a phase for the realization lets the output interact with the input and vice versa, creating a strange slowdown for V_{in} close to $0\,V$.
Other versions of this connection wire will be shown later with the XOR parts: the *bi-stability* given by the bus paradigm is the key to obtain a reliable behaviour.

3.2 Three-phase Twenty-four Molecule Wire

A more complex wire is the one reported here, realized with a single-line structure of twenty-four molecules on three phases. The use of three phases solves the slowdown seen with the mono-phase wire:

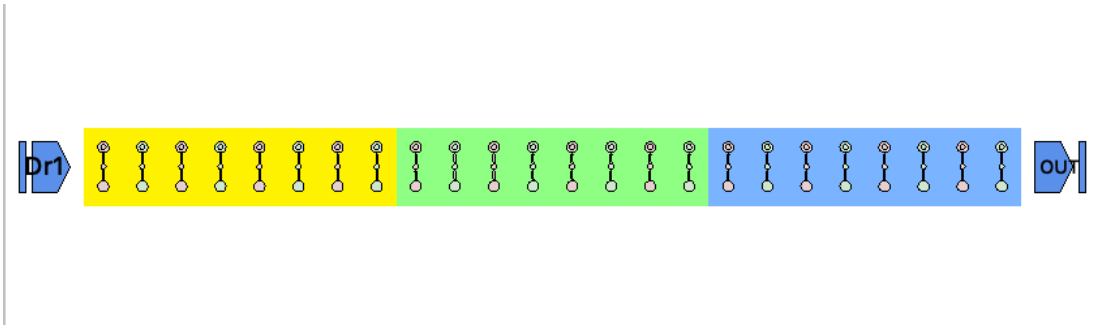


Figure 3.4: *Three-phase Twenty-four Molecule Wire* layout

The layout has been simulated following the automatic termination process: considering the eight molecules involved in the third phase of the layout, the termination is given by a single-line wire made up of eight molecules:

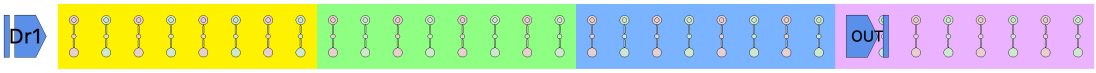


Figure 3.5: *Three-phase Twenty-four Molecule Wire* layout terminated

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\,V$ to $1\,V$:

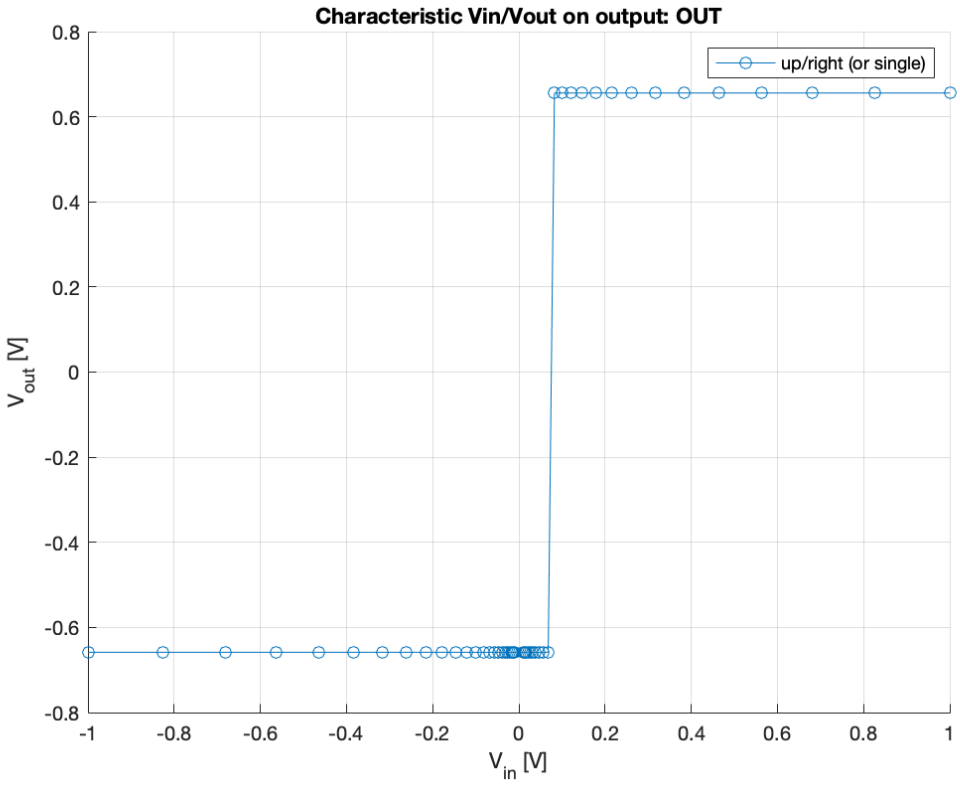


Figure 3.6: *Three-phase Twenty-four Molecule Wire* $V_{in} - V_{out}$ characteristic

3.3 Three-phase Twenty-four Molecule Bus

The same twenty-four molecules on three phases wire already shown but based on the bus paradigm has the following layout:

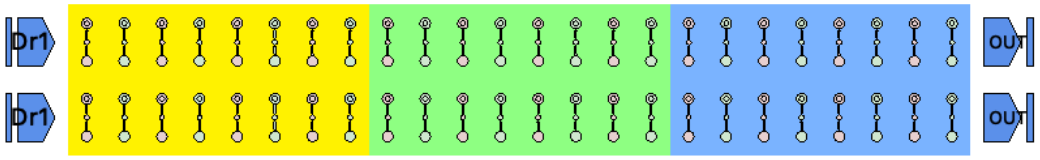


Figure 3.7: *Three-phase Twenty-four Molecule Bus* layout

The layout has been simulated following the automatic termination process. In this case the third phase of the layout involves a bus wire of *8x2* molecules: the termination is given by the same bus wire made up of *8x2* molecules:

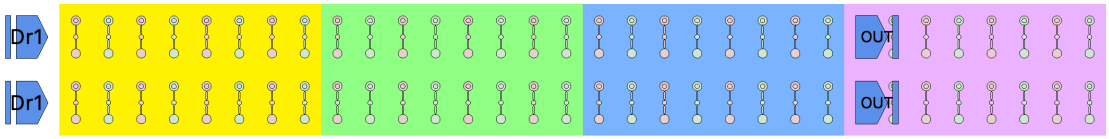


Figure 3.8: *Three-phase Twenty-four Molecule Bus* layout terminated

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\ V$ to $1\ V$:

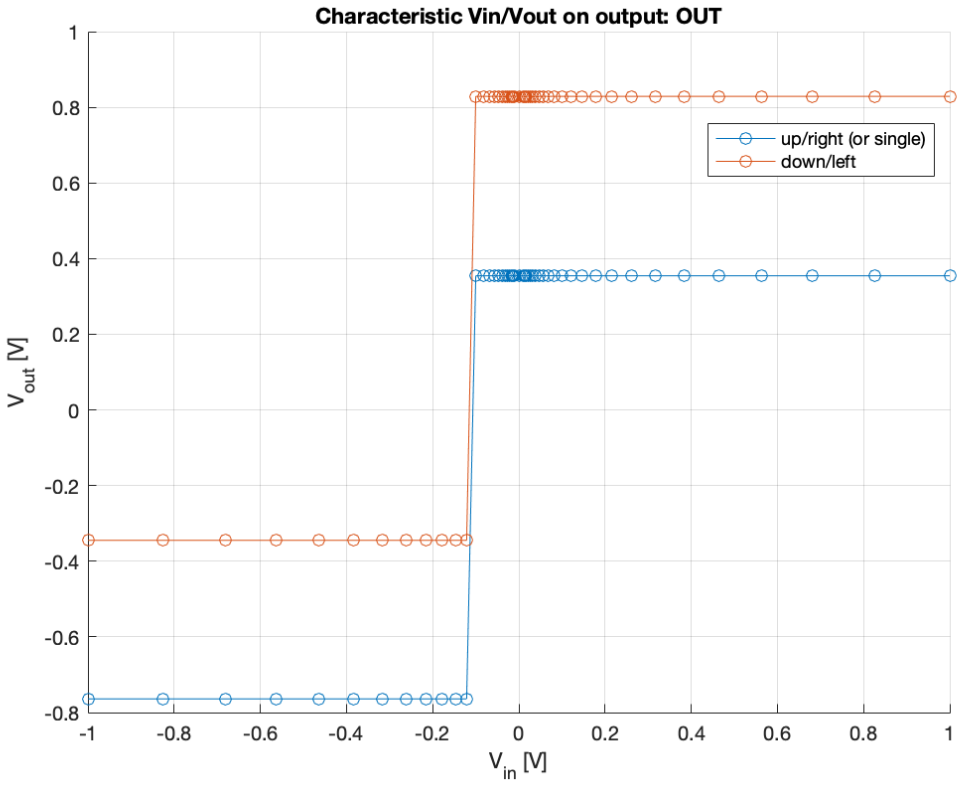


Figure 3.9: *Three-phase Twenty-four Molecule Bus* $V_{in} - V_{out}$ characteristic

The orange curve corresponds to the lower output while the blue one to the upper output. The output bias is given by the vertical electrostatic interaction between molecules as discussed in 1.2.2.

3.4 Three-phase L-connector@bus with upward output

A three-phase L-connector realized within the bus paradigm and with an upward output could has the following layout:

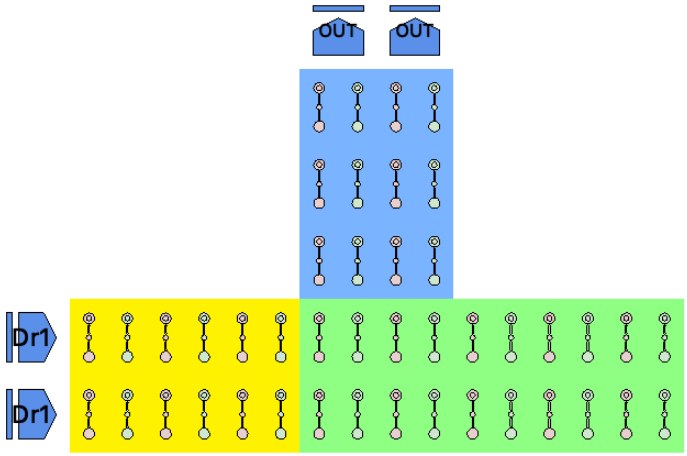


Figure 3.10: *Three-phase L-connector@bus with upward output* layout

The layout has been simulated with an automatic termination:

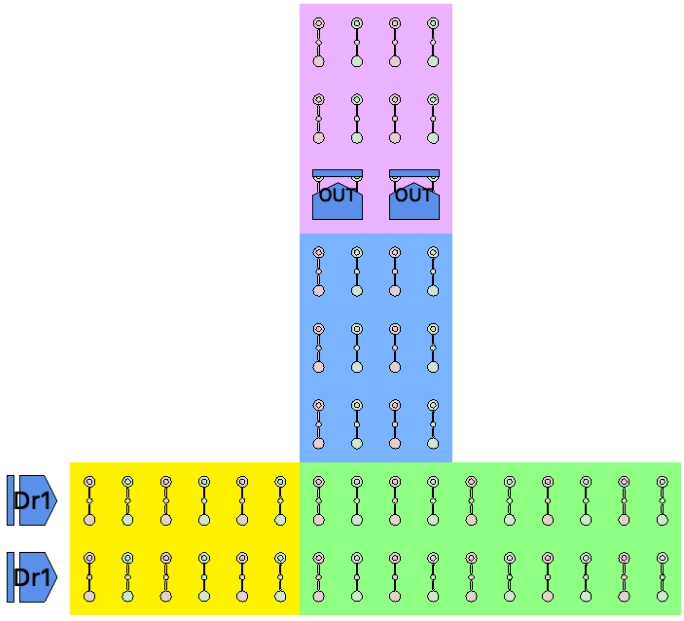


Figure 3.11: *Three-phase L-connector@bus with upward output* layout terminated

On the contrary to what has been shown with the other layouts, the output labels here are directed upward. The output labels direction is fundamental for the termination process because the script functionalities rely on this parameter to recognize the termination direction. Figure 1 shows the compliance between the termination and the output direction.

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\,V$ to $1\,V$:

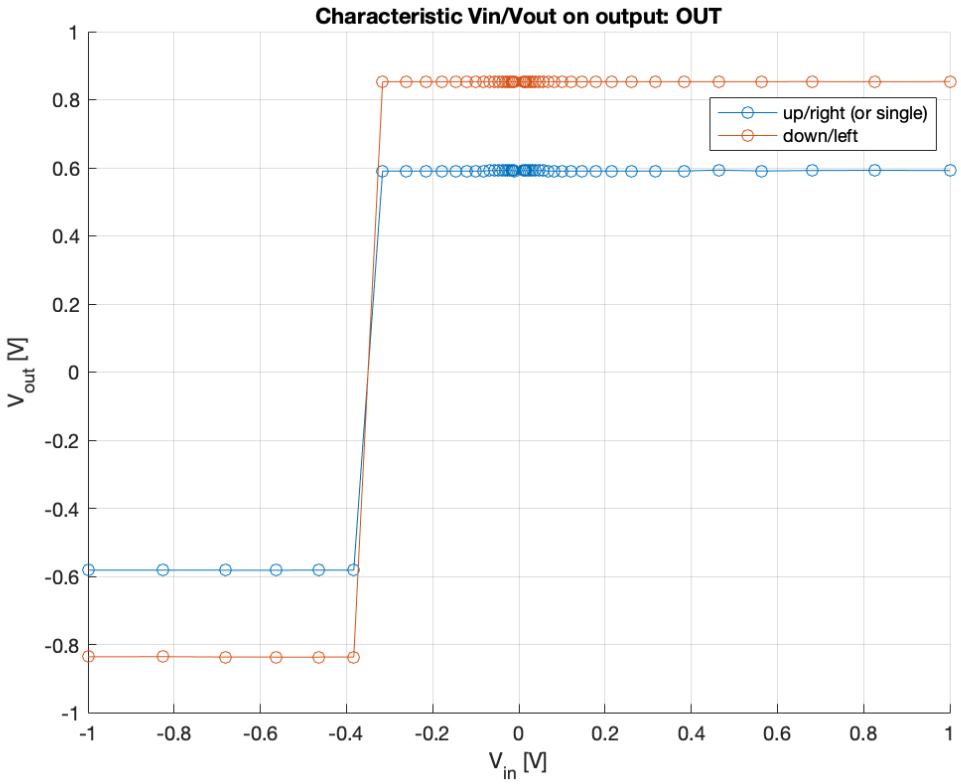


Figure 3.12: *Three-phase L-connector@bus with upward output* $V_{in} - V_{out}$ characteristic

The characteristic shows a particular negative biased behaviour that reveals a poor layout, probably given by the usage of not enough clock phases. An improved layout for the same cell will be shown with the XOR implementation, using four phases instead of three.

3.5 Three-phase L-connector@bus with downward output

The same three-phase L-connector already shown but with a downward output:

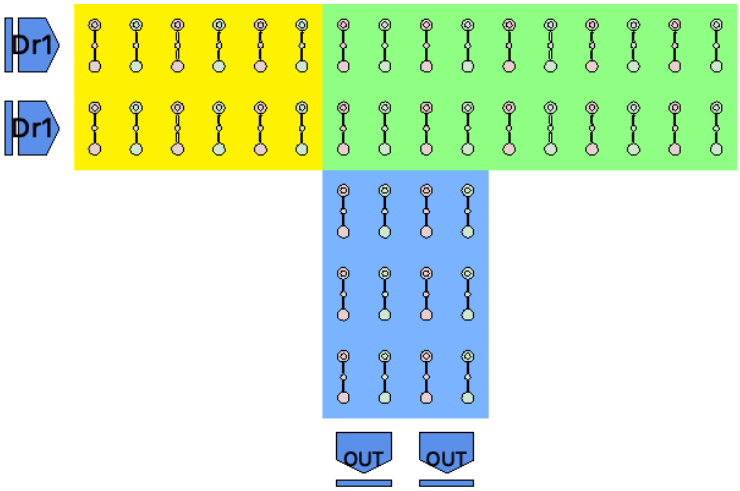


Figure 3.13: *Three-phase L-connector@bus with downward output* layout

The layout has been simulated with an automatic termination:

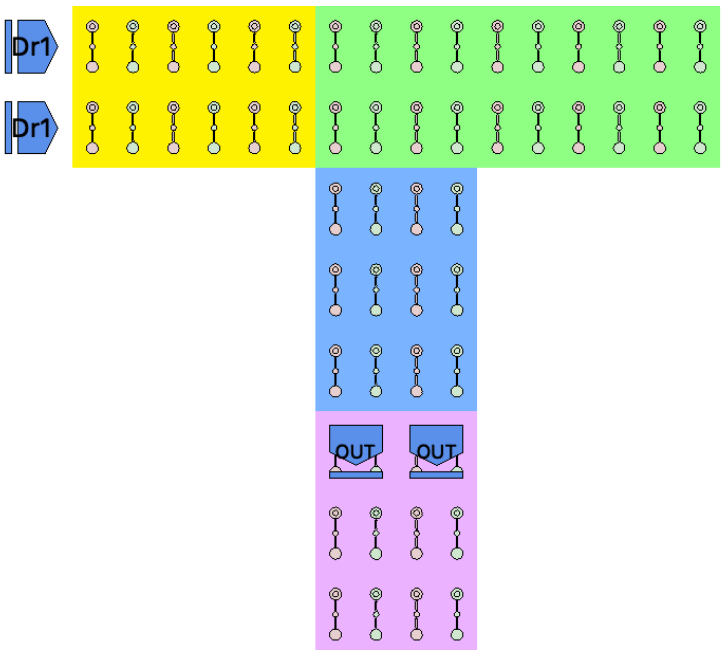


Figure 3.14: *Three-phase L-connector@bus with downward output* layout terminated

Here the output labels are directed downward. The automatic termination process finds the output angle, handling the termination direction correctly also in this case.
The *Debug Mode* characterisation is provided with a 51 *points* linear input sweep from $-1\ V$ to $1\ V$:

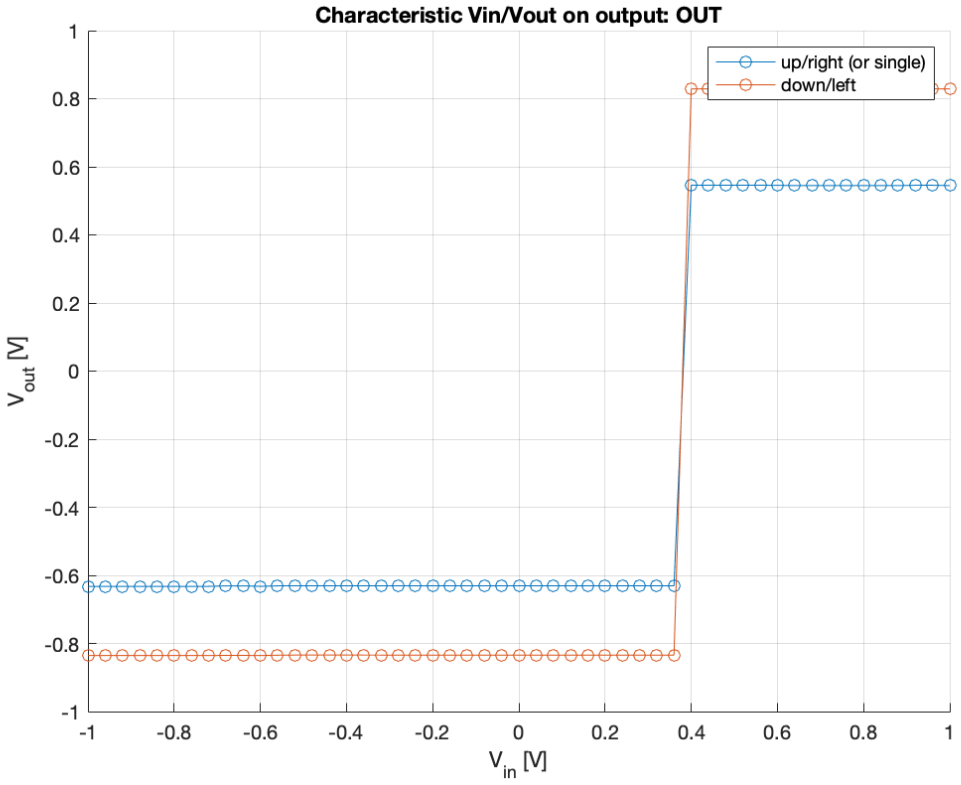


Figure 3.15: *Three-phase L-connector@bus with downward output* $V_{in} - V_{out}$ characteristic

Also in this case, the characteristic shows a particular positive biased behaviour that reveals the same poor layout as discussed before.

3.6 Four-phase inverter@bus

The first cell performing a logic operation among the wires shown before, is the *four-phase inverter@bus*:

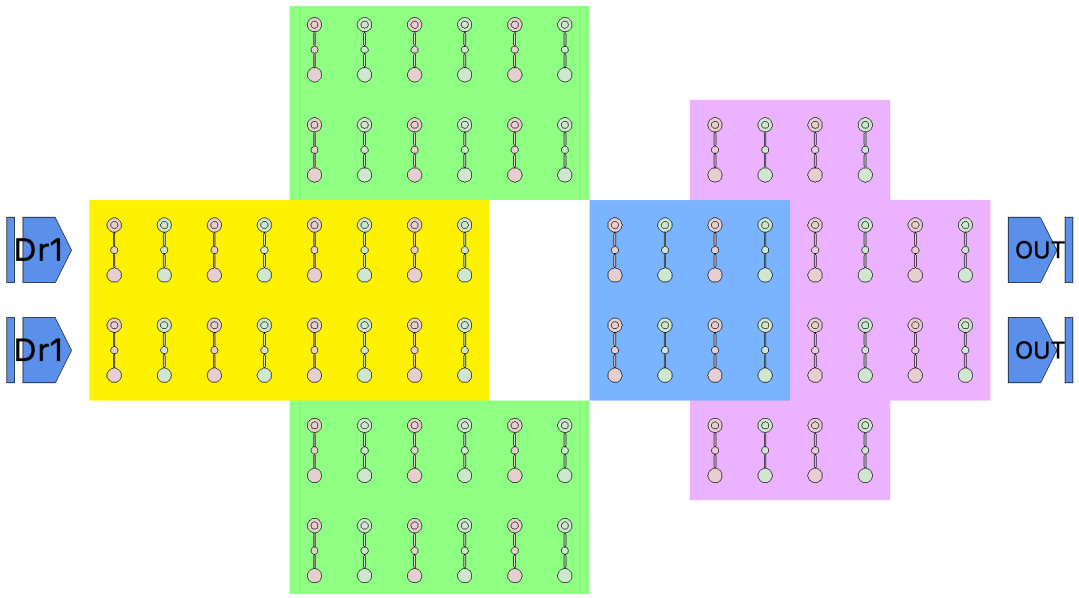


Figure 3.16: *Four-phase inverter@bus* layout

The layout has been simulated with an automatic termination:

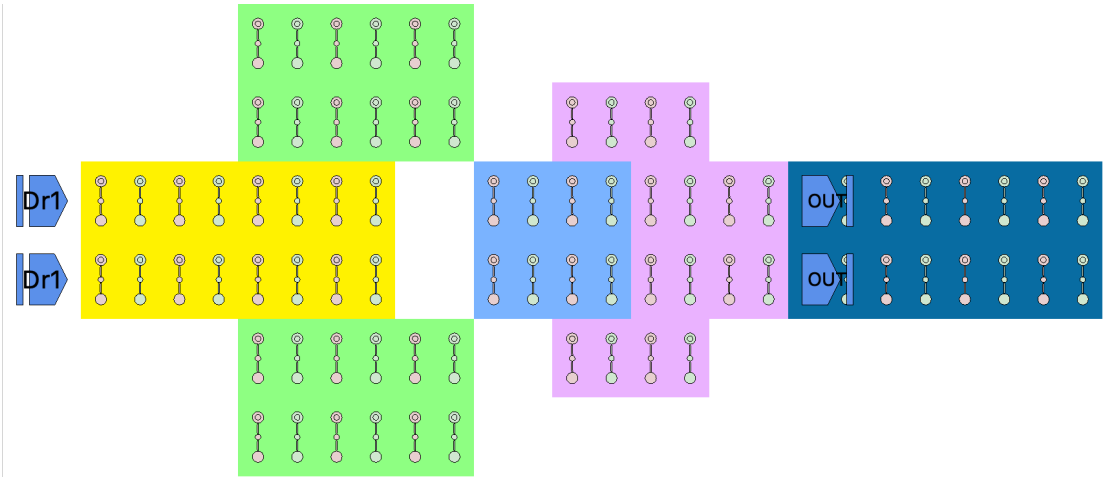


Figure 3.17: *Four-phase inverter@bus* layout terminated

The automatic termination procedure here considers the number of molecules involved in the fourth phase (pink). The fourth phase here is particular because of the flaps embracing the third phase. However, for the termination function, the fourth phase is unique and composed of sixteen molecules.

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\,V$ to $1\,V$:

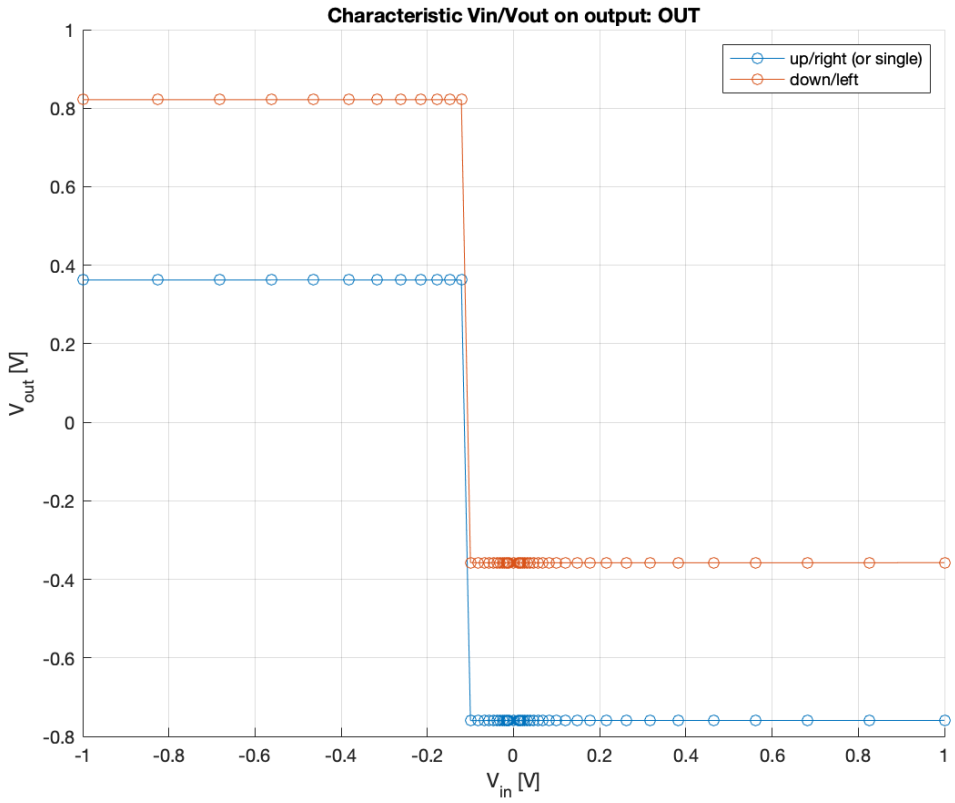


Figure 3.18: *Four-phase inverter@bus* $V_{in} - V_{out}$ characteristic

The layout is different from the one shown in figure 1.5 because of the necessity to work at least in simulation. The mentioned flaps as the bus paradigm are in this case mandatory to ensure a good robustness to the layout.

3.7 Three-phase Majority Voter

The core cell of the *MolFCN* technology is the majority voter that permits replicating the basic logic functions like the *logic-AND* and the *logic-OR*. A tentative single-line layout for this cell is the following:

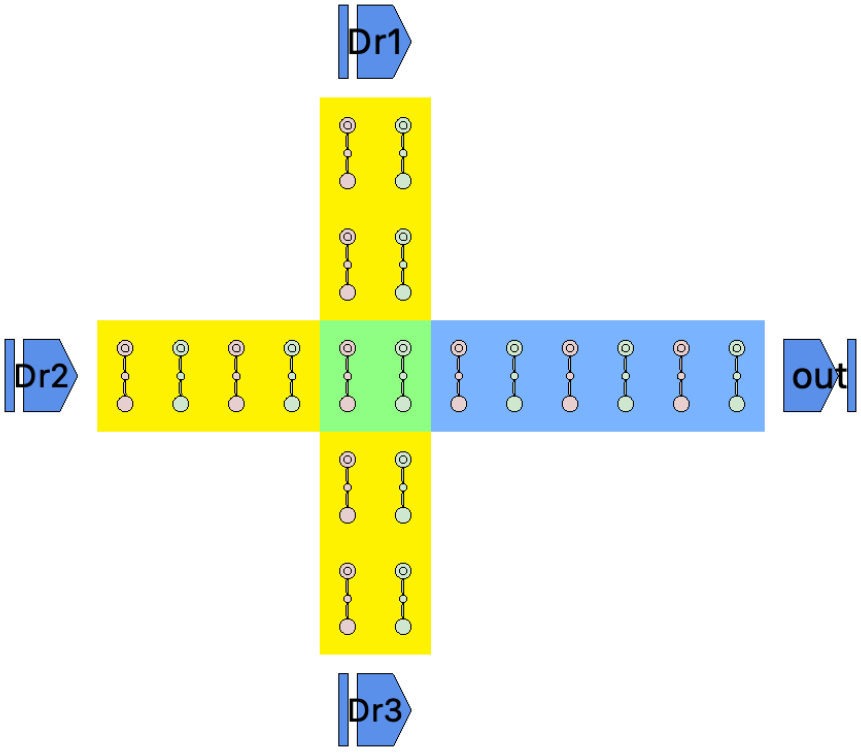


Figure 3.19: *Three-phase Majority Voter* layout

The vertical inputs of the layout (*Dr1* and *Dr3*) have the same direction as the horizontal one. This is not a design mistake: it is needed to guarantee the correct behaviour of the cell considering that the *QCAs* accessed along the vertical axis show a different face of them to the drivers compared to the horizontal one. The layout scheme shows this detail in the *QCA* representation: each *QCA* has drawn the two molecules as they will be considered by *SCERPA*. While *Dr2* sees the *QDs* of just a molecule, *Dr1* and *Dr3* see the same *QD* of the two molecules that compose the interested *QCA*. Following this drawing rule, the cell is simulated as it is driven by other cells instead of the fake drivers.
As a general rule, the drivers should be placed always with the direction used above without distinctions, to be sure of the correspondence between the simulation output and the expected cell behaviour.

The layout has been simulated with an automatic termination:

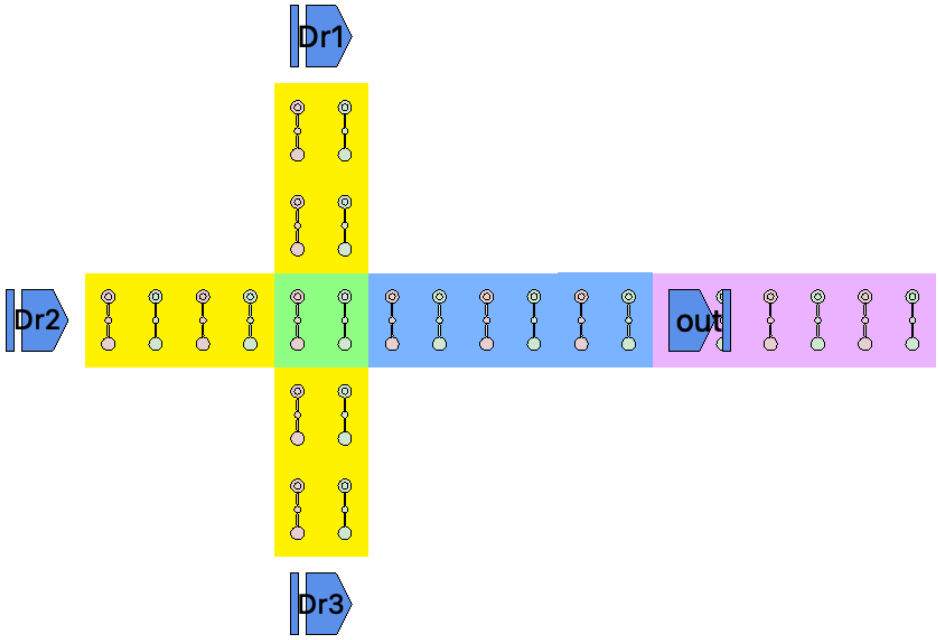


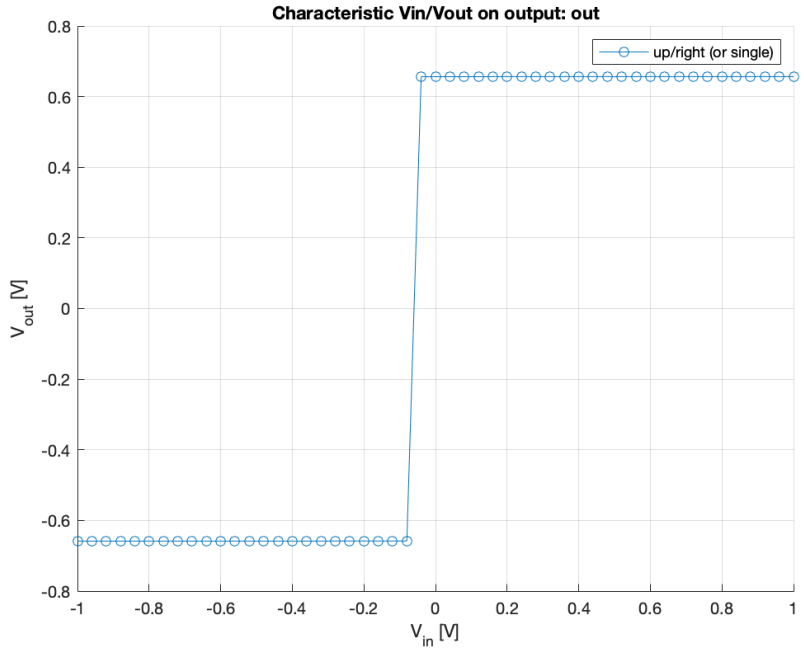
Figure 3.20: *Three-phase Majority Voter* layout terminated

The characterisation for cells with multiple inputs is more intricate than the other because of the necessity to test every non-trivial combination. The MV output switches when two inputs are different and the third one sweeps. Simulating this kind of input dataset, it is possible to trigger the output switch letting it be observable.

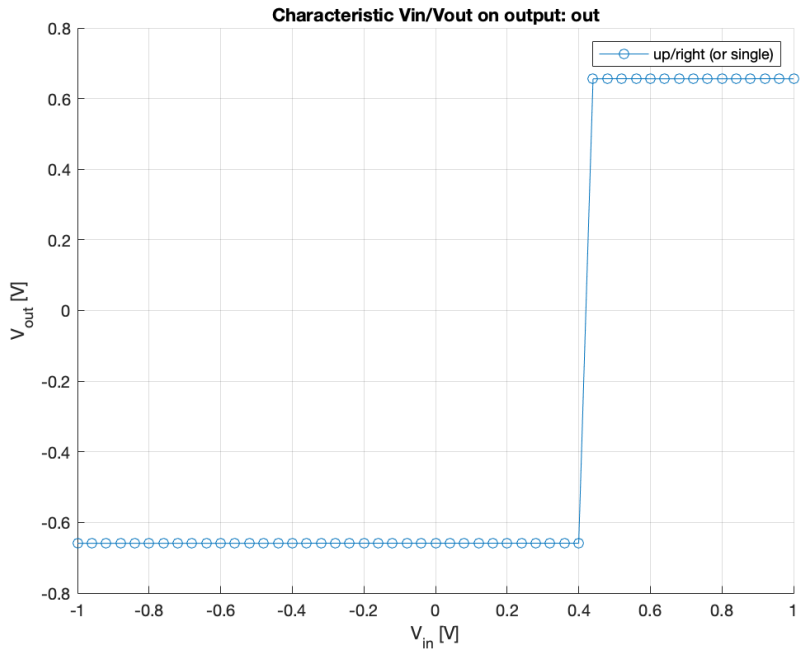
Table 3.1: Inputs combinations to trigger the *majority voter* output switch

Dr1	Dr2	Dr3
sweep	$-1\ V$	$1\ V$
sweep	$1\ V$	$-1\ V$
$-1\ V$	$1\ V$	sweep
$1\ V$	$-1\ V$	sweep
$-1\ V$	sweep	$1\ V$
$1\ V$	sweep	$1\ V$

The use of $\pm 1\ V$ permits the levels saturation and a good resolution with few sweep points.
The input dataset could be coupled considering the sweep applied at the same input because of the expected reasonably similar behaviour:

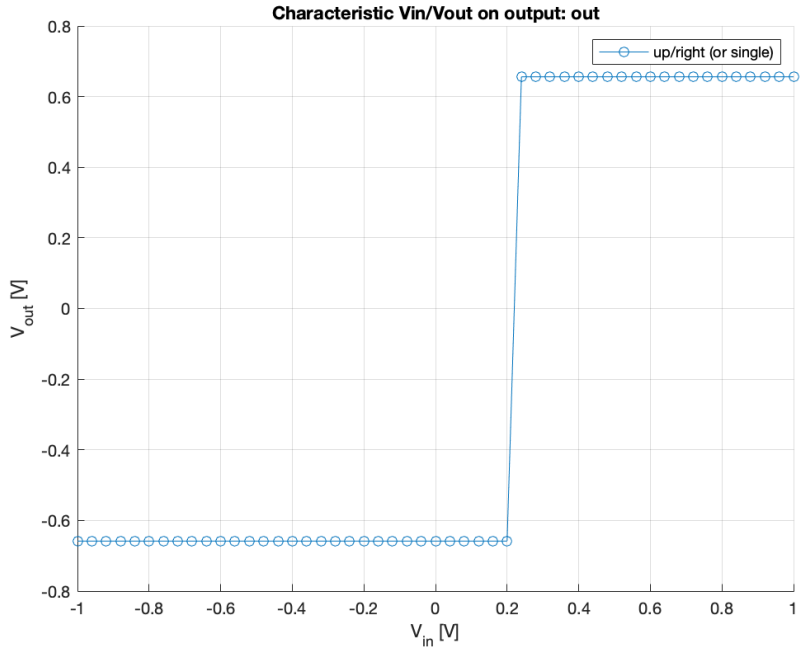


(a)

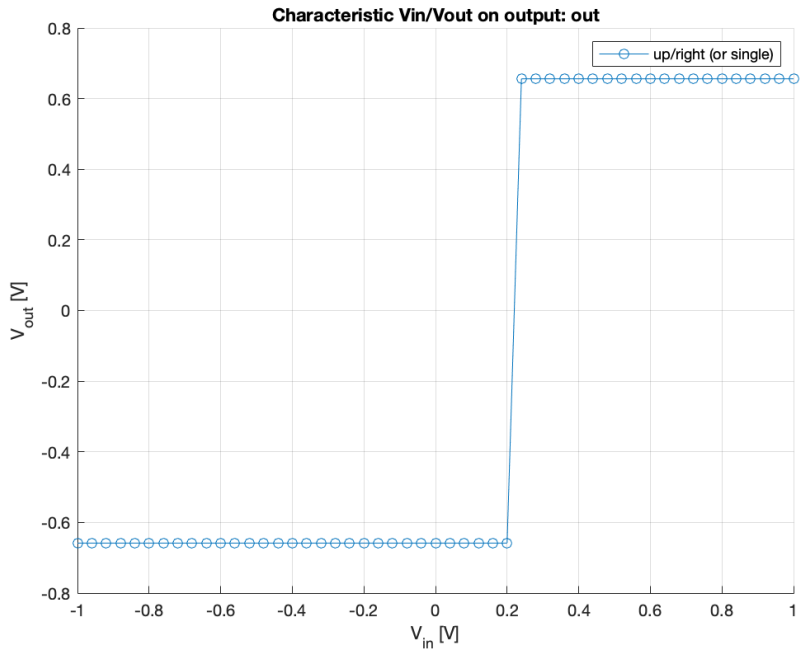


(b)

Figure 3.21: *Three-phase Majority Voter Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Input $Dr1 \longrightarrow sweep$, $Dr2 \longrightarrow -1\ V$, $Dr3 \longrightarrow 1\ V$; (b) Input $Dr1 \longrightarrow sweep$, $Dr2 \longrightarrow 1\ V$, $Dr3 \longrightarrow -1\ V$

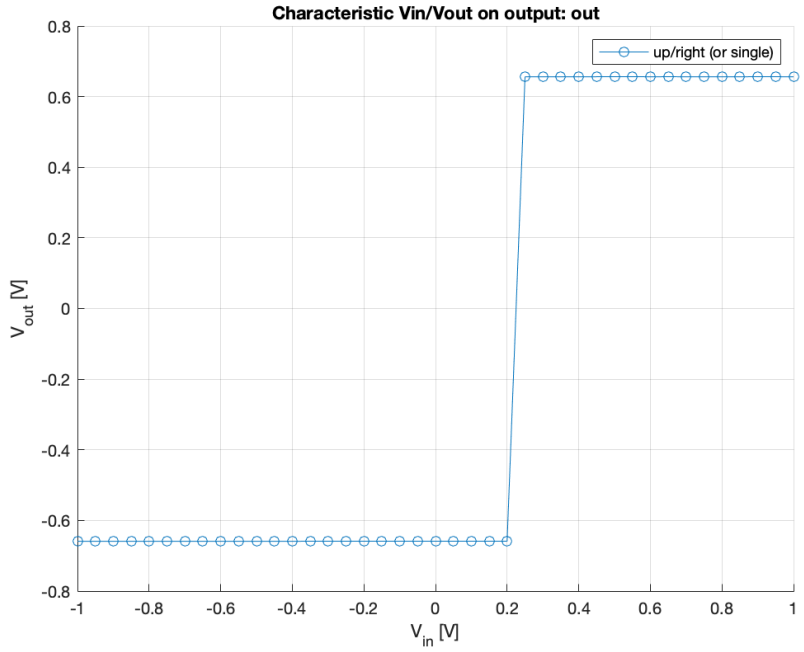


(a)

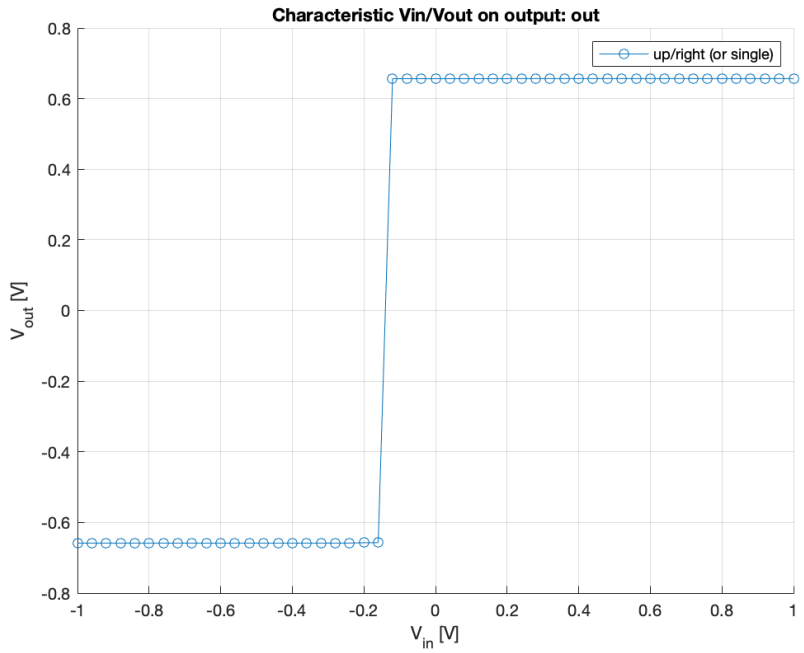


(b)

Figure 3.22: *Three-phase Majority Voter Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Input $Dr1 \longrightarrow -1\ V$, $Dr2 \longrightarrow sweep$, $Dr3 \longrightarrow 1\ V$; (b) Input $Dr1 \longrightarrow 1\ V$, $Dr2 \longrightarrow sweep$, $Dr3 \longrightarrow -1\ V$



(a)



(b)

Figure 3.23: *Three-phase Majority Voter Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Input $Dr1 \longrightarrow -1\ V$, $Dr2 \longrightarrow 1\ V$, $Dr3 \longrightarrow sweep$; (b) Input $Dr1 \longrightarrow 1\ V$, $Dr2 \longrightarrow -1\ V$, $Dr3 \longrightarrow sweep$

The graphs shown are obtained using a *41 steps* linear sweep. Although it is expected a similar behaviour for some of the input datasets, the single-line layout reveals some criticalities given by its low robustness. This is the reason why the bus paradigm is mandatory in cells design. In fact, the same layout realized within the bus paradigm shows a much better behaviour in general, as can be seen in the next section.

3.8 Three-phase Majority Voter@bus

The three-phase majority voter shown before, realized within the bus paradigm has the following layout:

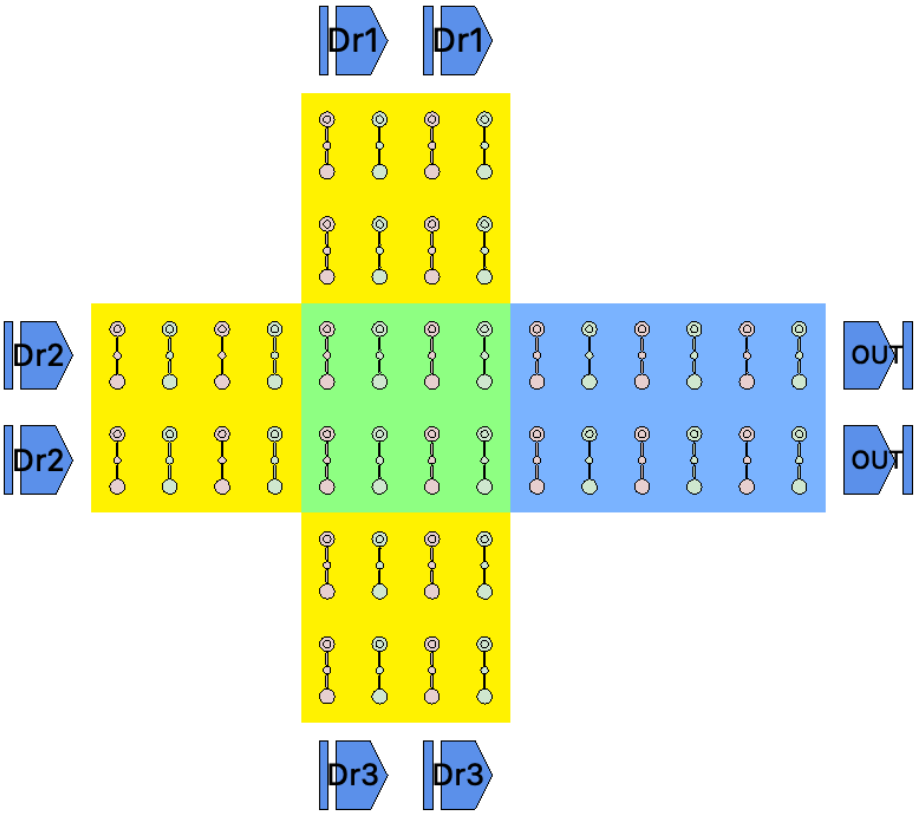


Figure 3.24: *Three-phase Majority Voter@bus* layout

In this case as for the single-line majority voter shown before, the drivers follows the same direction as rule of design, ensuring the expected cell behaviour.

The layout has been simulated with an automatic termination where the *6x2* molecules involved on the third phase are replicated on the output:

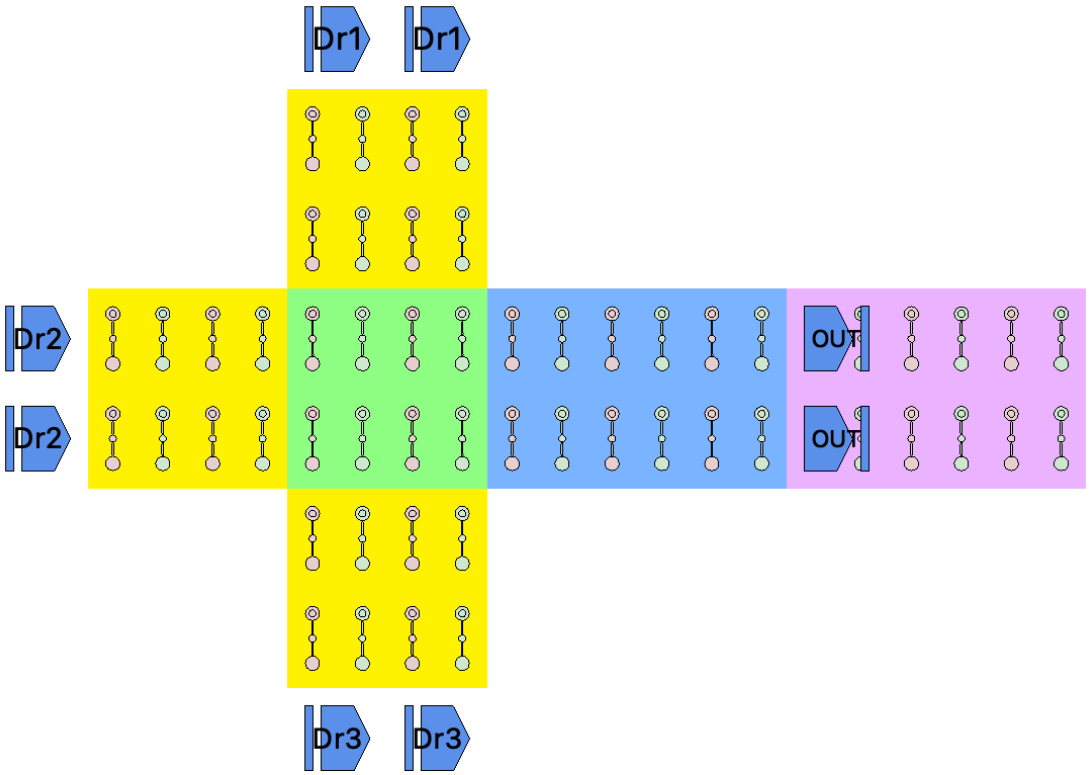
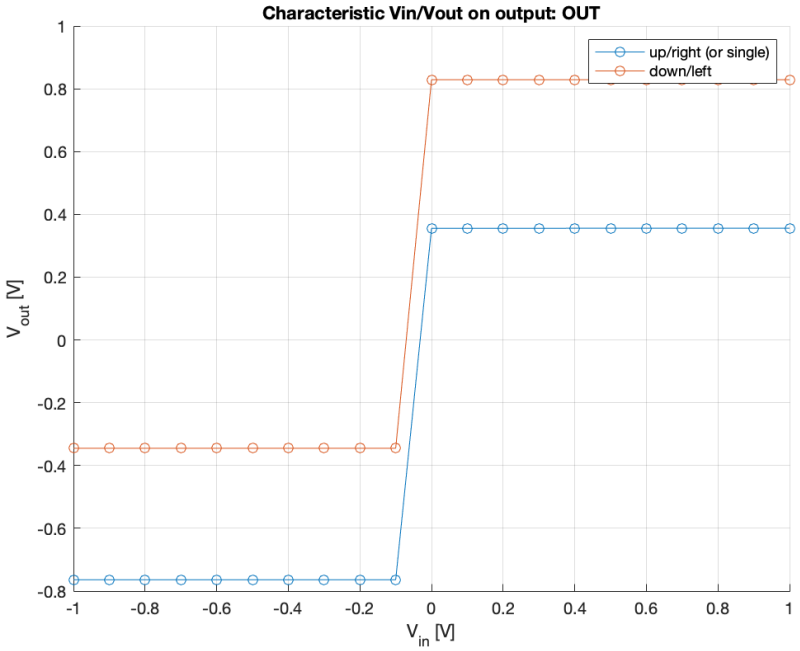
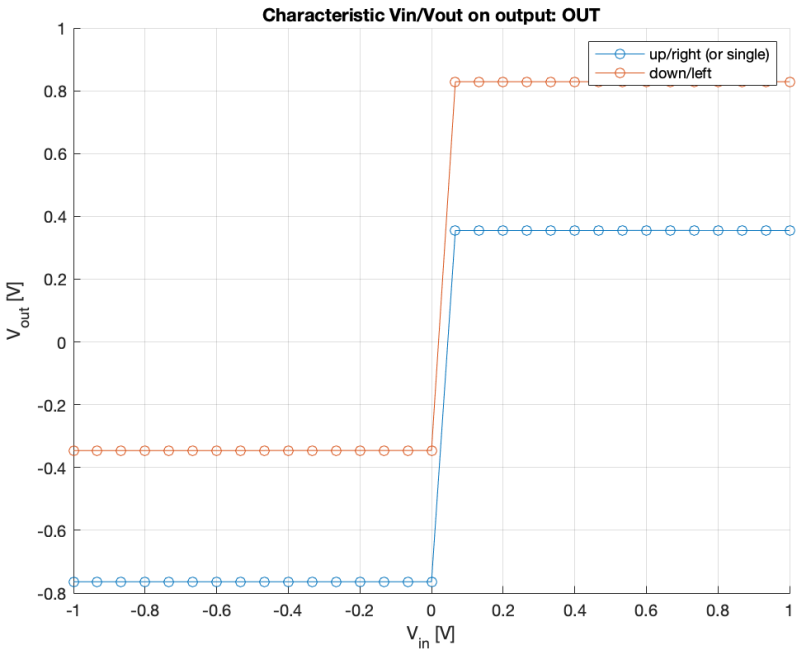


Figure 3.25: *Three-phase Majority Voter@bus* layout terminated

As shown in table 3.1, the non-trivial combinations to test are the ones where the output can switch revealing its behaviour.
The characteristics discussed so far have been simulated with the method examined in the previous sections: clearly, they are not showing measures taken on the real device but extracted from *SCERPA*. Usually, from a real characteristic also the gain can be defined looking at the transaction from a '0'-logic to '1'-logic and vice versa, but this is not the case.
In fact, the transaction slope is given by the *MATLAB* plot points interpolation and it is strictly dependent on the steps chosen to discretize the sweep interval.
The characteristics shown below would highlight this difference: the graphs are absolutely identical from the logical point of view, but the last ones are extracted with a logarithmic input sweep while the others exploit a linear sweep. The gain in the logarithmic ones seems to be a lot higher revealing the effect of the accumulated points that are consistently closer each to the others with respect to the equivalent linear case.

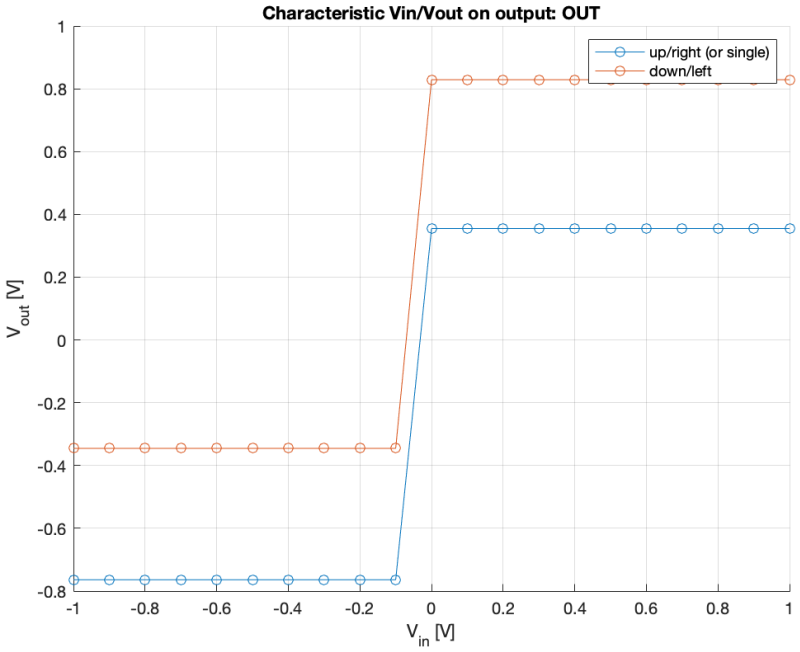


(a)

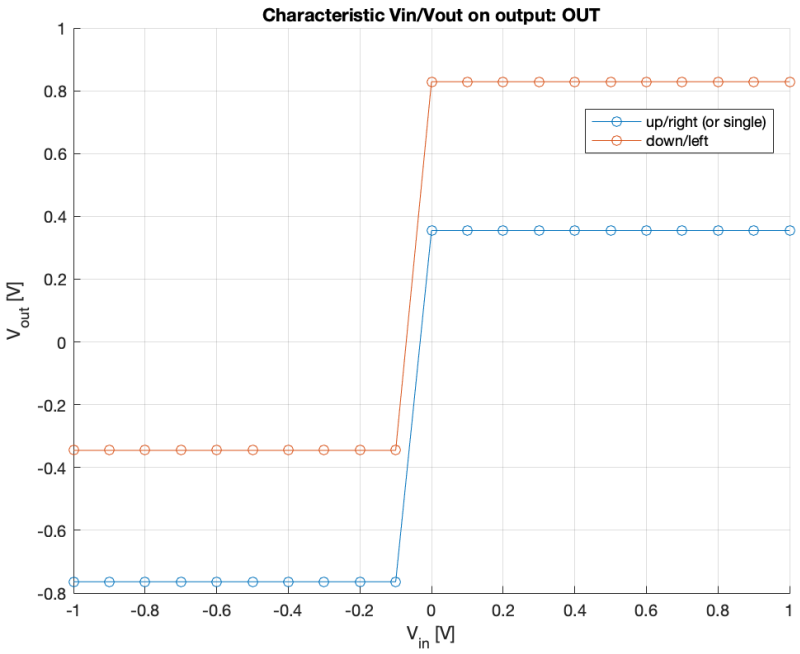


(b)

Figure 3.26: *Three-phase Majority Voter@bus Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Input $Dr1 \longrightarrow sweep$, $Dr2 \longrightarrow -1\ V$, $Dr3 \longrightarrow 1\ V$; (b) Input $Dr1 \longrightarrow sweep$, $Dr2 \longrightarrow 1\ V$, $Dr3 \longrightarrow -1\ V$

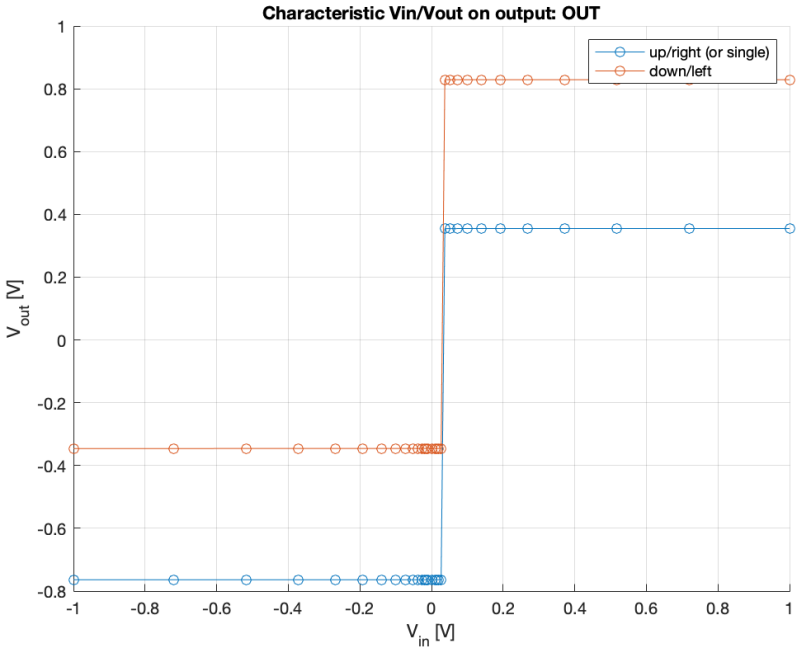


(a)

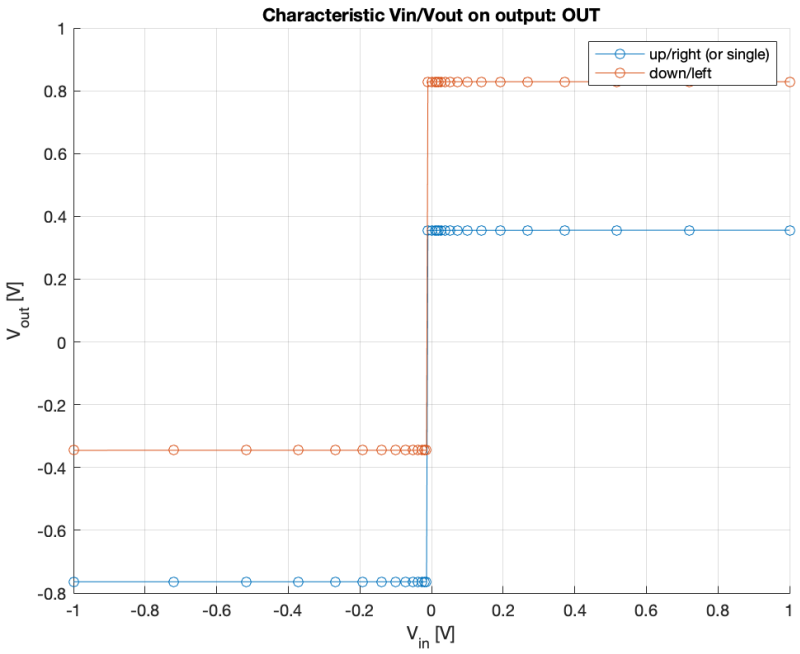


(b)

Figure 3.27: *Three-phase Majority Voter@bus Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Input $Dr1 \longrightarrow -1\ V$, $Dr2 \longrightarrow$ *sweep*, $Dr3 \longrightarrow 1\ V$; (b) Input $Dr1 \longrightarrow 1\ V$, $Dr2 \longrightarrow$ *sweep*, $Dr3 \longrightarrow -1\ V$



(a)



(b)

Figure 3.28: *Three-phase Majority Voter@bus Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Input $Dr1 \longrightarrow -1\ V$, $Dr2 \longrightarrow 1\ V$, $Dr3 \longrightarrow sweep$; (b) Input $Dr1 \longrightarrow 1\ V$, $Dr2 \longrightarrow -1\ V$, $Dr3 \longrightarrow sweep$

3.9 Four-phase T-connector@bus

The four-phase T-connector@bus is a valid cell to show the *characterisation tool* capabilities in the fan-out handling. The cell works as a simple wire propagating the information from the input to the outputs. A tentative layout for this cell is represented below:

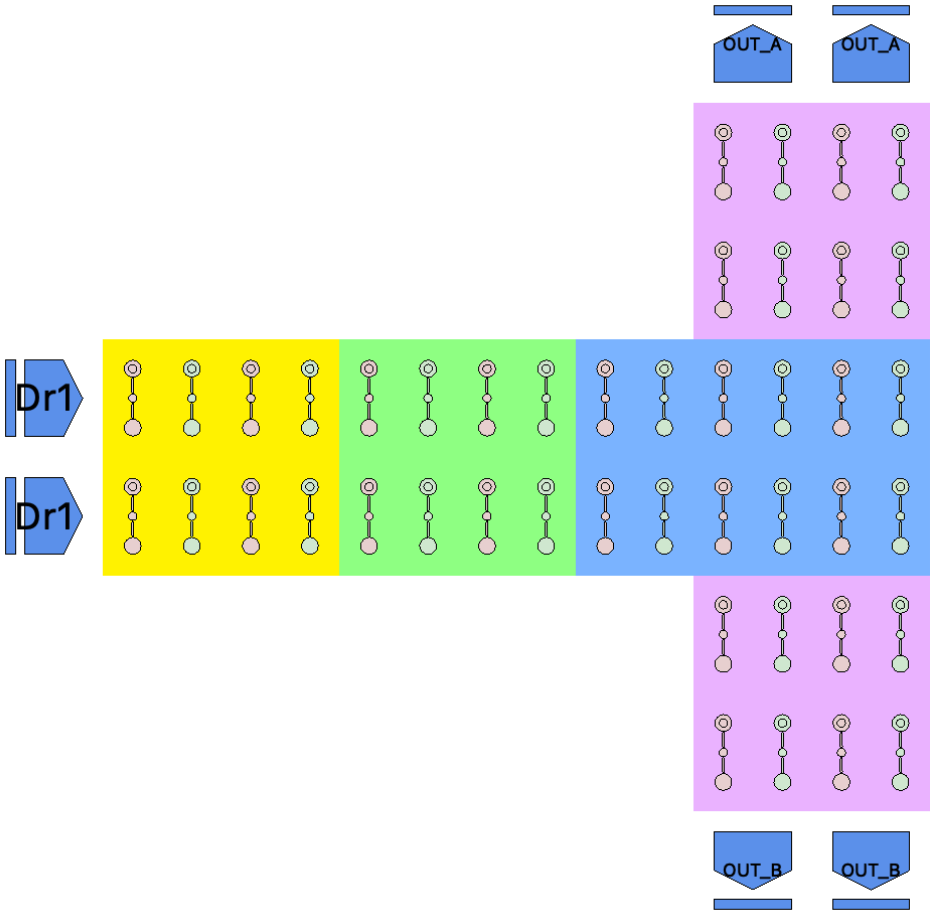


Figure 3.29: *Four-phase T-connector@bus* layout

The layout has been simulated with an automatic termination that considers both the output automatically as discussed in section 2.2.5. As a remark, the design rule on the outputs would have the same label name for each independent output while the dependent ones need to be equal. In this case, the upper output is named as *OUT_A* while the other *OUT_B*: the *characterisation tool* can differentiate between the outputs and apply the procedure correctly.

The layout after the termination procedure is shown below:

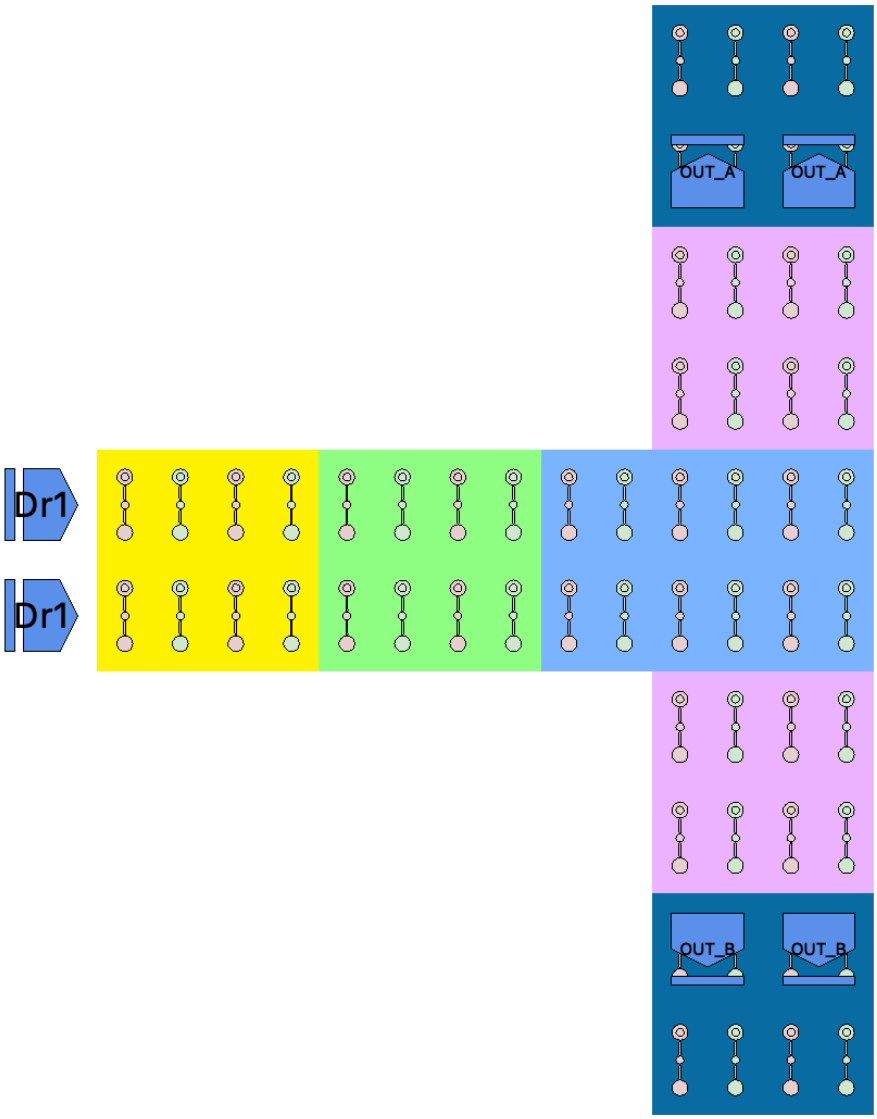
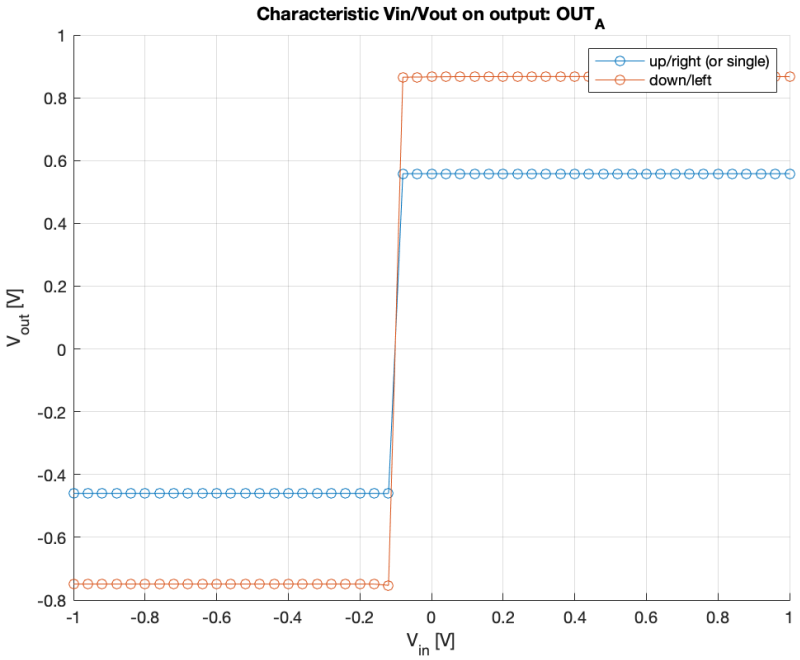


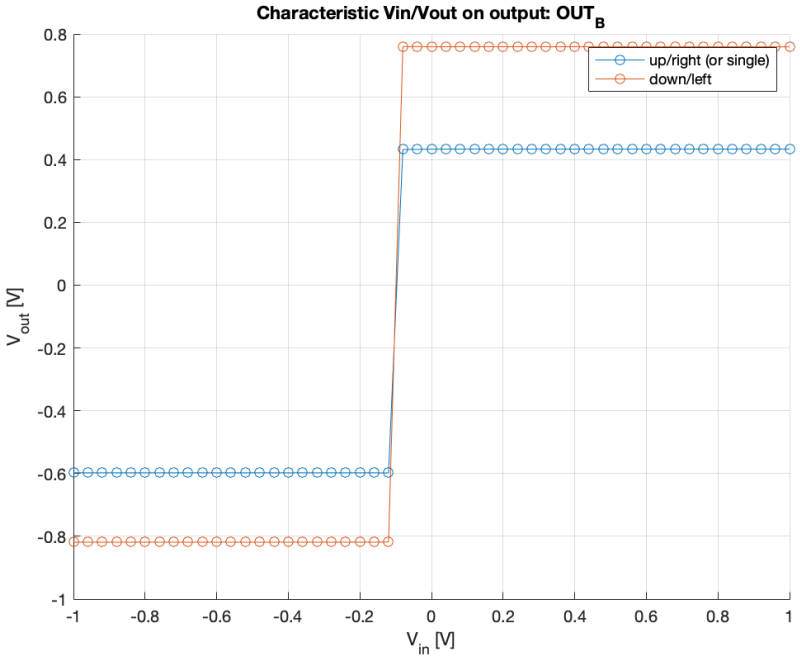
Figure 3.30: *Four-phase T-connector@bus* layout terminated

Despite the automatic termination, the number of molecules involved for each independent output is the correct one. Each independent output sees the molecules of its own last phase, ignoring the other ones in the layout[1].
The *Debug Mode* characterisation is provided with a 51 *points* linear input sweep from $-1\,V$ to $1\,V$, returning two plots, one for each output:

[1]This procedure is handled correctly on symmetrical layouts at this moment.



(a)



(b)

Figure 3.31: *Four-phase T-connector@bus Debug Mode* $V_{in} - V_{out}$ characteristic: (a) Output A, (b) Output B

Part III

Characterisation of a complex layout: XOR

Chapter 4

XOR cell

4.1 Description, characterisation and truth table

The *exclusive OR* is not a basic logic function but can be obtained by means of the *logic-AND*, *logic-OR* and *logic-INV* combination:

$$A \oplus B = A\overline{\mathrm{B}} + \overline{\mathrm{A}}B$$

where the truth table:

Table 4.1: Exclusive OR truth table

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

The XOR cell is a valid representative to evaluate the *characterisation tool* behaviour considering it is composed of several basic cells as *Wires*, *L-connectors*, *Majority Voters* and *Inverters*.
The layout that will be presented to encode the *logic-XOR* exploits four phases with a latency equal to three, meaning that these four phases are repeated three times from the inputs to the output. The previous sections highlighted the importance of a bus layout instead of a single-line one. For this reason, the XOR cell will be directly considered as bus-designed.

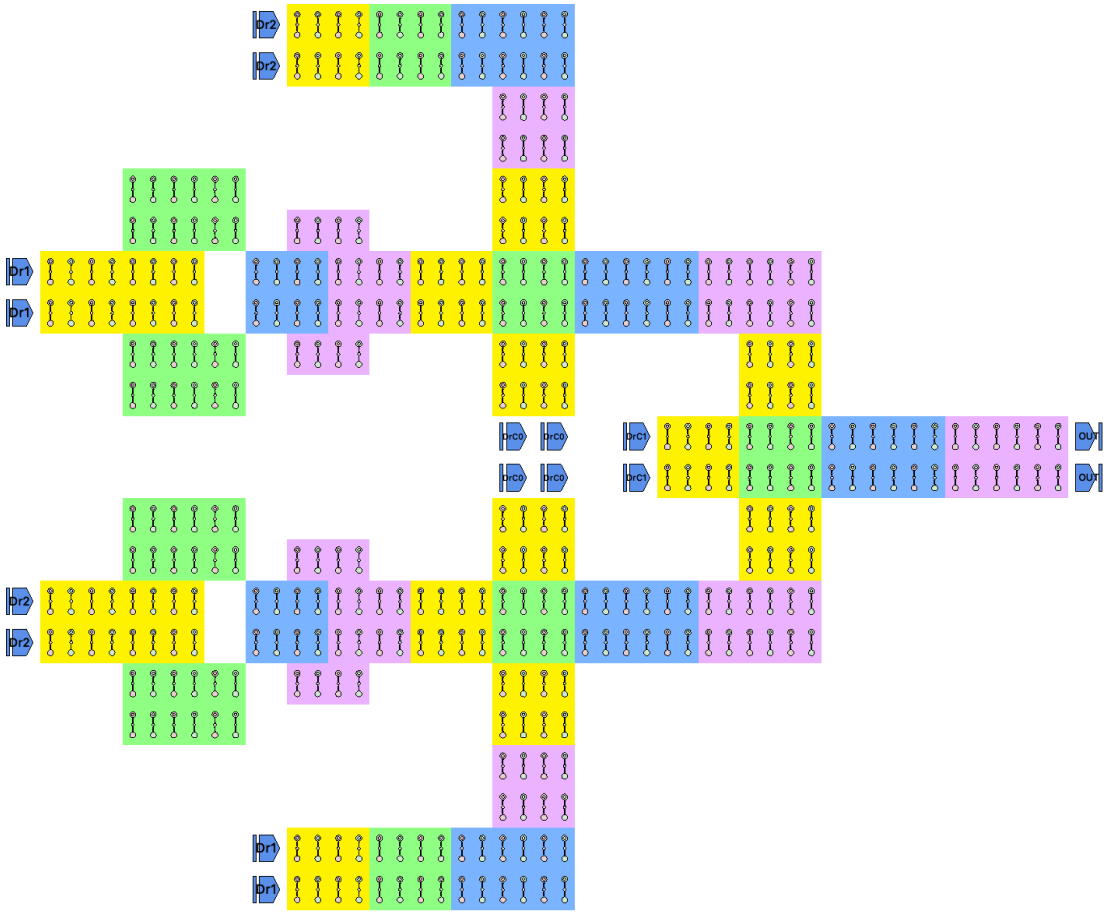


Figure 4.1: *Four-phase XOR@bus* layout

The XOR layout is composed of three MVs, two INVs, two L-connector and some six molecules wires with the output in every direction. The MVs have some of the inputs fixed to a certain voltage considering the necessity to let these behave like *logic-AND* or a *logic-OR*.
Before starting with the results, let's introduce the approach used to verify the *characterisation tool* goodness:

- **1)** The first important step to consider is the layout verification through *SCERPA*. The XOR cell can be treated as a basic block and characterised to verify its behaviour. This step is fundamental to create a reference to compare the results obtained by the characterisation through the single blocks libraries. For this reason, the *User Mode* of the *characterisation tool* can be exploited to test on the layout just the truth table input values: setting up the number of steps for the *driver_comb_creator()* function equal to two, the input combinations created are:

Table 4.2: Input combinations generated by the *driver_comb_creator()* function when the sweep number of steps is set up equal to two

A	B
$-1\ V$	$-1\ V$
$-1\ V$	$1\ V$
$1\ V$	$-1\ V$
$1\ V$	$1\ V$

Considering $\pm 1\ V$ as the saturation value, these input combinations correspond to the ones to test to extract the cell truth table;

- **2)** Once the cell layout is verified, a decomposition in single blocks can be introduced in order to characterise each of them in *User Mode*, creating the library files needed for the following step;
- **3)** The libraries created can be used to compute the output voltages of every single block of the complex architecture, moving from the inputs to the output. If the libraries extracted during the characterisation process are correct, for each input dataset must be found an almost perfect correspondence, meaning that the complex layout characterisation has been extracted correctly avoiding the direct use of *SCERPA*.

The first entry of the approach listed above refers to the extraction of the reference behaviour for the complex architecture under test. Using the *characterisation tool* within the *User Mode*, the library table extracted shows the expected behaviour:

Table 4.3: XOR cell *User Mode* simulation: *csv* library file

$Dr1$	$Dr1_c$	$Dr2$	$Dr2_c$	V_{outB}	V_{outA}	V_{outD}	V_{outC}
-1	1	-1	1	-0.7630	0.3548	-0.3446	0.8290
-1	1	1	-1	0.3539	-0.7637	0.8286	-0.3451
1	-1	-1	1	0.3539	-0.7637	0.8286	-0.3451
1	-1	1	-1	-0.7630	0.3548	-0.3446	0.8290

The outer molecules are the A and the C ones where can be observed a negative voltage[1] in the inner lines, correspondent to a *'1'-logic* following the standard agreement discussed in 1.2 [9, 10, 18].

[1]The input value to consider as a reference are the direct column: *Dr1* and *Dr2*.

4.2 Characterisation of single blocks

The XOR cell layout can be decomposed into basic blocks easy to characterise:

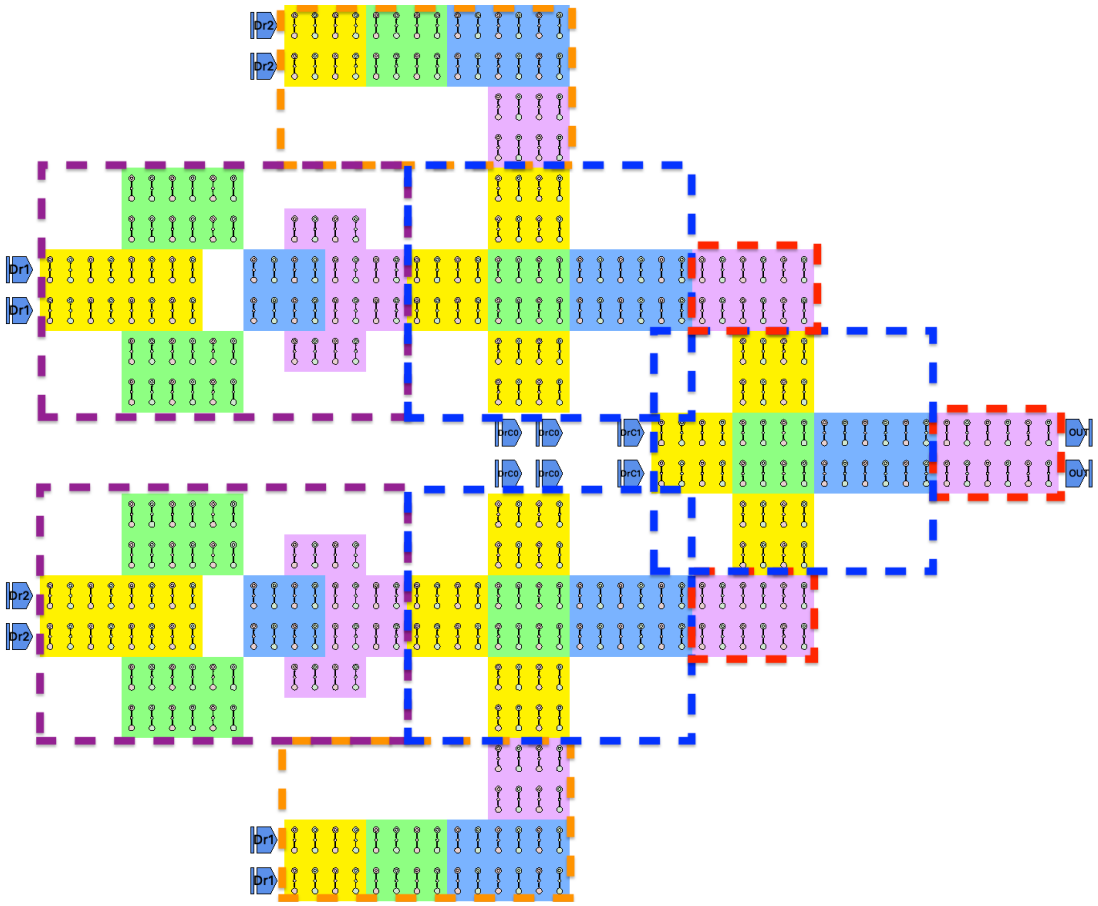


Figure 4.2: *Four-phase XOR@bus* layout decomposition

Some of the blocks presented have been analyzed in chapter 3, while the others need to be introduced yet:

- The purple blocks are *four-phase Inverters@bus* already analyzed in section 3.6;
- The blue blocks are *three-phase Majority Voters@bus* analyzed in section 3.8;
- In orange are represented two *four-phase L-connectors@bus*: the upper one with an upward output while the other with a downward output. These *L-connectors* are different from the ones shown in section 3.4 and 3.5 because of the clock phases involved. The characterisation of these will be discussed in the following section.
- The red blocks are *mono-phase six molecule busses* with the output directed horizontally or vertically. The characterisation of these will be discussed as the ones of the *L-connectors* in the following section.

4.2.1 Mono-phase Six Molecule Wire@bus

In section 3.1 is reported the characterisation of a *Single-Line Mono-Phase Six Molecule Wire*. Considering the bus paradigm involved in the XOR cell design, the *Mono-Phase Six Molecule Wires@bus* identified as the red blocks in figure 4.2 must be treated as completely new cells. In this perspective, the characterisation for each of them must be provided based on their output direction[2]:

Horizontal output

The first analyzed *Mono-Phase Six Molecule Wire@bus* has the output along the horizontal direction and is the direct bus expansion of the cell shown in section 3.1:

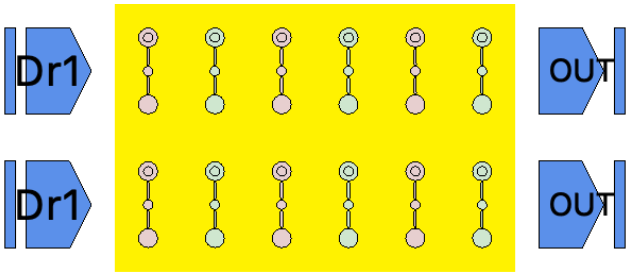


Figure 4.3: *Mono-phase Six Molecule Wire@bus* layout with horizontal output

The automatic termination procedure attaches a copy of the wire in phase two at the output, considering the mono-phase design:

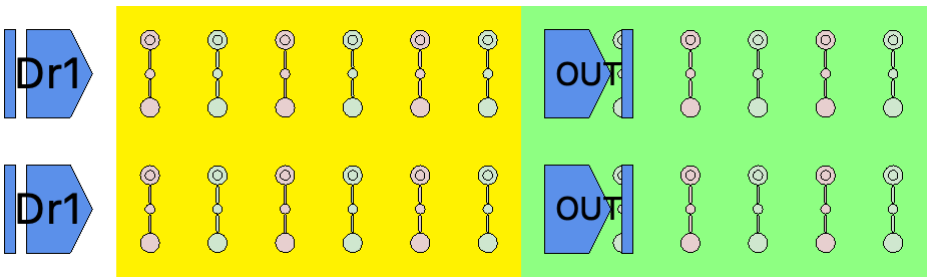


Figure 4.4: *Mono-phase Six Molecule Wire@bus* layout with horizontal output terminated

[2]In these sections are shown the *Debug Mode* results. However, the *User Mode* ones are needed for the XOR cell evaluation through libraries.

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\,V$ to $1\,V$:

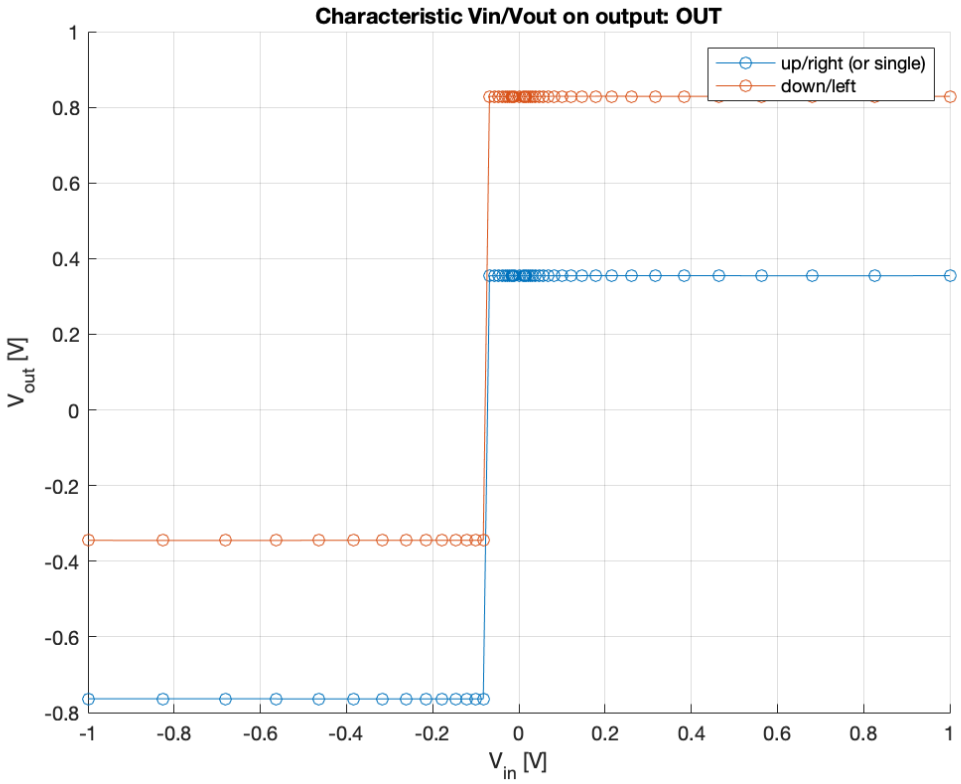


Figure 4.5: *Mono-phase Six Molecule Wire@bus* with horizontal output *Debug Mode* $V_{in} - V_{out}$ characteristic

The $V_{in} - V_{out}$ characteristic has a better behaviour compared to the one in figure 3.3, where the single-line one reveals a strange slowdown for input voltages close to $0\,V$.
The *bi-stability* provided by the bus paradigm is fundamental to ensure cell robustness. The mentioned comparison remarks this concept: the bus driven design must be considered mandatory for the *MolFCN* architectures.

Upward output

Moving the output upward, with a layout identical to the one already shown and using a custom length termination[3] (*8x2 molecules*):

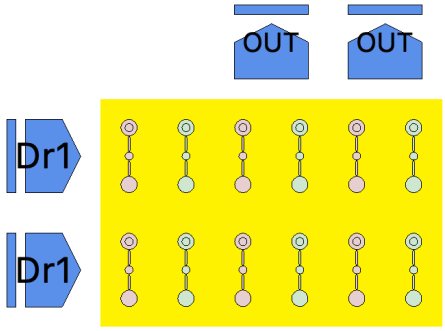


Figure 4.6: *Mono-phase Six Molecule Wire@bus* layout with upward output

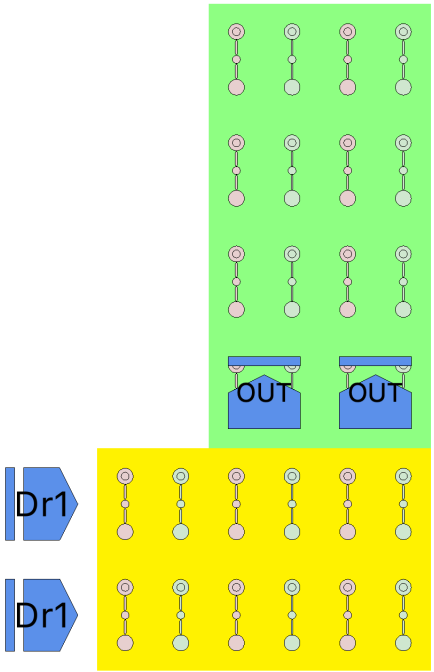


Figure 4.7: *Mono-phase Six Molecule Wire@bus* layout with upward output terminated

[3]The use of a custom length termination does not have a precise reason in this case. It highlights a *bi-stability* saturation effect: once the output has reached up a certain *bi-stability*, the addition of other molecules in the termination does not affect the behaviour. For this reason, the $V_{in} - V_{out}$ characteristic for this cell does not show different output voltage levels compared with 4.11, where an automatic length termination is used.

The *Debug Mode* characterisation is provided with a 31 *points* logarithmic input sweep from $-1\ V$ to $1\ V$:

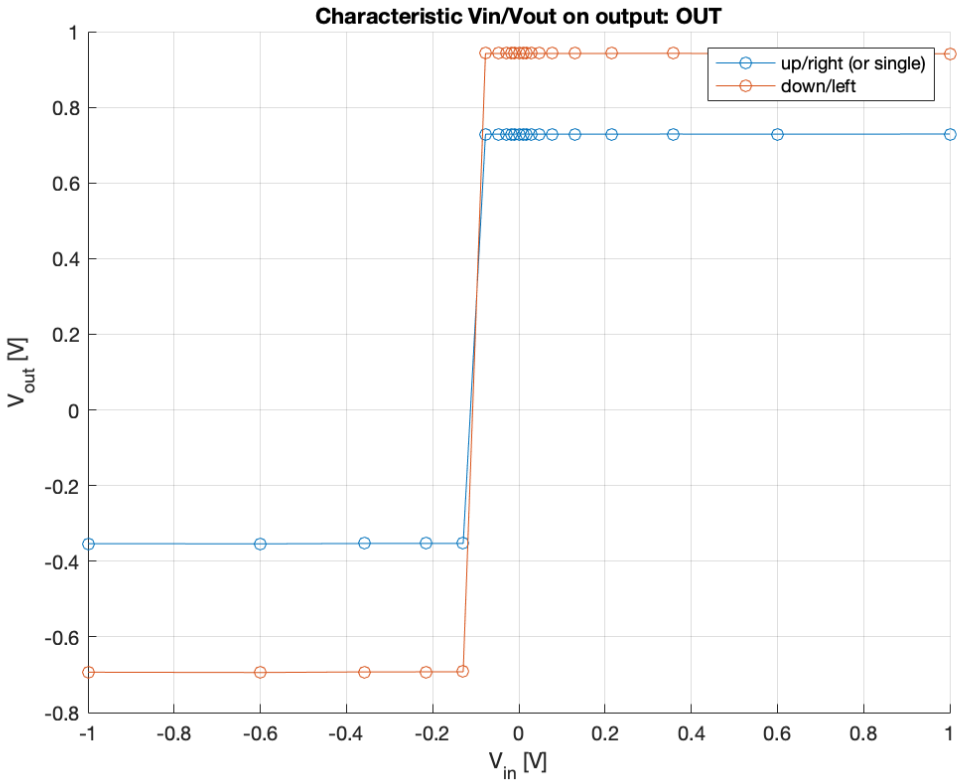


Figure 4.8: *Mono-phase Six Molecule Wire@bus* with upward output *Debug Mode* $V_{in}-V_{out}$ characteristic

The characterisation in *Debug Mode* shown above highlights a condition where the logarithmic sweep can be worse than a linear one. If the voltage transition is not close to the accumulation point ($V_{in} = 0\ V$), the resolution in the characterisation gets inadequate because of the not constant distance between the evaluation points.
A linear sweep does not suffer this criticality but the simulation overhead to reach an accuracy close to the one obtained with the logarithmic sweep in the accumulation point is unapproachable.

Downward output

The last *mono-phase Six Molecule Wire@bus* configuration involves a downward output and an automatic length termination:

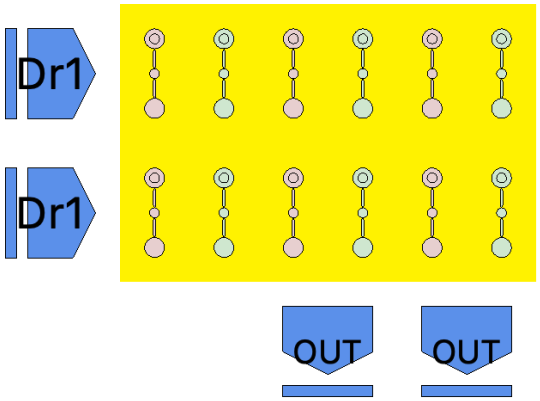


Figure 4.9: *Mono-phase Six Molecule Wire@bus* layout with downward output

In this case the termination does not consider a custom length as for the case with the upward output. The automatic termination is enough to ensure the bi-stability needed.

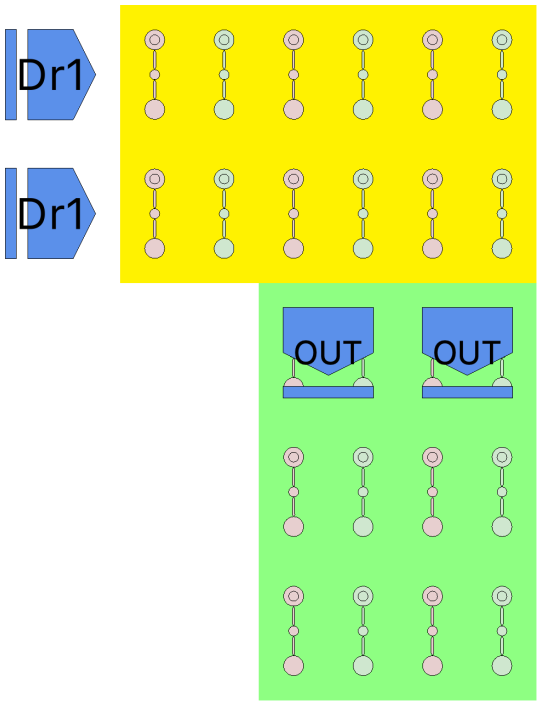


Figure 4.10: *Mono-phase Six Molecule Wire@bus* layout with downward output terminated

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\ V$ to $1\ V$:

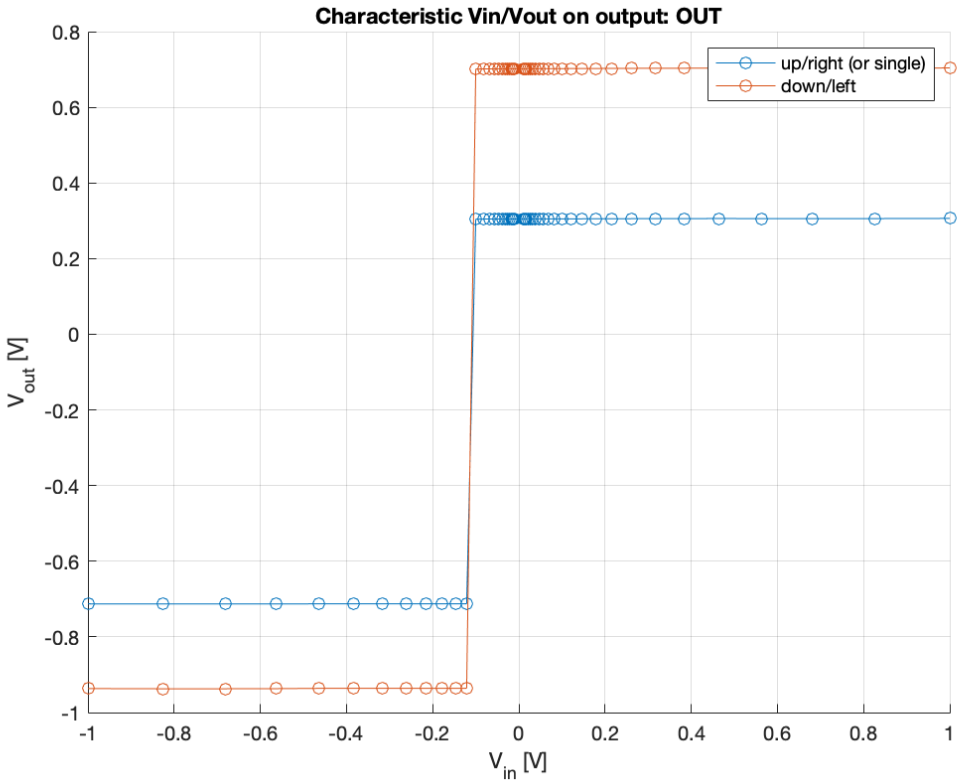


Figure 4.11: *Mono-phase Six Molecule Wire@bus* with downward output *Debug Mode* $V_{in} - V_{out}$ characteristic

In this case, the voltage transition is slightly shifted with respect to the accumulation point ($V_{in} = 0\ V$) but the use of a 51 points sweep provides an acceptable resolution in the characteristic.

4.2.2 Four-phase L-connector@bus with upward output

After the *Mono-phase Six Molecule Wire@bus* analysis, the last blocks to characterise are the *L-connectors* highlighted in orange in figure 4.2.
In sections 3.4 and 3.5, a layout for the *L-connectors* has been presented where the three clock phases returned poor performances: the *Debug Mode* $V_{in} - V_{out}$ characteristics in figures 3.12 and 3.15 show a behaviour particularly biased towards positive input voltages and negative ones respectively.
The poor performances achieved could be related to bad design choices: the number of clock phases involved in the layout may was not sufficient or the layout shape itself was not accurate enough.
For this reason, a different layout design for this cell has been used in the XOR development and it is analyzed here:

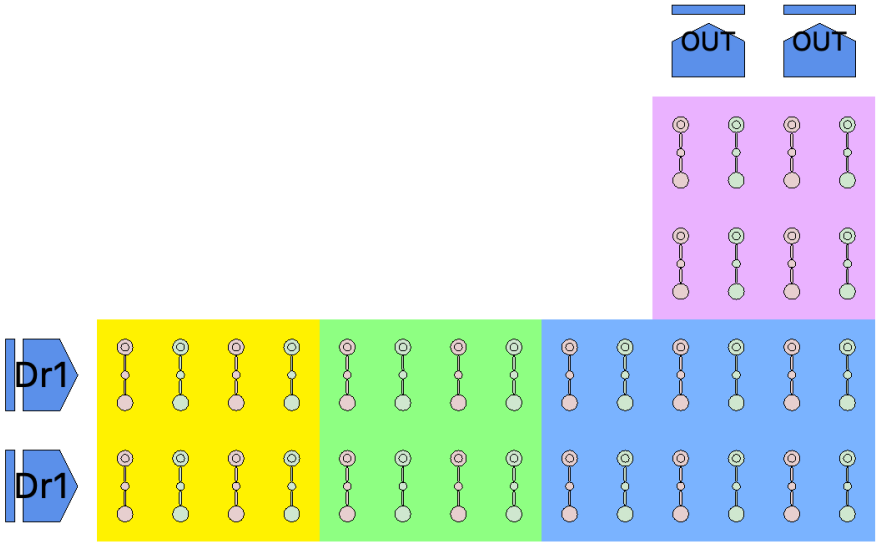


Figure 4.12: *Four-phase L-connector@bus* layout with upward output

Here a fourth phase has been added to the cell design, providing a layout probably complex but with almost the same number of molecules.
Considering the fourth phase length in the layout, the automatic termination procedure could be not enough to ensure the bi-stability needed. The custom length termination feature permits overcoming the problem without particular criticalities. The use of a custom length termination enables for a simulation closer to reality, considering the cell usage environment: in the XOR cell layout shown in figure 4.1, the *L-connectors* have several amounts of logic on their output.

The termination length chosen is a *8x2 molecules* as in other solutions presented before. It provides the bi-stability needed to obtain valid output voltage levels:

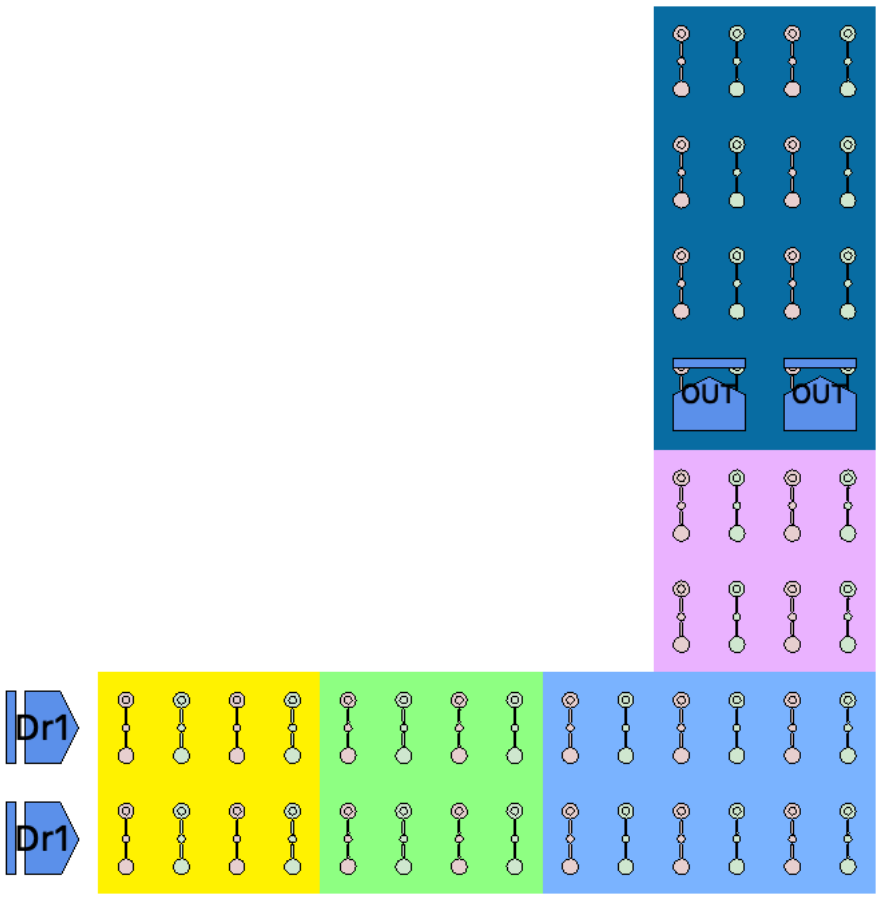


Figure 4.13: *Four-phase L-connector@bus* layout with upward output terminated

The *Debug Mode* characterisation is provided with a 51 *points* logarithmic input sweep from $-1\ V$ to $1\ V$ and the results are much better than the ones shown in figure 3.4 for the old *L-connectors* design:

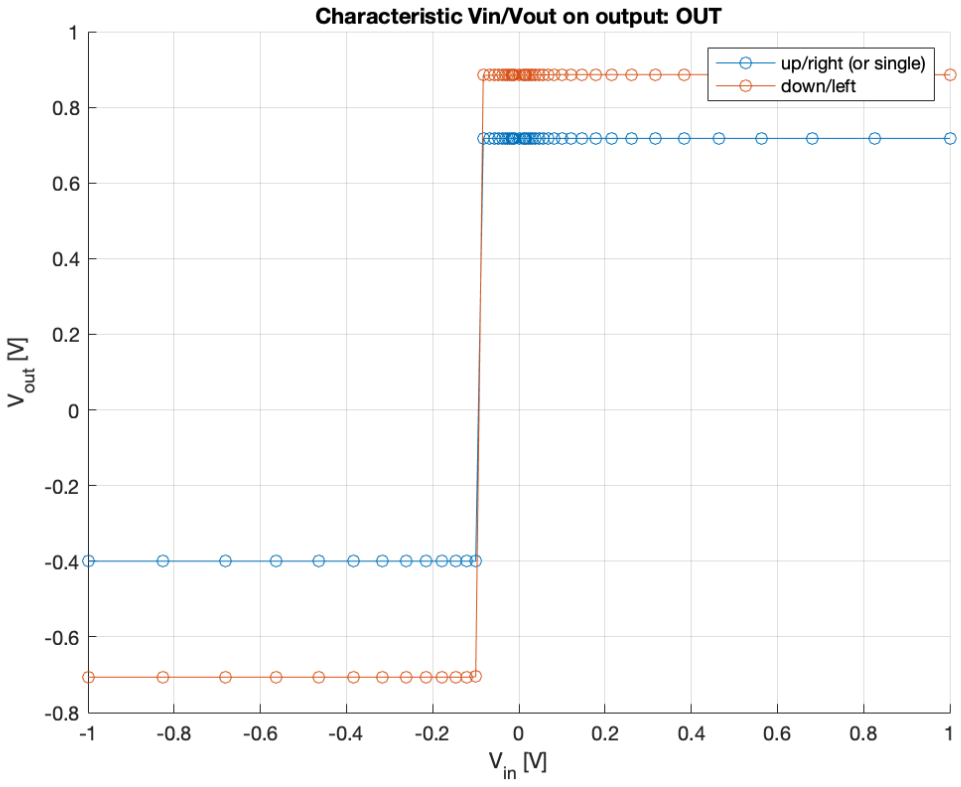


Figure 4.14: *Four-phase L-connector@bus* with upward output *Debug Mode* $V_{in} - V_{out}$ characteristic

4.2.3 Four-phase L-connector@bus with downward output

The same *L-connector* shown before but with a downward output:

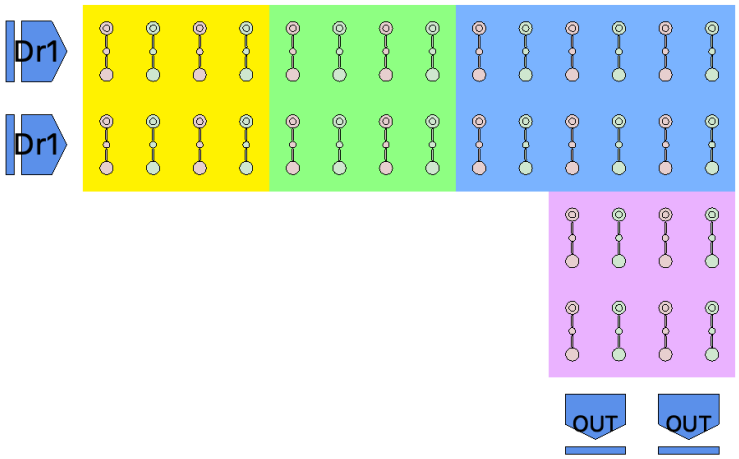


Figure 4.15: *Four-phase L-connector@bus* layout with downward output

Also here, a custom length termination has been chosen for the layout simulation:

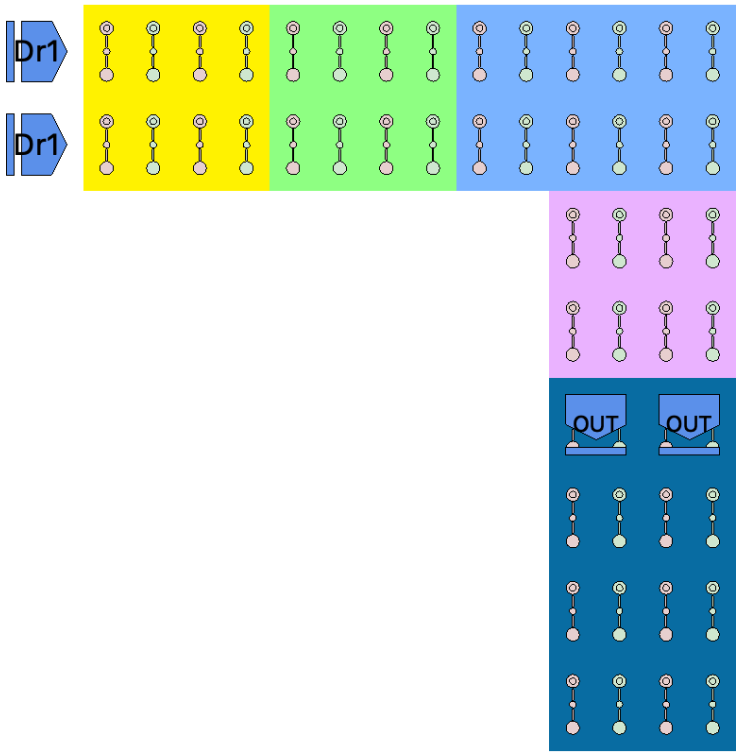


Figure 4.16: *Four-phase L-connector@bus* layout with downward output terminated

The *Debug Mode* characterisation is provided with a 31 *points* logarithmic input sweep from $-1\,V$ to $1\,V$:

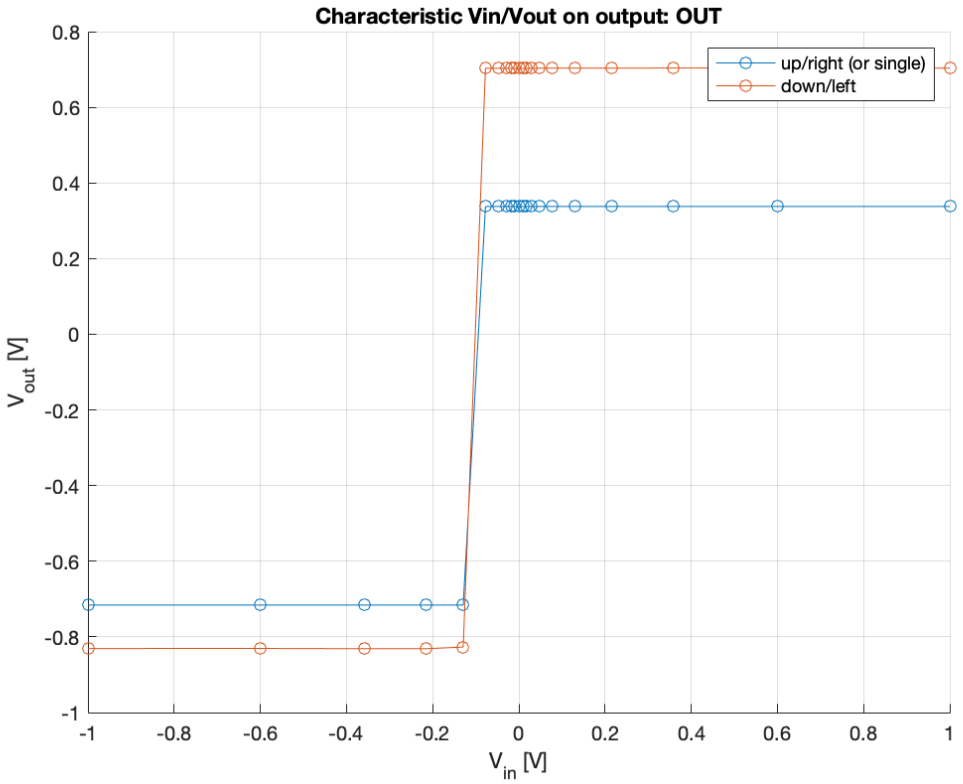


Figure 4.17: *Four-phase L-connector@bus* with downward output *Debug Mode* $V_{in} - V_{out}$ characteristic

Chapter 5

Verification

5.1 Methodology and tentative script

Having characterised each interested cell in *User Mode* deriving the libraries containing the information about their behaviour, these can be now applied to extract the XOR cell output voltage starting from the input dataset definition and bypassing *SCERPA*.
So far, a tool capable to decompose a generic complex cell, creating the architecture to fetch the libraries automatically, does not exist for this technology[1], for this reason, a script capable to manage the XOR cell layout has been designed with the idea of simulating the *netlist-like* future method.
The script follows a simple logic flow where, starting from the XOR inputs, it moves on, cell by cell, extracting the output voltages and using them as inputs for the following cells until the complex cell output is reached up.
The cells' output voltage extraction through libraries follows different strategies depending on the number of inputs of that cell. For this reason, a dedicated function *InOut_eval()* has been developed for the output voltage extraction in order to simplify the script for the XOR layout evaluation.
To introduce the description of the *InOut_eval()* function, the script involved in the XOR cell layout evaluation can be used as a driving example.

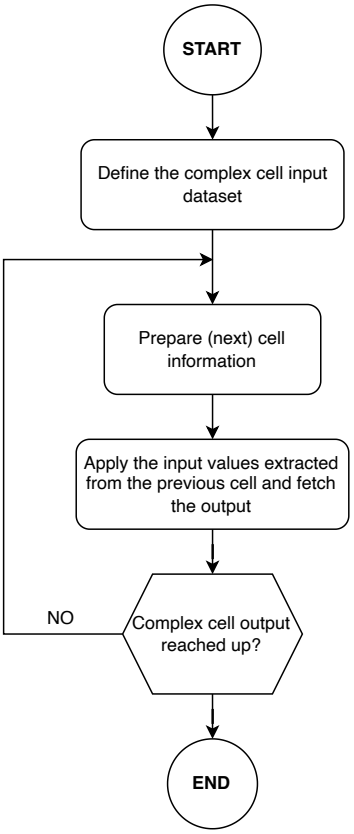


Figure 5.1: Complex cell evaluation flowchart

[1]The design of such a tool could be an interesting project for the future development.

The script flowchart is shown in figure 5.1. A code snippet from the function could clarify the logic flow well:

Listing 5.1: Majority Voter output voltage evaluation

```
%% Upper path MV
charactSettings.LibDeviceName = "majority_voter_bus_3phase_OUT";
circuit.Values_Dr = {
    'Dr1'
    'Dr1_c'
    'Dr2'
    'Dr2_c'
    'Dr3'
    'Dr3_c'
    };

% Dataset -> [Dr1 Dr1_c Dr2 Dr2_c Dr3 Dr3_c]
dataset = [UPPER_Vout_Lconn.OUT_A -UPPER_Vout_Lconn.OUT_A
   UPPER_Vout_inv.OUT_A -UPPER_Vout_inv.OUT_A DrC0 DrC0_c];
UPPER_Vout_MV = InOut_eval( dataset, circuit, charactSettings );
```

To evaluate the output of a cell, the *InOut_eval()* function needs to know the library name, the drivers' organization found inside the library and the input dataset to use for the fetch. Some of the inputs are given by previous cells while others are set up externally considering them fixed.
The XOR cell evaluation script is a cascade of this kind of code snippet, repeated for each cell involved in the XOR layout design. Once the complex cell output is reached, the evaluation script returns a data structure with the overall output voltages evaluated through the entire process. The discussed output voltages can be compared with the ones extracted by *SCERPA* and listed in table 4.3 to estimate the algorithm's goodness.

Few words need to be spent on the *InOut_eval()* function. It has been shown and briefly discussed in the listing 2.13 where the function could be used to bypass the *SCERPA* evaluation. The function needs a few parameters to work properly as shown in listing 5.1: the library name, the running mode set up, the drivers' organization and the input dataset for the evaluation.
The function has itself a *Debug Mode* while its used inside a launch script and this is the reason why the running mode must be always set up[2]. The function fetching operation exploits two different methods based on the cell number of inputs[3]:

- **#input** $<$ **3**: If the number of inputs is low, the output evaluation is performed

[2]The flags reported in table 2.3 handle also the *InOut_eval()* function. The table can be used as a reference for the function setup also outside any launch script.

[3]The *InOut_eval()* function design can handle cells with up to six inputs.

interpolating the entire library table and extracting the output values with a reasonably great resolution, whichever is the input dataset. However, this method cannot be applied with many inputs because of the interpolation process complexity and the memory required to map *N-dimensional* data structures, especially when the libraries contain a lot of entries.

- **#input $\geq$ 3**: If the interpolation method cannot be applied, the input dataset is rounded to the nearest one in the library, in order to extract the output voltages with a simple research algorithm. This approach could appear inappropriate but this is not the case:
  - The error provided by the rounding method strictly depends on the number of steps used in the characterisation process. The characterisation in *User Mode* must be done with a good number of points regardless of everything, ensuring a reasonable negligible error;
  - The cell characteristic is provided to be used in a digital environment where the output levels are stable and saturated to a certain value. The approximation, for this reason, produces an error that does not exist most of the time.

The algorithm goodness will be shown in the next section with the trial on the XOR cell. However, the function is born as a tentative that may work as a start point for a future expansion where it has to be implemented in a complex environment. The development of this evaluation function departs from the aim of this thesis project that would focus on the cell characterisation criticalities and on the creation of what the evaluation function would read as a library.

5.2 Results

Once the evaluation script is ready, it can be run providing instantly as result the output voltages of the XOR cell that can be compared with the ones extracted from *SCERPA* shown in table 4.3:

MATLAB Variable: OUTPUT — Page 1
Mar 6, 2022 — 4:53:43 PM

	Dr1	Dr2	OUT_B	OUT_A	OUT_D	OUT_C
1	-1	-1	-0.7630	0.3548	-0.3446	0.8290
2	-1	1	0.3539	-0.7637	0.8286	-0.3451
3	1	-1	0.3539	-0.7637	0.8286	-0.3451
4	1	1	-0.7630	0.3548	-0.3446	0.8290

Figure 5.2: XOR cell output voltages evaluated using the libraries previously extracted

The direct comparison between the *SCERPA* computed data and the ones reported above using the evaluation script gives back a perfect matching that confirms the evaluation algorithm and the *characterisation tool* goodness. The *SCERPA* computed data are tabled also here to make the comparison easier:

Table 5.1: XOR cell *User Mode* simulation for comparison with the evaluation script extracted data: *csv* library file

$Dr1$	$Dr1_c$	$Dr2$	$Dr2_c$	V_{outB}	V_{outA}	V_{outD}	V_{outC}
-1	1	-1	1	-0.7630	0.3548	-0.3446	0.8290
-1	1	1	-1	0.3539	-0.7637	0.8286	-0.3451
1	-1	-1	1	0.3539	-0.7637	0.8286	-0.3451
1	-1	1	-1	-0.7630	0.3548	-0.3446	0.8290

In terms of simulation time, the evaluation of a single input dataset through the script costs $\sim 200\ ms$ on a consumer mobile CPU: Intel® Core-i5 5257U 2.7GHz:

```
>> xor_lib_evaluated
Elapsed time is 0.188539 seconds.
Completed
```

Figure 5.3: Evaluation time required to extract a set of output voltages given an input dataset using libraries for the XOR cell

Considering the four input datasets needed to extract every output voltage for the comparison, a total elapsed time rounded to 800 ms can be taken as reference.
Using the same *MATLAB* instrument to measure the simulation time for the same XOR cell using *SCERPA*:

```
>> xor_2nd_ver
Elapsed Time 17.321333 minutes.
Completed
```

Figure 5.4: *SCERPA* simulation time in *User Mode* to test four independent input datasets

The difference in terms of time overhead is huge as expected considering that the library-based method rely on a pre-computation strategy. A summary table with the difference is reported below:

Table 5.2: Time overhead comparison between the *SCERPA*'s driven simulation and the library-based one

SCERPA's based simulation	*Library-based* simulation	Difference
$\sim 17.32min$	$\sim 755\ ms$	$\sim 1375x$

From a different point of view, the pre-computation strategy does not eliminate the simulation complexity. It just moves it out of the complex architecture simulation, anticipating it with the idea of re-use the computed data where is possible. It is also true that every new complex architecture could need for the characterisation of new cells until the database gets well-stocked with an appropriate set of cells libraries. This mid-step can slow the onset of visible benefits but, after a first libraries collection, the development of new and astonishing architectures can start enjoying the advantages of a library-based strategy.

Chapter 6

Conclusion and future perspectives

The *chracterisation tool* presented would be a piece of a complex software architecture for the simulation of systems not feasible so far. Clearly, it cannot be enough to approach the *VLSI* paradigm as today is done with the CMOS for instance, but it represents a good starting point together with *SCERPA*.
Actually, the *chracterisation tool* is capable to extract the $V_{in} - V_{out}$ behaviour of a cell with an approach fully automatic and reliable results. However, a generic tool for the cell layout re-organization in netlist is not available up to now, while the *InOut_eval()* function is already generic but could be improved for a possible massive usage with *MolFCN* architectures composed by a lot of cells.
A tentative future perspective could be related to the development of a tool capable to perform the automatic decomposition of a generic system layout in basic cells.
Once these got recognized, it could fetch the correct libraries, evaluating the system behaviour as it was shown for the XOR cell in the previous sections.
On the other hand, considering the *characterisation tool* implementation, for sure better procedures can substitute the non-optimal ones in the code, improving the actual performances. It is also true that this is not a stand-alone tool and, working with *SCERPA*, the performances bottleneck is not its fault. For this reason, the priority should be focused on the cell characterisation features as the *Self Operating Area (SOA)*.
The *MolFCN* technology relies on the electrostatic interaction for the information propagation, where it is the relative position between QCAs that induces the propagation path. Two close independent cells would interact through the generated electrostatic field introducing cross-talk noise in the signal and invalidating the information in the worse cases. The *SOA* evaluation would be an area parameter to compute for each characterised cell with which be compliant in the system design phase to avoid the problem discussed above.
The *characterisation tool* has been developed with the idea of being expanded, permitting an easy integration of sub-tools and other functions.

For instance, it provides a *text file* for each characterised cell, ready to be filled with any information as discussed in section 2.2.6.
Another useful feature for the *characterisation tool* could be the implementation of a dynamic mesh for the input value generation in simulation. For instance, the use of the logarithmic sweep reduces the computational complexity with respect to the linear one, assuming to fix the accuracy, because of the accumulation of points in region of $V_{in} = 0\ V$. Ideally, the $0\ V$ is the input voltage where the level transition is more likely to be. However, this is not always true as shown with the results in chapter 3. A dynamic mesh could adapt the accumulation point based on the cell layout and the clock phases to obtain a more valuable characterisation without paying the overhead of a finer mesh.

As it has been shown, the improvement possibilities are constrained just by our imagination and the ones listed above are just a starting idea of what could be done. For what I am concerned, the tool designed represents a valid instrument to explore the astonishing design space of the *MolFCN* technology. It can help saving time for any designer and researcher involved with it. Moreover, the time overhead has been presented as the main limitation in the *SCERPA* usage, and this tool provides promising evidence in that field.

Bibliography

[1] G. E. Moore. "Cramming more components onto integrated circuits". *Electronics, vol. 38, no. 8*, April 1965.

[2] M. Roser and H. Ritchie. "Technological progress", 2020. URL https://ourworldindata.org/technological-progress.

[3] S. Srivastava, S. Sarkar, , and S. Bhanja. "Estimation of upper bound of power dissipation in QCA circuits". *IEEE Transactions on Nanotechnology*, 8(1):116–127, January 2009.

[4] C. S. Lent, B. Isaksen, , and M. Lieberman. "Molecular quantum-dot cellular automata". *J. Amer. Chem. Soc.*, vol. 125(no. 4):pp. 1056–1063, 2003.

[5] A. Pulimeno, M. Graziano, D. Demarchi, and G. Piccinini. "Towards a molecular QCA wire: Simulation of write-in and read-out systems". *Solid-State Electron.*, vol. 77:pp. 101–107, November 2012.

[6] Y. Wang and M. Lieberman. "Thermodynamic behavior of molecular-scale quantum-dot cellular automata (QCA) wires and logic devices". *IEEE Trans. Nanotechnol.*, vol. 3(no. 3):pp. 368–376, September 2004.

[7] A. Orlov, A. Imre, G. Csaba, L. Ji, W. Porod, and G. H. Bernstein. "Magnetic quantum-dot cellular automata: Recent developments and prospects". *J. Nanoelectron. Optoelectron.*, vol. 3(no. 1):pp. 55–68, March 2008.

[8] A. O. Orlov. "Realization of a functional cell for quantum-dot cellular automata". *Science*, vol. 277(no. 5328):pp. 928–930, August 1997.

[9] Y. Ardesi, G. Turvani, G. Piccinini, and M. Graziano. "SCERPA Simulation of Clocked Molecular Field-Coupling Nanocomputing". *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 29(no. 3):pp. 558–567, March 2021.

[10] Y. Ardesi, R. Wang, G. Turvani, G. Piccinini, and M. Graziano. "Scerpa: a self-consistent algorithm for the evaluation of the information propagation in molecular field-coupled nanocomputing". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39(no. 10):pp. 2749–2760, October 2020.

[11] A. Pulimeno, M. Graziano, A. Sanginario, V. Causa, D. Demarchi, and G. Piccinini. "Bis-Ferrocene Molecular QCA wire: Ab Initio Simulations of Fabrication Driven Fault Tolerance". *IEEE Transactions and nanotechnology*, vol. 12(no. 4):pp. 498–507, July 2013.

[12] IEEE. "International Technology Roadmap for Devices and Systems: More Moore", 2020.

[13] Y. Ardesi, L. Gnoli, M. Graziano, and G. Piccinini. "Bistable propagation of monostable molecules in molecular field-coupled nanocomputing". *Proc. 15th Conf. Ph.D. Res. Microelectron. Electron. (PRIME)*, page 225–228, July 2019.

[14] F. Riente, U. Garlando, G. Turvani, M. Vacca, M. R. Roch, and M. Graziano. "MagCAD: A Tool for the Design of 3D Magnetic Circuits. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 3:65–73, 2017. doi: 10.1109/JXCDC.2017.2756981.

[15] U. Garlando, F. Riente, and M. Graziano. "FUNCODE: Effective Device-to-System Analysis of Field Coupled Nanocomputing Circuit Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2020.

[16] L. Verstraete, P. Szabelski, A. M. Bragança, B. E. Hirsch, and S. De Feyter. "Adaptive self-assembly in 2D nanoconfined spaces: dealing with geometric frustration". *Chem. Mater.*, 31(n°17):6779–6786, 2019.

[17] J. N. Randall et al. "Atomic precision lithography on si". *J. Vac. Sci. Technol. B, Microelectron.*, 27(6):2764–2768, 2009.

[18] R. Wang, M. Chilla, A. Palucci, M. Graziano, and G. Piccinini. "An effective algorithm for clocked field- coupled nanocomputing paradigm". *2016 IEEE Nanotechnology Materials and Devices Conference (NMDC)*, page 1–2, 2016.