POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

ALiAS - Analog Logic-in-Memory Arrays Synthesizer

Relatori:
Prof. Mariagrazia Graziano
Prof. Marco Vacca
Prof. Maurizio Zamboni

Candidato:
Fabrizio Ottati

Aprile 2020

Table of contents

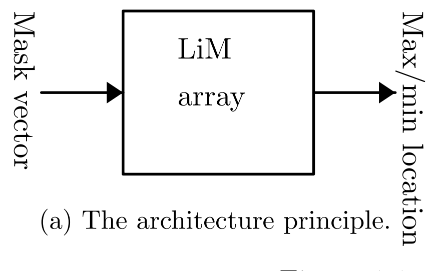
Chapter 1

Abstract

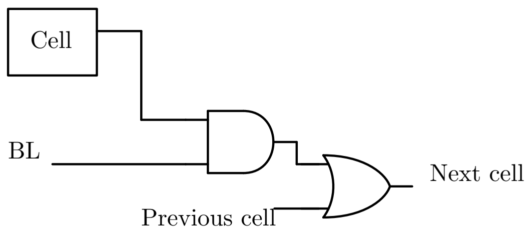
The Von-Neumann paradigm is based on the data exchange between CPU and memory: the processor reads the data from the memory, elaborates them and, then, writes the results back. However, CPU and memory do not work at the same speed: while in computing units the CMOS technology has kept progressing year after year, memory CMOS circuits have not, being fundamentally limited by the request of higher storage density (memory cells per unit area) that almost nullifies the speed enhancement brought by technology scaling.

For this reason, research and industry have searched for other solutions that allow to overcome the Von-Neumann bottleneck. One approach that is being explored is the Logic-in-Memory (LiM) one: the data are processed inside the memory array itself, without trasnferring them to the CPU registers.

In this work, a standard Content Addressable Memory has been redesigned in order to perform the maximum/ minimum value search in the array using the LiM approach. In particular, three variants of a memory cell design, which has been proposed in [1], are presented, whose general logic scheme is shown in Figure 1.1b. The algorithm execution is based on the bitwise-AND logic operation between the memory content and an external data word, called "mask vector". The logic scheme of the architecture is shown in Figure 1.3



(a) The architecture principle.



(b) The LiM cell.

Figure 1.1: Architecture and cell.

First, the memory cell circuit has been designed by hand in Cadence Virtuoso and tested inside an array; after this, the design characteristics have been tuned through several simulations and, then, the fundamental components (memory cell and sense amplifier) netlists have been extracted; subsequently, these have been embedded in a software tool called ALiAS (Analog Logic-in-Memory Arrays Synthesizer), designed by me.

The tool generates in a dynamic way (i.e. following the user specifications) the architecture netlist; the obtained testbench is simulated and characterized, extracting the performance and energy consumption measurements; the user chooses arbitrarily the array dimensions, the memory signals drivers characteristics (if they have to be ideal or real-world ones), the simulation speed (hardware resources assigned to the simulation) and clock period duration. Furthermore, standard memory arrays (SRAM and CAM) can be generated by the tool, in order to be used as reference in the results evaluation, and some results are presented in Figure 1.4.

In these graphs, the delays related to memory operations, such as write and read ones (indicated by `Write1Delay`, `Write0Delay`, `Read1Delay` and `Read0Delay`), the CAM search operation (indicated by `MatchDelay`) and the AND operation (indicated by `ANDDelay`), are presented. These have been estimated for many values (from 8 to 256 bits) of the memory width, with memory height set to 256 bits.

As expected, the LiM cells, denoted with the `ANDModel`, `ANDSTModel` and `ANDDYNModel` labels in the graphs of Figure 1.4, have worse performance in standard memory operations with respect to classical implementations, because of their higher circuital complexity; this can be noticed by the fact that, in `Write1Delay`, `Write0Delay`, `Read1Delay` and `Read0Delay`, the LiM cells have a larger delay associated with respect to CAM and SRAM.

However, the AND operation (`ANDDelay`) performs way better than the CAM search one (`MatchDelay`), and its implementation has no influence on the search performance thanks to the sensing scheme adopted, as it can be noticed in the `MatchDelay` graph where all the architectures are characterized by the same delay value.

In conclusion, adding complexity to the memory cell leads to decreased performance in standard operations; however, the possibility to perform the maximum/minimum value search completely in memory allows to avoid the whole memory content transferring to the CPU, which is a highly time and energy consuming task.

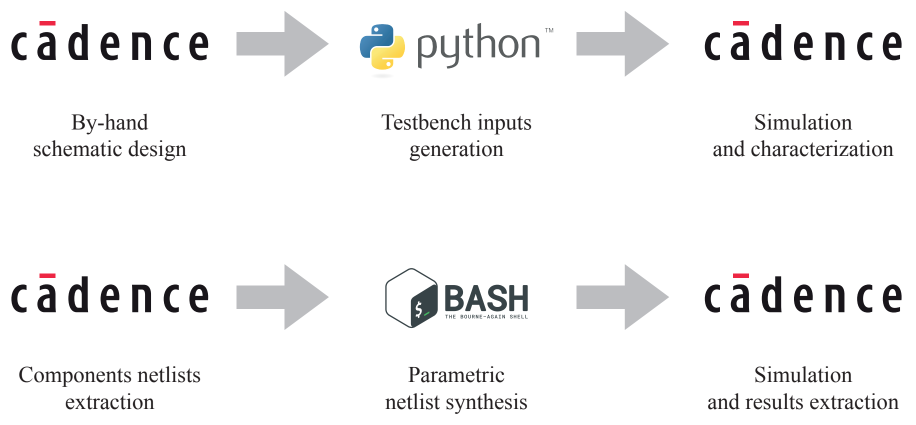

cadence
python
cadence
By-hand schematic design
Testbench inputs generation
Simulation and characterization
cadence
BASH THE BOURNE-AGAIN SHELL
cadence
Components netlists extraction
Parametric netlist synthesis
Simulation and results extraction

Figure 1.2: Design flow.

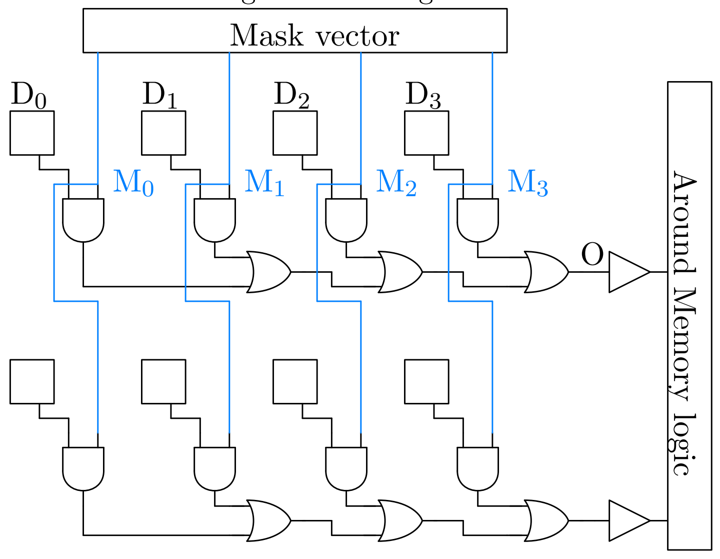


Figure 1.3: Array implementation.

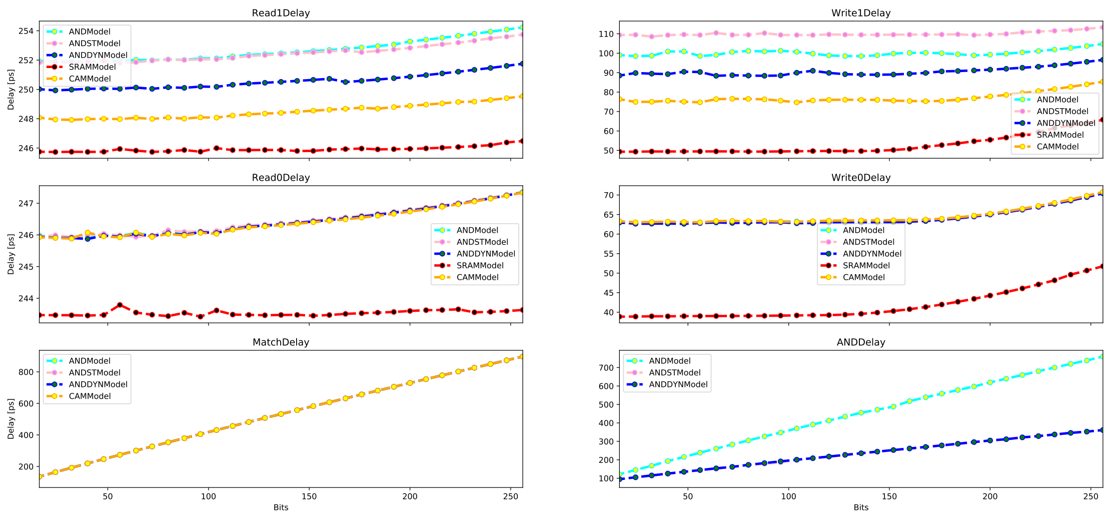


Figure 1.4: Delay of memory operations as function of word width

Chapter 2

State Of The Art

2.1 Static Random Access Memories

The Static Random Access Memory (SRAM) still plays a crucial role in VLSI circuits as embedded memory [2]. One of the most used cell topologies remains the 6T one, which is shown in Figure 2.1.

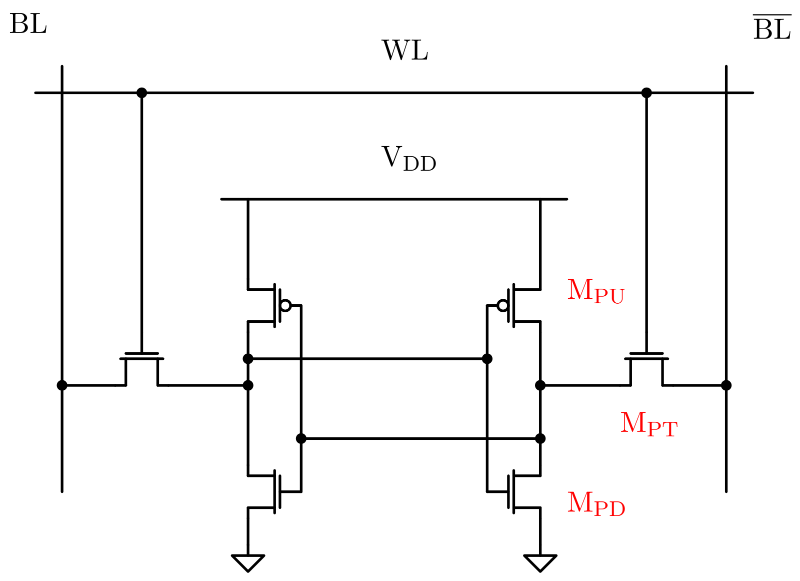


Figure 2.1: 6T SRAM cell

There are many motivations for the success of the SRAM as embedded memory:

- it provides the highest random access speed; hence, it is widely used for buffer and register memories in ASICs.
- it allows an high integration with logic circuits, because of its high compatibility of process and supply voltage.

As more processing elements are being integrated on the same chip, the request for embedded memory has increased through the years, in order to satisfy the increasing demand for larger memory bandwidth and capacity. For this reason, the 6T topology is still preferred, since it minimizes the area occupied by the memory cell.

It is well known that the memory core of the cell is made by two CMOS cross-coupled inverters, also referred to as "bistable ring", as reported in Figure 2.1. The cross-coupled inverters allow to retain the stored information as long as the cell is supplied, differently from the case of the Dynamic RAM, in which a "refresh" cycle is needed to allow the data retention.

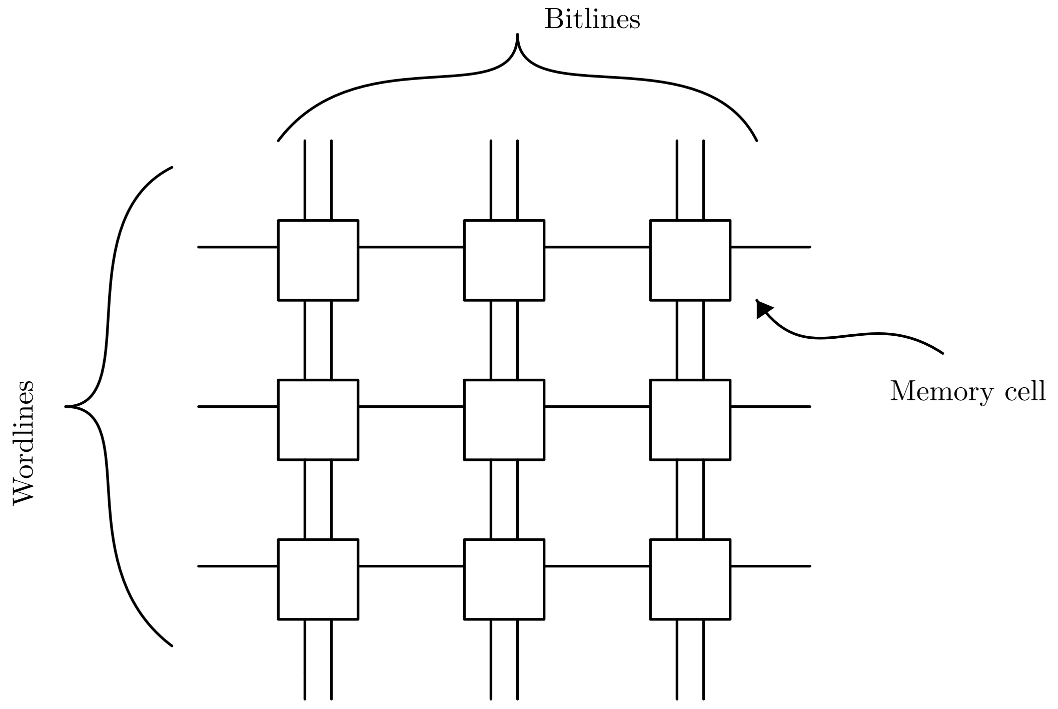


Figure 2.2: Typical memory array

In Figure 2.2 the typical memory array is shown.

The cells are arranged in an array of rows and columns: in the vertical direction, the cells are connected by the bitlines while, in the horizontal direction, they all share a single line, called wordline.

2.1.1 SRAM metrics

The Static Noise Margin (SNM) [3, 4] of a cell is defined as the maximum amount of noise that the memory cell can tolerate while retaining its data. The SNM is defined in three conditions:

- when the cell is not accessed (data retention SNM).
- when the cell is accessed during a read operation (read SNM).
- when the cell is accesses during a write operation (write SNM).

The way the noise margin is related to the different memory operations is presented in the following.

2.1.2 Read Operation

A read operation is performed through two subsequent cycles:

- the bitlines (BL) are precharged to the logic '1', which voltage level (usually) is the supply voltage V_{DD}.
- the cell to be read is selected by asserting the wordline (WL), enabling, in this way, the pass-transistors (PT) of the cell. One of the two bitlines is discharged by the cell: to be precise, the bitline on the side of the cross-coupled inverter on which the logic '0' is stored.

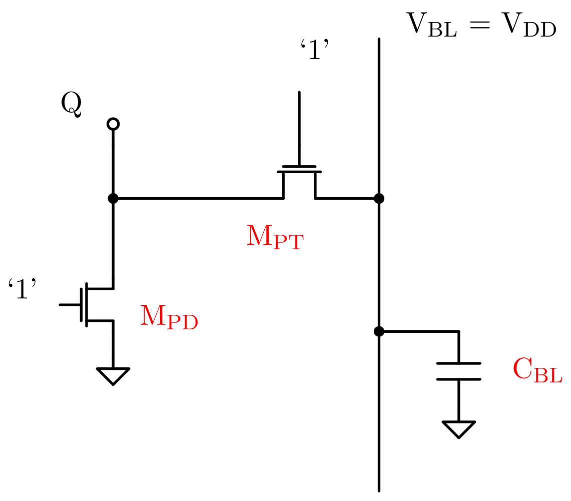


Figure 2.3: 6T SRAM cell, read section

The read section of the cell is shown in Figure 2.3.

A MOSFET can be modeled with a resistor when it is turned on and it operates in the linear region; the resistance value can be roughly estimated in the following way:

$$R_{MOS} = \frac{1}{\mu_n \cdot C_{OX} \cdot (W/L) \cdot (V_{DD} - V_{TH})}$$

where:

- μ_n is the electron mobility in the channel of the transistor.
- C_{OX} is the gate-oxide capacitance.
- (W/L) is the aspect ratio of the transistor.

- V_{DD} is the voltage applied to the gate of the transistor; in this case, this is equal to the supply voltage of the circuit.
- V_{TH} is the threshold voltage of the transistor.

When the pull-down transistor (PD) has a logic '1' on its gate, it connects the bitlines to ground through the pass transistor; since the bitline presents a large parasitic capacitance C_{BL}, given by the sum of the parasitic access capacitance of the cells connected to the line (in particular, the drain capacitance of the pass transistor) and the parasitic capacitance of the metal line, a charge-sharing phenomenon occurs: some charge is transferred from the bitline to the storage node Q; hence, the potential of Q is raised from the ground voltage.

The voltage value reached by the node Q (V_Q) depends on the voltage divider that occurs between the pass and pull-down transistors:

$$V_Q = \frac{R_{PD}}{R_{PD} + R_{PT}} \cdot V_{BL}$$

Since the output of the inverter on the right (Figure 2.1) is connected to the input of the one on the left, it may happen that V_Q rises above the trip-point of the inverter and, so, the cell content is overwritten, as shown in Figure 2.4; hence, the read operation can lead to the corruption of the data stored in the cell.

Referring to Figure 2.4, V_{WL} represents the voltage applied to the WL and, so, to the gate of the pass transistors; $V_{Q'}$ represents the voltage on the node which is placed on the other side of the bistable ring; V_{BL} represents the voltage on the bitline connected to the node Q through the pass transistor.

The correct behaviour of the cell during a read operation is shown in Figure 2.5. The voltage V_Q does not reach the inverter trip-point and, so, the data is read correctly from the cell (the bitline is discharged to the logic '0'), without corrupting it.

To avoid this phenomenon, it has to be ensured that:

$$R_{PD} < R_{PT}$$

This can be obtained by sizing the pull down transistor with an aspect ratio (W/L) larger than the one of the pass transistor:

$$(W/L)_{PD} > (W/L)_{PT}$$

This, of course, leads to an increase in the cell area.

The ratio between the electrical strengths of the pull-down and pass transistors is referred to as β-ratio [5], and it can be informally defined in the following way:

$$\beta_{SRAM} = \frac{R_{PT}}{R_{PD}}$$

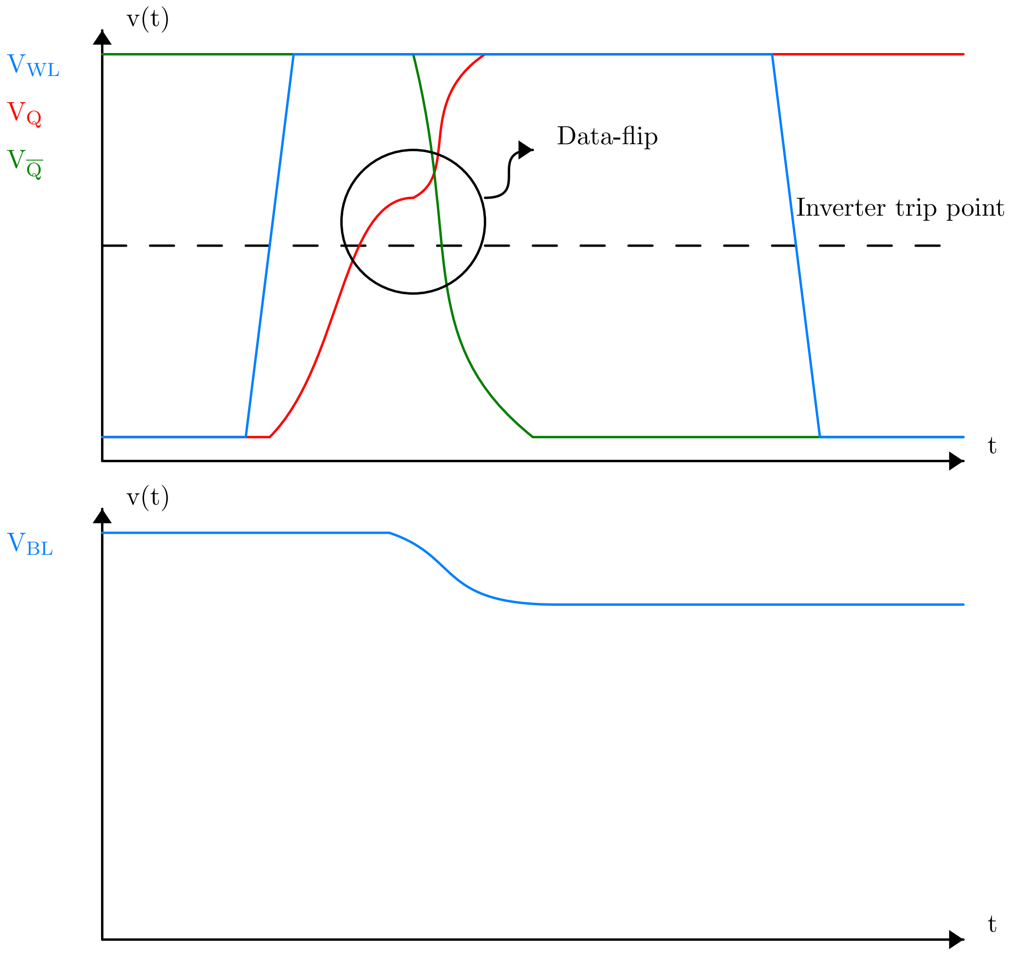


Figure 2.4: Read operation, data-flip

It has to be remarked that the rate with which the bitline is discharged is determined by the current absorbed by the cell during the read operation (I_{cell}); the latter depends on the aspect ratio of the pass and pull-down transistors; since these are in series to the bitline during the discharging phase, the transistor with the smaller aspect ratio determines the value of I_{cell}; hence, the pass transistor aspect ratio cannot be minimally sized, in order to guarantee a minimum value for the current absorbed by the cell (and also for other reasons, as discussed in subsection 2.1.3).

2.1.3 Write Operation

The write operation is performed by selecting the cell using the wordline and by putting the data to be written on the bitlines. For example, if a logic '1' has to be written to the cell, a logic '1' (i.e. a voltage equal to V_{DD}) is put on BL, while a logic '0' (i.e. a voltage equal to 0 V) is applied on $\overline{\text{BL}}$.

It has been stated that the pull-down and pass transistors are sized so that V_Q does

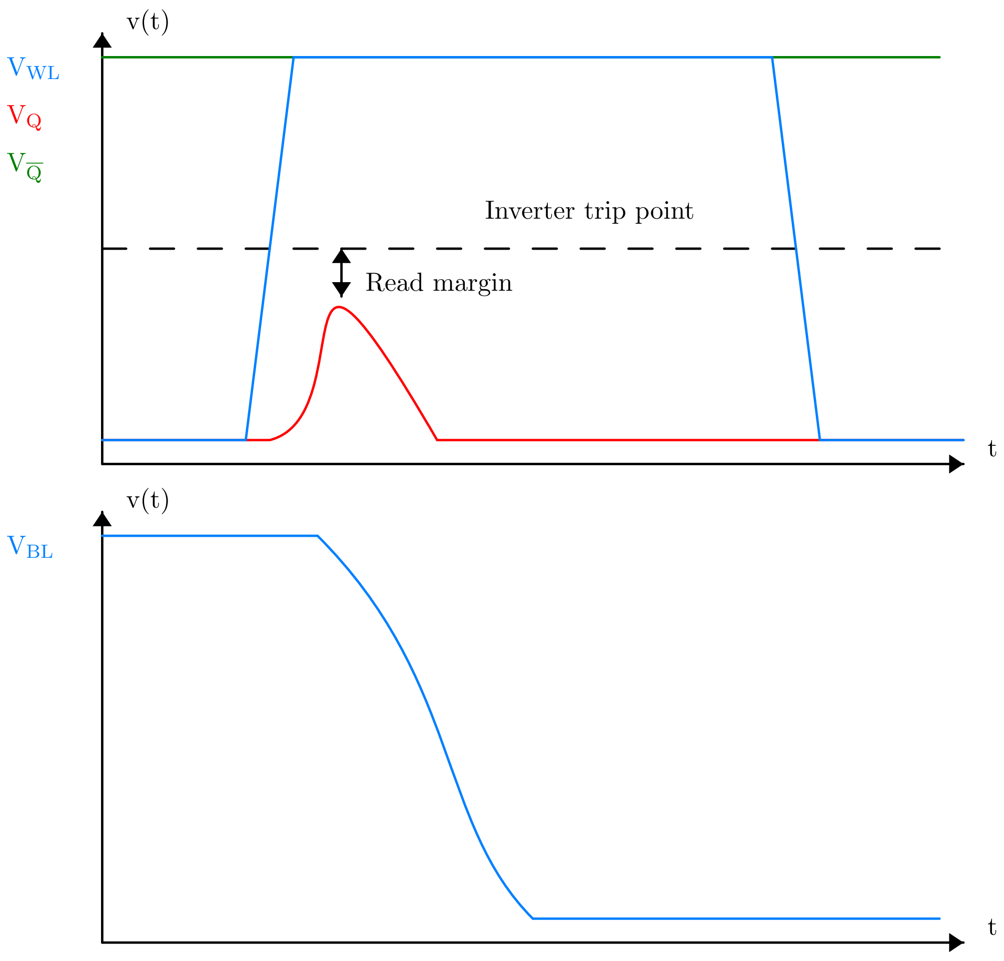


Figure 2.5: Read operation, correct behaviour

not rise above the trip-point of the bistable ring (Figure 2.3) when a logic ‘1’ is applied to the bitline; hence, a logic ‘1’ cannot be forced on the side on which a logic ‘0’ is stored.

For this reason, only by forcing a logic ‘0’ on the cell side on which a logic ‘1’ is stored, the cell content can be overwritten. This is automatically achieved in the differential structure shown in Figure 2.1, where two bitlines are connected to the cell sides.

The write section is shown in Figure 2.6.

Also here, a voltage divider arises between the pull-up (PU) and pass transistors; since the pull-up is connected to V_{DD} while the pass transistor is connected to ground, the voltage at the node Q is given by:

$$V_Q = \frac{R_{PT}}{R_{PT} + R_{PU}} \cdot V_{DD}$$

Hence, V_Q results to be larger than 0 V. This implies that if V_Q does not lower below the bistable ring trip-point, the data is not written to the cell, as shown in Figure 2.7.

To avoid this phenomenon, it has to be ensured that:

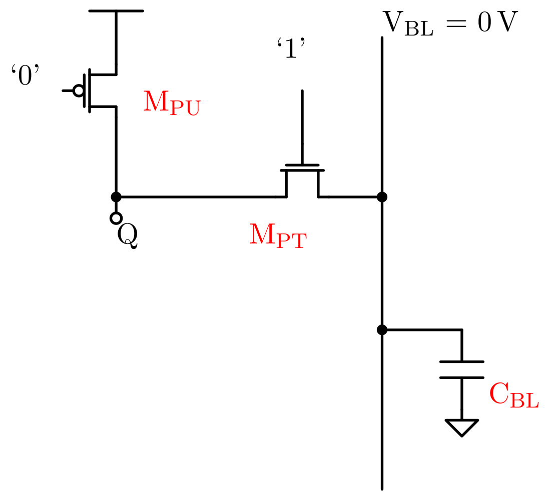


Figure 2.6: SRAM 6T, write section

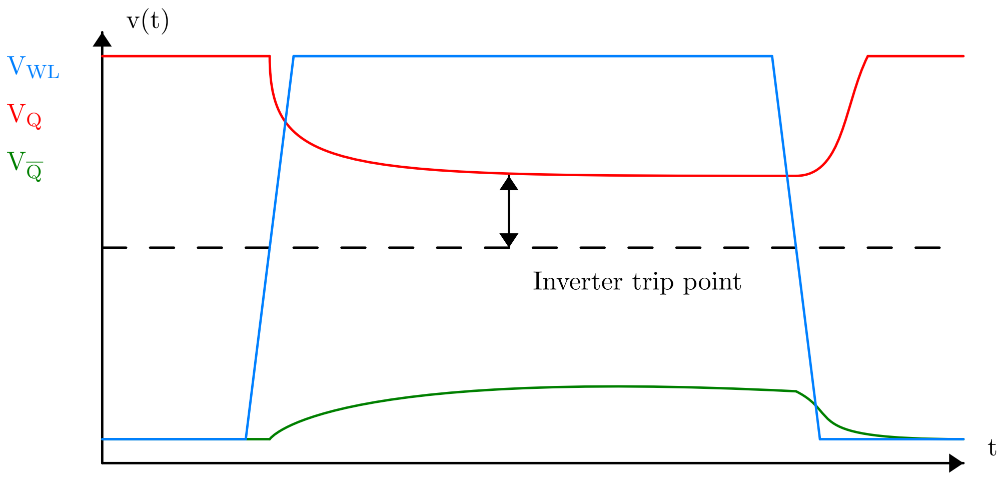


Figure 2.7: SRAM 6T, unsuccessful write operation

$$R_{PT} < R_{PU}$$

This can be achieved by choosing:

$$(W/L)_{PT} > (W/L)_{PU}$$

Since the mobility of the holes in the channel (μ_p) is lower than the one of the electrons (μ_n), the difference in the aspect ratio of the two transistors needed in order to achieve a certain write stability is lower than in the case of the pull-down and pass transistors for the read operation.

By properly sizing the pull-up and pass transistors [5], a successful write operation can be performed, as it is shown in Figure 2.9.

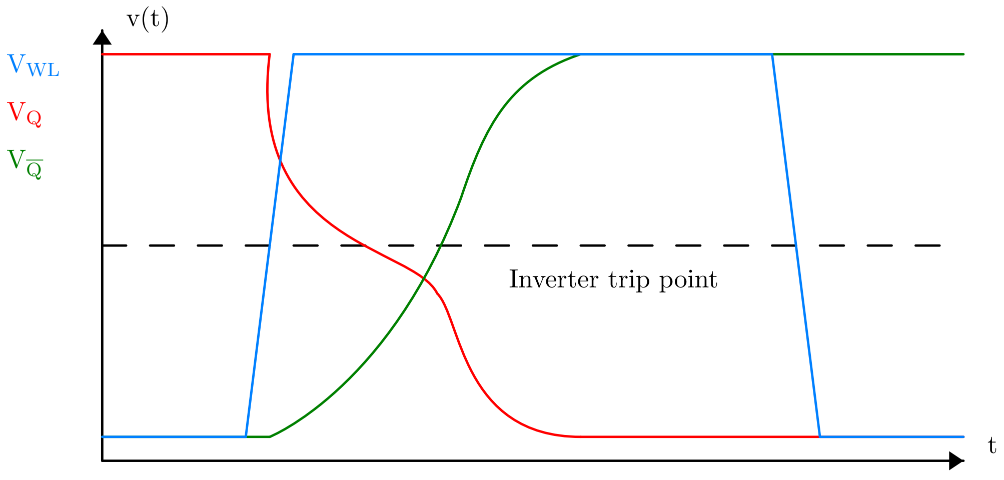


Figure 2.8: SRAM 6T, successful write operation

The ratio between the driving-strengths of the PT and PU transistors is referred to as γ-ratio [5]. It can be informally defined in the following way:

$$\gamma_{SRAM} = \frac{R_{PU}}{R_{PT}}$$

In conclusion: the pass transistor has to be sized with an aspect ratio larger than the pull-up; the pull-down has to sized with an aspect ratio larger than the pass transistor; the aspect ratio of the latter has to be large enough to provide a minimum value of the cell current I_{cell}.

2.1.4 Data Retention

When the memory cell is not selected, its pass transistors are disabled and the cell is said to be in “data retention mode”.

One could think of lowering the supply voltage of the cell ($V_{DD_{cell}}$), in order to reduce the static power consumption of the memory; however, below a certain value of $V_{DD_{cell}}$, the bistable ring is not able to retain the stored data and the information is lost [5]. This lower bound of the cell supply voltage is called “data retention voltage” (V_{hold}).

2.1.5 Cell current distribution

As stated before, the current absorbed by the cell, I_{cell}, determines the time needed by the cell to discharge the bitline, together with the latter parasitic capacitance and the discharge level needed, which is the bitline voltage level to be reached in order to allow the sense amplifier to correctly sense the bitline voltage variation.

The value of this current depends on the electrical strength of the pull-down and pass transistors, which are in series to the bitline; the electrical strength, in turn, is determined by the aspect ratio of those transistors. This represents an issue in terms

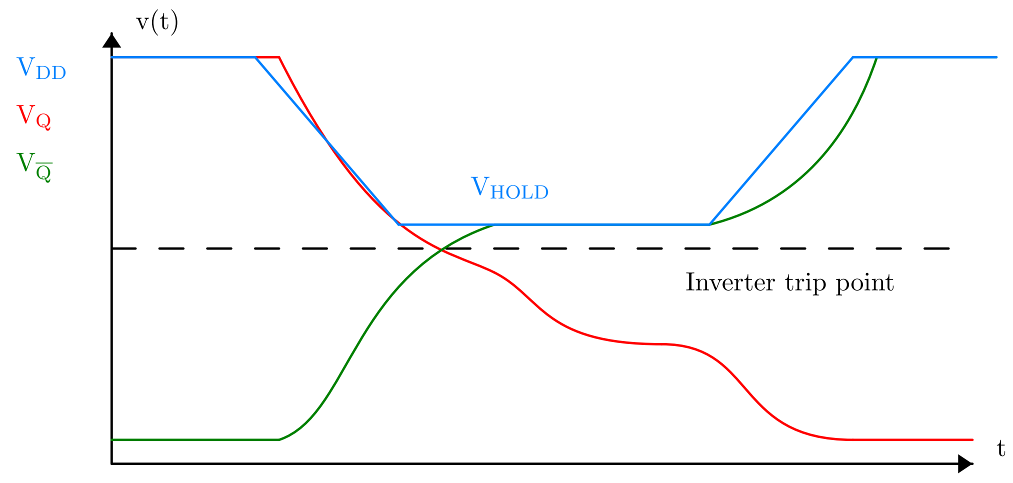


Figure 2.9: SRAM 6T, failed data retention with V_{DD} lowering

of area constraints of the cell, because the pull-down has to be designed with a aspect ratio larger than the access transistor, as discussed in subsection 2.1.2; hence, since to obtain a large cell current a large aspect ratio is needed for the pass transistor, an even larger one is required for the pull-down. This results in an increase in the cell area.

The technology scaling has led to a reduction in the bitline parasitic capacitance value, even if this has not been so large because, as the cell size is reduced, more cells are integrated in the same array, resulting in a limited reduction (or even in an increase, since interconnections do not scale as transistor dimensions) of the parasitic capacitance.

With the scaling, at the same time, the supply voltage of the cells V_{DD} has to be scaled: this results in a reduction of the transistors overdrive voltage, $V_{GS} - V_{TH}$, and, so, of I_{cell}; since with the the transistors size scaling the technological parameters (and, so, V_{TH}) are subjected to larger relative variations, also I_{cell} is characterized by larger variation, and so the cell access delay [6].

2.1.6 Cell stability

The variations of V_{TH} have an influence also on the γ and β factors (subsection 2.1.2 and subsection 2.1.3) [6], since the electrical strength of the transistors strongly depends on the overdrive voltage: in this way, the V_{TH} variations affect also the read and write margins of the cell.

For example, if the pull-down is designed with an aspect ratio larger than the pass-transistor one but the latter has a lower V_{TH} because of the process variations, the read margin obtained is lower than the desired one; it may even happen that the cell is unable of performing a correct read operation.

The same example can be carried out for the write operation. For this reason, circuital and design techniques are needed to overcome, at least in part, the technological limitations, in order to allow for a decent yield of the fabrication process.

It is well known that most of the die area in modern microprocessors is dedicated to the memory part of the circuit (for example, cache memories); this means that most of the transistors in a digital integrated circuit are associated to the memory part and, so, the latter plays a key-role in determining the power consumption of the chip; for this reason, one would like to use the lowest V_{DD} possible for the memory.

What limits the V_{DD} scaling is the read margin: because of the process variations, to achieve a certain value of the read margin the overdrive voltage of the transistors cannot be reduced too much: some slack has to be left in order to take into account the V_{TH} variations among the transistors of the cell.

2.1.7 Read and write assist techniques and circuits

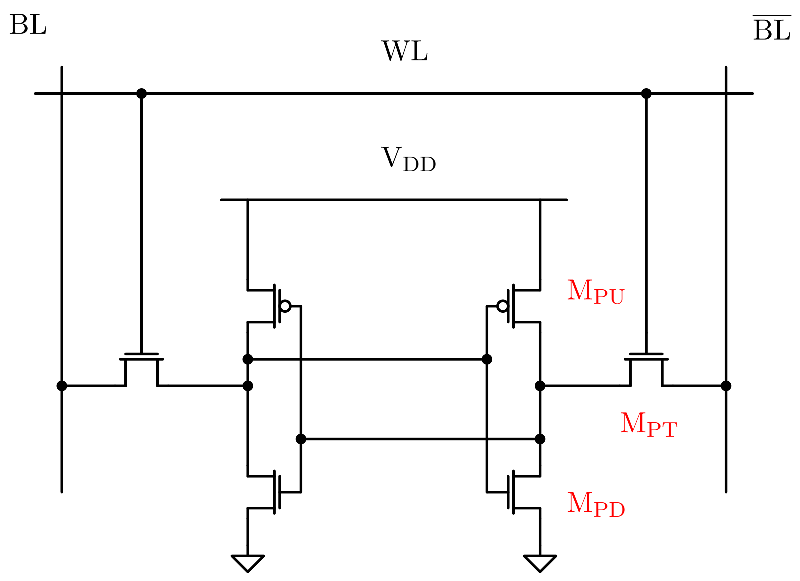


Figure 2.10: 6T SRAM cell

In order to increase the read margin, one could think of increasing the electrical strength of the pull-down with respect to the pass-transitor by boosting up the cell supply voltage $V_{DD_{cell}}$ [7] and, so, enlarging the pull-down overdrive voltage with respect to the pass-transistor one; in this way, the eventual V_{TH} difference between the two transistors is compensated and, so, the read margin is not reduced. This is accomplished by choosing:

$$V_{DD_{cell}} > V_{DD_{WL}}$$

Thanks to the difference between the cell supply voltage and the wordline potential, a partial (at least) compensation for the loss in the read margin due to the V_{TH} mismatch can be obtained.

However, there are some limitations to the value of $V_{DD_{cell}}$ that can be used in order to implement this technique:

- $V_{DD_{cell}}$ cannot be chosen too large, because the cell leakage current would increase too much (and so the static power consumption) and, also, the cell reliability would decrease (higher probability of failure)[6].
- one could think of lowering $V_{DD_{WL}}$ [8, 9, 10] instead of increasing $V_{DD_{cell}}$, but this would cause a reduction in the current absorbed by the cell during a read operation: in fact, since the pull-down and pass-transitor are in series to the bitline, the access-transistor would limit the cell current because of the reduced overdrive voltage.

To improve the write margin, one could think of applying a negative voltage to the bitline during the write operation [11], so that the potential of the node between the pass-transistor and the pull-up is brought to a lower value: in this way, the write operation becomes more reliable.

Another technique that could be implemented is the body biasing one: driving the well of the transistors with proper voltages, an improvement in both read and write margins can be obtained [12, 13, 14]:

- driving the wells of the pass-transistors so that their threshold voltage is increased during a read operation: in this way, their electrical strength with respect the pull-downs is reduced and, so, the read margin is improved.
- the wells of the pass-transistors could be driven so that their threshold voltage is reduced during a write operation: in this way, their electrical strength with respect the pull-ups is increased and, so, the write margin is improved.

2.1.8 Alternative cell topologies

With the topology proposed in Figure 2.11, called 8T cell [8, 6, 15], an ideal read margin can be obtained, since the read stage is decoupled from the memory stage [3]. This is achieved by adding two transistors and dedicated bitlines and wordlines, called respectively read-bitline (RBL) and read-wordline (RWL), which are dedicated to the read operation.

During a read operation, a transistor is used to sense the data stored inside the memory cell, by connecting its gate to the cell content; another transistor is connected to the RBL and it is enabled through the RWL. Connecting the gate of the sensing MOSFET to $\overline{\mathrm{D}}$, which is the side of the complemented stored data (a differential SRAM cell is implemented as core of the memory cell), the following behaviour is shown:

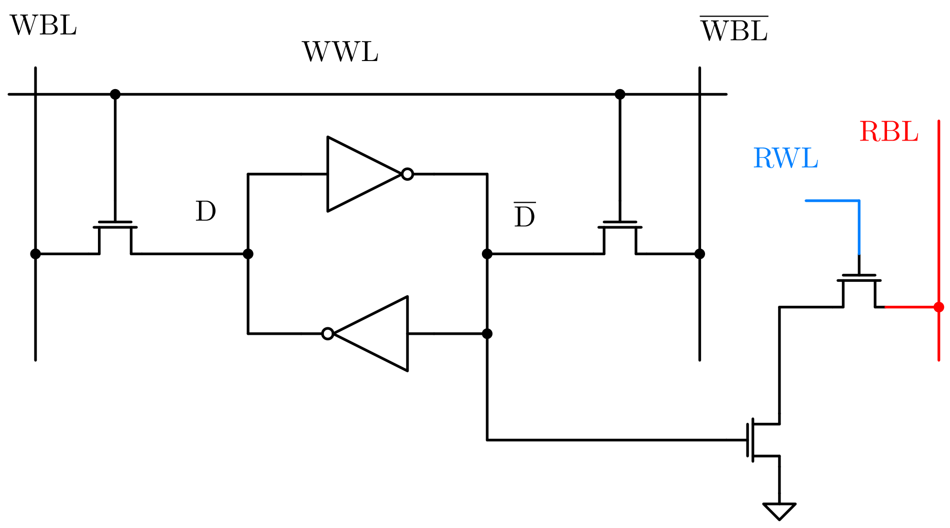


Figure 2.11: 8T SRAM cell

- when D = '1', the sensing transistor is disabled and the RBL is not discharged: a logic '1' is provided in output.
- when D = '0', the sensing transistor is enabled and the RBL is discharged: a logic '0' is provided in output.

Since the read stage is decoupled from the memory stage, it is not possible to overwrite the cell content during a read operation: the read margin is ideal. Also, the overhead in the cell area is not so large (30% increase) [15], because one does not need to use an aspect ratio for the pull-down larger than for the pass-transistor; hence, given the same current absorbed during the read operation (which is determined by the sizing of the sensing stage), smaller pull-downs are required with respect to the standard 6T cell.

Since there is no limitation to the pass-transistors sizing due to the read margin constraint, one could think of designing them to maximize the write margin: for instance, one could use a larger aspect ratio for them or choose a larger $V_{DD_{WL}}$ to increase their electrical strength.

It has to be precised that, thanks to the ideal read margin, the supply voltage of the cell can be reduced much more with respect to the 6T case: in fact, what limits the V_{DD} scaling in the 8T case is the static noise margin (SNM) of the cell, not the read one [3, 8].

The 8T cell, as the 6T one, is affected by the half-select problem [3]. Consider the circuit in Figure 2.12, which shows two 8T cells memory cells implemented in a memory row.

In general, all the cells of a row share the same wordline: this means that, when a

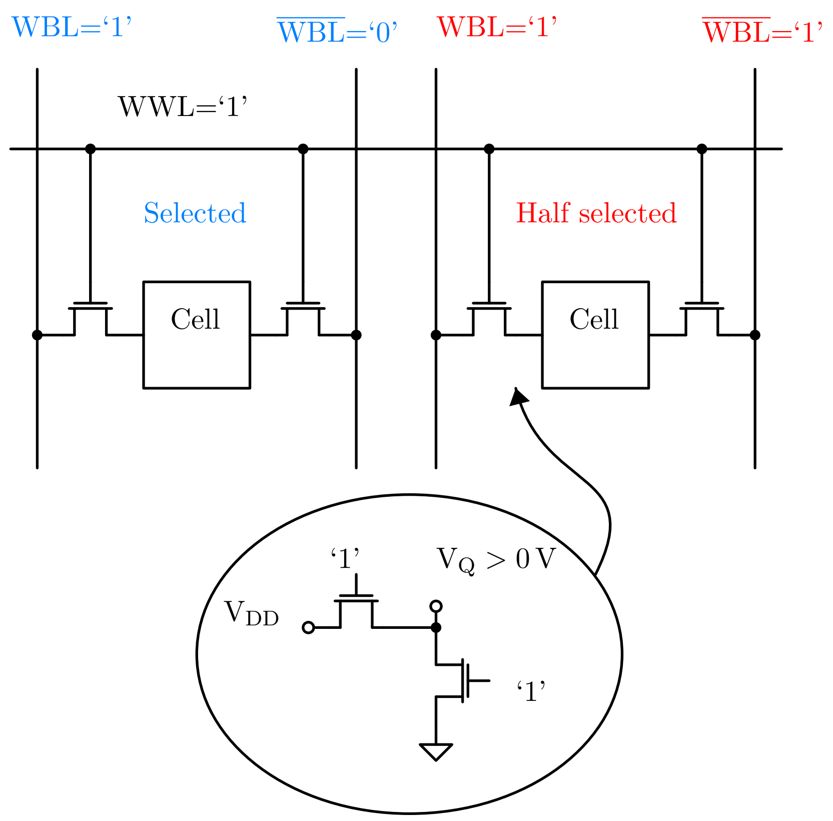


Figure 2.12: Half-select problem

data word (which is the portion of the memory row that has to be written or read) is selected, all the cells of the row have their access-transistors enabled; hence, the bitlines of the half-selected cells are kept at the logic '1' potential, so that their content is not overwritten.

However, the eventual noise on the bitlines can reach the memory core of the half-selected cells through the enabled pass-transistors, leading to the overwriting of their content, since the pull-downs and access-transistors of a 8T cell are not sized so that the stored data is not overwritten when the memory core is accessed.

To solve this problem, a new topology can be adopted, which is shown in Figure 2.13: the 10T cell [2, 16, 17].

In this circuit two more access transistors are added, which are the ones connected to COLUMN. To access the cell two decoding steps are needed: one to select the row of the cell and the other to select the column of the cell: in this way, the half-select problem is eliminated, since the half selected cells have one of the two access transistors on each side disabled and, so, their memory core remains separated from the bitlines.

The overhead in the cell area is large: four additional transistors are needed with respect the 6T cell. However, with this circuit ideal read margin and differential sensing are obtained.

It has to be noticed that there is an overhead also in the control circuit, since now

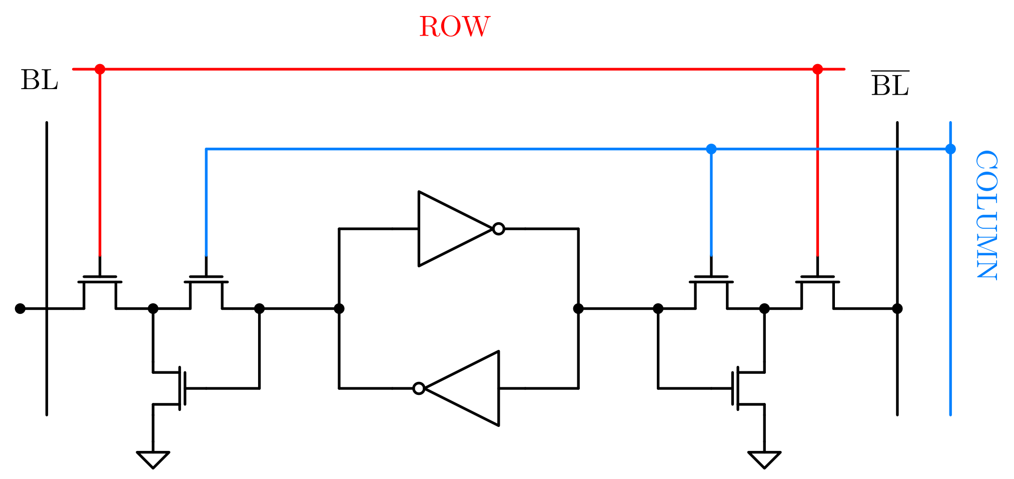


Figure 2.13: 10T SRAM cell

two decoding steps are needed (for row selection and column selection).

Leakage represents a problem from both power and functional points of view: the unselected cells in the column of the cell which is being accessed for a read operation absorb (inject) a current from (in) the bitline. This represents a problem because:

- if the selected cell is discharging the bitline and the unselected cells inject a current in it, the discharging time (and so the read delay) enlarges.
- if the selected cell is not discharging the bitline and the unselected cells absorb a current from it, the bitline voltage level can lower until it reaches the trip point of the sense amplifier, which results in a wrong read result.

It cannot be known a priori if an unselected cell will absorb a current from the bitline or inject a current in it, because it depends on the data stored in the cell; for this reason, many circuital solutions have been exploited to make the leakage contribution of an unselected cell independent on the stored data.

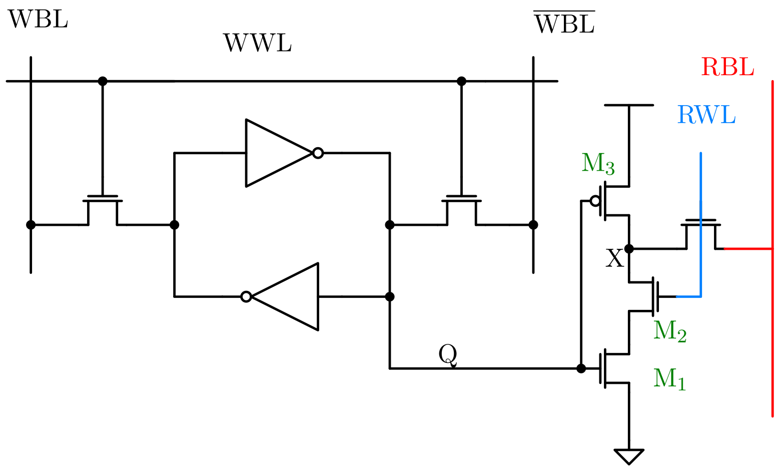


Figure 2.14: 10T single ended: data-independent leakage

In Figure 2.14 a possible solution is shown: it is a modified version of the 8T cell, in which the leakage on the RBL is independent on the stored data [18].

When Q = ‘0’ (Figure 2.15), M_3 is enabled and the node X is brought to V_{DD}.

When Q = ‘1’ (Figure 2.16), M_1 is enabled and, if the transistor are opportunely sized, the node X is brought to a voltage level close to V_{DD}, because M_2 is disabled (the cell is not selected).

It has to be noticed that the overhead in the cell area is not negligible, because of the two additional transistors required for this cell topology.

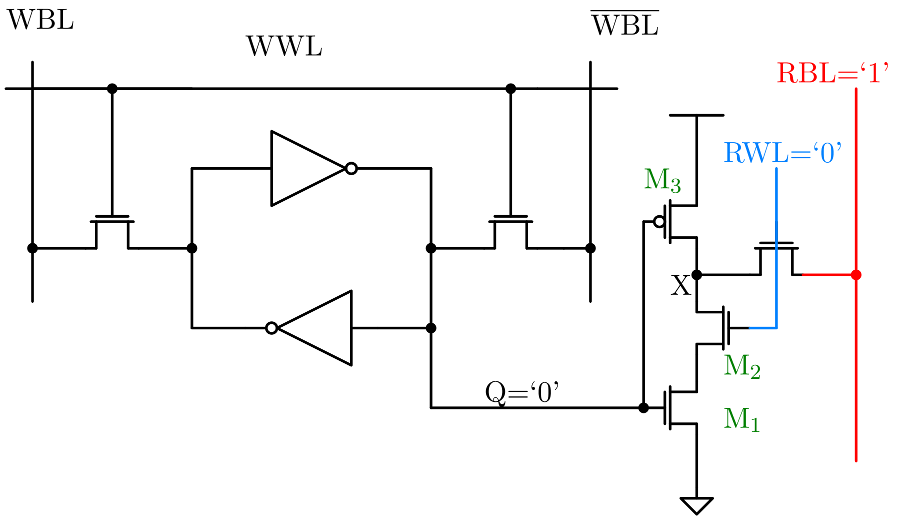


Figure 2.15: 10T single ended: data-independent leakage, Q=‘0’

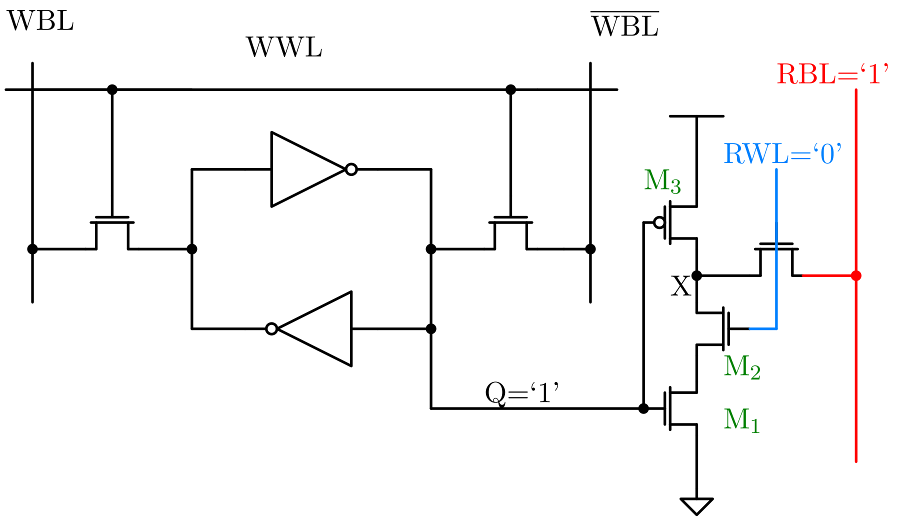


Figure 2.16: 10T single ended: data-independent leakage, Q=‘1’

2.2 Content Addressable Memories

The Content Addressable Memory (CAM) is a memory architecture which takes in input a datum and provides in output its address if this is stored inside the memory array [19, 20, 21, 22, 23, 24]. The datum, which is called search word, is compared against all the words stored inside the CAM, which are called stored words. If at least one of these comparisons gives a successful result, a match is declared.

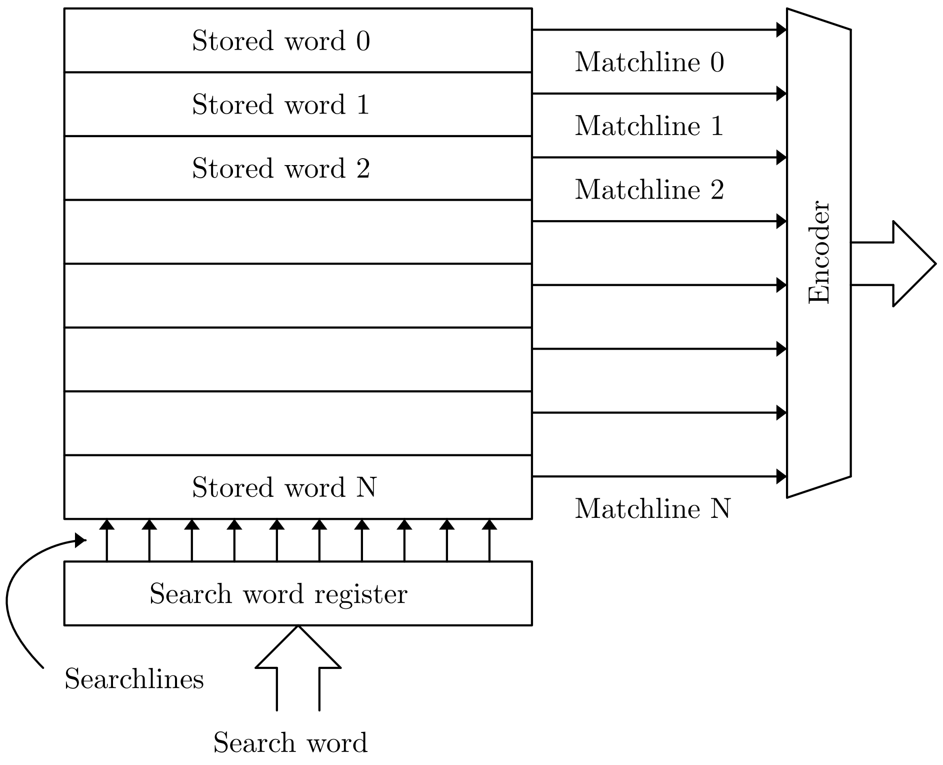


Figure 2.17: CAM block scheme

The search word is loaded on the searchlines, which connect the search word bits to the CAM cells for the bit-to-bit comparison. The search word is compared with all the stored words at the same time.

To each stored word a match-line is associated, and the value assumed by the latter at the end of the search process, which depends on the type of CAM cell implemented inside the memory, determines if there is a match between the search word and the associated stored word.

The values assumed by the match-lines are taken in input by an encoder, which provides in output the address (location) of the stored word that has given a match. If more than one memory word matches with the search one (like it happens in routers [25, 26]), a priority mechanism is implemented in the encoder.

Since the search operation is carried out on the whole memory array at once, the obtained search speed is very large. However, this speed comes at the cost of large chip area and power consumption; in particular, the latter is the critical parameter of a CAM: for this reason, reducing the power consumption without sacrificing area or search latency is the main thread of recent research in CAMs.

2.2.1 The CAM cell

There are two types of CAM cell: NOR and NAND cells. The most common is the NOR one.

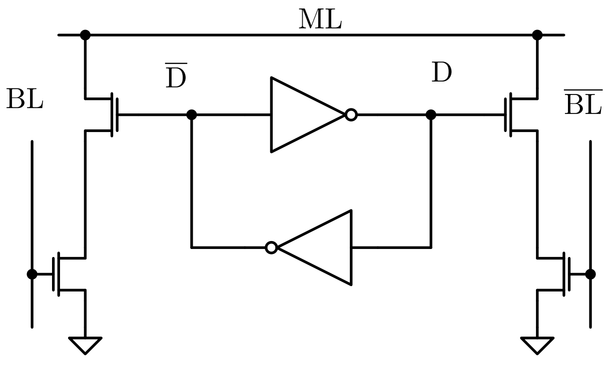


Figure 2.18: NOR cell

The NOR cell circuit is shown in Figure 2.18. The memory core of the cell is the bistable ring, like in SRAM cells; to this a comparison circuit, made of two pull-down paths, is added; the pass-transistors that allow the access (read and write operations) to the cell are not shown for the sake of clarity.

In the classical match-line scheme, the match-line (ML) is precharged to the logic ‘1’; then, the search operation begins by loading the search word on the searchlines, which are represented by BL and $\overline{\text{BL}}$ in Figure 2.18. From now on, searchlines and bitlines are considered to be the same thing, since the same lines are used handle the memory cells content and to perform the search operation.

If the cell content matches with the datum on the searchlines, both pull-down paths are disabled (one of the two series transistors is turned off) and the match-line remains charged at the logic ‘1’: a match is provided as result.

If the cell content does not match with the datum on the searchlines, one of the two pull-down paths is enabled and the match-line is discharged to the logic ‘0’: a mismatch is provided as result.

The NOR cells are arranged in a wired-OR configuration on the match-line, as it is shown in Figure 2.19: in this way, if at least one cell does not match with the search word, the match-line is discharged and a mismatch is provided as result.

In the NAND cell, shown in Figure 2.20, the comparison circuit is made of three pass-transistors: M_1, M_2 and M_3.

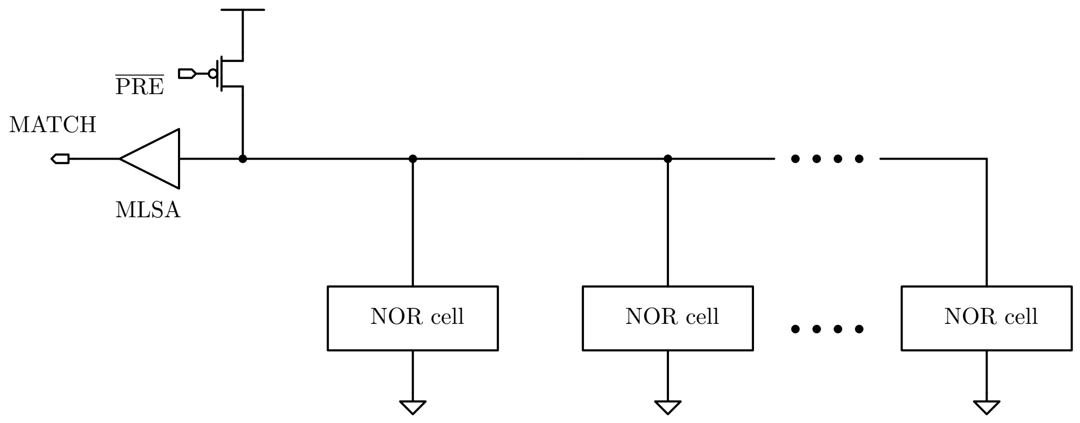


Figure 2.19: NOR match-line

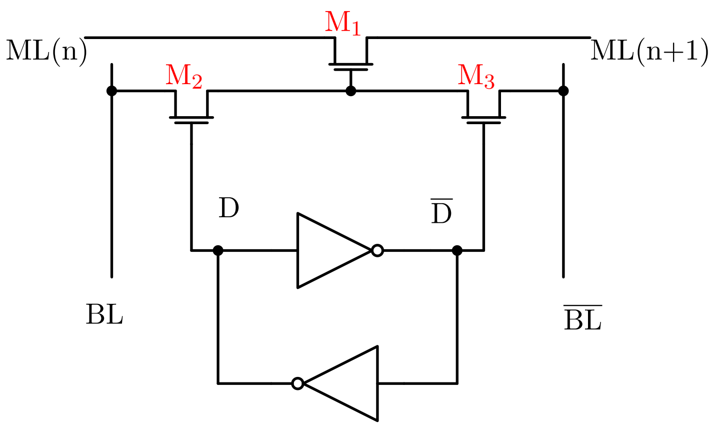


Figure 2.20: NAND cell

If the datum on the searchlines and the cell content match, one between M_2 and M_3 is enabled and connects the gate of M_1 to V_{DD}: the latter is turned on and it connects the match-line segments $ML(n)$ and $ML(n+1)$: a match is provided as result.

If there is a mismatch between the searched and the stored datum, the gate of M_1 is connected to the logic '0' and, so, the transistor is disabled: the two match-line segments are not connected and a mismatch is provided as result.

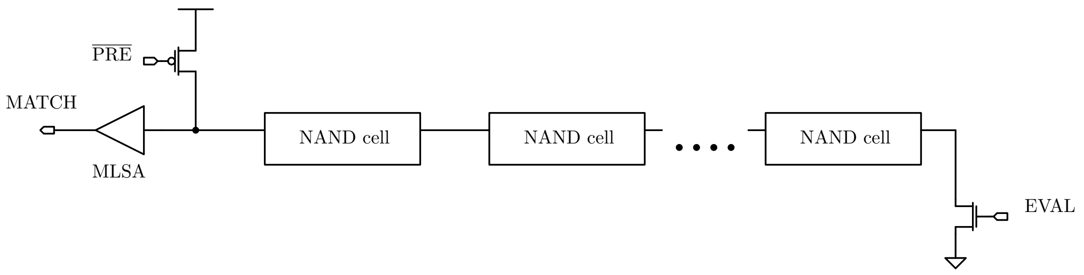


Figure 2.21: NAND match-line

The NAND cells are arranged in a series configuration on the match-line, as shown in Figure 2.21. In this way, if all the cells provide a match as result, the match-line is discharged. Otherwise, at least one cell interrupts the discharge path to ground and the match-line remains charged at the logic ‘1’.

In the scientific literature variants of the proposed cells can be found; however, the predominant cells are the ones presented previously and, so, only those will be furtherly analyzed in the following.

When a search operation is carried out on a CAM, only one stored word (or few stored words, in routers) provides a match as result, while all the others a mismatch; hence, the power consumption in a search operation is determined by the power involved in the one that gives a mismatch as result.

In the NOR match-line, using the classical sensing scheme, in the mismatch case the match-line is discharged: hence, the mismatch result implies a much larger power consumption with respect to the match case; since the mismatch result is the most common one in the memory, using a NOR cell and a classical sensing scheme implies a large power consumption.

In the NAND match-line, instead, the match-line is discharged in the match case: hence, the power consumption of the search operation is strongly reduced with respect the NOR implementation.

It has to be highlighted the fact that the NAND match-line is much slower than the NOR one: when the match-line has to be discharged, in the NOR match-line there are two transistors in series, in the worst case (only one among the N cells connected to the match-line provides a mismatch), that discharge the line; in the NAND case, instead, there are N transistors in series, where N is the width of the CAM word, that discharge the match-line. It has also to be noticed that, referring to Figure 2.20, the gate of M_1 is never brought to V_{DD} but to $V_{DD} - V_{TH}$, since it is connected to the supply voltage by a nMOS transistor; hence, its conductivity is reduced with respect to a fully-driven nMOS transistor, leading to a slower discharging of the match-line. Furthermore, the discharging delay of a NAND match-line increases quadratically with the number of cells present on the line [27].

For this reason, the NOR match-line is preferred to the NAND one, allowing a much faster search operation, while other sensing techniques are employed to reduce the power consumption, as it is explained in the following chapters.

Another disadvantage of the NAND match-line is given by the charge sharing problem that occurs in this kind of circuits (multiple transistors in series).

Referring to Figure 2.21, consider the case in which all the cells give a match as result except the last one (the one closest to the ground voltage). As soon as the searchlines are enabled in order to begin the search phase, the charge that has been injected in the input capacitance of the sense amplifier during the precharge phase is divided among the parasitic capacitances of the line, which are placed between the NAND cells (drain and source capacitances of the transistors), that have not been charged; hence, the voltage on the match-line may be lowered below the MLSA threshold voltage that, consequently, switches its output, leading to a wrong result in the search operation.

To avoid the charge sharing phenomenon, also the parasitic capacitances between the cells should be charged during the precharge phase; however, this leads to an increase in the power consumption, even if this is compensated by the fact that a NAND match-line power consumption is much lower that the NOR match-line one.

2.2.2 Ternary cells

In literature other kind of content addressable memories can be found: the Ternary Content Addressable Memories (TCAMs).

In this kind of memory, two bits are stored in each cell. The introduction of a second bit allows the definition of a third state for the memory cell content: the X state, which can be called don't-care state [19].

In fact, when a cell contains a X, it provides a match with the datum put on the searchlines independently of the value of the latter. This can be useful when one wants to search for all the stored words that match at least in part with a certain search word.

Also, in TCAMs there are two types of cells can be found: the NOR cell and the NAND cell.

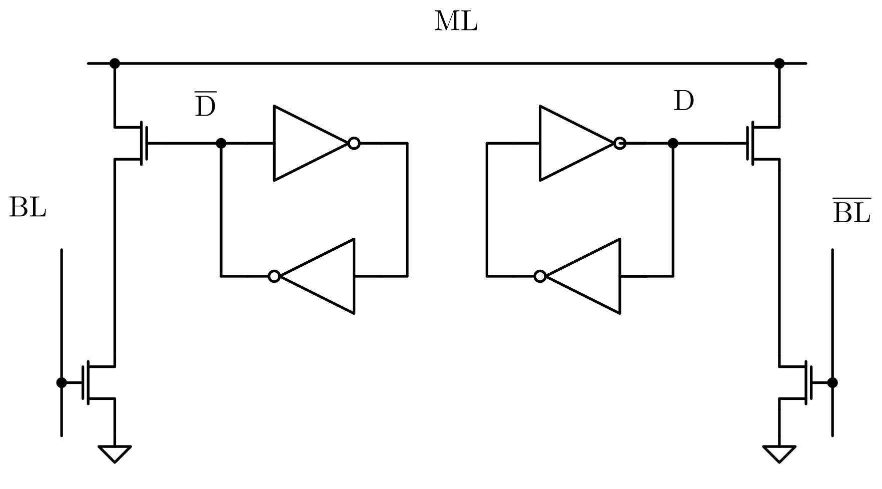


Figure 2.22: TCAM NOR cell

The NOR cell is shown in Figure 2.22. One can notice how there are two SRAM cells: one stores the data D and the other stores its complemented version, $\overline{D}$. As said before, three possible states can be stored in a TCAM cell: a logic '1', a logic '0' and a logic 'X'. The corresponding combinations of D and $\overline{D}$ are show in Table 2.1.

One can notice that, when a 'X' is stored inside the cell, both the match-line pull-down paths are disabled: hence, a match is provided by the cell as result, independently

Data stored	D	$\overline{D}$
1	1	0
0	0	1
X	0	0

Table 2.1: D and $\overline{D}$ allowed combinations

of the value of the data put on the searchlines.

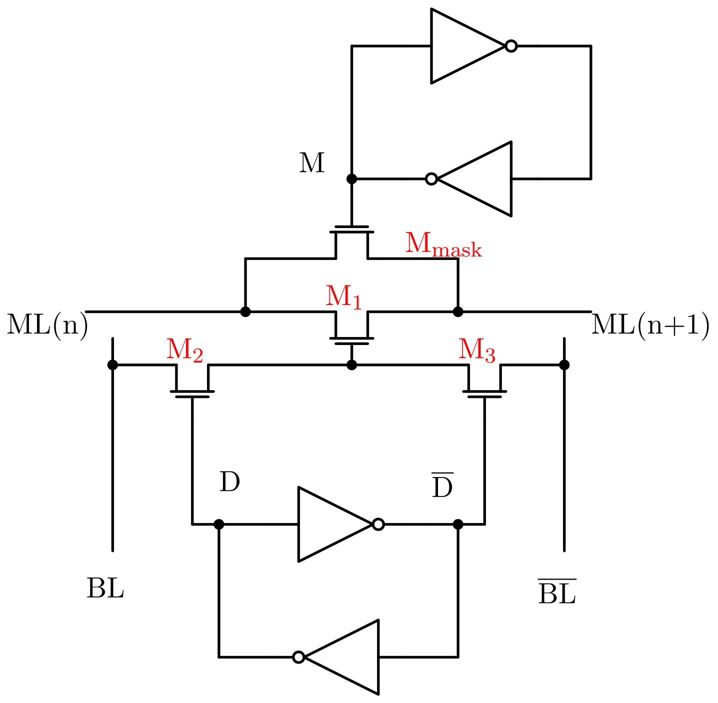


Figure 2.23: TCAM NAND cell

In Figure 2.23 the NAND TCAM cell is shown [28]: here the second bit is called mask bit, M. The presence of this bit, of course, modifies the behavior of the cell:

- when M = '0', the data stored inside the cell (D) determines if the result is a match or not.
- when M = '1', M_{mask} is enabled and, so, the cell provides always a match as result, since $ML(n)$ and $ML(n+1)$ are always connected independently of the value of D.

2.2.3 Match-line schemes

Many sensing schemes for the CAM have been proposed and they all focus on the NOR match-line, since it allows the fastest operation; for this reason, only this kind of match-line scheme is analyzed in detail in the following.

In order to compare the different match-line sensing schemes, a model [19] for the match-line has been derived, which is shown in Figure 2.24.

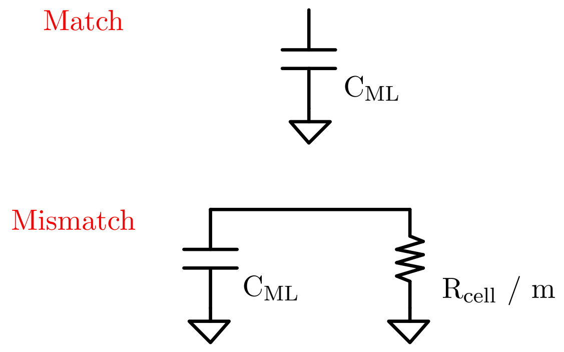


Figure 2.24: Match-line model

When the result of the search operation is a match, every CAM cell has both pull-down paths disabled; if one neglects the subthreshold current of the transistors, the match-line can be modeled with a capacitance C_{ML}, which is made by the drain capacitances of all the pull-down paths connected to the line.

When, instead, the result is a mismatch, there are m pull-down paths that are enabled, where m is the number of bits of the stored word that do not match with the search word. Each of these pull-down paths can be modeled with a resistance R_{cell}; since the cells are all in parallel (wired NOR configuration shown in Figure 2.19), the equivalent resistance of all the pull-down paths is given by R_{cell}/m.

During the evaluation phase, the match-line capacitance C_{ML} gets discharged by the pull-down resistances cited before. The larger is the number of mismatch bits, the faster will be the discharging of the match-line, since the equivalent resistance of the mismatch cells is reduced.

Precharge-high scheme

The precharge-high scheme, which circuit is shown in Figure 2.19, is the simplest sensing scheme that can be implemented in a CAM. The search operation consists of two phases:

- a precharge phase, in which the match-line is charged to V_{DD} ($\overline{\text{PRE}}$ = '0'). During this phase, the pull-down paths of the CAM cells have to be disabled by forcing BL = $\overline{\text{BL}}$ = '0', so that the line can be charged independently of the cells content.

- a search phase, in which the match-line is left floating ($\overline{\text{PRE}}$ = '1') and the search word is loaded on the searchlines. If search and stored word match, the line is not discharged (Figure 2.25); otherwise, the match-line is brought to the ground voltage by the pull-down paths of the cells (Figure 2.26).

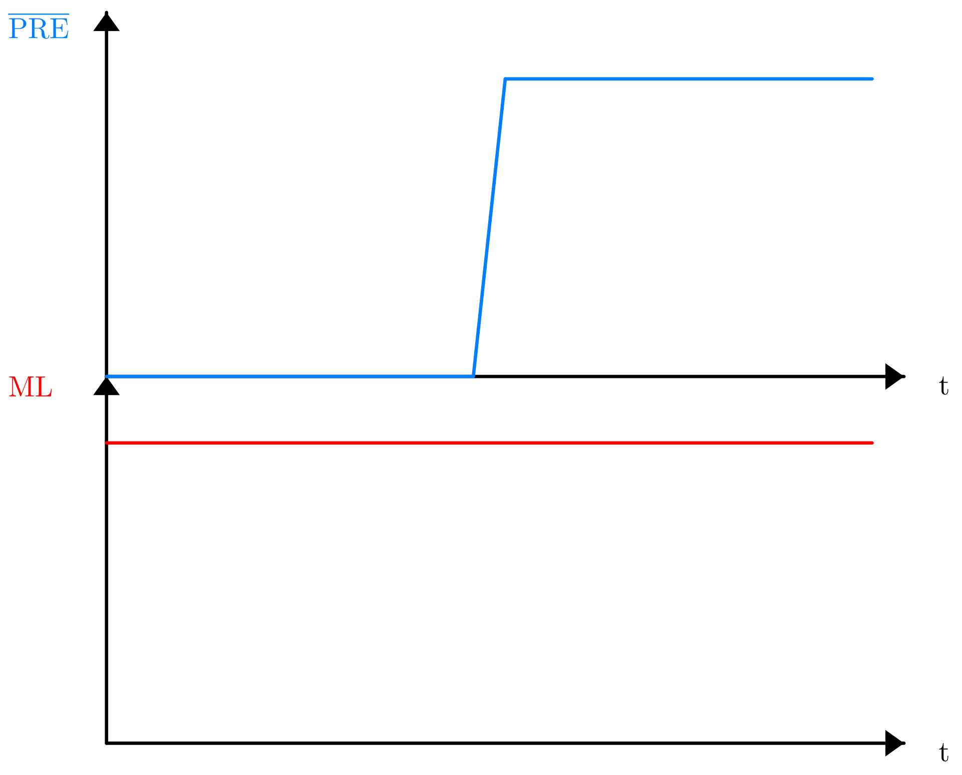


Figure 2.25: Precharge high scheme: match case

The precharge-high scheme implies the largest power consumption for the search operation, because the match-line of a stored word gets discharged in the mismatch case; hence, since almost all the stored words do not match with the search word, pratically all the match-lines get discharged in each search operation, which results in large power consumption. For this reason, this kind of sensing scheme has been practically abandoned in modern CAM implementations.

Using the model derived for the matchine in Figure 2.24, the energy consumption of a search operation which results in a mismatch is given by:

$$E_{mismatch} = C_{ML} \cdot V_{DD}^2 \tag{2.1}$$

where V_{DD} is the voltage to which the match-line has been previously charged (usually it is the supply voltage), and f_{search} is the frequency of the search operations performed on the memory.

One can notice how the classical $\frac{1}{2}$ factor is missing from Equation 2.1 shown above: this is because the match-line gets charged during the precharge phase and it gets discharged during the evaluation phase; hence, there are two commutations for search operation in the mismatch case.

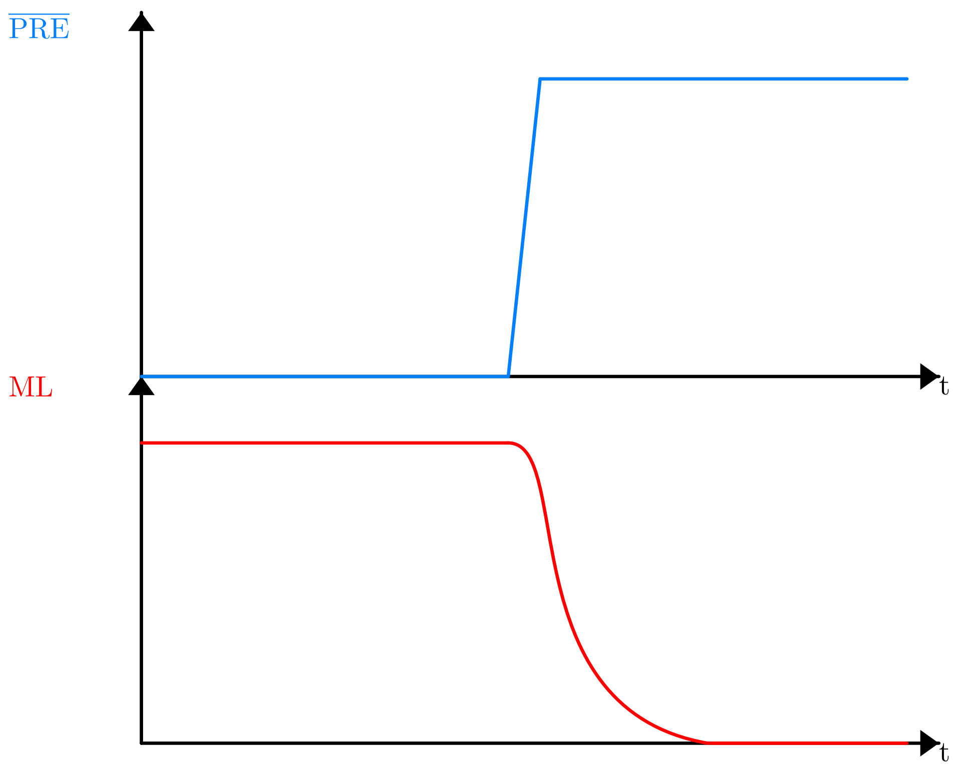


Figure 2.26: Precharge high scheme: mismatch case

Low swing scheme

In the low-swing scheme [29], the principle is to reduce the voltage swing on the match-line: this allows to reduce the power consumption of the mismatch result and to speed up the sensing phase [30], since this is now given by:

$$E_{mismatch} = C_{ML} \cdot V_{DD} \cdot V_{swing}$$

The energy consumption reduces linearly with the voltage swing on the match-line.

The difficulty in implementing this sensing scheme is to reduce the voltage swing on the match-line without introducing a second voltage reference inside the memory array, apart from the voltage supply.

Current-race scheme

The current-race scheme [31] works in the opposite way with respect to the precharge-high one. Also in this scheme, there are two phases: pre-discharge and evaluation.

In the pre-discharge phase ($\overline{\text{PRE}}$ = '0', $\overline{\text{EN}}$ = '1') the match-line is discharged through a pull-down transistor, while the input of the inverter, shown in Figure 2.27, is charged to V_{DD}; hence, the inverter output is brought to 0 (MLSAO='0).

In the evaluation phase ($\overline{\text{PRE}}$ = '1', $\overline{\text{EN}}$ = '0'), the current source I_{ML} is connected

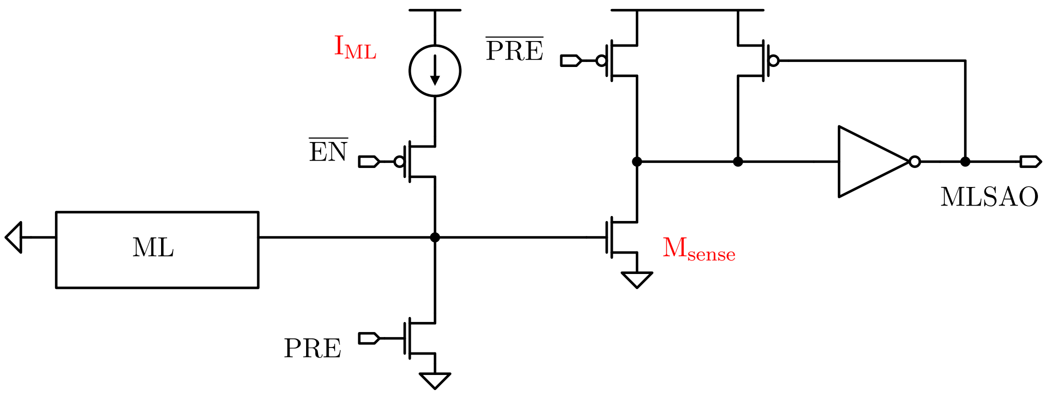


Figure 2.27: Current-race scheme

to the match-line; then, there are two possible outcomes:

- if search and stored words match, all the pull-down paths of the match-line are disabled and the line gets charged, since it behaves like a capacitance (Figure 2.24); hence, M_{sense} is turned on and it discharges the inverter input, forcing it to switch its output (MLSAO='1'): a match is provided as result.
- if search and stored words do not match, the match-line does not get charged because there is at least one pull-down path enabled in the line (Figure 2.24); hence, M_{sense} is not turned on and, so, the inverter input is kept charged to V_{DD}, and its output remains constant (MLSAO='0'): a mismatch is provided as result.

Using the current-race scheme, the energy involved in the mismatch case is:

$$E_{mismatch} = \frac{1}{2} \cdot C_{ML} \cdot V_{DD}^2$$

The energy consumption is halved with respect to the precharge-high scheme case, since the match-line does not get charged in the mismatch case; it gets charged only in the match one.

However, in the equation above the energy associated to the current injected by the current source in the pull-down paths has been neglected; in fact, in actual applications of this sensing scheme the $\overline{\text{EN}}$ signal is disabled after a short time interval, in order to save energy.

The value of I_{ML} is crucial in order for the sensing circuit to work correctly; in fact, I_{ML} has to be low enough for a mismatch to not turn into a match: this means that, in the mismatch case, the voltage of the match-line has to be always lower than the threshold voltage of M_{sense} or, at least, low enough to prevent the threshold voltage of the inverter to be crossed. In fact, when M_{sense} is turned on, it has to bring the inverter input to the logic '0' while this is fixed to the logic '1' by the pMOS keeper transistor shown in Figure 2.27.

The voltage to which the match-line is charged in the mismatch case is:

$$V_{mismatch} = I_{ML} \cdot \frac{R_{cell}}{m}$$

where, as usual, R_{cell} is the resistance of each cell pull-down path and m is the number of mismatch bits. Of course, the worst case is associated to $m = 1$.

If one implements a clock circuit inside the memory that disables the current source as soon as a match is detected, the energy consumption associated to the match case is given by:

$$E_{match} = C_{ML} \cdot V_{DD} \cdot V_{TH}$$

where V_{TH} is the threshold voltage of the sense amplifier (actually, it is a little bit lower, in order for the sense amplifier to not switch its ouput). One can notice how the formula displayed above is very similar to the one obtained with the low swing sensing scheme (section 2.2.3).

Another advantage of the current-race scheme is that, since the precharge of the match-line has been substituted with a pre-discharge cycle, the CAM cells do not need to be disabled before the search cycle, because there is no need to precharge the match-line with this sensing scheme; hence, the match-lines do not need to be brought to the logic zero during the predischarge cycle and, then, to the search datum value during the evaluation cycle: the datum can be loaded on the searchlines at the beginning of the predischarge cycle reducing, in this way, their switching activity and, so, the associated power consumption.

Another thing that can be noticed is that, since the datum is loaded on the searchlines one cycle before the actual search operation, the search speed is independent on the searchline drivers delay, because the searchline transistors (Figure 2.18) are already enabled/disabled at the beginning of the evalutation cycle.

Selective precharge scheme

With this scheme, the energy allocated to the search operation depends on the datum stored inside the selected stored word [32].

Consider the circuit presented in Figure 2.28. The first bit of the word is implemented with a NAND cell, while all the other bits are implemented using NOR cells. During the precharge phase, the first bit of the search word is loaded on the first cell, while all the others are disabled: in this way, if the content of the first cell matches with the first bit of the search word, the line gets charged; otherwise, the pass transistor of the NAND cell is disabled and the line does not get charged. In this way, if there is a mismatch on the first bit, the line does not change its state, since it does non get charged at all; otherwise, the result of the search operation is determined by the remaining bits.

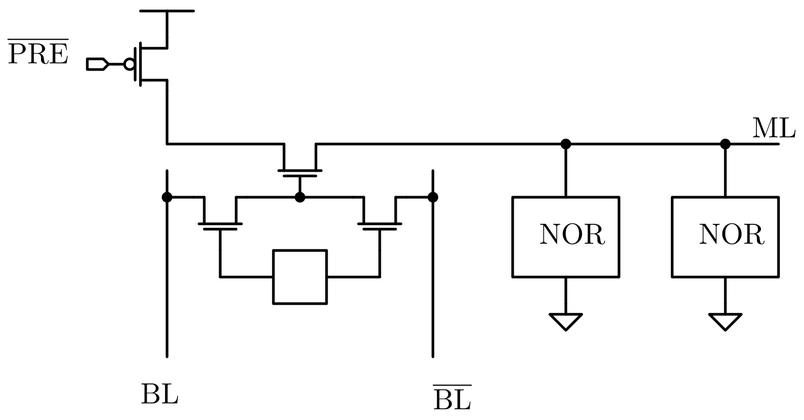


Figure 2.28: Selective-precharge scheme

This sensing scheme is very convenient in architectures where the data starts to differ from the first bits, since the mismatch are detected already in the precharge phase of the search operation. In fact, it is widely used to reduce the power consumption in low-power CAM architectures [33, 34, 35, 36, 37, 38].

Pipeline scheme

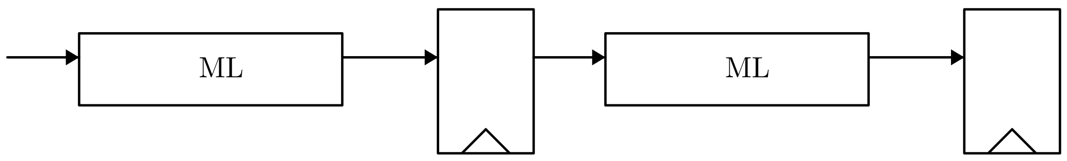


Figure 2.29: Pipeline scheme

The pipeline sensing scheme [39, 40, 41] is a generalization of the selective-precharge one (section 2.2.3): the match-line is divided in sections, and the search operation is carried out sequentially one section at time.

The first section of the stored word is compared with the first section of the search one: if there is a match, the comparison moves on the next section; otherwise, the search operation is ended and a mismatch is provided as result. The result is a match only if all the sections of the stored word match.

In this way, if the first section of the match-line gives a mismatch, the energy required for the comparison of the remaining sections is saved.

Current-saving scheme

This is an enhanced version of the current-race scheme (section 2.2.3): the difference is in the fact that a different amount of energy is allocated to the match-line depending on the result of the search operation [42, 43].

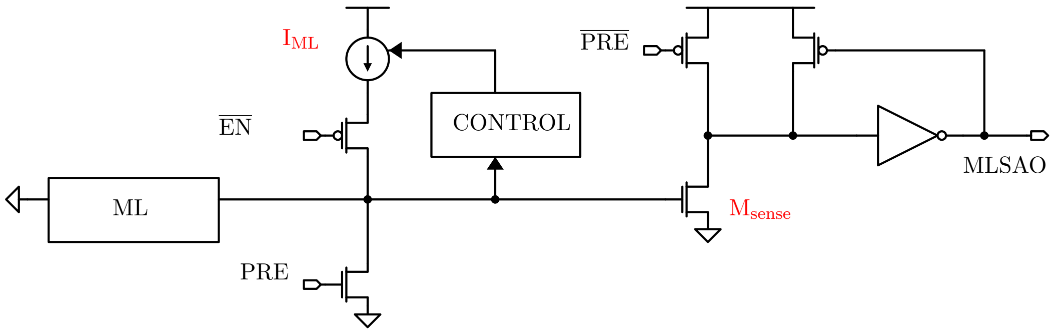


Figure 2.30: Current-saving scheme

When there is a match, the match-line gets charged by I_{ML}; as the line voltage arises, the current delivered to it is increased using the feedback circuit shown in Figure 2.30. As a consequence, a large current is delivered in the match case, while I_{ML} is kept to a minimum value in the mismatch one, since the line does not get charged.

In this way, an energy larger than in the mismatch case is allocated to the match result, which is the (far) less frequent one.

2.2.4 Searchline schemes

The switching activity of the searchlines has a considerable impact on the energy consumption of a CAM; hence, several techniques [40, 44, 39, 45] have been proposed to reduce the activity of these lines during a search operation.

In the standard precharge-high scheme (section 2.2.3), the searchlines are brought to the logic '0' during the precharge phase, in order to disable the CAM cells; then, at the beginning of the search phase, the search datum is loaded on the searchlines. This means that, during the precharge cycle, one of the two searchlines is brought to the logic '0'; during the evaluation one, one of the two searchlines is brought to the logic '1'; hence, two searchlines per search operation change their logic state, and the total energy per search cycle associated to a couple of searchlines (since there are two searchlines for each CAM column) is given by:

$$E_{SL} = C_{SL} \cdot V_{DD}^2$$

where C_{SL} is the parasitic capacitance associated to each searchline. This is given by the gate capacitance of the nMOS transistor of the CAM cell which is connected to the searchline (Figure 2.18).

In order to reduce the energy consumption of the searchlines, one has to work on their parasitic capacitance (for example, by minimizing the aspect ratio of the transistors connected to them) and their switching activity.

One way to reduce the switching activity could be to not disable the searchlines during the precharge phase of the search cycle. This is accomplished in the current-race (section 2.2.3) and current-saving (section 2.2.3) schemes, in which during the predischarge phase the CAM cells do not need to be disabled; hence, there is no need to bring the searchlines to the logic '0' during this phase.

Chapter 3

SRAM design

In this chapter the design and verification of the SRAM memory array is discussed; three arrays have been designed and analyzed, which dimensions are 8x8, 16x16 and 32x32 bits, using the ST FD-SOI 28nm technological library in Cadence Virtuoso.

3.1 The SRAM cell

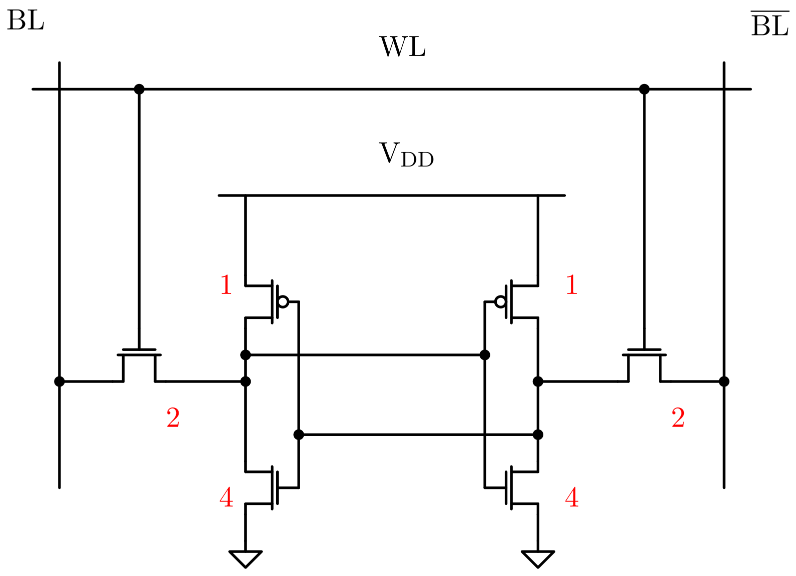


Figure 3.1: 6T SRAM cell

The cell adopted in the design is the classical 6T one, which is shown in Figure 3.1. The simplest possible cell is implemented, since the goal of this work is to design a memory capable of performing logic operations; hence, the memory portion of the cell has to be minimized in order to save area.

In the circuit, the aspect ratio of each transistor is shown as a multiple of the minimum aspect ratio, which is given by:

$$(W/L)_{min} = \frac{80\,\text{nm}}{30\,\text{nm}}$$

Hence, a transistor to which a number equal to n is associated in the schematic has an aspect ratio given by:

$$(W/L)_n = (W/L)_{min} \cdot n$$

The cell transistors have been sized following the guidelines presented in section 2.1.

The ratio between pull-down and pass transistor is chosen equal to 2, in order to avoid the overwriting of the cell content during a read operation. In this way, the ratio between the driving capabilities of the two transistors can be approximately computed as:

$$\frac{(W/L)_{PD}}{(W/L)_{PT}} = 2 \rightarrow \frac{R_{PD}}{R_{PT}} \approx 2$$

The ratio between pass transistor and pull-up is chosen equal to 2, in order to allow the write operation to be correctly performed; hence, the ratio between the driving capabilities of the two transistors can be approximately computed as:

$$\frac{(W/L)_{PT}}{(W/L)_{PU}} = 2 \rightarrow \frac{R_{PT}}{R_{PU}} \approx 4$$

One could notice that the difference in the driving capabilities between the pull-up and the pass transistor is larger than the one between the pull-down and the pass transistor; hence, the write margin of the cell is larger than the read one. In fact, the difference in the electrons and holes mobility ($\mu_n/\mu_p \approx 3$) alone determines an enough large difference in the driving capabilities of pull-up and pass transistor; however, the aspect ratio of the latter is doubled in order to obtain an acceptable value for the current absorbed by the cell during a read operation.

Also, in the equation above it is evident that, even if $\mu_n/\mu_p \approx 3$, the ratio between the driving capabilities is only doubled with respect the aspect ratios proportion, instead of being tripled. This is due to the fact that, in a planar MOSFET, at the interface between the gate oxide and the silicon substrate (where the channel actually forms) defects are present: hence, the difference in the mobilities values is reduced.

3.2 Driver circuits

In order to get realistic results from the simulations, driver circuits are included in the design. These are used to drive the wordlines, bitlines and sense amplifier outputs.

For simplicity, all the additional inverter stages, needed to properly drive the memory lines, have been omitted from the circuits shown in the following.

3.2.1 Bitlines driver

The bitlines driver circuit is shown in Figure 3.2.

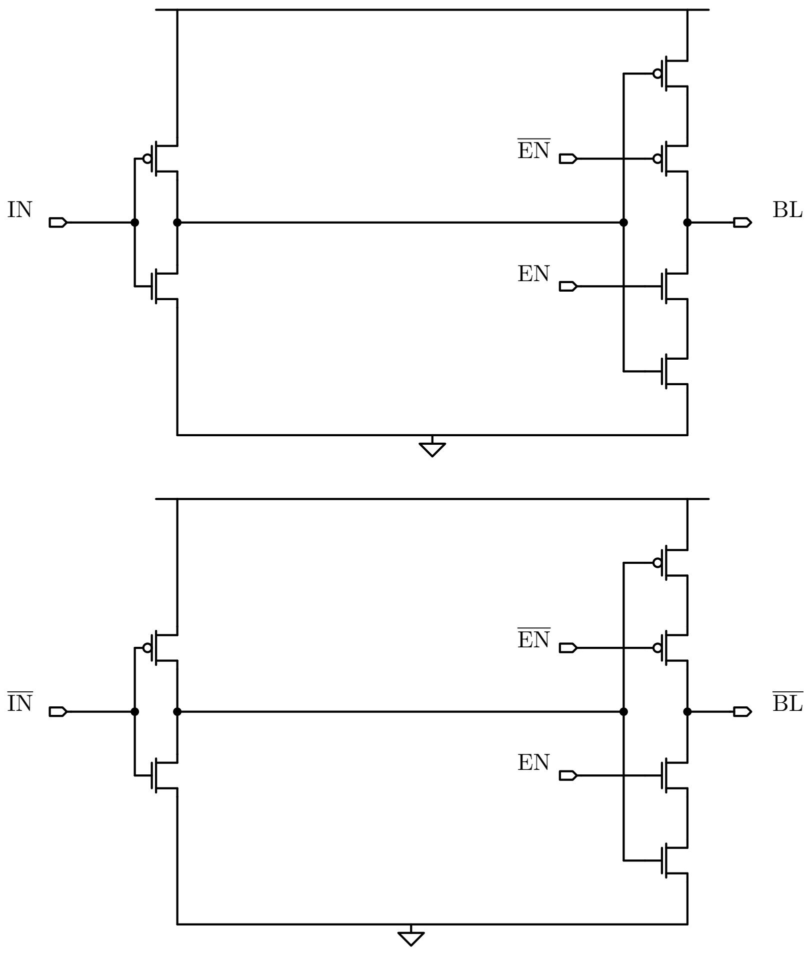


Figure 3.2: Bitlines driver

This component is used to drive the bitlines of the memory. For each couple of bitlines (i.e. for each memory column), a driver is instantiated. All the drivers are driven by ideal voltage sources in the Cadence Virtuoso testbench.

It can be noticed how the output section of the driver is a tristate inverter: in fact, the bitlines have to be electrically separated from the drivers during the precharge and read cycles, so that they can be charged to V_{DD} and, then, one of them can be discharged by the cell which is being accessed during the read operation.

When EN='1', the output section of the driver is enabled and, so, connected to the bitlines; when EN='0', the output section is disabled and the driver is electrically separated from the bitlines.

3.2.2 Precharge circuit

The precharge circuit is shown in Figure 3.3. This component is used to precharge the bitlines before the read operation.

When $\overline{\text{PRE}}$ = '0', the bitlines are connected to V_{DD} through the two side pMOS transistors and equalized by the central pMOS; when $\overline{\text{PRE}}$ = ‘1’, all the transistors are disabled and the circuit is electrically isolated from the bitlines.

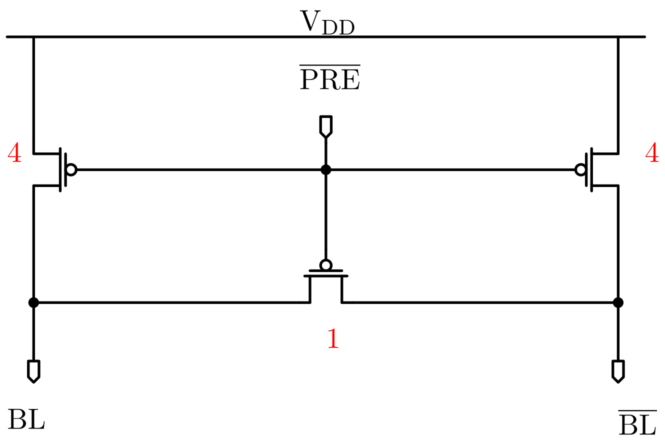


Figure 3.3: Precharge circuit

3.2.3 Wordline driver

The wordline driver is made by simple inverter stages, since there is no need to isolate the driver from the wordline: in fact, the wordline voltage has to be always well defined, so that the cells can be isolated from the bitlines when not selected.

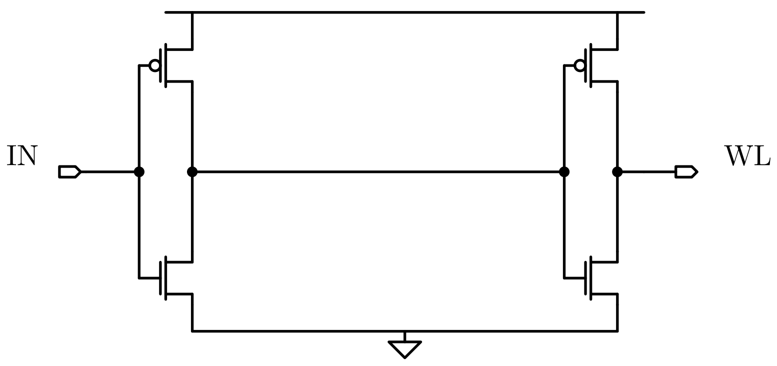


Figure 3.4: Wordline driver circuit

3.3 Sense amplifier

A classical voltage sense amplifier topology has been chosen [46], which circuit is shown in Figure 3.5.

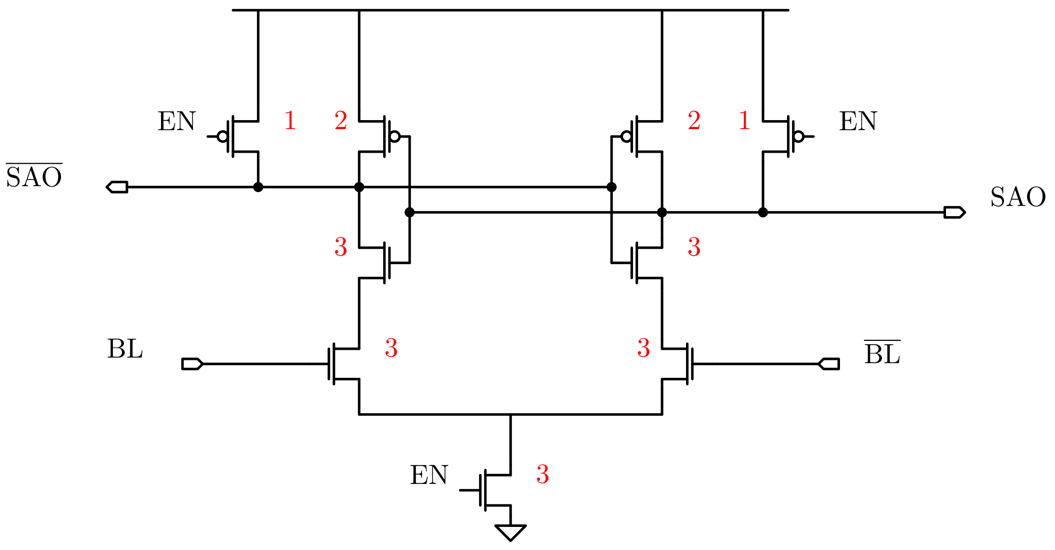


Figure 3.5: Sense amplifier circuit

The input section of the sense amplifier is a differential couple, biased by the nMOS shown on the bottom of the figure; the latter is enabled by activating the EN signal (EN=‘1’). The load of the differential couple is a standard latch, made by two cross-coupled inverters.

During the precharge phase of the read operation, the bitlines and the sense amplifier outputs are charged to V_{DD}; during the evaluation phase, the bitlines are left floating and the cell to be read is connected to the lines; hence, the cell discharges one of the bitlines, depending on the stored value.

This results in the developing of a differential voltage at the sense amplifier input, which is amplified by the nMOS differential couple; this drives the upper latch, which is consequently brought in a stable “digital” configuration, that reflects the bitlines voltage values (and, so, the cell content).

It takes some time for the cell to discharge the bitline and, so, for a differential voltage between the bitlines to develop; this means that the sense amplifier cannot be enabled together with (i.e. at the same time instant) the cell because, in that case, the differential voltage at the input of the amplifier could be not well defined and, so, the SA could provide an incorrect result in output, as it is shown in Figure 3.6. In fact, an actual differential amplifier is affected by an input offset voltage: so, even if the differential voltage applied at the input of the amplifier is null, the circuit provides a non null differential output (which, in the memory case, leads to an incorrect output result for the read operation).

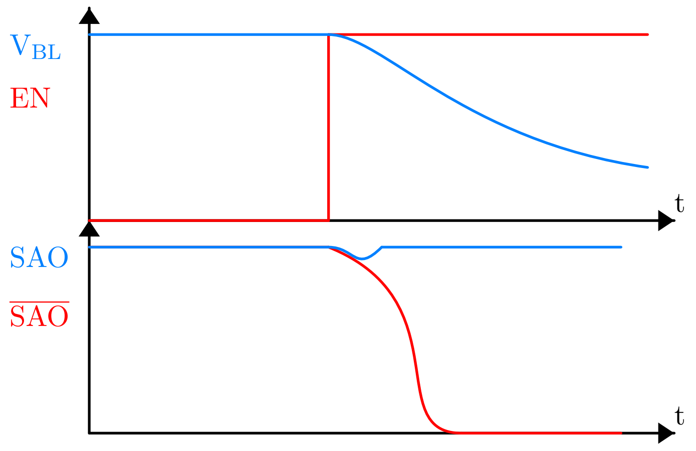


Figure 3.6: Incorrect sensing during read operation

For this reason, the enable signal of the sense amplifier has to be delayed with respect to the one that selects the cell, so that the amplifier starts to sense its input only when a certain minimum differential voltage has developed between the bitlines, as it is shown in Figure 3.7; this is achieved by delaying the EN signal using a cascade of logic inverters, whose number depends on the delay that one wants to obtain which depends, in turn, on the discharging speed of the bitlines (and, so, on the array size).

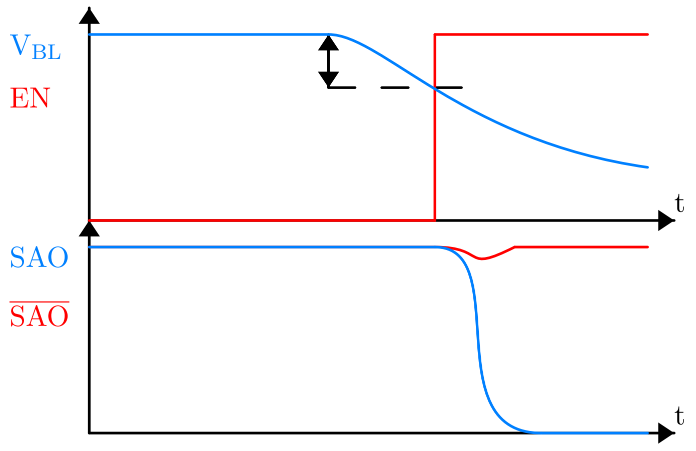


Figure 3.7: Correct sensing during read operation

Of course, this has an influence on the read delay: the larger is the SA delay, the larger will be the sensing delay; hence, the enabling delay has to be kept to a minimum value. The value of the delay, however, depends on the array size: as this increases, the parasitic capacitance of the bitlines enlarges and, so, it takes more time for the cell to discharge the line and, hence, for a certain differential voltage to be developed at the SA input.

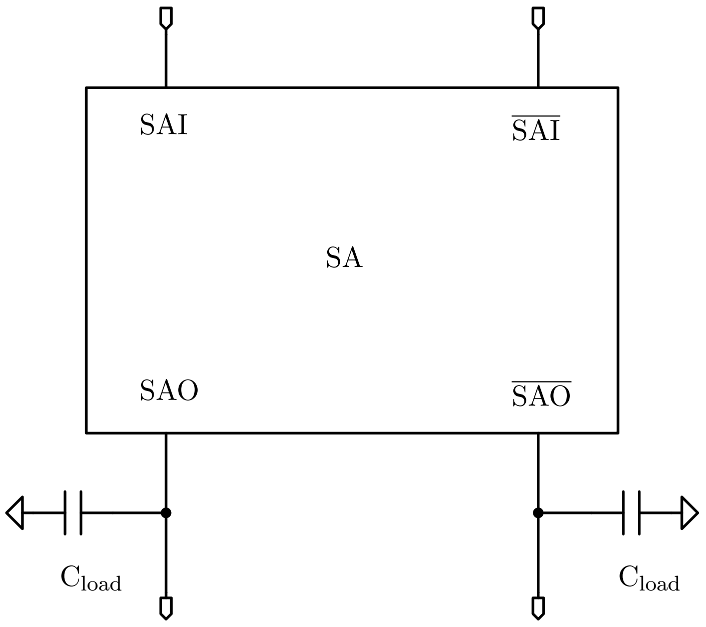


Figure 3.8: Capacitive loads of the sense amplifier

As said before, capacitive loads have been placed at the sense amplifier outputs, in order to emulate the presence of potential busses that the SA needs to drive; however, the value of these capacitances has been limited since, otherwise, it would have influenced too much the power and performance measurements: if C_{load} is very large, the power consumption and the delay of the read operation are completely determined by the sense amplifier, and the influence of the array size on these results would be null.

3.4 Interconnections parasitics

In order to take into account (at least in part) the parasitic capacitance and resistance of the interconnections, *RC* circuits have been placed between each cell, in both vertical and horizontal directions, as it is shown in Figure 3.9.

In this way, the position of the accessed cell inside the array has an influence on the performance of the operation. For example: if the sense amplifiers are placed at the bottom of the memory array, the cells placed on the first row will be slower than the ones placed on the last row, because they are separated from the sense amplifiers inputs by a larger number of *RC* circuits; for the same reason, if the wordline drivers are placed on the left of the memory array, the cells placed on the right of the array will be slower than the ones placed on the left during an access operation, being separated by a larger distance from the wordline drivers.

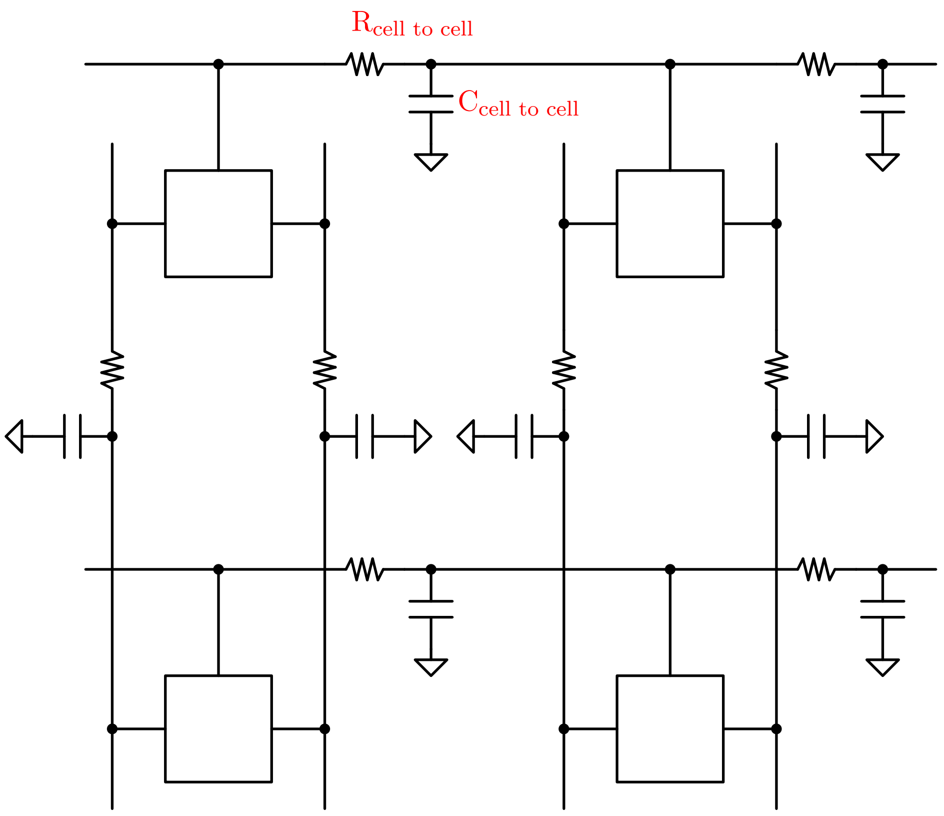


Figure 3.9: Interconnections parasitics model

3.5 Testbench

The testbench circuit is presented in Figure 3.10: the wordline drivers are placed on the left of the array; the bitline driver and precharge circuits are placed on top of the array; the sense amplifiers are placed on the bottom of the array.

3.5.1 The simulation flow

The design has been conducted at schematic level in Cadence Virtuoso, using the ST FDSOI 28nm technological library. The first element to be designed has been the SRAM cell; then, this has been arranged in an array, like it is partially shown in Figure 3.9.

After the creation of the array, the drivers for the input signals have been designed, such as the wordline drivers, bitline drivers and precharge circuits; then, the sense amplifier topology has been chosen and implemented.

All these hardware elements have been connected and organized in a testbench, discussed in subsection 3.5.3, which has been simulated using the SPECTRE simulator embedded in Cadence Virtuoso.

In order to generate the input signals for the testbench, a Python script has been created. This allows to easily set up the simulation, allowing the definition of every

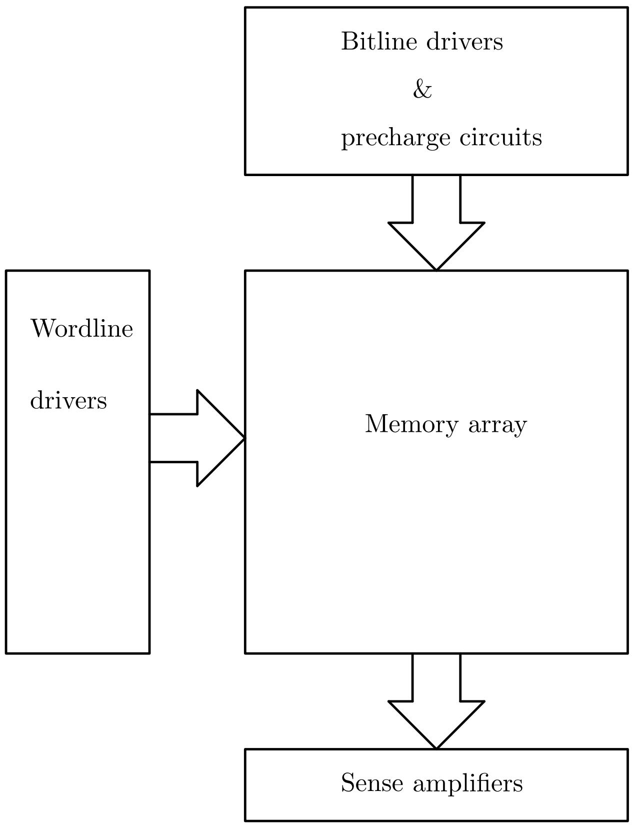


Figure 3.10: Testbench circuit

kind of memory operation through few simple steps. The Python code is discussed in subsection 3.5.2.

From the testbench, the array output and input signals have been extracted, together with the whole testbench instantaneous power consumption. This information has been elaborated in Cadence Virtuoso to estimate:

- the average power consumption per operation.
- the energy consumption per operation.
- the delay associated to write and read operation.

The same design and simulation flow has been followed for each memory architecture implemented. For this reason, in Figure 3.11 also the delay of the logic implemented inside the memory are listed in the results.

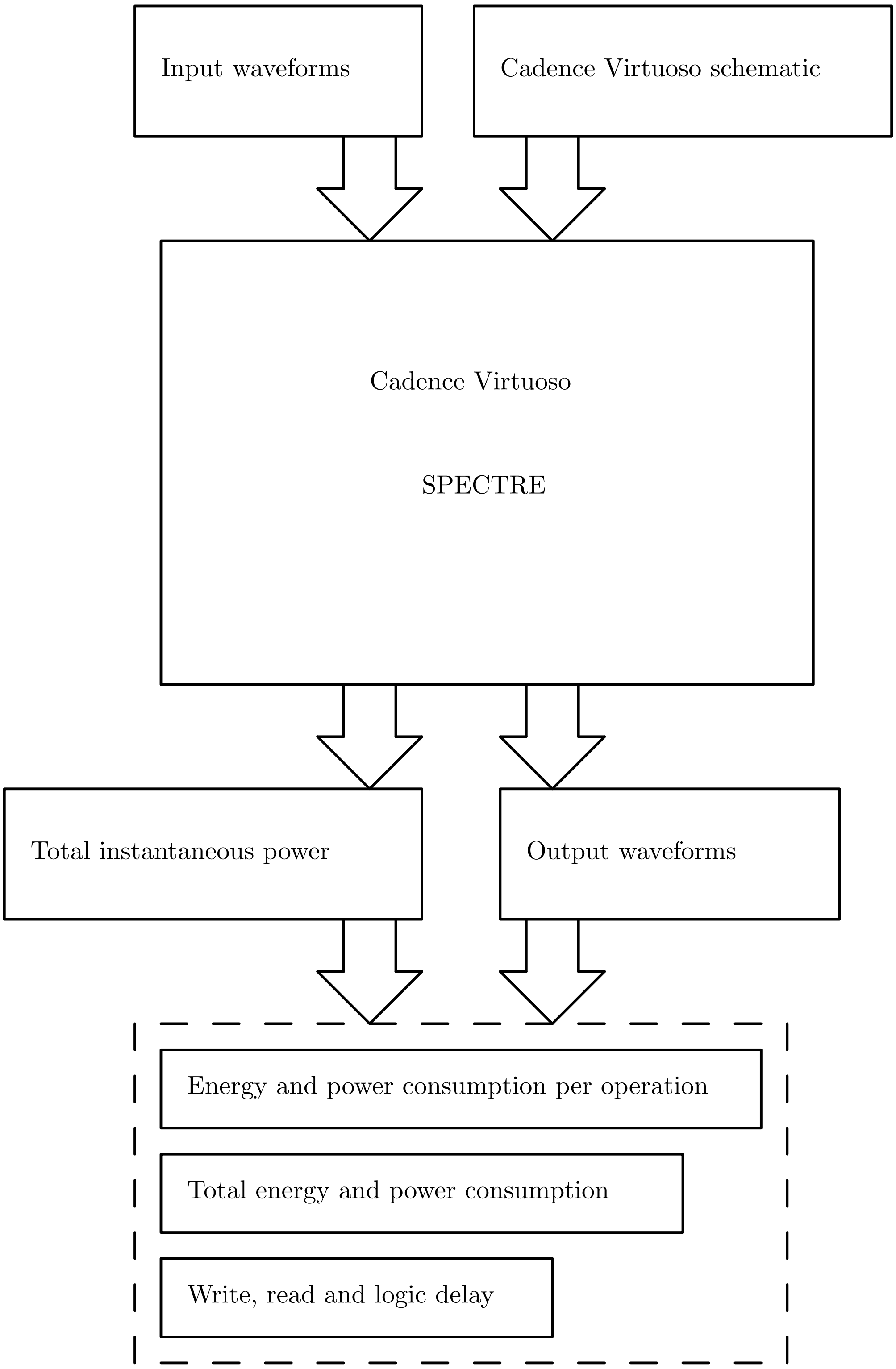

Input waveforms
Cadence Virtuoso schematic
Cadence Virtuoso
SPECTRE
Total instantaneous power
Output waveforms
Energy and power consumption per operation
Total energy and power consumption
Write, read and logic delay

Figure 3.11: Simulation flow

3.5.2 Python script for input signals generation

```python
#! /usr/bin/env python
import sys

# Simulation parameters
T_ck = 1e-9*float(sys.argv[1])
t_rise = T_ck / 1000 #Rise and fall time of the signals.
Vdd = float(sys.argv[2])
```

First of all, the simulation parameters are defined:

- `Vdd` represents the supply voltage of the circuit. The value $V_{DD} = 0.92\,\mathrm{V}$ has been chosen since it is the same supply voltage used by the Synopsys Design Compiler for the synthesis of digital circuits to which the performance of the LiM array would be compared.
- `T_ck` represents the cycle duration. The default value of this is 1 ns, but it can be increased via command line.
- `t_rise` is the rise and fall time of the signals generated by this script. In this case, it is chosen to be 1000 times smaller than the cycle duration.

```python

def write_cycle_to_file(file_pointer, value, cycle):
    if cycle == 0:
        file_pointer.write('0' + " " + str(value) + "\n") #Beginning of
                                                           the simulation.
    else:
        file_pointer.write(str(cycle*T_ck + t_rise) + " " + str(value) +
                                                           "\n") #After a time
                                                           interval equal to t_rise,
                                                           we assign the new value to
                                                           the waveform.
    file_pointer.write(str((cycle+1)*T_ck) + " " + str(value) + "\n")
                                                    #We keep the same signal value
                                                    until the end of the cycle.
```

Here, a function called `write_cycle_to_file()` is defined: this writes the signals values in the current simulation cycle to the proper files.

First, if the first cycle is being analyzed (`cycle=0`), then the value of the signal at the beginning of the simulation is defined; otherwise, the actual value of the signal is assigned to the waveform after a time equal to `t_rise` with respect to the beginning of the cycle. Then, the same value is assigned to the waveform at the end of the cycle.

```python
# Signals of the testbench.
signals = { 'WLFirstRow': {'file_pointer': None, 'value': 0,
                                    'default_value': 0},
'BL': {'file_pointer': None, 'value': Vdd, 'default_value': Vdd},
'BLn': {'file_pointer': None, 'value': Vdd, 'default_value': Vdd},
'Precharge_n': {'file_pointer': None, 'value': Vdd, 'default_value':
                                    Vdd},
'EnableSA': {'file_pointer': None, 'value': 0, 'default_value': 0},
'EnableBLDriver': {'file_pointer': None, 'value': Vdd, 'default_value':
                                    Vdd},
}
```

+

Here, the signals of the testbench are defined in a Python dictionary. To each signal, three quantities are assigned:

- the pointer to the file to be written, `file_pointer`.
- the value assumed by the signal, `value`.
- the default (non-active) value of the signal, `default_value`.

```
# The files associated to the signals are opened for writing.
for key in signals.keys():
    signals[key]['file_pointer'] = open((key + ".csv"), 'w+')
```

Here, the signals file pointers are assigned to the files to be written, that are opened in write mode.

```
# Operations to simulate.
operations = ("TestWriteSpeed0", #0
"Idle", #1
"TestWriteSpeed1", #2
"Precharge_cycle", #3
"Read", #4
"Idle", #5
"TestWriteSpeed0", #6
"Precharge_cycle", #7
"Read") #8
```

These lines of code define the operations to be simulated, that are saved in a Python tuple. First, a dummy `Write1` operation is performed in order to initialize the cell content; then, two write-read cycles are performed.

It has to be noticed that an `Idle` operation precedes every write operation apart from the initialization one: this is because the bitlines have to be at the same value before every write operation in order to get a valid energy measurement for the write operation.

After the `Read` operation at `#4`, since a logic '1' has been read, the bitlines have the following values (refer to Figure 3.1): BL='1', $\overline{\text{BL}}$='0'. Hence, to subsequently write a logic '0', both bitlines would be subjected to a commutation: BL='1'→'0', $\overline{\text{BL}}$='0'→'1'.

One the contrary, before the `TestWriteSpeed1` operation, in which a logic ‘1’ is written, the bitlines configuration is BL=$\overline{\text{BL}}$=‘1’, since in an `Idle` operation the bitlines are brought to the logic ‘1’ by the drivers; hence, only one bitline changes its state in the subsequent write operation: $\overline{\text{BL}}$=‘1’→‘0’.

From this it can be deduced that larger energy and delay would be involved in the `TestWriteSpeed0` operation with respect to `TestWriteSpeed1` if there was not the `Idle` operation in between the read and write operation.

```python
cycle=0 # First cycle

for operation in operations :

    # All signals are assigned their non-active values a priori, and they are activated only when needed in memory operations.
    for key in signals.keys():
        signals[key]['value'] = signals[key]['default_value']

    # Depending on the kind of operation, the signals values are properly assigned.
    if operation == "TestWriteSpeed1":
        signals['WLFirstRow']['value'] = Vdd
        signals['BL']['value'] = Vdd
        signals['BLn']['value'] = 0

    elif operation == "TestWriteSpeed0":
        signals['WLFirstRow']['value'] = Vdd
        signals['BL']['value'] = 0
        signals['BLn']['value'] = Vdd

    elif operation == "Read":
        signals['WLFirstRow']['value'] = Vdd
        signals['EnableSA']['value'] = Vdd
        signals['EnableBLDriver']['value'] = 0

    elif operation == "Precharge_cycle":
        signals['Precharge_n']['value'] = 0
        signals['EnableBLDriver']['value'] = 0

    elif operation == "Idle":
        pass

    else:
        print("Error! " + i + " is not an allowed operation.\n")
        exit(1)
```

Here, the signals behavior in each cycle is defined. First, all the signals are assigned their default values, and they are activated only in the cycles where they need to; second, for each operation the corresponding signals combination is defined.

```
    # The signals configuration is written to the files.
    for key in signals.keys():
        write_cycle_to_file(signals[key]['file_pointer'],
                                                    signals[key]['value'],
                                                    cycle)

    cycle = cycle + 1 #We go to the next cycle.

# The files are closed.
for key in signals.keys():
    signals[key]['file_pointer'].close()
```

At the end, all the signals values of the current simulation cycle are written to their files that will be used by the simulator; after this, all the files are closed and the program ends.

If one wants to add a new operation to the simulation, a new case has to be added to the `if` statement with the corresponding signal combination, together with a new operation in the `operations` vector.

If one wants to add a new signal to the memory, a new corresponding entry has to be added to the `signals` dictionary, and the signal value in each memory operation has to be configured.

3.5.3 Cadence Virtuoso schematic

In Cadence Virtuoso a voltage source is associated to each `.txt` file that is generated through the Python script, as it is shown in Figure 3.12.

The first array designed is the 8x8 one. In order to obtain it, a 8 bits memory column is created, which schematic is shown in Figure 3.13. It can be noticed how a parasitic *RC* circuit is present between each cell in both vertical and horizontal directions. For testing purposes, also the data stored in the first row and last row cells are extracted, in order to verify that the cell content is successfully written after a write operation, or that it is not overwritten during a read one.

On top and bottom of the column, pins for the bitlines are added (`BL_top`, `BLn_top`, `BL_bottom` and `BLn_bottom`), so that the column can be connected to other columns in the vertical direction. At the left and right sides of the column, pins for the wordlines are added (`WL_in<0:7>` and `WL_out<0:7>`), in order for the column to be connected also in the horizontal direction.

Then, the 8x8 array is created using eight 8 bit columns, arranged as it is shown in Figure 3.14.

In this way, a 8x8 array component is obtained and it can be used in the testbench as Device Under Test (DUT). The component is shown in Figure 3.15.

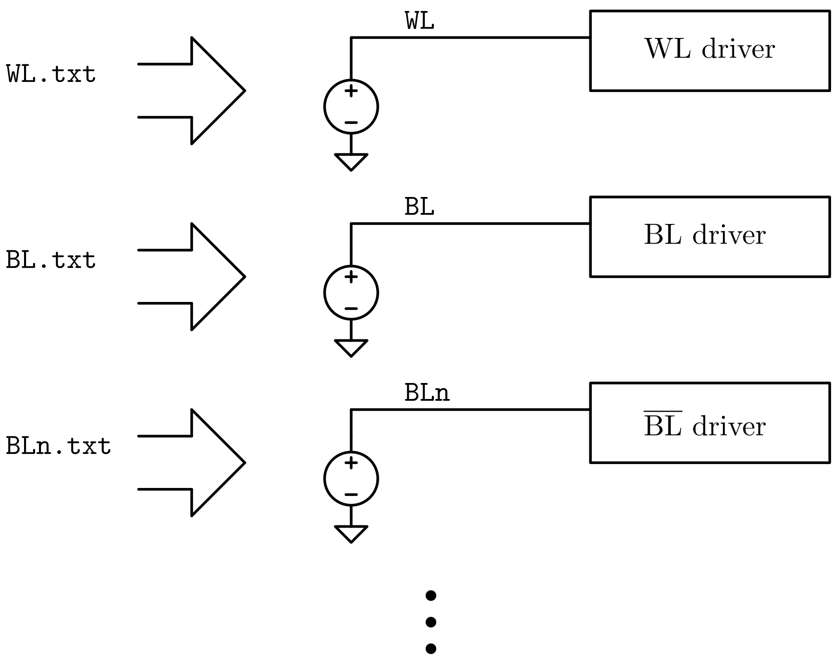


Figure 3.12: Voltage sources for the input signals

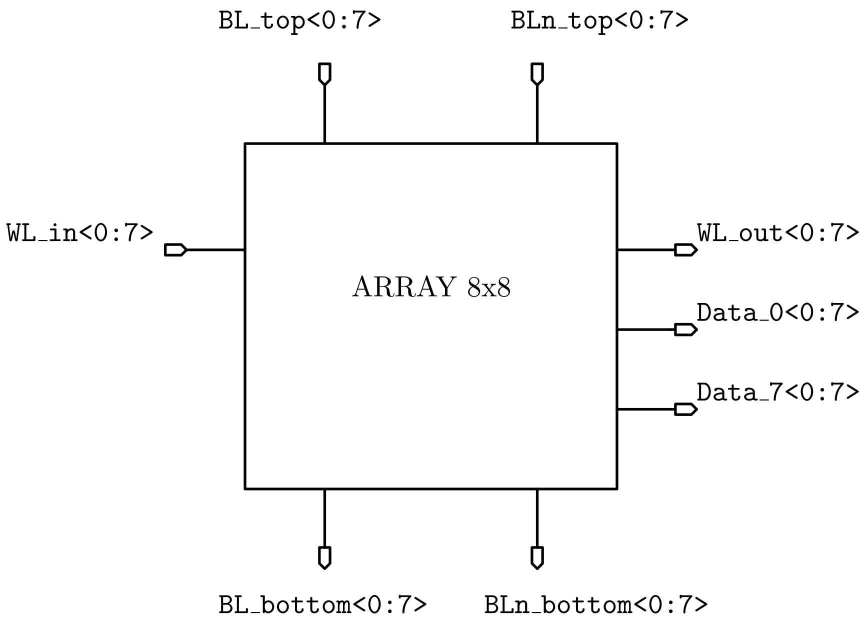


Figure 3.15: 8x8 array component

Then, the array component is inserted in the testbench schematic, which is shown in Figure 3.16. It can be noticed how only one wordline (`WL<0>`) of the array is driven to the logic '1', while all the others are forced to the logic '0', in order to isolate those memory rows from the bitlines; in fact, only one row of the memory is written and read in the simulation, in order to estimate the delay and power consumption of a single

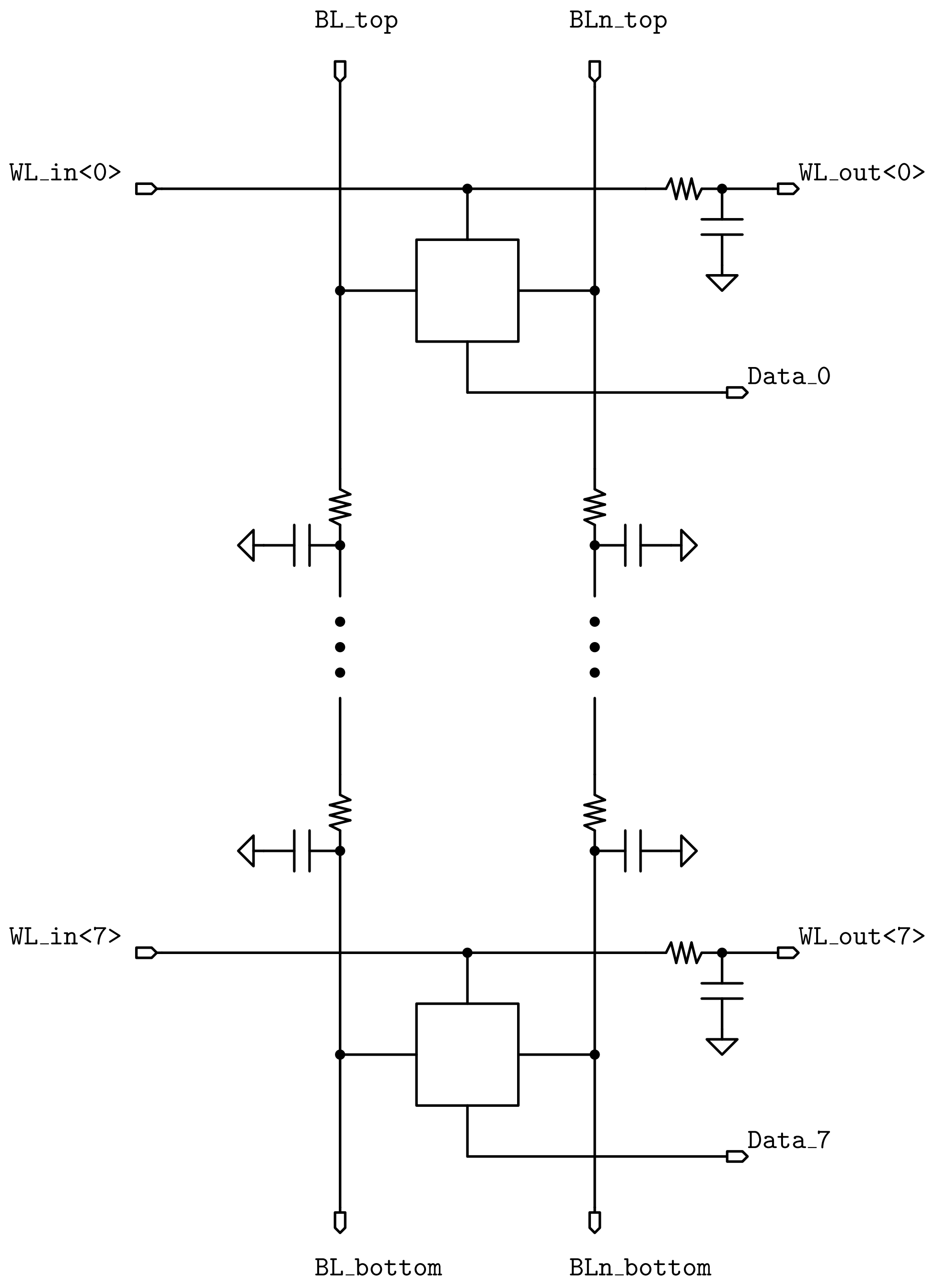


Figure 3.13: 8 bit memory column

memory row, even if conditioned by the presence of the other memory lines. Usually, in fact, only one row is processed in a memory array at time.

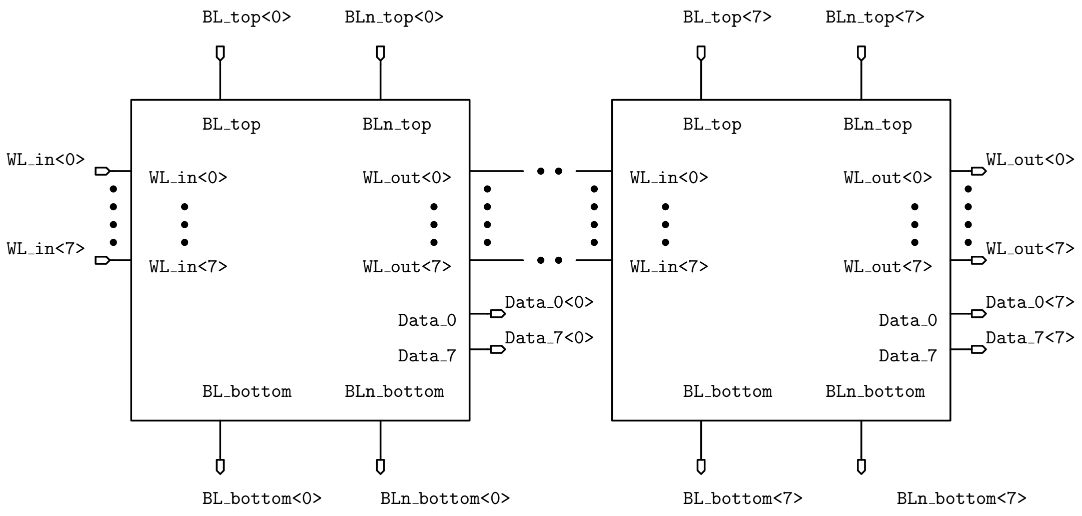


Figure 3.14: 8x8 array

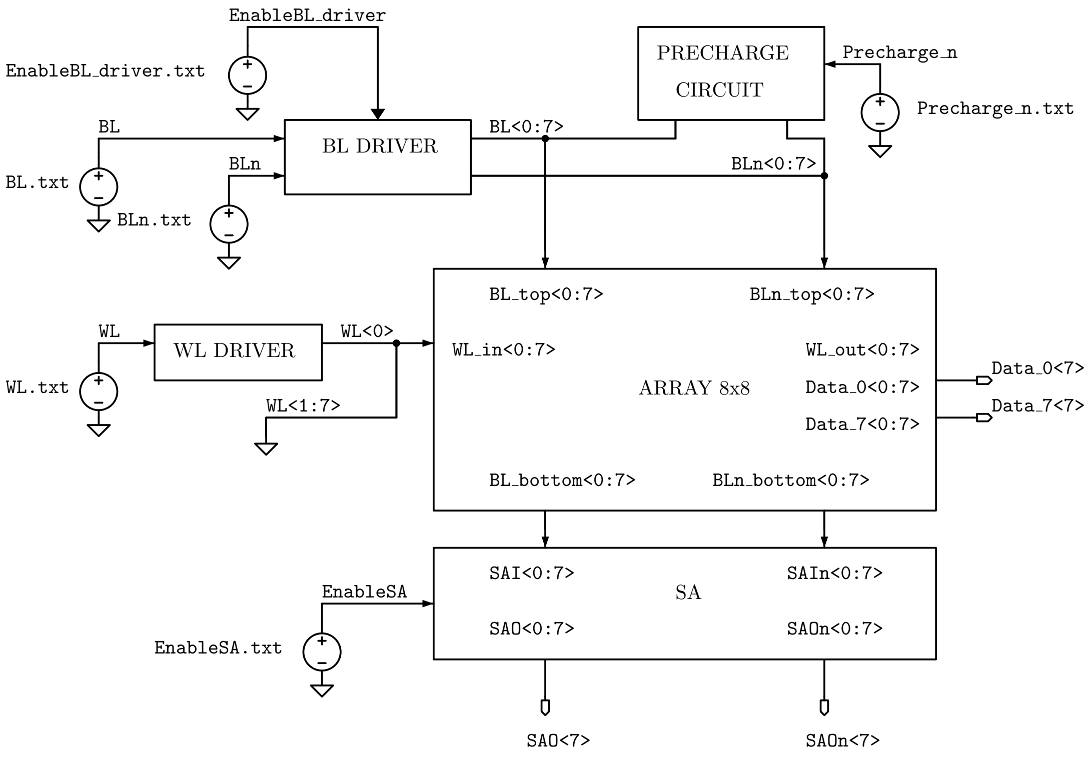


Figure 3.16: Testbench schematic

All the driver circuits are driven by the voltage generators presented in Figure 3.12. The signals that are extracted from the testbench are:

- the control signals `Precharge_n`, `EnableBL_driver`, `BL`, `BLn`, `EnableSA`.
- the output of the drivers `WL<0>`, `BL<0:7>`, `BLn<0:7>`.
- the outputs of the array `SAI<7>`, `SAIn<7>`, `Data_0<7>`, `Data_7<7>`.

- the outputs of the sense amplifiers SAO<7> and SAOn<7>.

3.5.4 Larger arrays design

Starting from a 8x8 array, larger arrays can be easily obtained. For example, if one wants to obtain a 16x16 array, four 8x8 arrays have to be combined, as it is shown in Figure 3.17.

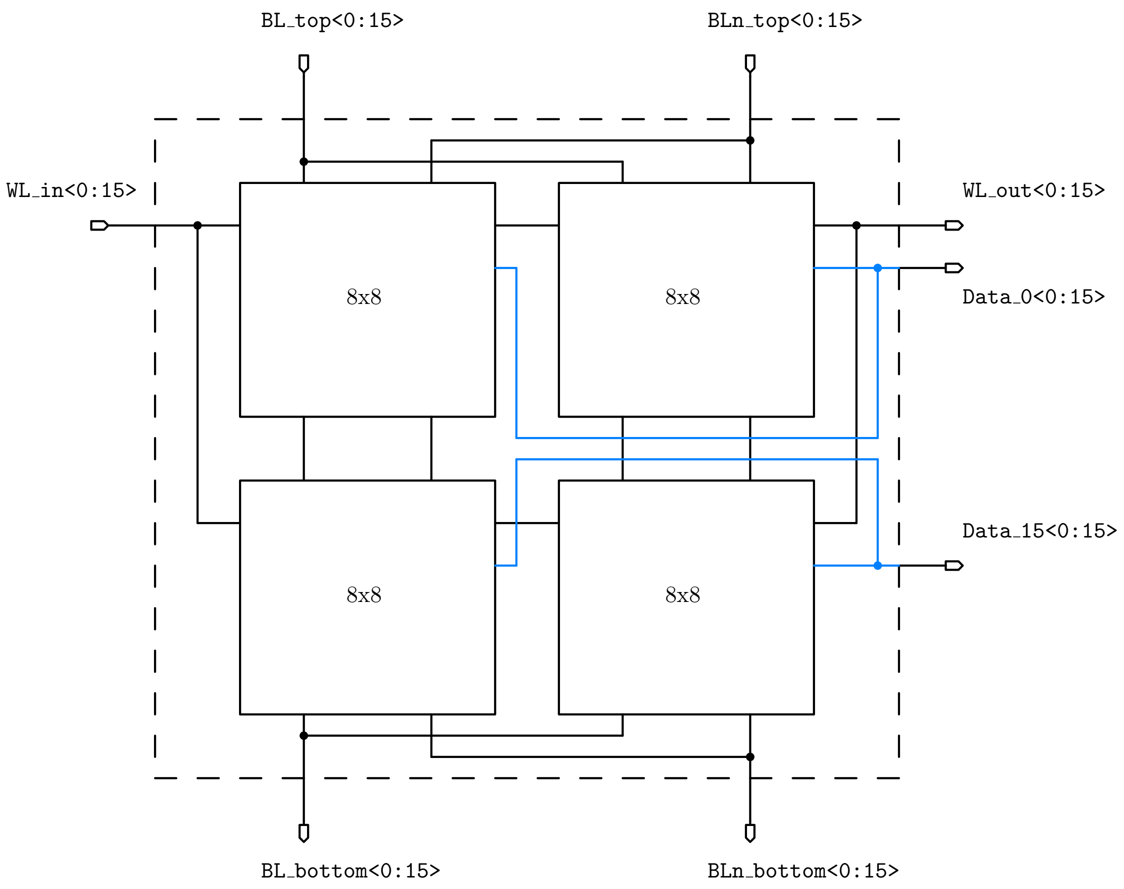


Figure 3.17: 16x16 array obtained from 8x8 arrays

In turn, the 16x16 array can be used to obtain a 32x32 one, and so on. Of course, the design has to be conducted by hand at schematic level in Cadence Virtuoso, and the surrounding circuitry has to be adapted to the new array dimensions.

In fact, as the array size increases, the resistive-capacitive load that the surrounding circuits have to drive enlarges; hence, the driving capability of those has to be improved.

Also, the sense amplifier has to be modified, since the resistive-capacitive load that a cell has to drive during a read operation enlarges as the array size increases. In particular, the delay of the enabling signal has to be enlarged, in order to guarantee a correct sensing during the read operation.

3.5.5 Simulation environment

The choice of the signals to be analyzed in the testbench is important; in fact, the signals that are considered are the following:

- the wordline of the first memory row, `WL<0>`.
- the content of the last cell (from left to right) of the first row, `Data_0<7>`.
- the content of the last cell of the last row, `Data_7<7>`.
- the input of the last sense amplifier (from left to right), `SAI<7>` and `SAIn<7>`.
- the output of the last sense amplifier, `SAO<7>` and `SAOn<7>`.

The reason of this is presented in Figure 3.18.

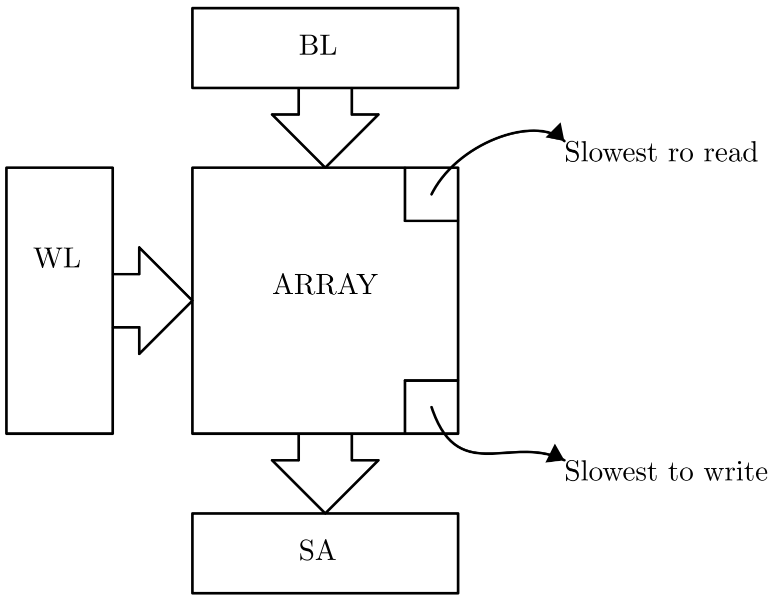


Figure 3.18: Worst cases for write and read operations.

When reading a cell, the worst case, which means the one to which the largest read delay is associated, is related to the cell placed on the first row and on the last column of the array:

- being placed on the first row, the distance from the sense amplifier input and, so, the resistive-capacitive load that the cell has to drive with respect the SA input node, are maximized; hence, this represents the worst case for the data sensing delay.
- being placed on the last column, the distance from the wordline driver and, so, the resistive-capacitive load that this has to drive with respect to the cell position, are maximized; hence, this represents the worst case for the cell access delay.

When writing to a cell, the worst case is related to the one placed on the last row and on the last column of the array:

- being placed on the last row, the distance from the bitline driver outputs and, so, the resistive-capacitive loads that this has to drive with respect to the cell position, are maximized; hence, this represents the worst case for the write delay.

- being placed on the last column, this represents the worst case for the cell access delay.

Hence, the delay of the read operation is computed considering the first row of the array and the output of the corresponding sense amplifier (`SAO<7>`); the delay of the write operation is computed considering the last row of the array and the last cell content (`Data_7<7>`).

The read delay is calculated between the output of the sense amplifier `SAO<7>` and the `WL` signal, which is the output of the wordline signal voltage generator; hence, it is computed using an ideal signal as reference, and not the output of the wordline driver. The graphical definition of this delay is reported in Figure 3.19. It can be noticed how the delay has been calculated as the 50% delay between the signals `WL` and `SAO<7>`.

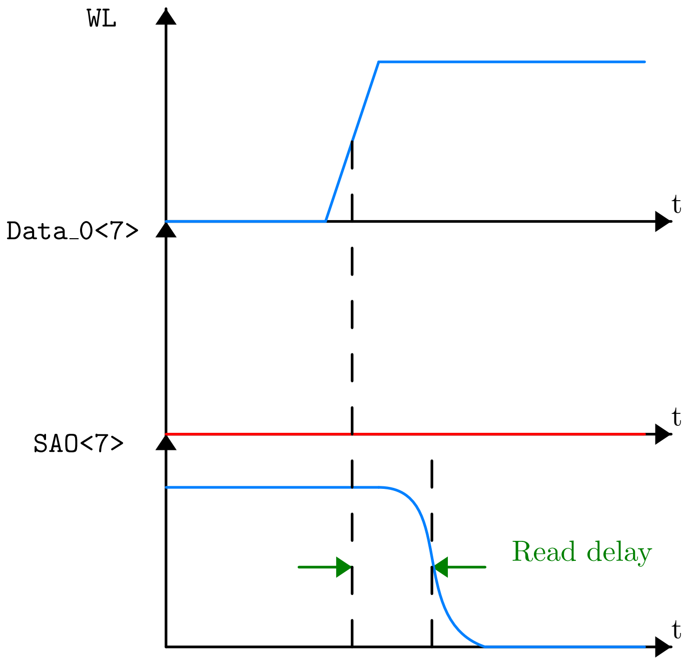


Figure 3.19: Read delay definition.

The write delay is defined in an analogous way, using the `WL` signal and the content of the last cell of the last row `Data_7<7>`, as shown in Figure 3.20.

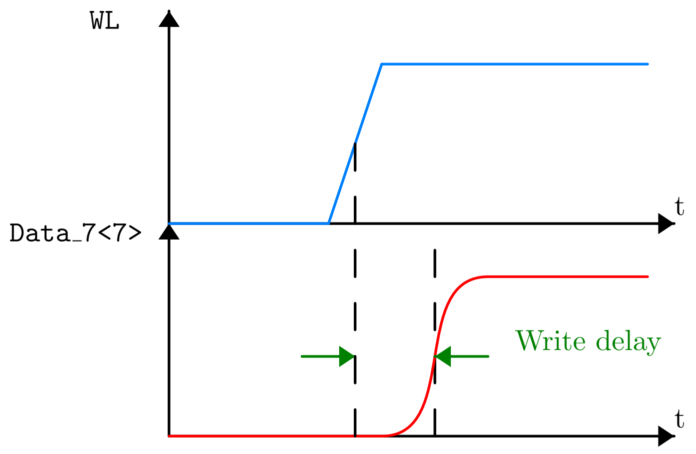


Figure 3.20: Write delay definition.

For what concerns the power consumption calculation, the instantaneous total power consumption, reported as `Power` in Figure 3.21, is evaluated in SPECTRE and, then, elaborated to obtain the average power consumption and the energy consumption of each operation.

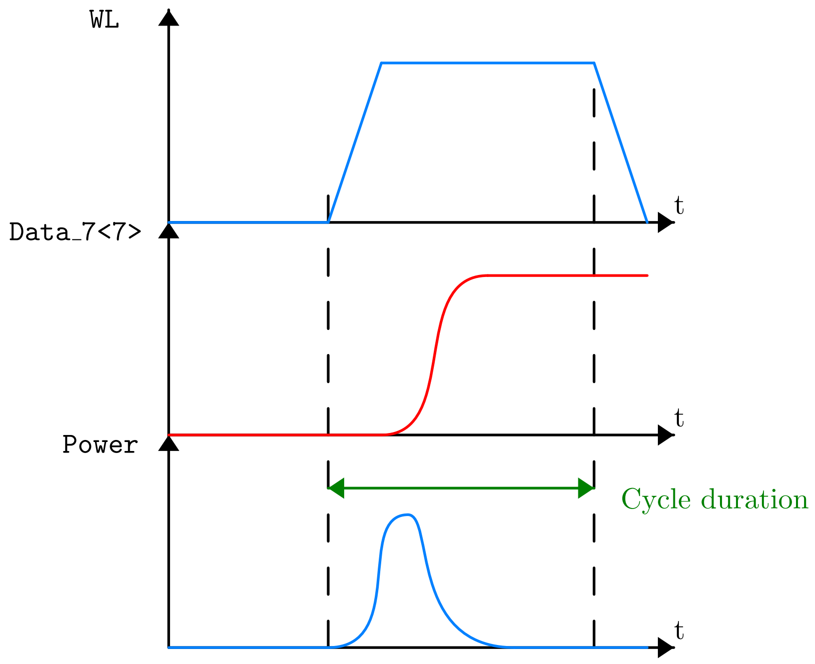


Figure 3.21: Power consumption evaluation in a cycle

In order to obtain the average power consumption of an operation, which is defined as $P_{operation}$, the instantaneous power, $p(t)$, is integrated along the cycle in which the considered operation is performed. In this way, the energy consumption of the operation $E_{operation}$ is obtained:

$$E_{operation} = \int_{cycle} p(t)dt$$

Then, this quantity is divided by the simulation duration, T_{sim}; in this way, $P_{operation}$ is obtained:

$$P_{operation} = \frac{E_{operation}}{T_{sim}}$$

The average value is not computed with respect the cycle duration but with respect the simulation duration because, in this way, the weight of the operation with respect to the total average power consumption is obtained.

3.6 Simulation waveforms

In Figure 3.22, the output waveforms of a Virtuoso simulation are presented: in particular, the write and read operations are executed one after the other.

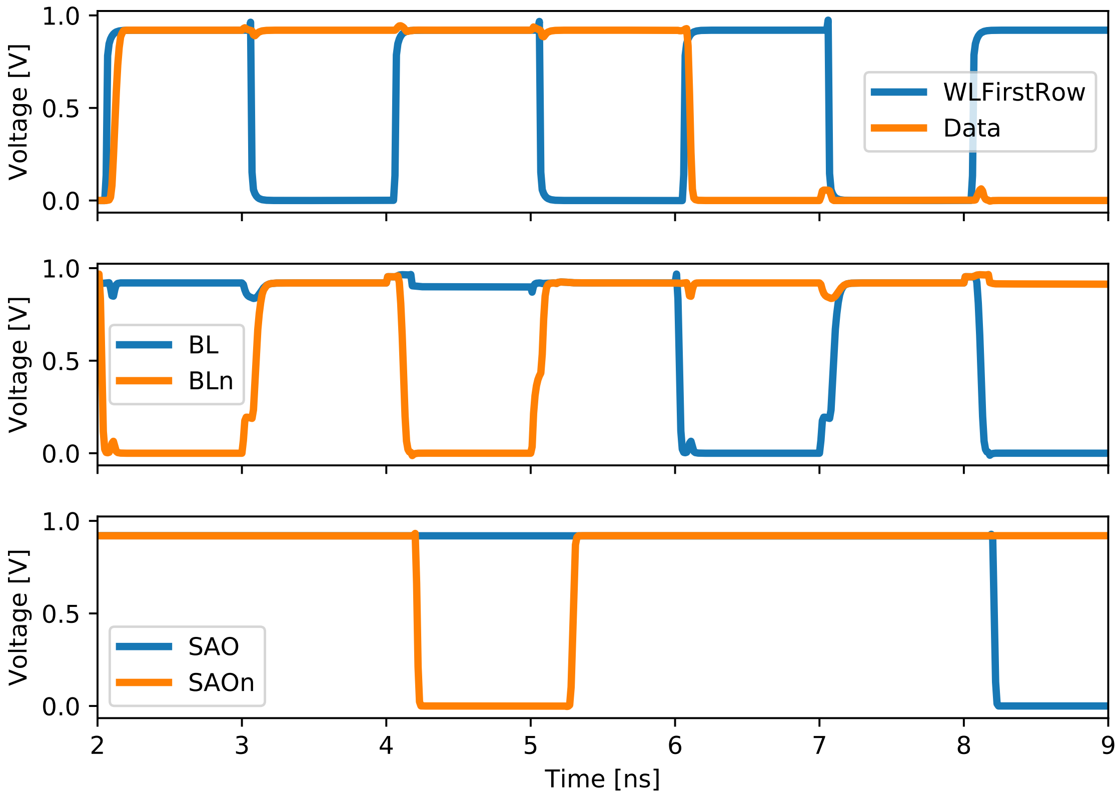


Figure 3.22: Write and read cycle

In the first cycle, a logic ‘0’ is written to the cell:

$$\texttt{WL = 1,BL = 0,BLn = 1} \rightarrow \texttt{Data = 0}$$

In the second cycle, the bitlines are precharged:

$$\texttt{Prechage\_n = 0} \rightarrow \texttt{BL = 1,BLn = 1}$$

In the third cycle, the cell content is read:

$$\texttt{WL = 1} \rightarrow \texttt{SAI = 0} \rightarrow \texttt{SAO = 0}$$

In Figure 3.22 some signals have been omitted for the sake of clarity.

Chapter 4

CAM design

In this chapter, the design and verification of a CAM memory is discussed; three arrays have been designed and analyzed, which dimensions are 8x8, 16x16 and 32x32 bits, using the ST FD-SOI 28 nm technological library in Cadence Virtuoso.

4.1 The CAM cell

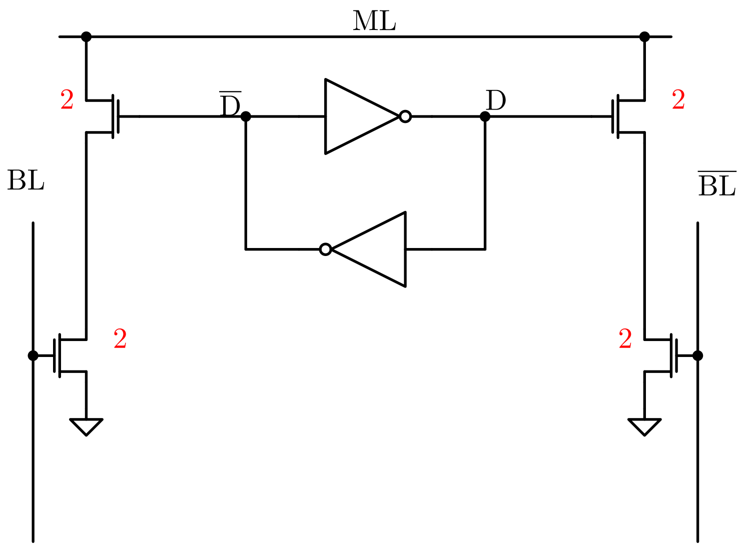


Figure 4.1: CAM cell implemented.

The SRAM memory core is sized as discussed in section 3.1. For what concerns the CAM section, the transistors have been sized choosing an aspect ratio doubled with respect to the minimum one, as it is shown in Figure 4.1; the reason behind this choice is explained in section 4.5.

4.2 Driver circuits

The drivers implemented in the CAM architecture are the same used for the SRAM one(section 3.2), since the original SRAM bitlines are used as searchlines for the CAM operation: there is no need to use dedicated column lines for the search operation, since the latter is never carried out together with a write or read operation.

4.3 Match-line sense amplifier

The sensing scheme implemented is the current-saving one (Figure 2.30), and the sense amplifier architecture proposed in [43] has been chosen, which circuit is shown in Figure 4.2.

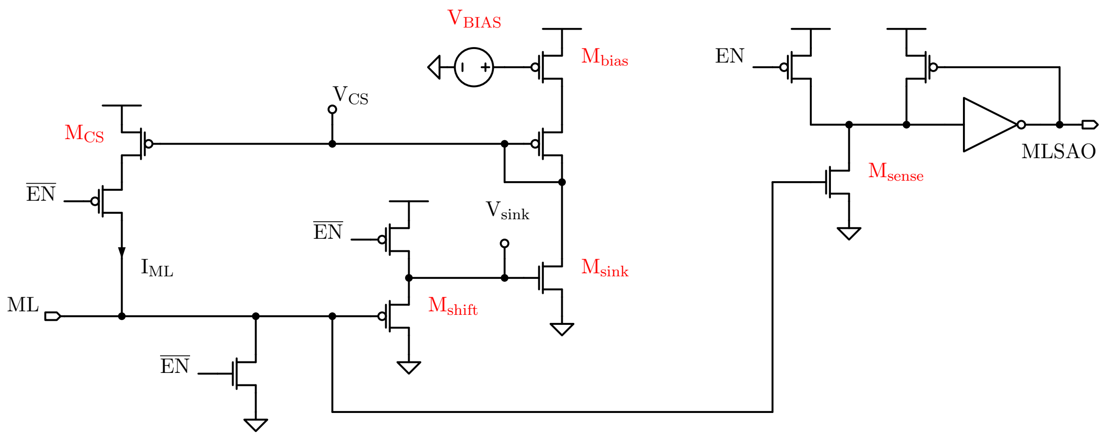


Figure 4.2: Match-line sense amplifier

As explained in section 2.2.3, the search operation consists of two phases: a pre-discharge phase and an evaluation phase.

From Figure 4.2, it can be noticed how only one enable signal, $\overline{\text{EN}}$, is used for the sense amplifier; this is, then, internally inverted obtaining EN, in order to be used as pre-discharge signal when $\overline{\text{EN}}$=‘1’. This implies that when the sense amplifier is not enabled, it is in the pre-discharge configuration.

The pre-discharge phase begins by deactivating the enable signal ($\overline{\text{EN}}$ = ‘1’) of the MLSA (Match-Line Sense Amplifier): the match-line (ML in Figure 4.2) is discharged to the ground voltage through a pull-down transistor; at the same time, the input of the inverter is charged to V_{DD} through a pull-up pMOS.

Then, the evaluation phase starts by activating the enable signal ($\overline{\text{EN}}$ =‘0’). At the beginning of the cycle, the voltage on the match-line is close to 0 V; hence, the gate voltage of M_{sink}, V_{sink}, is close to V_{TH}, since M_{shift} is in common-drain configuration, which practically means that:

$$V_{sink} \approx V_{ML} + V_{TH}$$

For this reason, the gate voltage of M_{CS}, V_{CS}, is close to V_{DD}, because M_{sink} is in common-source configuration, which is practically an inverting configuration; however, V_{CS} is large enough to turn it on and, so, a current I_{ML} is injected in the match-line.

What happens from now on, depends on the result of the comparison between the search word and the stored word:

- if the two words match, the line behaves like a capacitance (Figure 2.24) and it starts to get charged; hence, the voltage on the match-line arises and, so, V_{sink} increases: this results in the decrease of V_{CS} and, so, M_{CS} conducts a larger current. Since I_{ML} enlarges, the voltage on the match-line arises in a faster way, speeding up the match sensing; as soon as the threshold voltage of M_{sense} is crossed, it is turned on and discharges to ground the inverter input which, at this point, switches its output (MLSAO = ‘0’ $\rightarrow$ ‘1’): a match is provided as result.

- if the words do not match, the match-line does not get charged and the inverter input voltage does not vary: a mismatch is provided in output.

The MLSA working principle is shown also in Figure 4.4, for the match case, and in Figure 4.3, for the mismatch case.

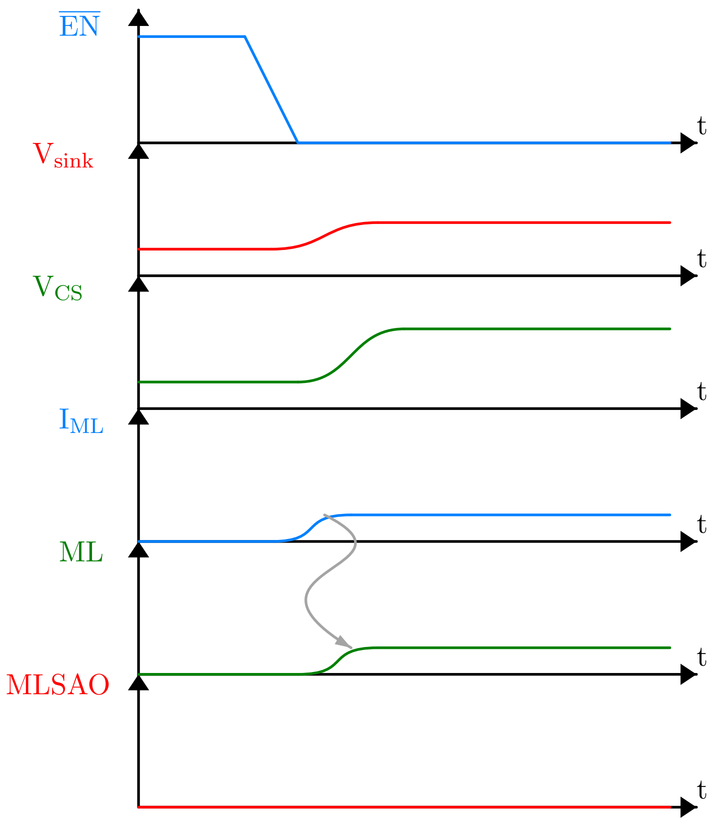


Figure 4.3: Mismatch case

In Figure 4.3 it is shown how, as soon as the MLSA is enabled by $\overline{\text{EN}}$=‘0’, a current is injected in the match-line, causing a slight increase in the line voltage due to the fact that the discharging paths have a certain resistance R_{cell}, which is not null; however, the match-line potential is not high enough to activate the MLSA and, so, its output remains at the logic ‘0’.

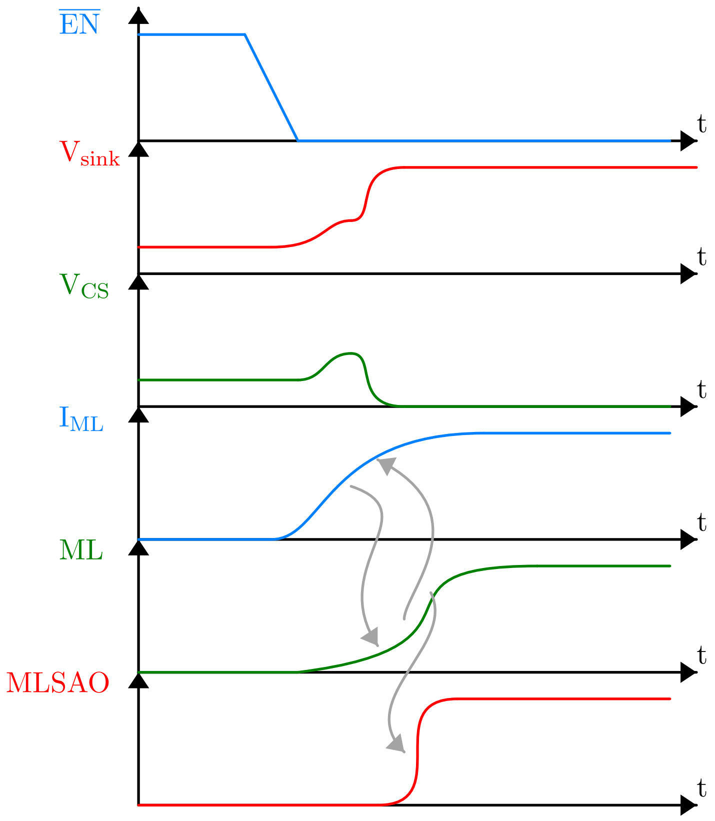


Figure 4.4: Match case

In Figure 4.4 it is shown how the match-line is charged to the logic ‘1’ by the current generator and a match result is provided in output. In this graph, it is highlighted the positive feedback between the current injected in the line and the voltage on this: as a current is injected in the match-line, the voltage on this starts to increase; this leads, thanks to the positive feedback, to an increase in I_{ML}, that in turn makes the line voltage rise in a faster way, until the MLSA output switches from ‘0’ to ‘1’.

Another advantage of this sensing scheme is that the search delay does not depend on the bitlines drivers strength.

In the standard sensing scheme, where the match-line is precharged and, then, discharged, all the cells are disabled by setting the bitlines to the logic ‘0’ during the precharge phase, so that the line can be charged. In fact, referring to Figure 4.1, if both the bitlines are set to the logic ‘1’, at least one of the pull-down paths is enabled, since one between D and $\overline{\mathrm{D}}$ is at the logic ‘1’; if only one between BL and $\overline{\mathrm{BL}}$ is at the logic ‘1’, it may happen that one of the pull-down paths is enabled if the cell content matches with the datum present on the bitlines; hence, the only way to ensure that all the cells are disabled is to set all the bitlines to the logic ‘0’.

This means that one has to load the datum on the bitlines at the beginning of the search cycle and, since the cells can start to discharge the line only when the transistors connected to the bitlines (Figure 4.1) are enabled, this results in a contribution to the search delay. In particular, the delay value is dependent on the row position inside the array: the farer this is from the bitlines drivers, the larger is the delay contribution associated to the search datum loading.

In the current-race scheme, this is not an issue: since the line is pre-discharged before the search operation, there is no need to disable the cells; hence, one can load the search datum on the bitlines during the pre-discharge operation: this means that the gate voltage of the transistors connected to the bitlines will be already stable at the beginning of the search cycle and, so, the drivers strength and the row position have no influence on the delay value.

4.4 Dummy match-line scheme

In section 4.3 it has been explained that in the mismatch case the match-line does not get charged, and the current injected by the MLSA in the line flows through the pull down paths of the cells to ground; hence, energy is wasted, since this current keeps flowing until the end of the search cycle. To limit the energy waste, a dummy match-line sensing scheme (also called “replica” scheme) is employed.

In this scheme, a dummy memory row is added, whose content always matches the search word independently of the value encoded in the latter. The output of the associated sense amplifier is then used to disable all the other MLSAs as soon as the dummy line provides a match; in this way, the current generator is disabled and no energy is wasted, since the current flow to ground (mismatch case) is stopped as soon as possible.

The line is created using dummy memory cells: these are cells in which only the transistors connected to the match-line are present, while all the others are removed; furthermore, the gate potentials of these transistors are chosen so that the cell gives a match as result (i.e. both cell pull down paths are disabled). An example of dummy memory cell is shown in Figure 4.5.

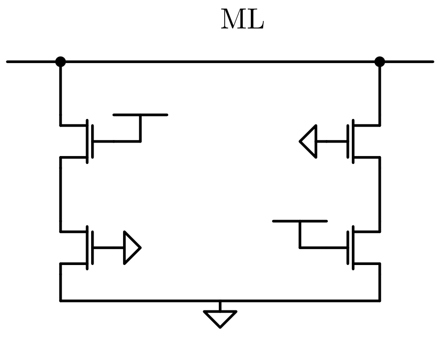


Figure 4.5: Dummy match-line cell.

In Figure 4.5, it can be noticed how each pull-down path is made of an enabled and a disabled transistors in series, as it happens in a real CAM cell whose content matches the search bit.

These dummy cells are arranged in a line (Figure 4.6) which is connected to the input of a sense amplifier; the output of this is used as enable signal of the MLSAs connected to the real memory rows, as it is shown in Figure 4.7.

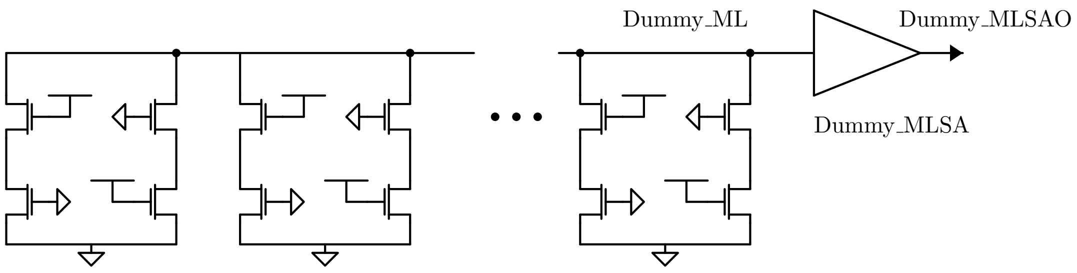


Figure 4.6: Dummy match-line

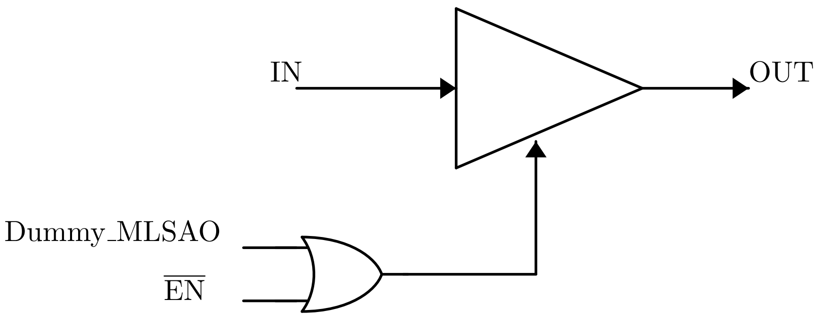


Figure 4.7: Dummy match-line sensing scheme.

In Figure 4.7 it is shown that the output of the dummy MLSA is OR-ed with the external $\overline{\text{EN}}$ signal: in this way, as soon as one of the two signals is equal to ‘1’, the enable signal of the sense amplifier is brought to ‘1’, disabling its current generator and, consequently, cutting the current flow in the match-line; in other words, as soon a match occurs on the dummy match-line, all the sense amplifiers of the array are disabled.

The complete memory scheme is shown in Figure 4.8.

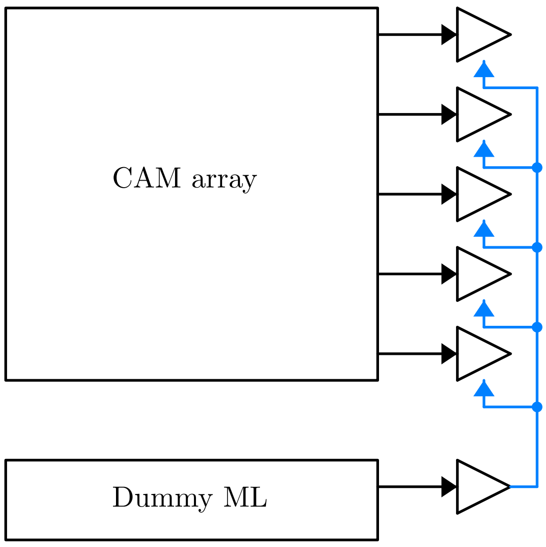


Figure 4.8: Dummy match-line memory scheme

In Figure 4.8 the OR gates embedded inside the MLSAs are not shown for the sake of clarity.

It can be noticed how all the sense amplifiers of the array are connected in parallel to the dummy MLSA: hence, the output stage of this has to be sized in order to properly drive all the other amplifiers.

To summarize, all the sense amplifiers are enabled using an external $\overline{\text{EN}}$ signal; then, as soon as the dummy match-line gets charged, the dummy MLSA switches its output from '0' to '1', and all the sense amplifiers are consequently disabled. Hence, it is evident that the time that it takes for the dummy match-line to get charged determines the time available to the other amplifiers for correctly sensing their inputs: for this reason, the array architecture has to be conceived so that the worst case search delay is associated to the dummy match-line position, in order for each amplifier in the array to be able to correctly evaluate the search operation result on its line.

4.5 Cell sizing

Consider the CAM NOR cell shown presented in Figure 4.1, which is reported in the following.

In a classical sensing scheme where the line is precharged to V_{DD}, the pull-down paths of the cells discharge the match-line in the mismatch case; hence, the sizing of the CAM transistors determines the sensing speed, since it controls the current absorbed by the cell and, so, the discharging speed of the match-line.

For the current-race scheme implemented this is not true: the line gets charged only in the match case, so what controls the sensing speed is the value of the current injected in the match-line, since it determines the slew-rate of the line voltage; hence, it seems

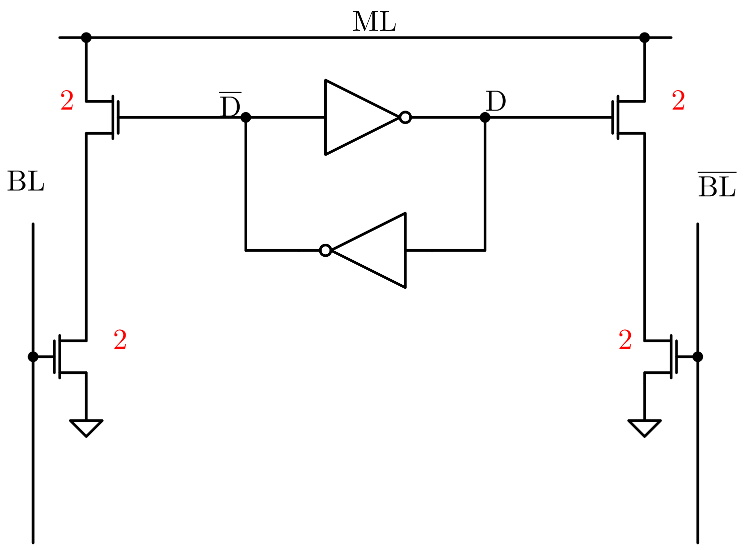


Figure 4.9: CAM cell implemented.

that the sensing speed does not depend on the cell transistor dimensions, which would suggest that the CAM transistors may be minimally sized in order to reduce the cell area. However, this is not true (or, at least, it is true only in part).

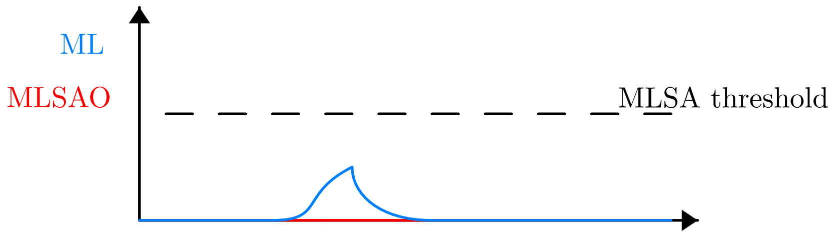


Figure 4.10

In Figure 4.10 it is shown what happens on the match-line during a mismatch. Since the pull-down paths of the cells behave as resistors, when a current is injected the line voltage arises: if the cells resistance is not too large, this voltage is limited and, if it does not cross the amplifier threshold, a correct mismatch result is registered, as shown in Figure 4.10; if, instead, the cells resistance is too large, the voltage on the line may cross the MLSA threshold and, so, lead to the incorrect evalutaion of a match result, as it is shown in Figure 4.11.

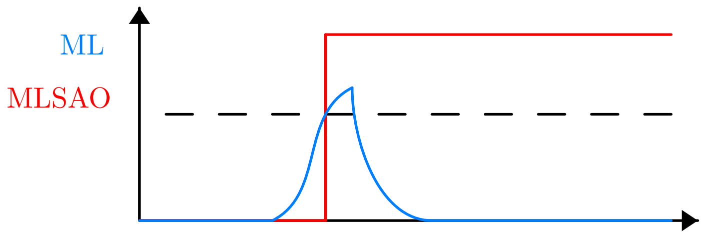


Figure 4.11

This phenomenon is particularly critical in the case in which all the cells provide a match result except one: in this condition, the cells resistance is maximized, since only one pull-down path in the whole line is enabled.

What determines the cell resistance are the dimensions of the pull-down transistors: hence, these cannot be minimally sized; however, they cannot be made as large as one wants, and the consequent cell area increase is not the only reason.

When a pull-down path is disabled, it does not behave as an ideal open circuit: it still conducts a current, even if this is very small. However, when the memory width is large (for example, in a range from 64 to 144 bits), there are many pull-down paths (two per cell) attached to the match-line; since these are in parallel, all the transistors subthreshold currents sum up.

Referring to the match-line model presented in Figure 2.24 and reported in the following, these cells can be viewed as an equivalent resistor, obtained from all the cell resistors in parallel. It has to be noted that the R_{cell} value in the match case in much larger than in the mismatch one, since it is referred to transistors that are turned off, while during a mismatch these are turned on.

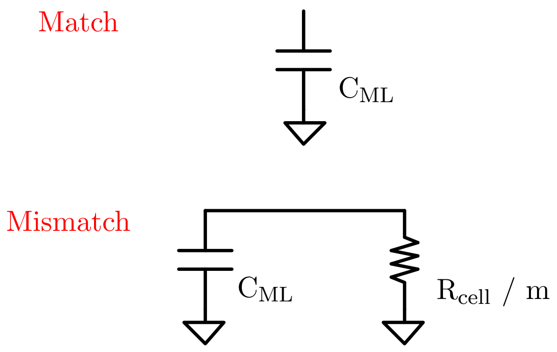


Figure 4.12: Matchline model

The R_{cell} of each cell, when this provides a match, is very large; however, the resulting equivalent resistance cannot be neglected: this leads to a non negligible current absorbed by the cells during a match that slows down the line charging, leading to a reduced sensing speed.

It can be noticed how the dummy match-line scheme allows also to limit the “voltage bump" shown in Figure 4.10: since the sensing interval is limited to a portion of the search operation cycle, the match-line voltage does not have enough time to "completely rise"; thus, the error probability in the search result evaluation is furtherly reduced thanks to this technique.

In conclusion:

- the cell transistors cannot be minimally sized, since this may lead to incorrect sensing during a search operation.
- their dimensions cannot be chosen too large for two reasons: cell area increase and match sensing speed reduction.

4.6 Testbench

The testbench has been organized in a way similar to the SRAM one described in section 3.5. For example, the simulation flow adopted is the same.

4.6.1 The testbench circuit

The testbench circuit is presented in Figure 4.13.

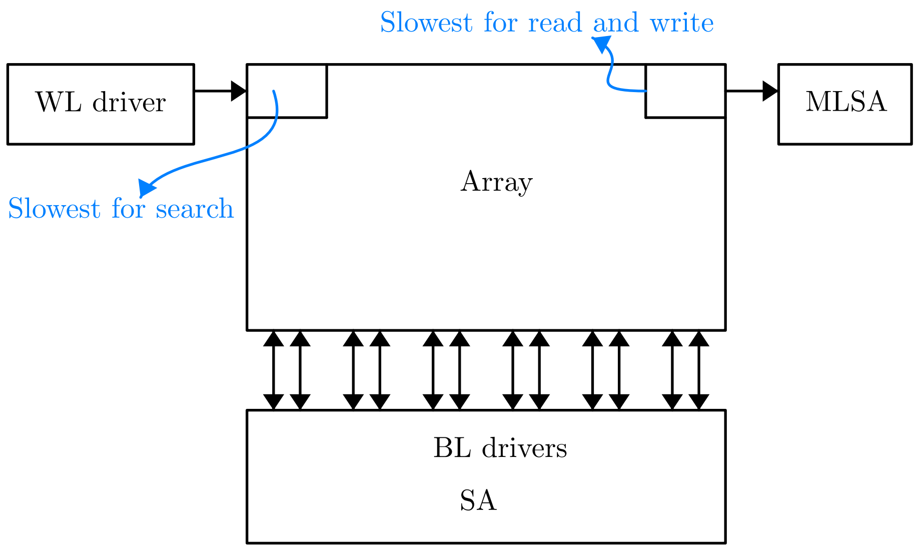


Figure 4.13: Testbench schematic

A different approach has been taken with respect to the SRAM case in order to perform all the operations on only one row: the bitline drivers have been moved to the bottom of the array, so that the worst cases for read and write operations could coincide. In fact, as discussed in subsection 3.5.5, in this way the cell placed on the last column and the first row, is the most distant one from both wordline and bitlines drivers and the sense amplifier, and so the largest write and read delays are associated to its position.

For what concern the search delay, the cell placed on the first column and first row has been chosen, since it is the most distant one from the match-line sense amplifier input. Since the delay of the search operation does not depend on the row position in the array, the first row has been used for the search testing, in order to perform all the memory operations on the same row.

4.6.2 Python code

The code adopted for the CAM simulation is very similar to the SRAM one presented in subsection 3.5.2. Of course, additional signals and operations are needed for the CAM testing.

```
# Signals of the testbench.
signals = { 'WLFirstRow': {'file_pointer': None, 'value': 0,
                                    'default_value': 0},
'BLFirstCol': {'file_pointer': None, 'value': Vdd, 'default_value': Vdd},
'BLnFirstCol': {'file_pointer': None, 'value': Vdd, 'default_value':
                                    Vdd},
'BLLastCol': {'file_pointer': None, 'value': Vdd, 'default_value': Vdd},
'BLnLastCol': {'file_pointer': None, 'value': Vdd, 'default_value': Vdd},
'Precharge_n': {'file_pointer': None, 'value': Vdd, 'default_value':
                                    Vdd},
'EnableSA': {'file_pointer': None, 'value': 0, 'default_value': 0},
'EnableBLDriver': {'file_pointer': None, 'value': 0, 'default_value':
                                    Vdd},
'EnableMLSAn': {'file_pointer': None, 'value': Vdd, 'default_value':
                                    Vdd},
'DisableFirstCol': {'file_pointer': None, 'value': 0, 'default_value': 0}
}
```

With respect to the code presented in subsection 3.5.2, it can be noticed how two different bitline signals are used: one for the first column (used to test the search delay and operation) and the other for the last column (read and write delays and operation).

Also, the MLSA signals have been added to the testbench with respect to the SRAM implementation. It has to be noticed that no precharge (actually, “pre-discharge” with this sensing scheme) signal is present: this is because the matchline is automatically discharged when the MLSA is not enabled and, so, there is no need for a dedicated precharge signal.

```
# Operations to simulate.
operations = ("Write_1", #0
"MLSAPredischarge_1", #1
"Search_1", #2
"MLSAPredischarge_0", #3
"Search_0", #4
"Idle", #5
"Write_0", #6
"Idle", #7
"MLSAPredischarge_1", #8
"Search_1", #9
"MLSAPredischarge_0", #10
"Search_0", #11
"Idle", #12
"TestWriteSpeed1", #13
"Precharge_cycle", #14
"Read", #15
"Idle", #16
"TestWriteSpeed0", #17
"Precharge_cycle", #18
"Read") #19
```

Here, the operations to be simulated are presented. First, in `#0`, a logic '1' is written to all the cells; then, all the cells except the one placed on the first column are disabled and a logic '1' is loaded on the first column bitlines, in order to test the match results (`#1` and `#2`); after this, a logic '0' is loaded to test the mismatch case. This sequence is repeated after writing a logic '0' to all the cells (`#6` to `#11`).

It has to be noticed that there are two kinds of pre-discharge operation: one for the logic '1' (`#1`), in which a logic '1' is put on the bitlines during the pre-discharge operation, and one for the logic '0'. This allows to take advantage of the sensing scheme independence from the bitlines configuration during the pre-discharge phase, as explained in section 4.3.

Starting from `#13`, the standard memory operations (read and write) and tested. First, a logic '1' is written to the cell on the last column and, then, the content of the latter is read; second, the a logic '0' is written and, then, read.

4.6.3 Cadence Virtuoso schematic

A schematic very similar to the one presented in Figure 3.16 has been derived for the CAM testing, using the same design flow; first, the cell and sense amplifier topologies have been produced in Virtuoso; second, a 8 bits column has been derived and used to obtain a 8x8 array; this, after being tested to tune the design characteristics, has been used to derive larger arrays (16x16 and 32x32).

4.7 Simulation waveforms

In Figure 4.14, a simulation run is shown. As described in subsection 4.6.2, first a logic '1' is written on the whole row (in the graph, the data of the first column cell, `DataFirstCol`, is considered); then, a logic '1' and a logic '0' are searched, leading to a match in the first case (`MLSAO` goes to '1') and to a mismatch in the second case (`MLSAO` stays at '0'). In the second part of the simulation, a logic '0' is written (`DataFirstCol` goes to '0') and the previous search pattern is performed: when a logic '1' is searched,

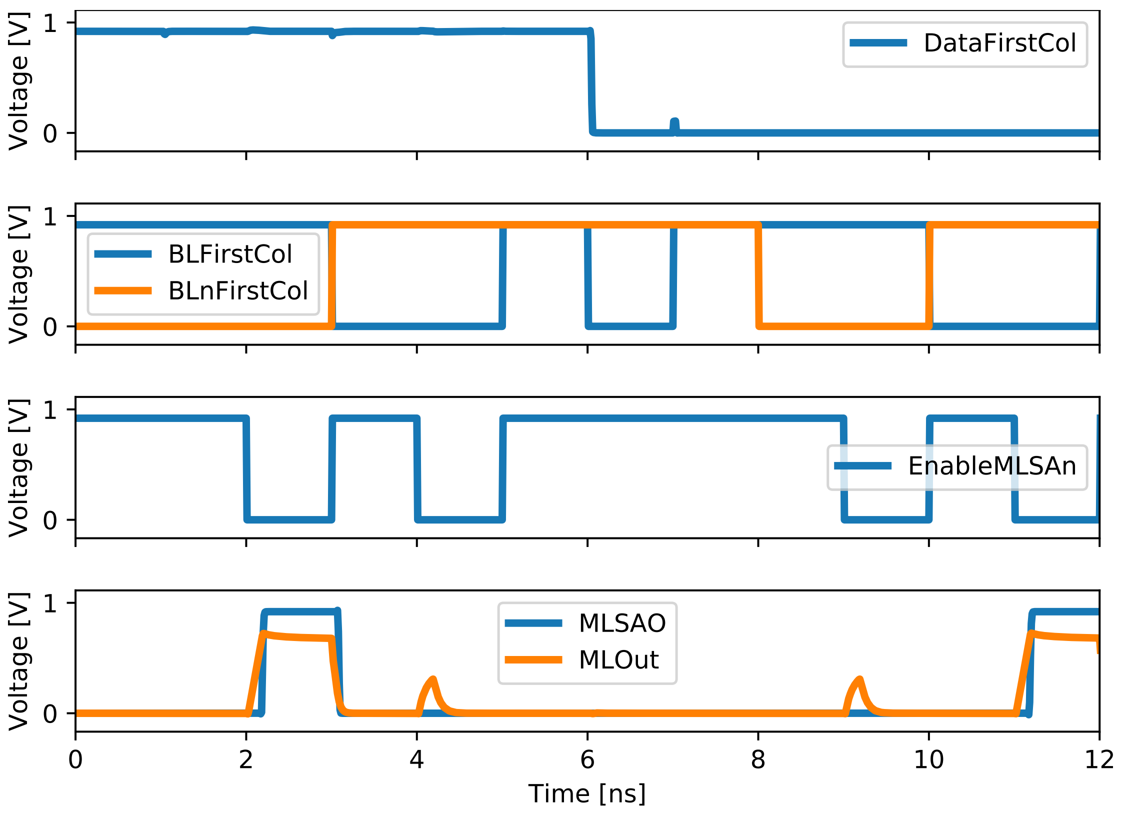


Figure 4.14: Simulation run.

a mismatch is provided; when a logic ‘0’ is searched, a match is provided.

It can be noticed how the match-line does not get fully charged. This is due to the dummy-line sensing scheme: as soon as the output of the dummy MLSA switches, all the other MLSAs are disabled and, so, the lines stop charging. This allows to furtherly reduce the energy involved in the sensing operation.

The simulation run has been carried out on a 32x32 array.

Chapter 5

LiM array design

In this chapter, a LiM cell design, based on the one proposed in [1], is presented. Three arrays have been produced, as in SRAM and CAM cases, whose sizes are 8x8, 16x16 and 32x32 bits.

The design has been carried out using the same procedure adopted in the precedent ones, and the result extracted are the same: energy consumption and delays for each memory operation.

5.1 The algorithm

In [1] a LiM architecture for the maximum/minimum value search in a memory array is proposed. The algorithm is based on the bitwise AND operation between the memory content (i.e. every word stored in memory) and an external datum called "mask vector", which is performed in parallel on all memory rows at the same time.

In a classical architecture, made by a processor that reads data from the memory and elaborates it, to find the maximum/minimum stored value one would need to read all the memory data and elaborate it inside the processing unit. This is, of course, an expensive operation from both time and energy points of view, since many clock cycles are needed to read each word and a large amount of energy is wasted on the busses between the memory and the CPU for the data transfer. For this reason, it would be more convenient to perform this operation completely in memory, as it is proposed in [1], so that no time and energy have to be spent to access the memory content.

In Figure 5.1, the steps needed to find the maximum value among unsigned binary data are shown. With `number` the memory datum is denoted, while with `mask` the mask vector is indicated.

At each step, the bitwise AND operation is performed between each memory datum and the mask vector, and the results of these operations are recorder by the logic that surrounds the array. The algorithm starts with the mask vector value "1000", that is used to check which rows have the MSB (Most Significant Bit) equal to '1': in fact, considering unsigned binary values, the largest data have the MSB equal to '1', while the others have a smaller encoded value.

At the beginning of the algorithm execution, all the words are listed in a search list (in particular, all the rows addresses are considered valid), since each datum is a maximum value candidate. The words in which the AND outputs are different from '1'

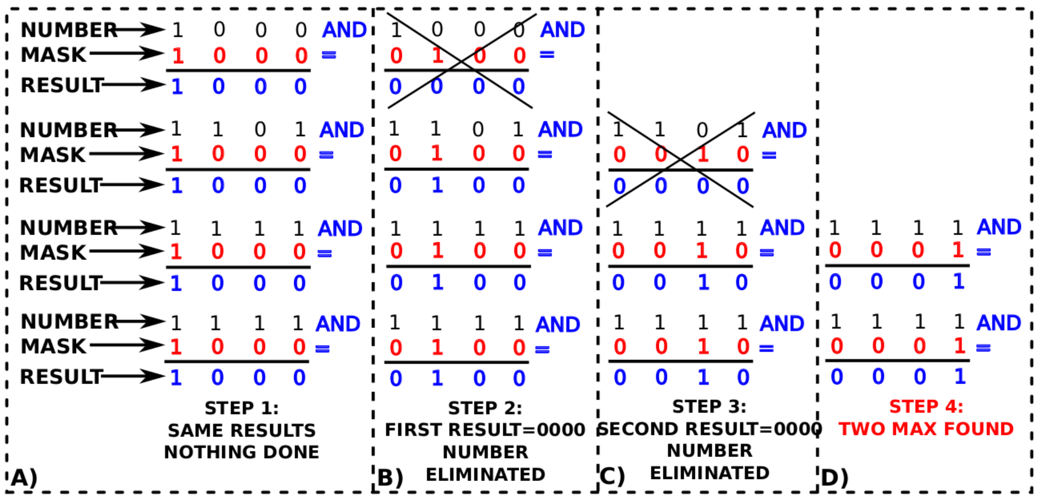


Figure 5.1: The algorithm [1].

are removed from the search list since they are no more maximum candidates, while the others are kept. In the first step shown in Figure 5.1 no word is removed from the search list since all the AND results are equal to '1' (only the column correspondent to the mask bit equal to '1' is considered).

In the second step, the same operation is performed changing the mask value: the "0100" value is used to check which words have the MSB-1 (the bit after the MSB) equal to '1'. As in the previous case, the words that "match" with the mask are kept in the search list, while the other ones are removed.

These steps are repeated until the LSB (Least Significant Bit) of the words is reached: at the end of this step, only one word will remain in the search list (unless there are two or more identical data stored in memory, as it happens in Figure 5.1), and the address of this is provided in output as the maximum value location.

It has to be highlighted that the number of steps needed to find the maximum stored value is equal to the array width, since one bit of every stored word is processed at time; hence, the search latency is independent of the number of stored data. In a processor architecture, instead, the number of data to be read has a strong influence of the search latency, since at least one clock cycle is needed to read each word and to store it in the processor registers.

5.2 The memory cell

The memory cell is a NOR CAM cell which has been modified to perform the bitwise AND operation. The cell logic scheme is shown in Figure 5.2.

In Figure 5.2, the `Cell` block denotes the memory core, which implements the CAM and SRAM functions and whose schematic has been presented in Figure 4.1. A logic AND gate is added to the cell and it is used to perform the AND operation between the

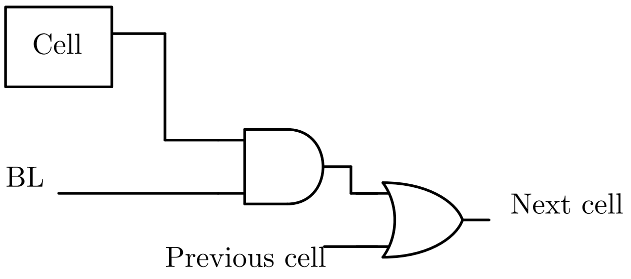


Figure 5.2: The memory cell (logic scheme).

cell content and the datum put on the bitline; the output of this is sent to an OR gate, which receives the other input from the OR of the previous cell on the row and sends its output to the next memory cell on the same row.

The array logic scheme is shown in Figure 5.3. In this, the cells disposition inside the architecture is shown: as explained before, each cell is arranged in a chain of OR gates, whose result is recorded by the logic that surrounds the memory array and used by this to update the search list during the maximum/minimum value search.

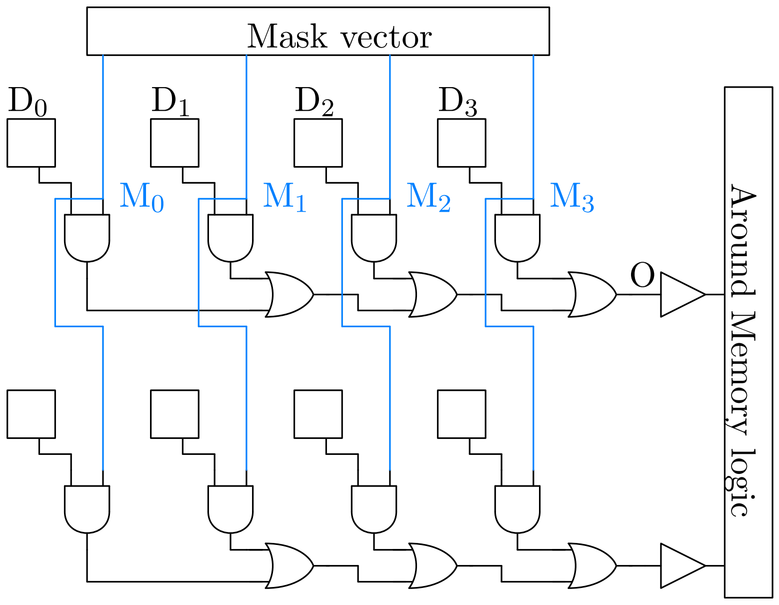


Figure 5.3: The array (logic scheme).

In the algorithm presented in section 5.1, a cell is selected by setting the corresponding mask bit to ‘1’, while the other cells are disabled (i.e. the corresponding AND gate output is forced to ‘0’) by setting the associated mask bits to ‘0’. This implies that the result of the OR chain depends only on the result of the AND between the selected cell content and the corresponding mask bit, which is set to ‘1’.

It may be better explained with an example. Consider the following: referring to a 4 bits word, the result at the end of the OR chain shown in Figure 5.3, denoted with O, is given by:

$$\mathrm{O} = \mathrm{D}_0 \cdot \mathrm{M}_0 + \mathrm{D}_1 \cdot \mathrm{M}_1 + \mathrm{D}_2 \cdot \mathrm{M}_2 + \mathrm{D}_3 \cdot \mathrm{M}_3$$

where with $\mathrm{D_i}$ and $\mathrm{M_i}$ the i-th cell content and i-th mask bit value, respectively, are denoted.

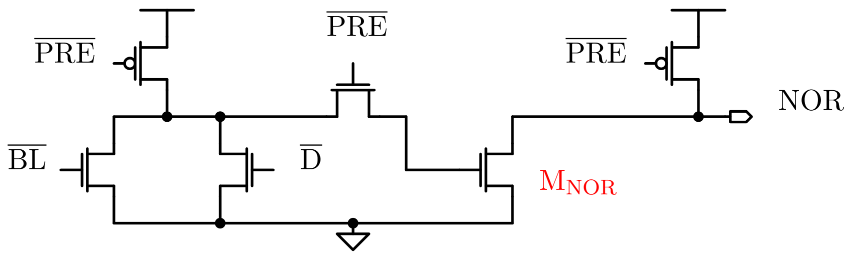
If the cell D_0 only is selected (i.e. M_0=‘1’, $\mathrm{M}_{1,2,3}$=‘0’), the equation can be rewritten in the following way:

$$\mathrm{O} = \mathrm{D}_0 \cdot 1 + \mathrm{D}_1 \cdot 0 + \mathrm{D}_2 \cdot 0 + \mathrm{D}_3 \cdot 0$$
$$\rightarrow \mathrm{O} = \mathrm{D}_0$$

Hence, the result depends exclusively on the selected cell content.

5.2.1 The proposed cell

The cell schematic proposed in [1] is shown in Figure 5.4.



$\overline{\mathrm{PRE}}$
$\overline{\mathrm{PRE}}$
$\overline{\mathrm{PRE}}$
NOR
$\overline{\mathrm{BL}}$
$\overline{\mathrm{D}}$
$\mathrm{M_{NOR}}$

Figure 5.4: The proposed cell

For the sake of clarity, the CAM and SRAM parts of the cell are omitted from the schematics, since the circuit shown in Figure 4.1 has not been modified.

This cell includes an AND gate implemented in dynamic CMOS logic. It has to be noticed that a non-inverting function is realized without using an output inverting stage for the gate, while in CMOS logic only inverting gates can be realized (NAND, NOR and so on). In this case, thanks to the fact that the negated cell content $\overline{\mathrm{D}}$ and the negated bitline $\overline{\mathrm{BL}}$ are available in the architecture, only one CMOS stage is needed to obtain a logic AND function.

Consider the AND part of the cell, shown in Figure 5.5.

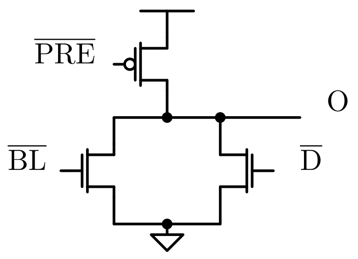


Figure 5.5: AND gate of the proposed cell.

The logic function implemented by the gate in Figure 5.5 is:

$$\text{O} = \overline{\overline{\text{BL}} + \overline{\text{D}}}$$

Using the Bool identities, the equation can be rewritten as:

$$\text{O} = \text{BL} \cdot \text{D}$$

Hence, a logic AND function between the cell content D and the bitline datum BL is obtained. The output of this is sent through a pass-transistor, enabled by deactivating the $\overline{\text{PRE}}$ signal, to the gate of the M_{NOR} transistor, which is connected to a line that links all the cells on the row, called "NOR line" (Figure 5.4).

The AND operation is separated in two phases:

- a precharge phase, in which the output of the AND gate and the NOR node are charged to '1'.
- an evaluation phase, in which the AND inputs are evaluated: if they are both equal to '1' (BL = D = '1' $\rightarrow$ $\overline{\text{BL}}$ = $\overline{\text{D}}$ ='0'), the AND output remains at '1' and M_{NOR} is enabled, thus discharging the NOR node; if note, the AND output is discharged to '0', turning off M_{NOR} and, so, preventing the discharging of the NOR node, which remains at the logic '1'.

The pass transistor shown in Figure 5.4 is needed to separate the output of the AND from the gate of M_{NOR} during the precharge phase, so that this is not enabled, thus allowing the NOR line to be charged.

The NOR line connects all the M_{NOR} transistors of the cells in a wired-OR configuration. Before the AND evaluation, the line is precharged to the logic '1'; then, if at least one cell in the row provides a '1' as result of the AND, the correspondent M_{NOR} is enabled and it discharges the line. In this way, adding an inverting stage at the end of the NOR line, a cascade of OR gates that connects all the cells is obtained, as it is shown in Figure 5.6.

It has to be noted that the PMOS transistor connected to the NOR node shown in Figure 5.4 is shared among the line and, so, it is not present inside the cell. It has been

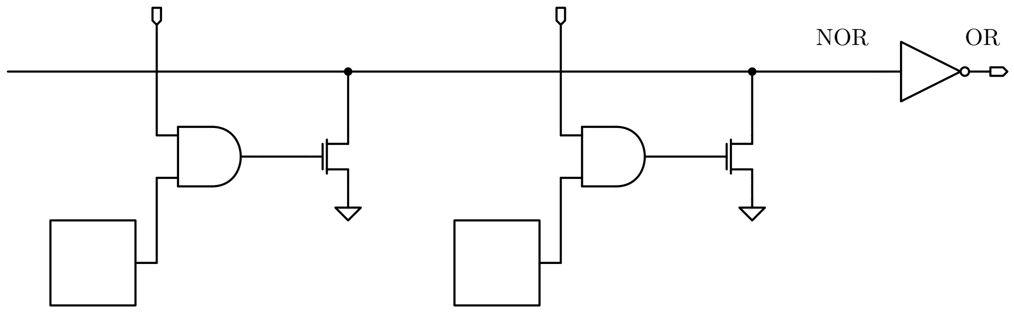


Figure 5.6: The NOR line.

included in the schematic only for the cell behavior explanation.

5.2.2 Adjustments to the original design

The scheme presented in Figure 5.4 has been used only to present the cell working principle; some modifications to the schematic are needed in order to make it working from a circuital point of view.

First of all, footer transistors (i.e. a transistor that is put between the ground pin of the logic gate/transistor and the actual ground to disable it when needed) have to be added to the cell, so that both the AND and NOR output can be charged to V_{DD} during the precharge phase. In fact:

- for the AND gate it would not be possible to precharge the AND node to the logic '1' since, if one between $\overline{\text{D}}$ and $\overline{\text{BL}}$ is equal to '1', one of the transistors would connect the AND node to ground and, so, it would not allow its potential to arise, consuming also energy because of the current flowing to ground.
- for the NOR gate, the pass-transistor connected to $\overline{\text{PRE}}$ does not ensure the disabling of M_{NOR} since it only connects the gate of the pull-down transistor to an undefined potential, which can be large enough to enable M_{NOR} and, so, to not allow the NOR node to be charged.

The schematic with the foot transistor is shown in Figure 5.7

When $\overline{\text{PRE}}$='0', the footer is disabled and the cell is in precharge phase; when $\overline{\text{PRE}}$='1', the footer is enabled and the cell is in evaluation phase.

Another problem relative to the pass-transistor in Figure 5.7 is the fact that it does not allow the gate voltage of M_{NOR} to be charged to V_{DD}, since it is a NMOS: this results in the reduced conductivity of M_{NOR} and, so, in its slower operation. Since a footer transistor has been introduced in the cell, the pass transistor is not needed anymore, since M_{NOR} is disabled thanks to the footer; hence, it can be removed, obtaining the design presented in Figure 5.8.

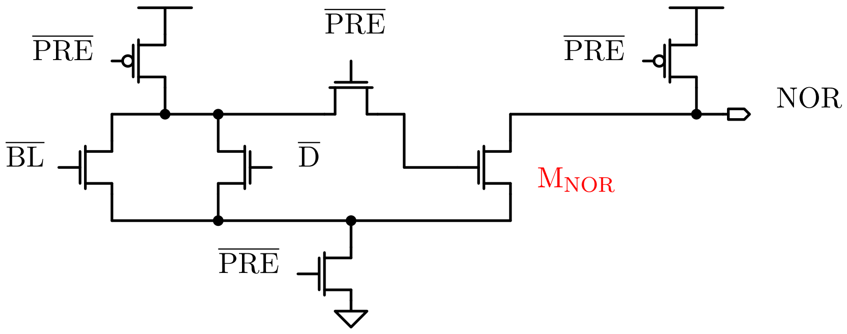


Figure 5.7: The modified cell.

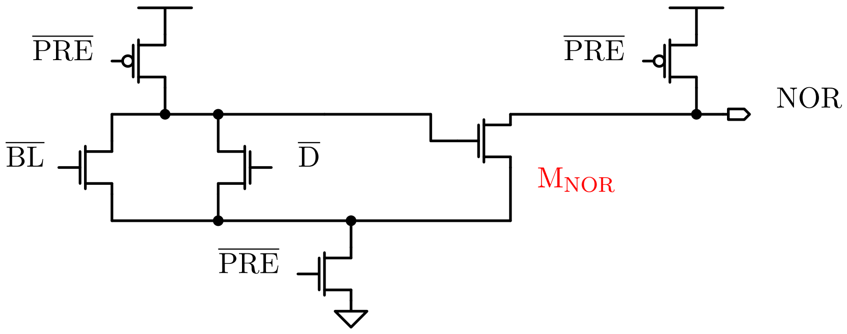


Figure 5.8: The final cell.

5.3 The sensing scheme

The sensing scheme adopted for the LiM array is the same of the CAM one, since it allows to reduce the cell area, as it will be described in the following, and to compare the CAM and LiM performance referring to the same sense amplifier.

Hence, a current race scheme is employed, together with a dummy-line sensing scheme to reduce the energy consumption associated to the LiM operations.

5.4 The implemented cell: dynamic AND version

The cell implemented is shown in Figure 5.9.

It can be noticed how the NOR transistor (i.e. the one connected to the NOR node) has no footer to disable it. This can represent an issue in a standard sensing scheme, since it would prevent the NOR line to be charged during the precharge phase. However, implementing a current-race scheme, the problem is solved: in fact, the line is pre-discharged instead of being pre-charged and, so, there is no need for the NOR transistor to be disabled during this phase. Hence, a transistor can be removed from the cell, leading to a reduction in the area occupation and, also, to an improvement in

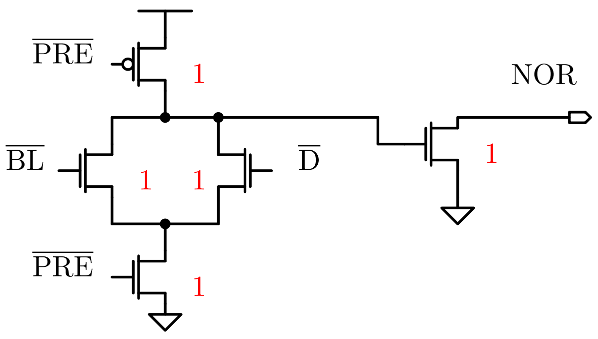


Figure 5.9: The implemented cell

the sensing performance.

To summarize: during the precharge phase ($\overline{\text{PRE}}$ = '0'), the PMOS transistor is enabled and charges to V_{DD} the AND node, while the footer transistor is disabled and prevents the AND node to be connected to ground; during the evaluation phase ($\overline{\text{PRE}}$ = '1'), the PMOS is disabled while the footer is enabled, so that the the AND node can be potentially discharged according to the inputs evaluation and, so, the NOR transistor is potentially disabled.

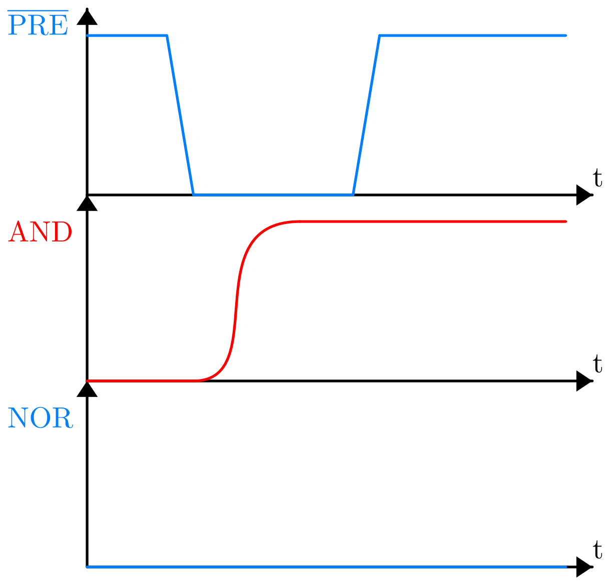


Figure 5.10: Dynamic cell behavior: AND='1' case.

In Figure 5.10 an AND operation in which the result is '1' is presented.

During the precharge phase ($\overline{\text{PRE}}$='0'), the AND output is precharged to '1', while the NOR output is predischarged to '0' (current-race scheme); during the evaluation phase ($\overline{\text{PRE}}$='1'), the AND output remains charged to '1' and, so, the NOR transistor is enabled: the NOR output remains at the logic '0'.

In Figure 5.11, instead, an AND operation in which the result is '0' is presented.

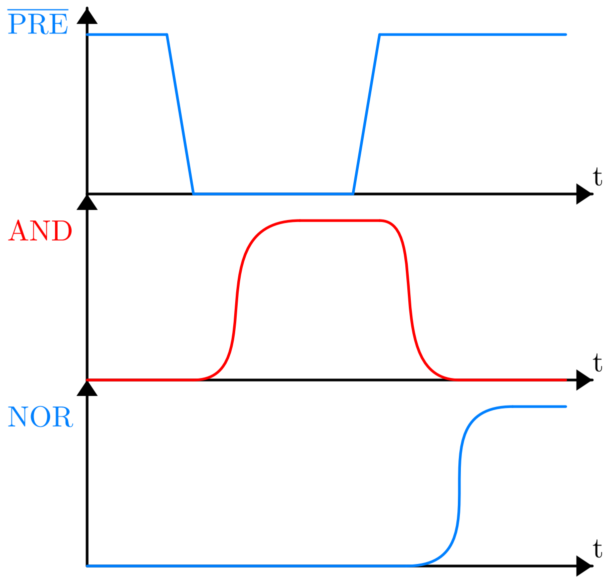


Figure 5.11: Dynamic cell behavior: AND=‘0’ case.

Here, during the evaluation phase ($\overline{\text{PRE}}$=‘1’), the AND output is discharged to ‘0’ and, so, the NOR transistor is disabled: hence, the NOR output is charged to the logic ‘1’.

Cell behavior					
D	**BL**	$\overline{\text{D}}$	$\overline{\text{BL}}$	**AND**	**NOR**
0	0	1	1	$1 \rightarrow 0$	$0 \rightarrow 1$
0	1	1	0	$1 \rightarrow 0$	$0 \rightarrow 1$
1	0	0	1	$1 \rightarrow 0$	$0 \rightarrow 1$
1	1	0	0	1	0

Table 5.1: Dynamic cell behavior.

In Table 5.1 the cell behavior as function of the inputs is reported.

5.5 The static cell

A CMOS static AND version of the cell presented in Figure 5.9 has been realized, in order to compare it with the dynamic one. The schematic is shown in Figure 5.12.

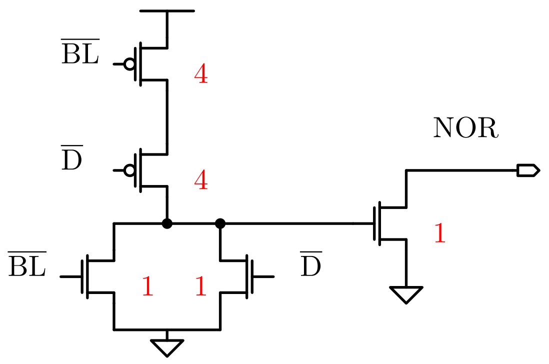


Figure 5.12: The static AND implementation.

In Figure 5.12 the transistors have been sized using the standard CMOS design rules for static logic gates: since the two PMOS transistors are in series, an aspect ratio equal to 4 (i.e. 4 times larger than the minimum aspect ratio) has been chosen instead of 2; for what concerns the NMOS transistors, since only one MOSFET for pull-down path is present, a minimum aspect ratio has been chosen.

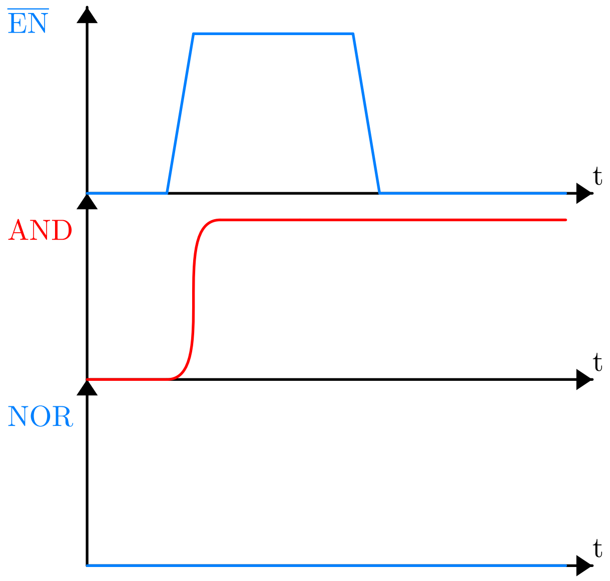


Figure 5.13: Static cell behavior: AND='1' case.

In Figure 5.13 an AND operation in which the result is '1' is presented.

Here the enable signal of the sense amplifier $\overline{\text{EN}}$ is reported, since no precharge phase is needed for the cell. When the enable signal is not active ($\overline{\text{EN}}$='1'), the NOR line is predischarged; subsequently, the sense amplifier is enabled ($\overline{\text{EN}}$='0'), the AND output is evaluated. Since AND='1', the NOR transistor is enabled and the NOR line does not get charged.

It has to be noticed that during the predischarge phase (i.e. while $\overline{\text{EN}}$='1') the AND inputs are evaluated and, in this case, the output changes from '0' to '1'. In fact, as discussed in section 4.3, the inputs of the operation are provided already in the predischarge phase, so that the bitlines drivers speed does not influence the sensing performance.

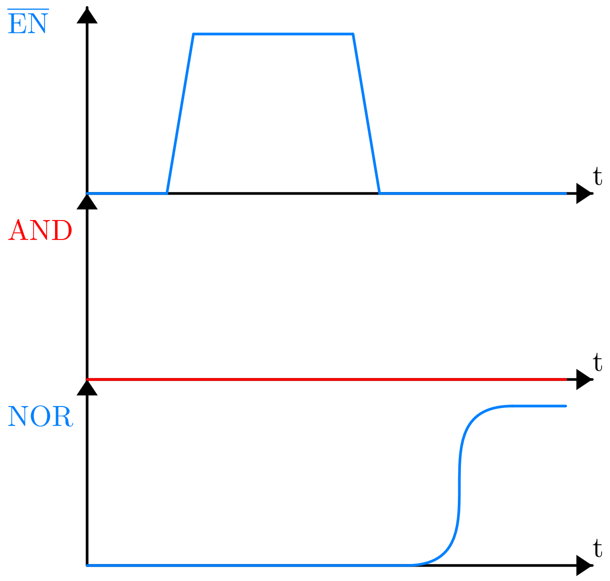


Figure 5.14: Static cell behavior: AND='0' case.

In Figure 5.14, instead, an AND operation in which the result is '0' is presented. Here AND='0' and, so, the NOR transistor is disabled, allowing the line to be charged to '1'.

Cell behavior					
D	**BL**	$\overline{\text{D}}$	$\overline{\text{BL}}$	**AND**	**NOR**
0	0	1	1	0	$0 \rightarrow 1$
0	1	1	0	0	$0 \rightarrow 1$
1	0	0	1	0	$0 \rightarrow 1$
1	1	0	0	1	0

Table 5.2: Static cell behavior.

In Table 5.2 the cell behavior as function of the inputs is reported. With respect to Table 5.1, it can be notice how the AND output is static, differently from what happens in the dynamic cell. This is because, in this case, a static CMOS gate is implemented instead of a dynamic one.

5.6 The special purpose cell

In section 5.2 it has been explained how the AND operation is used to select the cell whose content has to be checked: the AND output of the unselected cells is forced to '0' by providing a mask bit equal to '0', while the selected cell is given in input a mask bit equal to ‘1’; hence, to implement a full AND gate inside the cell seems to be needless for this particular algorithm (presented in section 5.1). For this reason, one can think of using a logic gate that allows only to select the cell on the row and to check its content, without performing a full AND logic operation with a dedicated gate. This has been done in the cell shown in Figure 5.15.

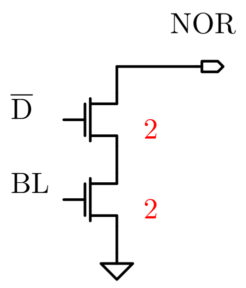


Figure 5.15: The special-purpose cell.

In this implementation, the cell is selected by setting BL=‘1’, while it is unselected by setting BL=‘0’. In this way, the unselected cells on the row are unable to discharge the NOR line, since their footer transistor is disabled, and the result of the operation is completely determined by the selected cell content.

Referring to the current-race sensing scheme, the content-check operation (called AND operation in analogy with the previous cells) is performed in the following way:

- the cell stores a logic ‘1’ ($\overline{\text{D}}$=‘0’): the pull-down path of the cell is disabled and the NOR line is charged to V_{DD}: a logic ‘1’ is registered as result.
- the cell stores a logic ‘0’ ($\overline{\text{D}}$=‘1’): the pull-down path is enabled and the NOR line is not charged: a logic ‘0’ is registered as result.

From the previous observations it may be deduced that the NOR line actually behaves like an OR line: when the cell content is equal to ‘0’, the line remains at ‘0’; when the cell content is equal to ‘1’, the line is charged to ‘1’. Hence, there is no need for the inverting stage at the end of the row shown in Figure 5.6. However, the inverting stage is not needed in any case: registering a NOR or an OR result makes no difference as long as the logic around the array, that gets the results and handles the maximum search list, knows which kind of result is provided (i.e. if the result is a NOR, it will be potentially inverted inside the logic block).

In Figure 5.16 the waveforms of an AND operation in which the result is ‘1’, are shown.

Differently from the static and dynamic cells cases, no intermediate AND output is reported, since the cell is made by a single stage; instead, the cell inputs, the negated cell content $\overline{\text{D}}$ and the bitline value BL, are shown. Hence, the result of the AND to be considered is the one of the logic operation D · BL.

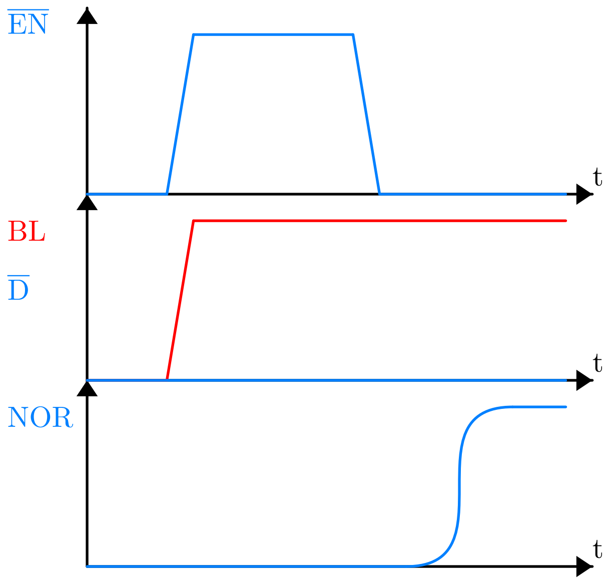


Figure 5.16: Special-purpose cell behavior: AND=‘1’ case.

The cell is selected at the beginning of the predischarge phase by setting BL=‘1’. In the case reported in Figure 5.16, the cell content D is equal to ‘1’ (hence $\overline{\text{D}}$=‘0’) and, so, the NOR line is charged to ‘1’. As stated before, on the NOR line the actual AND result (i.e. D · BL) is reported and, so, this behaves as an OR line, in practice.

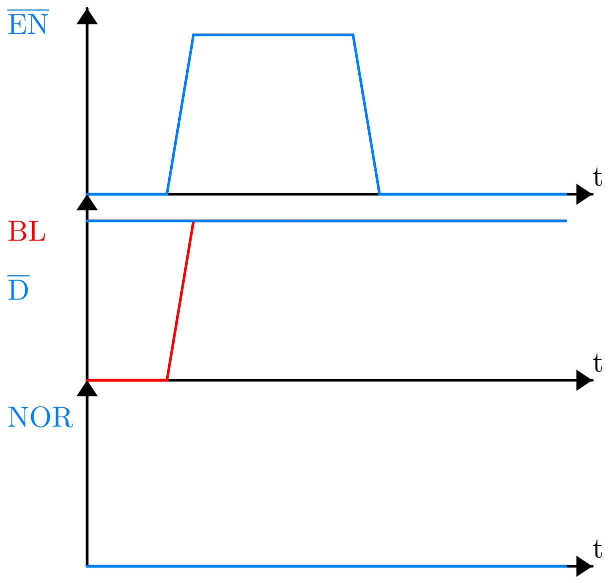


Figure 5.17: Special-purpose cell behavior: AND=‘0’ case.

In Figure 5.17, instead, the cell content is equal to ‘0’ (hence $\overline{\text{D}}$=‘1’). Hence, the pull-down path is enabled and the NOR line is not charged.

In Table 5.3 the cell behavior as function of the inputs is reported. It can be noticed how the cell content is evaluated only when BL=‘1’, otherwise the pull-down path of

Cell behavior				
D	**BL**	$\overline{\text{D}}$	D · BL	**NOR**
0	1	1	0	0
1	1	1	1	$0 \rightarrow 1$
-	0	-	-	$0 \rightarrow 1$

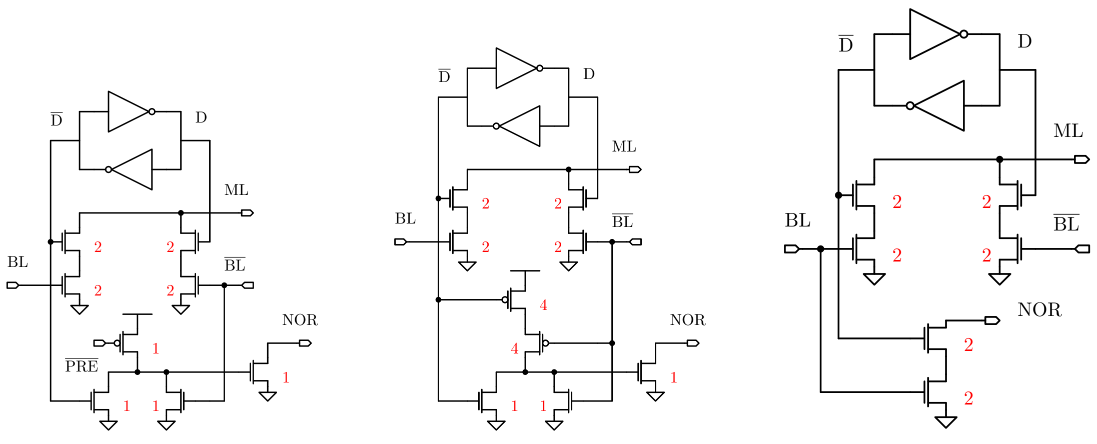
Table 5.3: Special-purpose cell behavior.

the cell is always disabled and, so, the NOR line is always charged.

When BL=‘1’, the result of the AND operation (in practice, the cell content since BL=‘1’) is reported in output

5.7 Comparison between the cells

In the following, the complete cell designs are reported.



(a) Dynamic cell. (b) Static cell. (c) Special-purpose cell.

Figure 5.18: The complete cells.

Transistors area		
$A_{dynamic}$	A_{static}	A_{SP}
$5 \cdot A_{min}$	$11 \cdot A_{min}$	$4 \cdot A_{min}$

(a) Transistors area for each cell.

Number of transistors per cell		
Dynamic	Static	Special-purpose
4	5	2

(b) Number of transistors in each cell.

Table 5.4: Comparison of the cells.

To determine the area of each cell, the area occupied by each transistor has been considered. The area of a minimum transistor (i.e. with aspect ratio equal to 1) is referred to as A_{min}:

$$A_{min} = W \cdot L$$

where W and L are the minimum transistor width and length.

Since the channel length L is fixed for every transistor, the increase in the area with respect to A_{min} is equal to the increase in the width W: for example, a transistor with aspect ratio equal to 2 (i.e. with a width twice as larger than the minimum one) is characterized by an area equal to $2 \cdot A_{min}$.

For each transistor, the area is computed in this way and the areas are summed up to estimate the cell one. The results are reported in Table 5.4a. For example, the static cell area is obtained in the following way:

$$A_{static} = (4 + 4 + 1 + 1 + 1) \cdot A_{min}$$
$$A_{static} = 11 \cdot A_{min}$$

In Table 5.4a the transistor areas of the three proposed cells are compared; only the transistors associated to the LiM functionality of the cells are considered, since the SRAM and CAM core is the same for the three cells.

It can be noticed how the cell area is larger in the static implementation than in the dynamic one, as it happens in standard logic gates. However, the dynamic implementation leads to increased dynamic power consumption, since dynamic logic is characterized by a switching activity larger than the static one, and it requires an additional signal $\overline{\text{PRE}}$ to be distributed on each row of the array to the cells.

As expected, the special-purpose cell is the one with the lower area, even if with a small margin with respect the dynamic one; however, one has to take into account that the number of transistors in the dynamic cell is doubled with respect the special-purpose one (Table 5.4b) and, so, the interconnections overhead probably leads to a larger area occupation for the dynamic cell.

Another characteristic that differentiates these cells from each other is their capacitive load on the bitlines: the more complex the cell is, the larger the number of transistors connected to the bitlines is and, so, the slower the bitlines operation will be. In fact, the larger the bitlines capacitance is, the larger will be the delay associated to write and read operations.

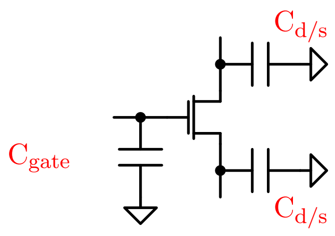


Figure 5.19: Parasitic capacitances of a MOSFET

The parasitic load associated to the cell can be taken into account in a numerical way considering the parasitic capacitances of the transistors connected to each bitline.

Defining with $C_{d/s}$ the drain/source capacitance and with C_{gate} the gate capacitance of a minimally sized transistor, these parameters can be easily estimated for an arbitrary sized MOSFET.

For simplicity, no difference is made for PMOSFETs and NMOSFETs in the parasitic approximation, since this is a very approximated model. These parasitics are shown in Figure 5.19.

In non-minimum transistors, the parasitic capacitances values have to be multiplied by the aspect ratio, since the capacitance is directly proportional to the transistor width (in particular, to its area that, since all the transistors are characterized by the same length, coincides with the width). For example, the parasitic capacitances of a transistor with aspect ratio equal to 2 are given by $2 \cdot C_{s/d}$ and $2 \cdot C_{gate}$. Following this principle, a the parasitic capacitances on the bitlines for each cell can be derived.

Referring to the cells schematics shown in Figure 5.21, the parasitic capacitance associated to each bitline have been derived and they are shown in Table 5.5. A comparison is made between the values in Figure 5.20, in which the capacitance has been normalized with respect to C_{gate}, assuming $C_{d/s} = \frac{C_{gate}}{3}$, which is an usual approximation.

Parasitic capacitances		
Cell	**BL**	$\overline{\textbf{BL}}$
SRAM	$2 \cdot C_{s/d}$	$2 \cdot C_{s/d}$
CAM	$2 \cdot C_{s/d} + 2 \cdot C_{gate}$	$2 \cdot C_{s/d} + 2 \cdot C_{gate}$
Dynamic	$2 \cdot C_{s/d} + 2 \cdot C_{gate}$	$2 \cdot C_{s/d} + \mathbf{3} \cdot C_{gate}$
Static	$2 \cdot C_{s/d} + 2 \cdot C_{gate}$	$2 \cdot C_{s/d} + \mathbf{7} \cdot C_{gate}$
SP	$2 \cdot C_{s/d} + \mathbf{4} \cdot C_{gate}$	$2 \cdot C_{s/d} + 2 \cdot C_{gate}$

Table 5.5: Parasitic capacitance comparison among the cells.

In Figure 5.20 it can be noticed how the dynamic (`DYN`) and static (`STAT`) cells have the same influence on the bitline BL as the CAM one: this is due to the fact that all the AND transistors are connected to $\overline{\text{BL}}$, as it can be observed in the dedicated column.

As expected, the most complex cell, the static one, is also the one with the larger parasitic capacitance associated. The most simple cell, the special-purpose (`SP`), is characterized by a bitline capacitance larger than the dynamic one: this is due to the fact that the special-purpose cell has a MOSFET with aspect ratio equal to 2 connected to BL (Figure 5.18), while the dynamic cell has a transistor with aspect ratio 1 connected to $\overline{\text{BL}}$.

It can be noticed how the static implementation has a larger capacitive load with respect the dynamic one, since it has more and larger transistors connected to $\overline{\text{BL}}$: while in the static cell two transistors are connected to $\overline{\text{BL}}$, one of which is sized with an aspect ratio equal to 4, in the dynamic cell only one minimum transistor is present.

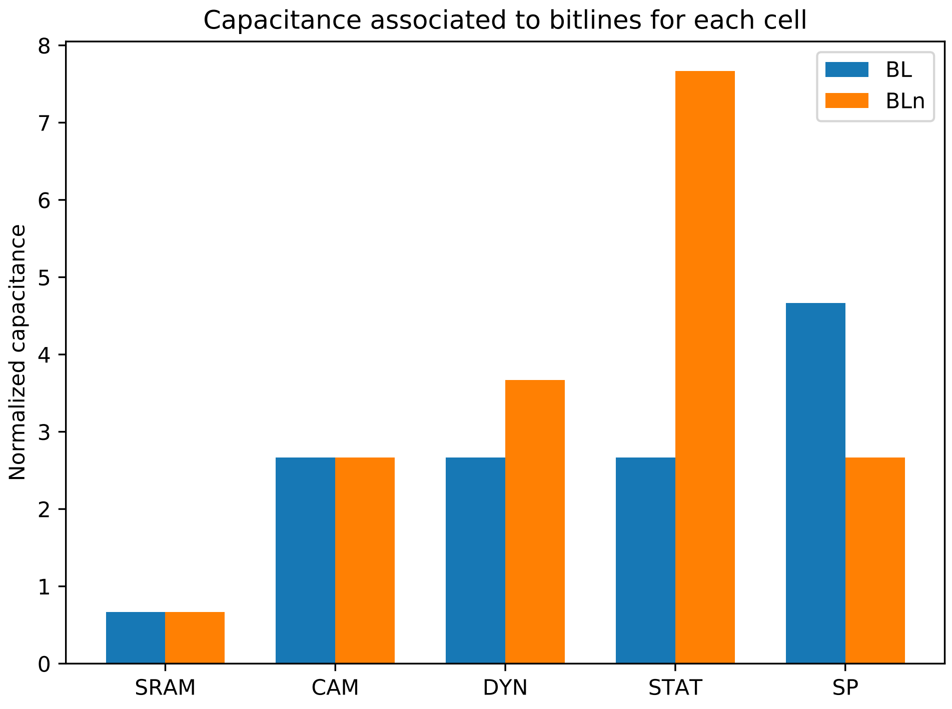


Figure 5.20: Parasitic capacitance comparison among the cells.

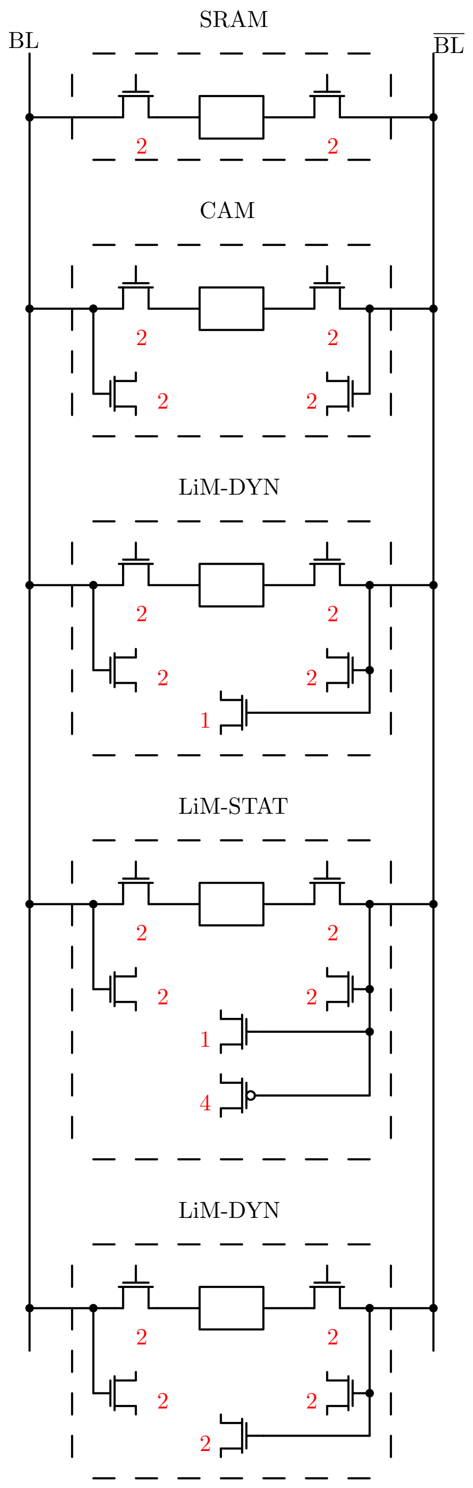


Figure 5.21: Capacitive load on bitlines of the cells.

5.8 Testbench

The testbench has been derived in a way similar to the CAM architecture, as it will be described in the following.

5.8.1 Testbench circuit

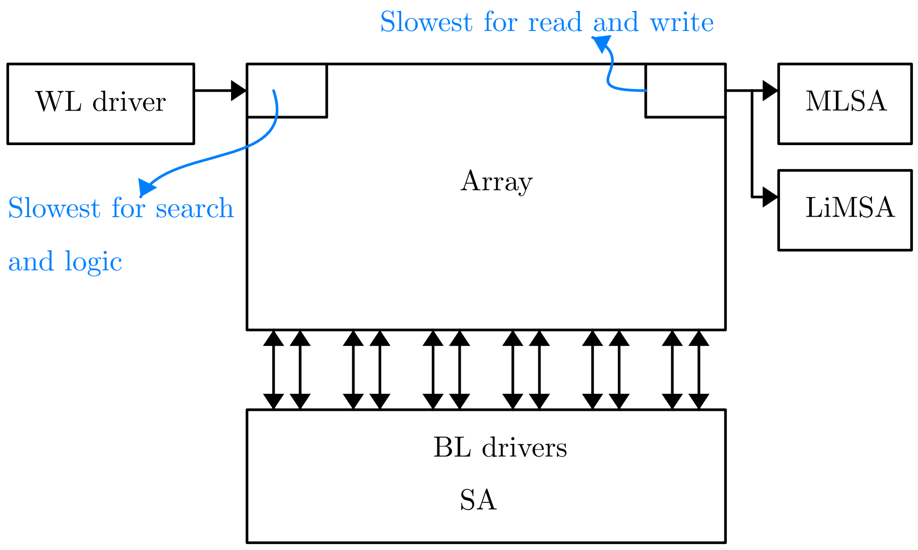


Figure 5.22: The testbench circuit.

In Figure 5.22 the testbench circuit is shown. As in the CAM architecture, the first cell on the first row and column is used to test the logic operations performance, since it is the most distant one from the sense amplifier input, which is put on the right of the array.

The rest of the driver and sensing circuits are arranged as in the CAM architecture.

5.8.2 The sensing scheme

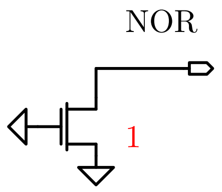
As explained in section 5.3, the same sensing scheme of the CAM architecture has been adopted, using dummy lines to limit the energy consumption of the memory operations.

For each cell, the correspondent dummy cell has been derived and arranged in a dummy line. The resulting cell schematics are shown in Figure 5.23.

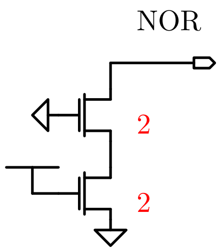
Since the sense amplifier topology implemented for each cell is the same, the same dummy-line scheme as been adopted, following the one presented in section 4.4.

An unique dummy row is used for both CAM and LiM sensing. Considering the case of the dynamic cell, the dummy cell of this line is shown in Figure 5.24.

In each cell of the dummy line, there is a part dedicated to the LiM sensing and a part dedicated to the CAM sensing, as it happen in the array cells. In this way, only one additional row in the array is needed, resulting in a reduced area overhead.



(a) Dynamic and static dummy cell.



(b) Special-purpose dummy cell.

Figure 5.23: The dummy cells.

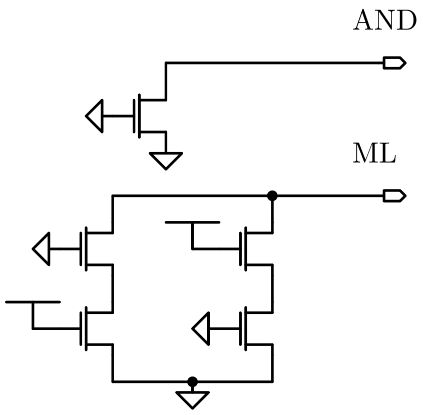


Figure 5.24: Cell of the dummy line.

For what concerns the sense amplifier, the same topology adopted in Figure 4.2 is used. Since this is a non inverting topology, the signal sensed on the line is simply amplified, without being inverted.

The adoption of the same sense amplifier topology for all the LiM cells leads to a different behavior between the dynamic and static cells, and the special-purpose one, as it is explained in the following.

For simplicity, the line to which the cells on a row are connected will be called “AND line” from now on.

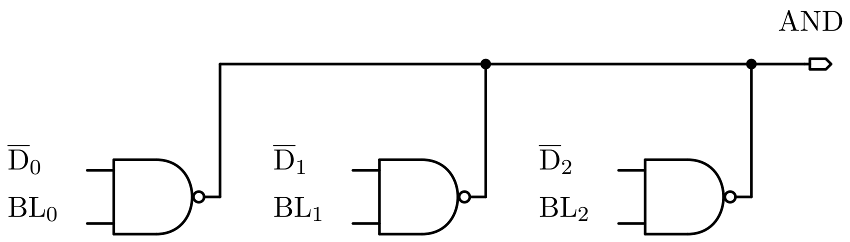


Figure 5.25: The AND line with the special-purpose cell.

In Figure 5.25 an AND line with special-purpose cell logic models is shown. The NAND gate reported is a dynamic gate: when its output is equal to ‘1’, it behaves like a tristate driver and, so, its output is floating, allowing the line to remain charged; when its output is equal to ‘0’, it discharges the line.

The equivalent line equation is the following

$$\mathrm{AND} = \overline{(\overline{\mathrm{D}_0} \cdot \mathrm{BL}_1)} \cdot \overline{(\overline{\mathrm{D}_1} \cdot \mathrm{BL}_1)} \cdot \overline{(\overline{\mathrm{D}_2} \cdot \mathrm{BL}_2)}$$

From the equation it can be deduced that as soon as one of the NAND outputs is equal to ‘0’, the line is brought to ‘0’, and only in the case in which all three of the outputs are equal to ‘1’, the line remains charged.

To select a cell, one has to set BL=‘1’, while to disable it one forces BL=‘0’. In fact, the output of a NAND is equal to ‘1’ if at least one of its inputs is equal to ‘0’; hence, to disable a dynamic NAND gate it is enough to set one of its inputs (in this case BL) to ‘0’.

Consider the case in which only the cell $\overline{\mathrm{D}_0}$ is selected (hence BL_0=‘1’ and $\mathrm{BL}_{1,2}$=‘0’). A two-inputs NAND gate with one of the inputs fixed to ‘1’ behaves as a logic inverter: hence, on its output and, so, on the AND line the cell content D_0 is reported; hence, the result on the AND line coincides with the result of the cell internal AND operation.

From a logic point of view, it happens the following thing:

$$\begin{gathered}\mathrm{AND} = \overline{(\overline{\mathrm{D}_0} \cdot 1)} \cdot \overline{(\overline{\mathrm{D}_1} \cdot 0)} \cdot \overline{(\overline{\mathrm{D}_2} \cdot 0)} \\ \rightarrow \mathrm{AND} = \overline{\overline{\mathrm{D}_0}} \cdot \overline{0} \cdot \overline{0} \\ \rightarrow \mathrm{AND} = \mathrm{D}_0 \cdot 1 \cdot 1 \\ \rightarrow \mathrm{AND} = \mathrm{D}_0\end{gathered}$$

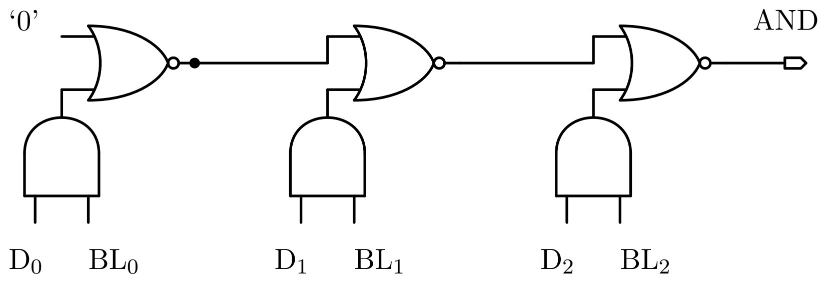


Figure 5.26: The AND line with the dynamic and static cells.

In Figure 5.26 the same line but with the dynamic and static cells models is reported. In this case, all the gates behaves like standard static ones.

As in the previous case, a cell in unselected by setting BL=‘0’: in this way, the corresponding AND output is set to ‘0’ and, so, the associated NOR result depends only on the other input. In fact, a NOR gate with an input equal to ‘0’ behaves like a logic inverter.

The first cell NOR has a ‘0’ on the other input because there is no other cell before it. This situation can be represented by setting the correspondent input to ‘0’, so that the output of the NOR is not determined by it.

The line equivalent equation is the following:

$$\mathrm{AND} = \overline{0 + \mathrm{D}_0 \cdot \mathrm{BL}_0 + \mathrm{D}_1 \cdot \mathrm{BL}_1 + \mathrm{D}_2 \cdot \mathrm{BL}_2}$$

Considering the same case as before (BL_0=‘1’ and $\mathrm{BL}_{1,2}$=‘0’), the AND value depends only on the value of D_0, since all the other cells are disabled. In particular, the inverted D_0 is reported on the line, as it can be deduced by analyzing the line equation:

$$\mathrm{AND} = \overline{0 + \mathrm{D}_0 \cdot 1 + \mathrm{D}_1 \cdot 0 + \mathrm{D}_2 \cdot 0}$$
$$\rightarrow \mathrm{AND} = \overline{0 + \mathrm{D}_0 + 0 + 0}$$
$$\rightarrow \mathrm{AND} = \overline{\mathrm{D}_0}$$

In conclusion: in the special-purpose cell case, the cell content is reported on the AND line; in the static and dynamic cells cases, the inverted cell content is reported.

5.8.3 Python code

For each of the cell types, a specific Python code has been developed for the simulations, which are presented in the following.

Dynamic cell

```
'EnableANDSAn': {'file_pointer': None, 'value': Vdd, 'default_value':
                                          Vdd},
```

```
'PrechargeAND_n': {'file_pointer': None, 'value': Vdd, 'default_value':
                                          Vdd}
```

Here the signals of the testbench are presented. For the sake of clarity, only the new ones are shown.

With respect to the CAM architecture, two input signals have been introduced: `EnableANDSAn`, used to enable the NOR line sense amplifier; `PrechargeAND_n`, used to precharge the dynamic AND gates outputs before the AND evaluation.

```
# Operations to simulate.
operations = ("Write_1", #0
"MLSAPredischarge_1", #1
"Search_1", #2
"MLSAPredischarge_0", #3
"Search_0", #4
"ANDPredischarge", #5
"AND", #6
"ANDPredischarge_disable", #7
"AND_disable", #8
"Idle", #9
"Write_0", #10
"Idle", #11
"MLSAPredischarge_1", #12
"Search_1", #13
"MLSAPredischarge_0", #14
"Search_0", #15
"ANDPredischarge", #16
"AND", #17
"ANDPredischarge_disable", #18
"AND_disable", #19
"Idle", #20
"TestWriteSpeed1", #21
"Precharge_cycle", #22
"Read", #23
"Idle", #24
"TestWriteSpeed0", #25
"Precharge_cycle", #26
"Read") #27
```

Here the operations simulated are shown. First, the cells are written to logic '1' (`#0`); second, the CAM functionality is tested (`#1` to `#4`); third, the LiM functionality is tested (`#5` to `#8`). In the LiM operations testing, two operations are used: `AND` and `AND_disabled`.

```
    elif operation == "AND":
        signals['BLnFirstCol']['value'] = 0
        signals['EnableANDSAn']['value'] = 0
```

In `AND`, the proper AND operation is tested: the mask bit of the selected cell is set to '1' while all the other bits are set to '0'. In the code presented above, the $\overline{\text{BL}}$ is set to '0', while all the other bitlines (BL in first column and both bitlines in the last column) are set to '1' (default value for the bitlines). In fact, only the bitline $\overline{\text{BL}}$ controls the AND operation in the dynamic and static cells and, so, there is no need to change the value of both bitlines.

```
    elif operation == "AND_disable":
        signals['BLnFirstCol']['value'] = Vdd
        signals['EnableANDSAn']['value'] = 0
```

In `AND_disabled`, instead, all the mask bits are set to '0' in order to check the proper behavior of the LiM logic and sensing scheme. In the code above the $\overline{\text{BL}}$ is set to '1' in order to disable the cell. This is made explicitly in the code even if not needed only for the sake of clarity: in fact, the bitlines default value is '1'.

This set operation is tested again after having written a '0' to the cells (`#10` to `#19`);

after this, the read and write operations performance are tested (#21 to #27).

Static cell

With respect to the dynamic cell test code, only the PrechargeAND_n signal is omitted, since no cell internal precharge is needed in the static implementation. The rest of the code, however, is identical.

Special-purpose cell

With respect the dynamic cell code, also here the PrechargeAND_n signal is omitted for the same reason explained before, and some changes are made to the AND and AND_disable operations.

```
    elif operation == "AND":
        signals['BLFirstCol']['value'] = Vdd
        signals['BLLastCol']['value'] = 0
        signals['EnableANDSAn']['value'] = 0
```

In the AND operation, since in the special-purpose cell the AND is connected to the bitline BL (Figure 5.15), the value of this has to be modified; in particular, its active value for the AND operation is equal to '1' and, so, this has to be set on the first column. The other cells, instead, are deselected by forcing their BL to '0'.

```
    elif operation == "AND_disable":
        signals['BLFirstCol']['value'] = 0
        signals['BLLastCol']['value'] = 0
        signals['EnableANDSAn']['value'] = 0
```

In the AND_disable operation, all the cells are deselected by setting their BL to '0'.

5.8.4 Cadence Virtuoso schematic

A schematic very similar to the one presented in Figure 3.16 has been derived for the LiM array testing, using the same design flow; first, the cell and sense amplifier topologies have been produced in Virtuoso; second, a 8 bits column has been derived and used to obtain a 8x8 array; this, after being tested to tune the design characteristics, has been used to derive larger arrays (16x16 and 32x32).

5.8.5 Waveforms

In the following, the simulation waveforms for the three array types are reported.

It can be noticed how in Figure 5.29 an opposite behavior with respect to Figure 5.28 and Figure 5.27 is obtained for ANDOut, which represents the AND line voltage, and, consequently, ANDSAO, which is the LiM sense amplifier output. The reason of this has been explained in section 5.3.

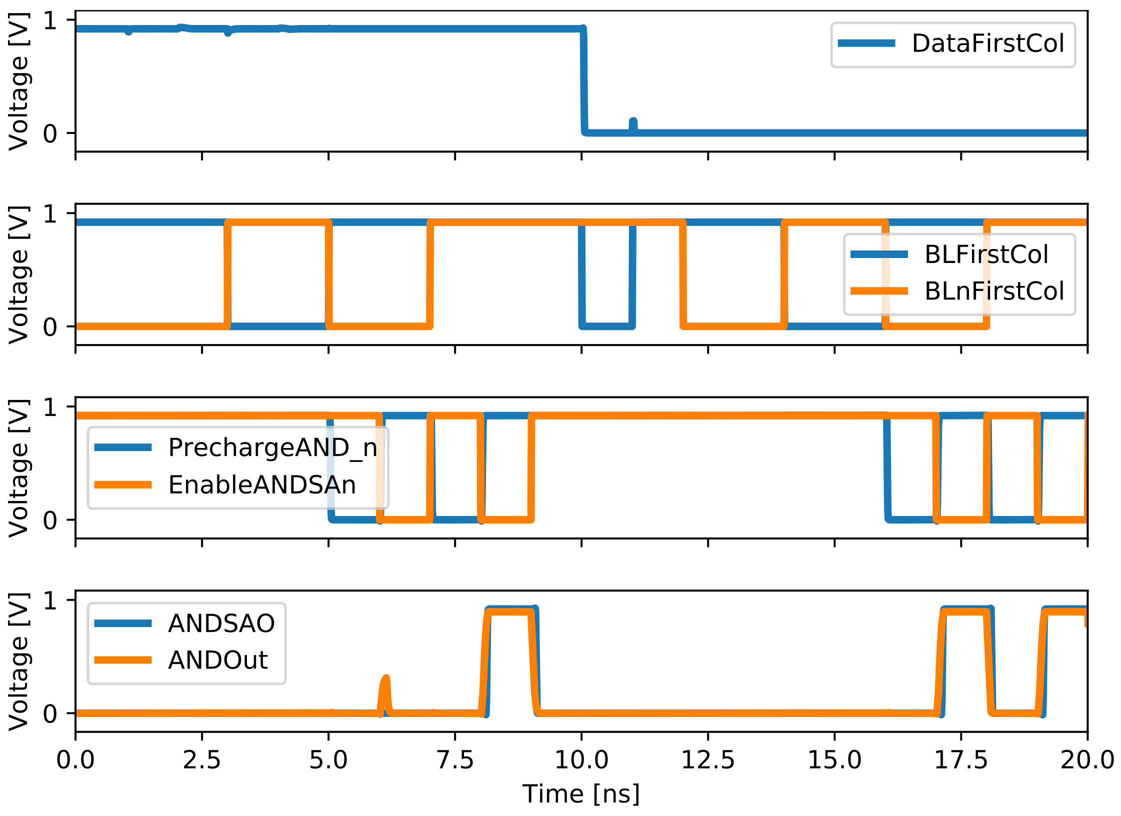


Figure 5.27: Dynamic cell waveforms.

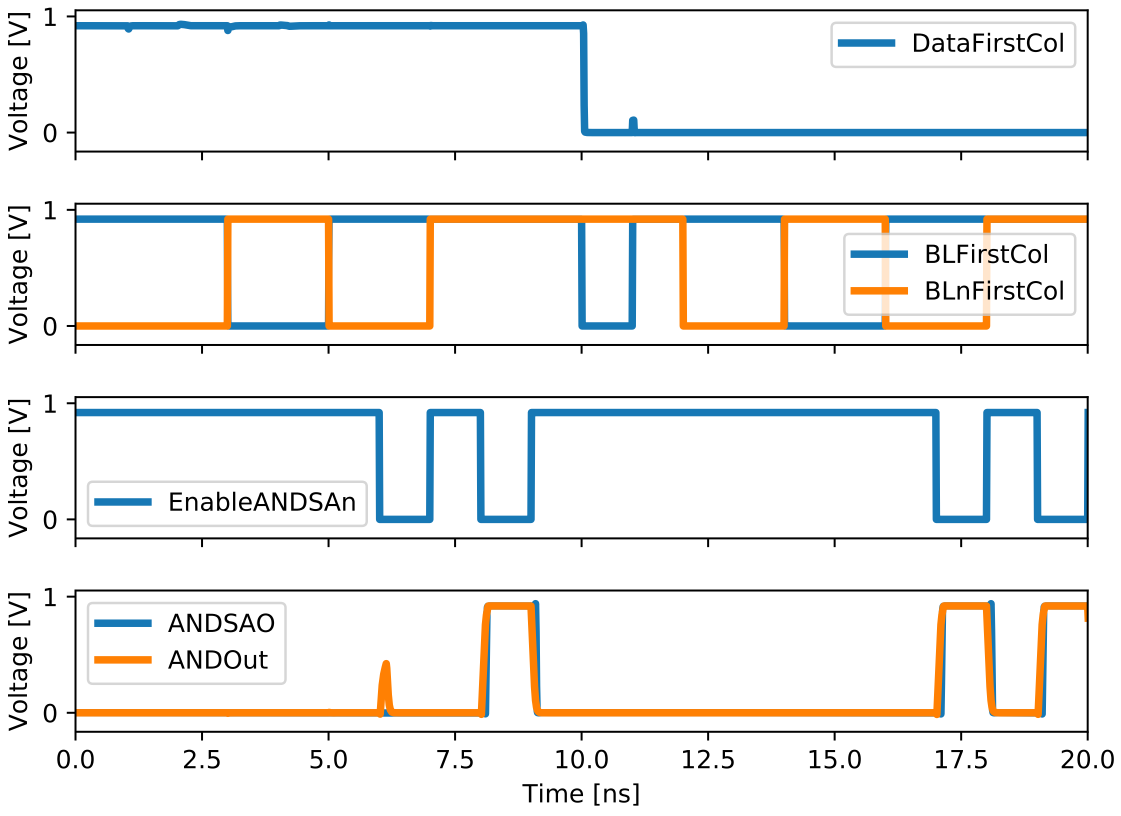


Figure 5.28: Static cell waveforms.

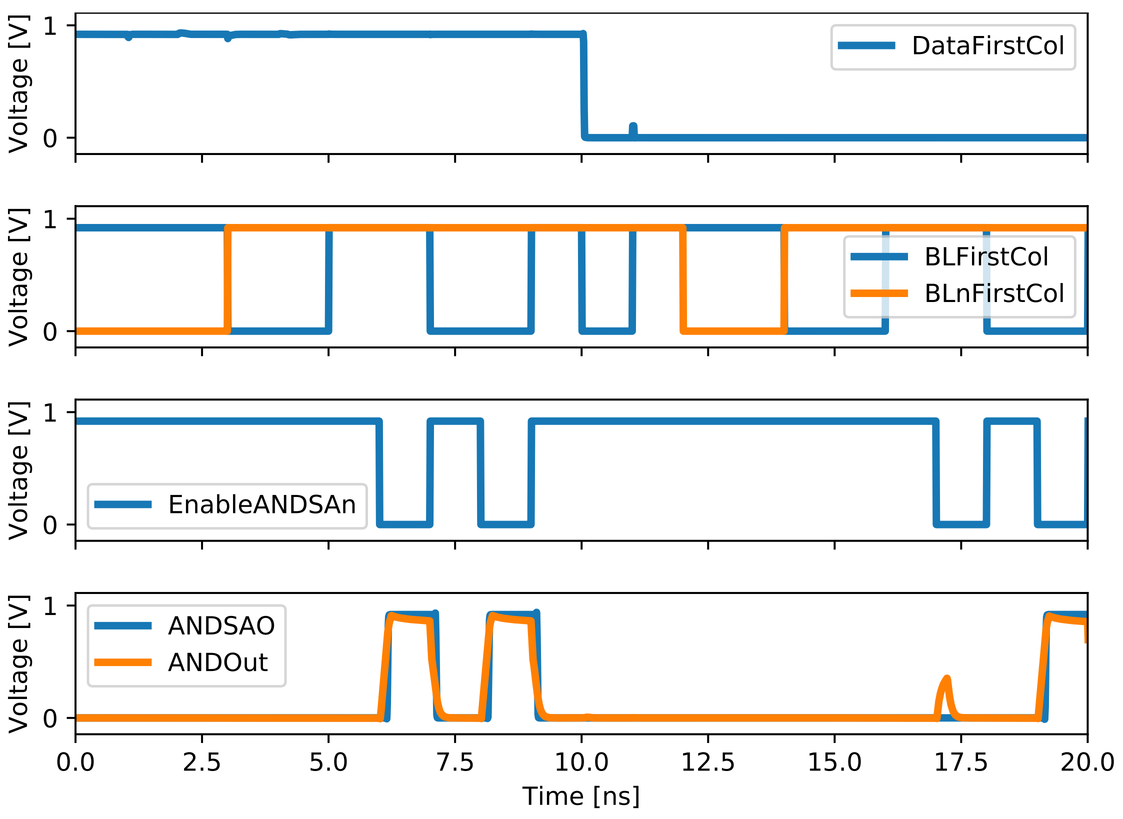


Figure 5.29: Special-purpose cell waveforms.

Chapter 6

ALiAS

ALiAS

Awesome LiM Arrays Synthesizer

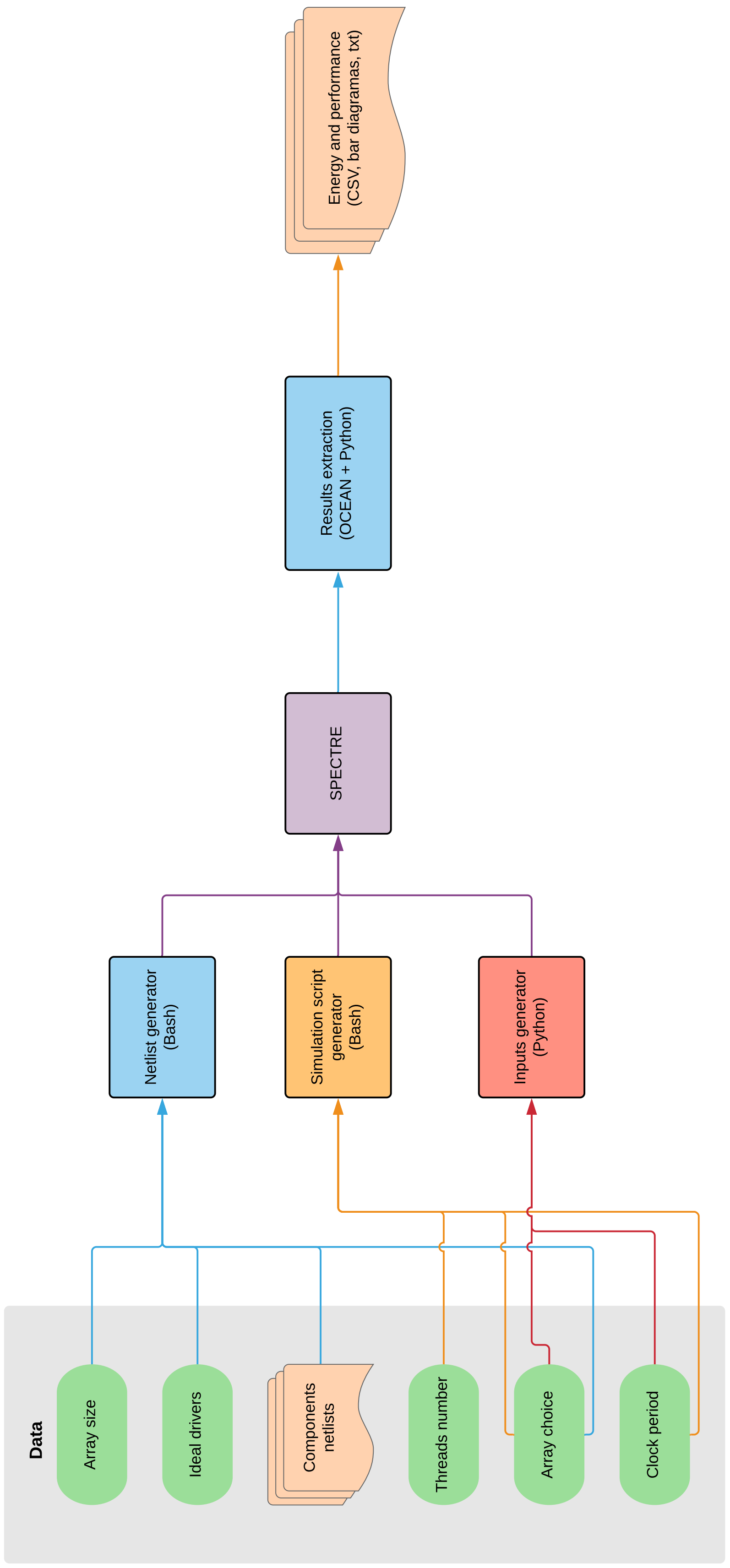
In this chapter, the ALiAS tool (Analog Logic-in-Memory Arrays Synthesizer) is presented. This program synthesizes the SPECTRE (SPICE proprietary language of Cadence) netlist of a memory array, simulates the netlist in Cadence Virtuoso (SPECTRE simulator) and extracts from the simulation performance and energy consumption of the array, saving the results in proper files formats.

As of today, the accepted array types are:

- the model of a SRAM array.
- the model of a CAM array.
- the model of the LiM array presented in chapter 5, implementing the three cell types.

The tool code has been realized using the Bash and Python languages.

6.1 Design flow



Data
Array size
Ideal drivers
Components netlists
Threads number
Array choice
Clock period
Netlist generator (Bash)
Simulation script generator (Bash)
Inputs generator (Python)
SPECTRE
Results extraction (OCEAN + Python)
Energy and performance (CSV, bar diagramas, txt)

Figure 6.1: Design flow.

In Figure 6.1, the tool design flow is shown. On the left of the scheme, inside the grey box, the inputs of the tool are presented:

- `Array size`: provided by the user in form of height and width of the array to be generated.
- `Ideal drivers`: the user gets to choose if real drivers have to be instantiated inside the testbench, in order to take into account their non idealities. The drivers instantiated are the ones presented in section 3.2 and they are automatically designed by the tool.
- `Components netlists`: these are the netlists of the fundamental components that characterize the array architecture, such as sense amplifier and memory cell topologies. These have to be generated by the designer and included in the tool. In fact, being custom components, they cannot be automatically generated, since their circuits are highly dependent on the architecture type (i.e. logic operations implemented in the array). Of course, the code has to be rearranged for new array types to be generated.
- `Threads number`: Cadence Virtuoso allows to choose the number of CPU threads to be assigned to the simulation. The number of threads to be used is set to a default value of 4, which can be modified by the user when the program is launched.
- `Array choice`: the kind of array to be generated.
- `Clock period`: the clock period of the simulation, which default value is 1 ns. This value can be increased so that particularly large (hence slow) arrays can be correctly simulated.

The user choices are then provided to three code blocks:

- `Netlist generator`: this is a block that, given the array type, its size and the user options, generates the netlist of the array and the corresponding testbench.
- `Simulation script generator`: this block generates the OCEAN (scripting language for Cadence Virtuoso) script that handles the simulation of the testbench and the results extraction, starting from the array type, the number of threads to be assigned to the simulation and the clock period of the latter.
- `Input waveforms generator`: this is a block written in Python that generates the input stimuli for the testbench according to the chosen array type and clock period.

Then, the resulting testbench is simulated using the Cadence SPECTRE simulator and the performance and energy consumption of the array are extracted using an OCEAN script generated by the tool.

At the end, the results are organized using proper file formats inside the results directory provided by the user:

- a plain text file for the energy and delay measurements.
- CSV files for the waveforms generated in the simulation.
- bar diagrams that can be used to compare the performance and energy consumption of the memory operations.

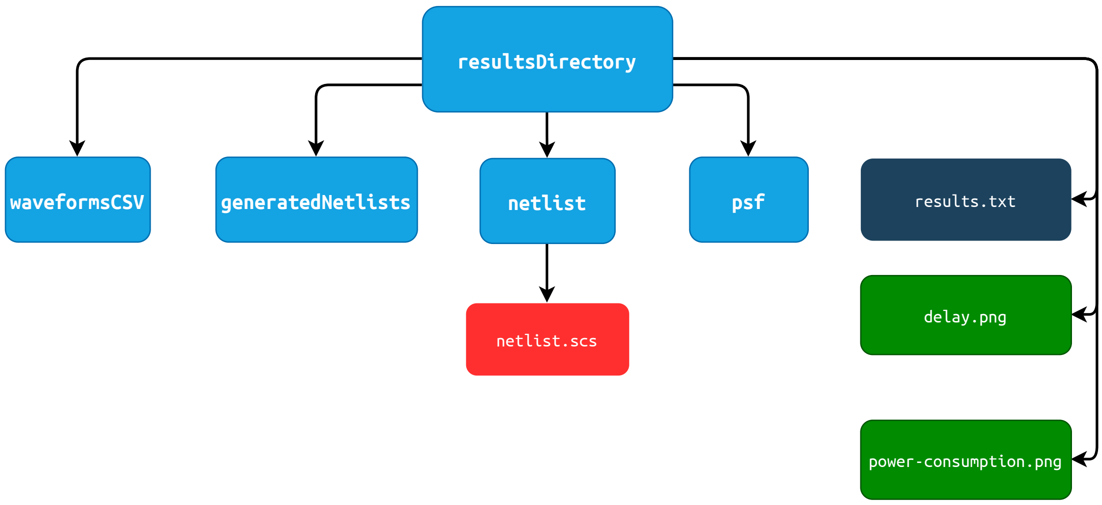


Figure 6.2: The results directory structure.

In Figure 6.2 the results directory structure generated after during a simulation is reported with all the files and directories generated by ALiAS:

- `waveformsCSV`: this directory contains the CSV files of the testbench waveforms generated during the simulation.
- `generatedNetlists`: here the netlist files generated by the tool are saved.
- `netlist`: this is the working directory of the SPECTRE simulator. Here the testbench netlist, called `netlist.scs`, is stored, together with the files used by SPECTRE to perform the simulation.
- `psf`: here the results files generated by SPECTRE are stored.
- `results.txt`: in this file the delay and energy consumption values of each memory operation are saved.
- `delay.png`: this is a bar diagram generated in Python starting from the `results.txt` file, where the operations performance are compared.
- `power-consumption.png`; this is another bar diagram generated in Python in which the power consumptions of the memory operations are compared. Also this file is generated from `results.txt`.

6.2 The tool structure

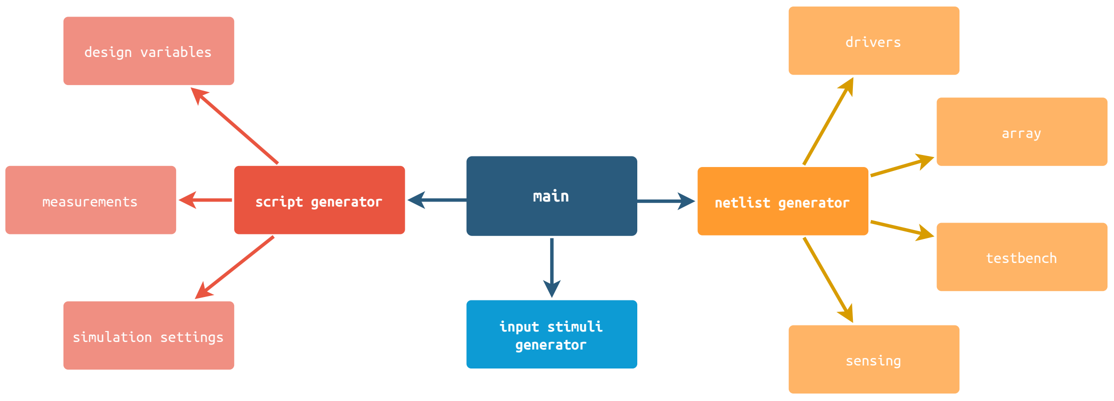


Figure 6.3: Program structure

The program structure is presented in Figure 6.3.

As stated before, the program is made by three main blocks: the `netlist generator`, the `script generator` and the `input stimuli generator`. Each of these blocks will be analyzed in detail in the following, using as as guiding example the LiM array that implements the special-purpose cell (section 5.6).

6.2.1 Netlist generator

The `netlist generator` produces the netlists of the drivers, array, testbench and sensing circuitry (dummy rows, delay circuits etc.).

The testbench architecture

In order to properly analyze the code structure, it is better to have in mind the testbench circuit organization, which is shown in Figure 6.4.

The complete array is substituted with a simplified model which is equivalent from a performance point of view.

Since all the memory operations are tested on only one row, as discussed in the previous chapters, only this is instantiated inside the testbench, together with a memory column.

The row is not a standard one: this is made by real cells , which circuit is shown in Figure 6.5, that are put at the ends of the row, and by dummy ones that are placed in between these.

A real cell is instantiated on the first column (beginning of the row) and it is used to test the performance of the logic operations, such as the CAM search operation and the LiM AND operation. This position is used since it is the farthest one from the match-line and AND-line sense amplifiers inputs, as discussed in section 4.6, that are put next to the last column, as shown in Figure 6.4.

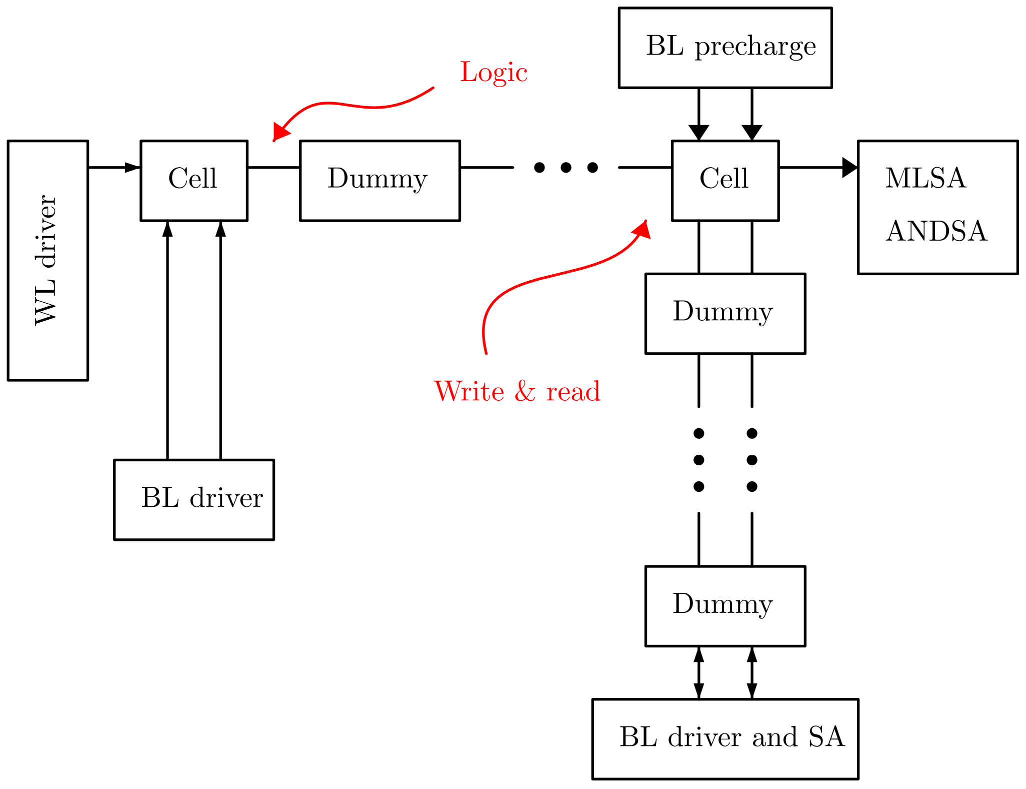


Figure 6.4: The array model.

Another real cell is placed at the end of the row (last column) and it is used to evaluate the performance of the write and read operations. This position is chosen since it is the farthest one from the wordline and bitline drivers and the read sense amplifier, as discussed in section 4.6.

In Figure 6.5, the complete schematic of the cell is shown. It can be noticed how, for each pin of the cell, a *RC* circuit has been employed to mimic the interconnections parasitic parameters. Hence, each pin of the cell has been divided in an input and an output ones, with the *RC* circuit in between.

From this schematic, the dummy cells topologies are derived.

The dummy-row cells are not proper memory cells: they are used only to take into account the row size in the performance measurements, which means that inside these cells only the transistors connected to the row signals, such as the wordline, the match-line and the AND-line, are present, while the others are removed. Simplifying the cell circuit, the simulation complexity is reduced and the resulting performance results are really similar to the ones that one would obtain using a complete array.

The circuit of a dummy row cell is shown in Figure 6.6.

In Figure 6.6, for each row signal the corresponding transistors are instantiated, together with a *RC* circuit that takes into account the parasitics of the metal line and, hence, separates the input of each cell from the output:

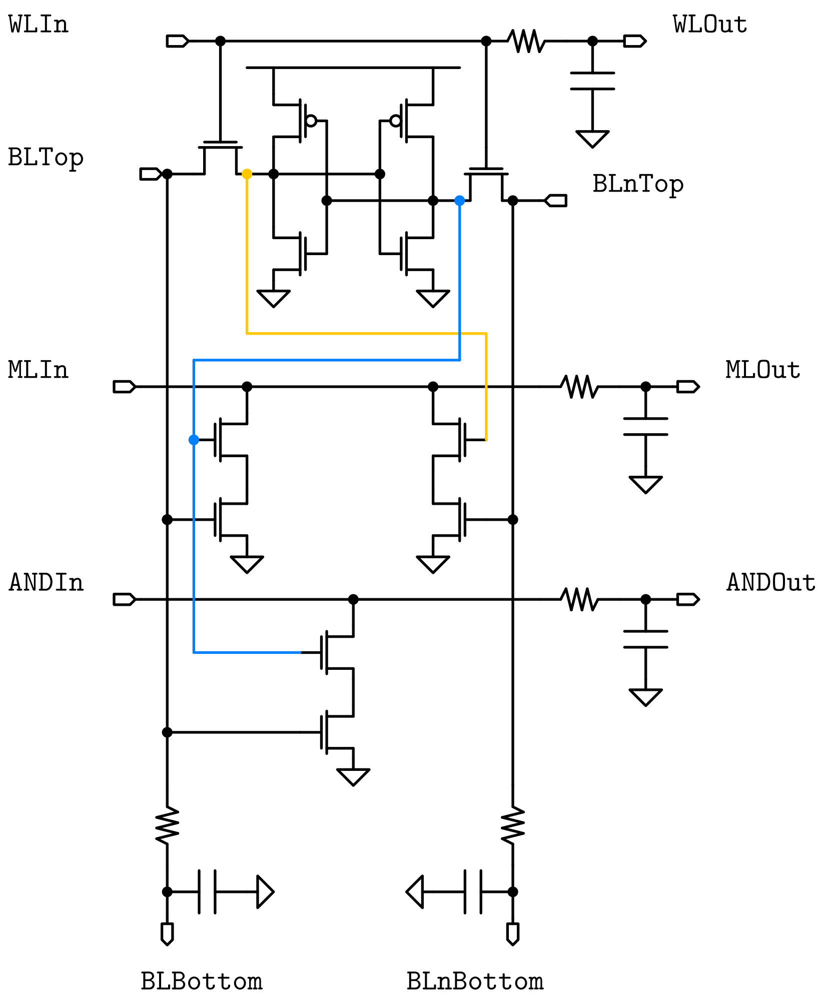


Figure 6.5: Special-purpose LiM cell.

- for the wordline, the pass-transistors of the cell are left in order to take into account the influence of their gate capacitance on the line operation.

- for the match-line, the CAM pull-down paths are instantiated, since these slow down the line operation with their drain capacitance and represent a contribution to the leakage of the line with their subthreshold current. Of course, the pull-down paths are disabled by properly choosing the transistors gate potentials, since the dummy cells must have no influence on the logic operation executed on the ML apart from their parasitic contribution.

- for the AND line, the transistors of the logic part of the cell are employed in order to take into account their drain capacitance and their leakage. As in the match-line case, the pull-down path is disabled by properly choosing the gate voltages of the transistors.

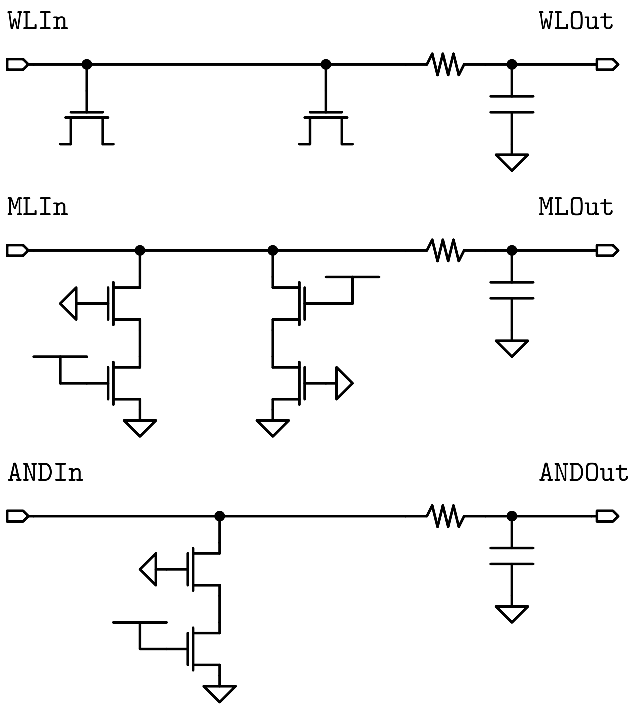


Figure 6.6: The dummy-row cell.

Since the cell on the last column is used to test the write and read operations, its distance from the bitline drivers and sense amplifier has to be taken into account: this is accomplished by including a memory column in the testbench, made by dummy cells designed in a way similar to the dummy-row ones, as shown in Figure 6.4.

The schematic of a dummy-column cell is shown in Figure 6.7:

- for the SRAM part, the pass-transistors are instantiated since they slow down the bitlines operation with their drain capacitance. Also, a partial replica of the cell SRAM core is reported in order to take into account the leakage current absorbed from/injected in the bitlines by the unselected rows. This replica emulates a cell that stores a logic '1', since a pull-up transistor is added on the BL side and a pull-down one is put on the $\overline{\text{BL}}$ side.

- for the CAM part, the pull-down transistors connected to the bitlines are instantiated in order to take into account their gate capacitance.

- for the LiM part, the pull-down transistor connected to BL is employed to take into account its gate capacitance.

In Figure 6.4 other memory blocks are shown:

- the last column bitline drivers, that are used to test the write operation, which circuit is shown in Figure 3.2.

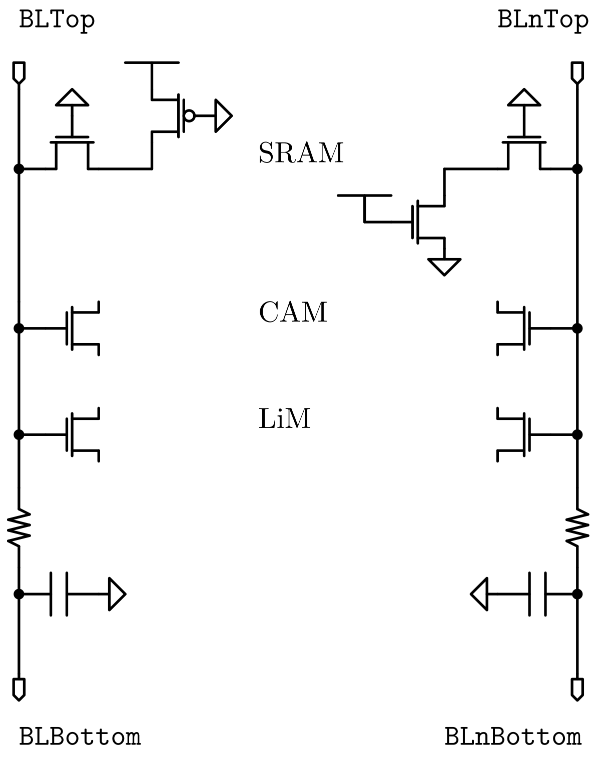


Figure 6.7: The dummy-column cell.

- the last column sense amplifier, used to test the read operation, which circuit is shown in Figure 3.5.
- the last column precharge circuit, which circuit is shown in Figure 3.3, used for the read operation.
- the wordline driver, used to test both write and read operations, which circuit is shown in Figure 3.4.
- the MLSA and ANDSA, which are the sense amplifiers used for the search and AND operations, which circuit is shown in Figure 4.2.

In Figure 6.4 only one MLSA and one ANDSA are shown; however, in a real memory array one sense amplifier per line (AND line and match-line) and row is needed. The number of rows and, so, of sense amplifiers has an influence on the performance and power consumption of the array: in fact, in the dummy-line sensing scheme, the dummy sense amplifier has to drive all the other SAs enable signal in order to disable them. This results in an increase in the delay associated to the dummy sense amplifier output and, so, in a larger energy consumption, since the real SAs are disabled after a larger time interval. In fact, all the sense amplifiers current generators are disabled as soon as their enable signal changes value: hence, the longer the time interval needed for this to happen, the larger will be the conduction time of the sense amplifiers and, so, the associated energy consumption.

This has to be taken into account in the array model presented. In order to do it, dummy loads have to be instantiated for the dummy SAs, which take the place of the missing sense amplifiers.

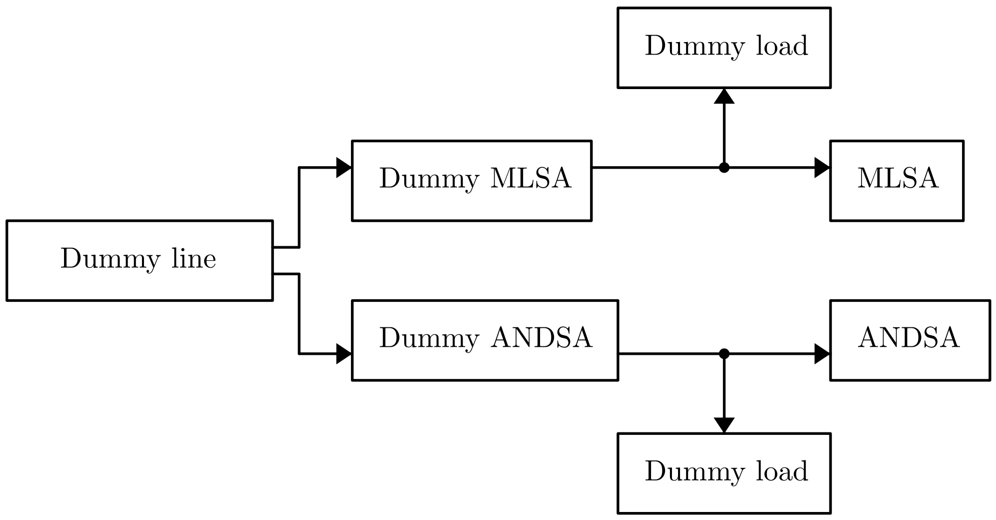


Figure 6.8: The dummy load scheme.

As previously explained, in the instantiated testbench each dummy sense amplifier has to drive only one sense amplifier of the same kind: the dummy MLSA drives the real MLSA and the dummy ANDSA drives the real ANDSA. However, in a real array the dummy sense amplifier should drive a number of amplifiers equal to the number of rows of the array, since one SA is instantiated for each row. To take into account this, a dummy load is attached to the output of each dummy SA, as shown in Figure 6.8.

As it has been shown in section 4.4, each real sense amplifier contains an OR gate that is driven by the system enable signal and the dummy SA. Hence, the load seen by the dummy sense amplifier for each SA that it drives, is made by an OR gate. For this reason, the dummy load used to emulate the presence of the remaining sense amplifiers is made by input sections of OR gates all connected in parallel, as it is shown in Figure 6.9. The number of OR gates is equal to the number of rows in the array (i.e. the array gate).

The OR gates used for the dummy load are not full CMOS logic circuits: only the transistors connected to one of the inputs are used, in order to minimize the area occupied by the load and to prevent unnecessary switching activity in these gates. The OR gate internals are shown in Figure 6.9.

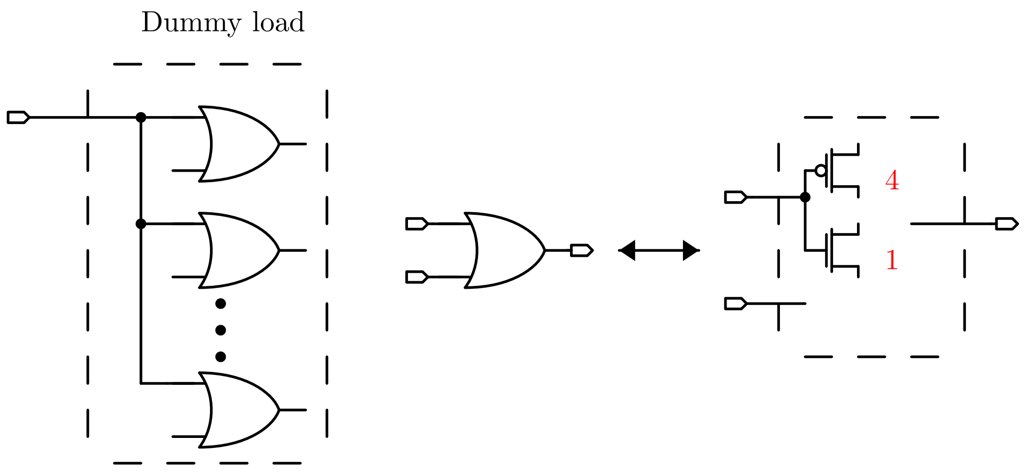


Figure 6.9: The dummy load.

The code structure

In Figure 6.10 the `netlist generator` code structure is shown, using different colors to indicate different file types: cyan for the directories, blue for Bash files, red for SPECTRE netlists.

For each array type, a dedicated directory is created, which contains the Bash files that generate the corresponding netlists. As stated before, only the LiM array is considered in this analysis for simplicity.

Hence, the `netslist generator` structure is the following:

- `arrayGenerator.sh` creates the array netlist.
- `dummyLineGenerator.sh` creates the dummy row used to implement the dummy-line sensing scheme.
- `TBGenerator.sh` produces the testbench netlist, instantiating all the blocks generated by the tool and the signal generators that provide the input stimuli to the circuit.
- `main.sh` generates the final netlist that is provided to the simulator by organizing in an unique file the netlists produced by the scripts.

A directory called `Common` is used to store the Bash files that generate the memory blocks common to all the array types, such as the bitline and wordline drivers (`BLDriverGenerator.sh` and `WLDriverGenerator.sh`), the block that generates the delay for the enable signal of the sense amplifier used to read the memory content discussed in section 3.3 (`SADelayGenerator.sh`) and the Bash functions used to instantiate each of the memory blocks in the testbench netlists (`componentsFunctions.sh`).

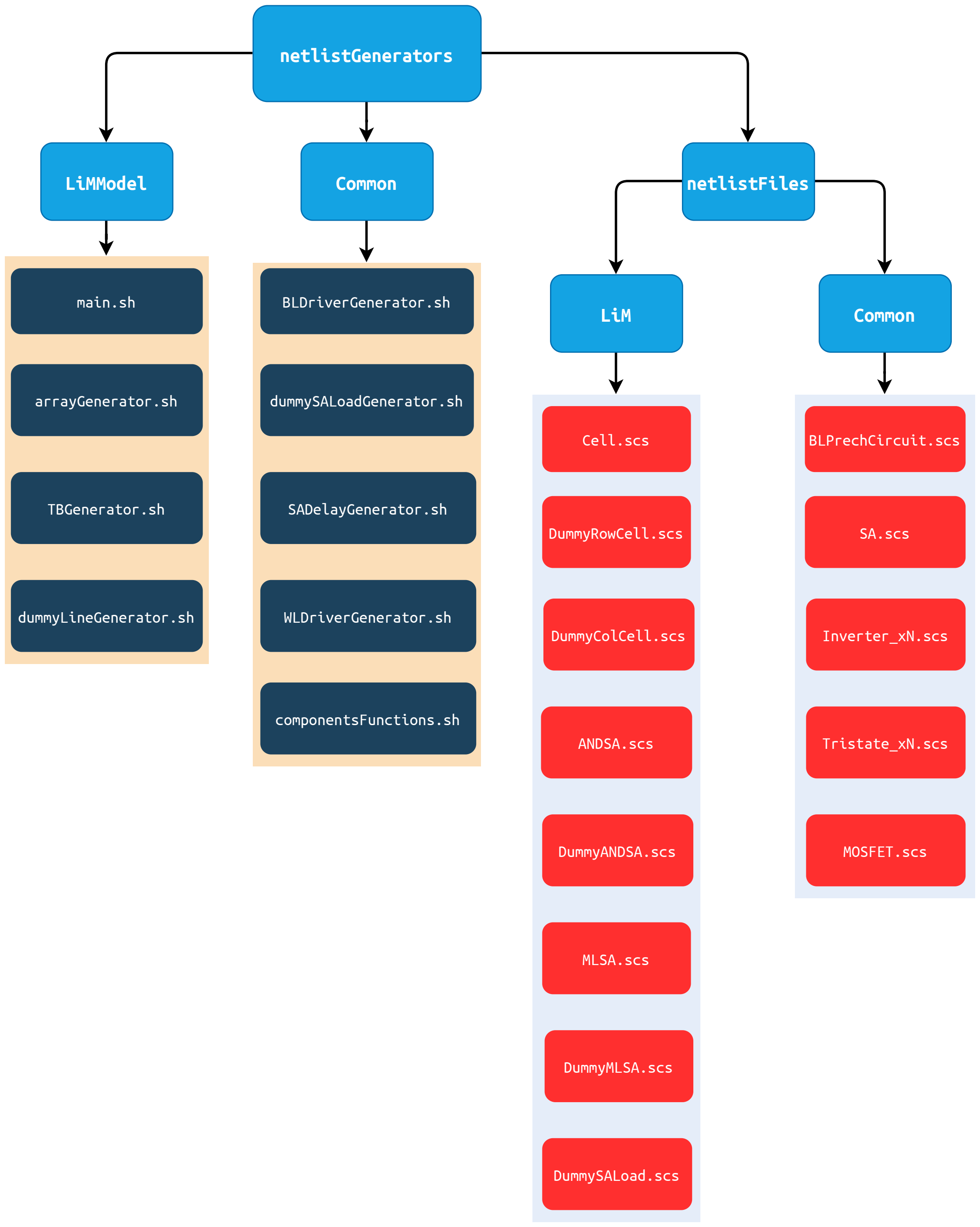


Figure 6.10: Netlist generator block.

In the directory `netlistFiles`, the SPECTRE netlists of the by-hand designed components are contained. For every array type, specific netlists are dedicated, which are stored in apposite directories. For the LiM case:

- `Cell.scs`: the memory cell, shown in Figure 6.5.
- `DummyColCell.scs`, `DummyRowCell.scs`: the dummy column and row cells, shown

in Figure 6.7 and Figure 6.6 respectively.

- `ANDSA.scs`: the AND line sense amplifier (LiM).
- `DummyANDSA.scs`: the dummy AND line sense amplifier, used to implement the dummy line sensing scheme.
- `MSLA.scs`: the ML sense amplifier (CAM).
- `DummyMLSA.scs`: the dummy MLSA.
- `DummySALoad.scs`: the load of the dummy MLSA and ANDSA. This is the netlist of a single OR gate, like the one presented in Figure 6.9.

In the directory `Common` the netlists of the blocks common to all the array types are stored:

- `BLPrechCircuit.scs`: the circuit used to precharge and equalize the bitlines before a read operation.
- `SA.scs`: the sense amplifier used to read the memory content.
- `Inverter_xN`: a standard CMOS inverter which driving strength (i.e. aspect ratio) can be choosen arbitrarily when instantiated. It is a parametric block used to realize the drivers.
- `Tristate_xN`: a standard CMOS tristate inverter. As in `Inverter_xN.scs`, also for this component the driving strength can be chosen arbitrarily. This circuit is used for the output section of the bitline drivers.
- `MOSFET.scs`: the netlist of the MOSFET model.

WL driver generator algorithm

The `WLDriverGenerator.sh` script generates the netlist of the wordline driver using as input parameter the array width. The larger is the row width, the stronger (from a driving point of view) the wordline driver has to be, which means that many inverter stages have to be instantiated in the driver chain.

The algorithm, whose flowchart is shown in Figure 6.11, starts from the base case of an array whose width is equal to 8 bits. At the beginning of the execution, the aspect ratio for the inverters to be placed in the driving chain, `W/L`, is chosen equal to 1 and an input inverter is placed:

```
W/L=1
place input inverter
```

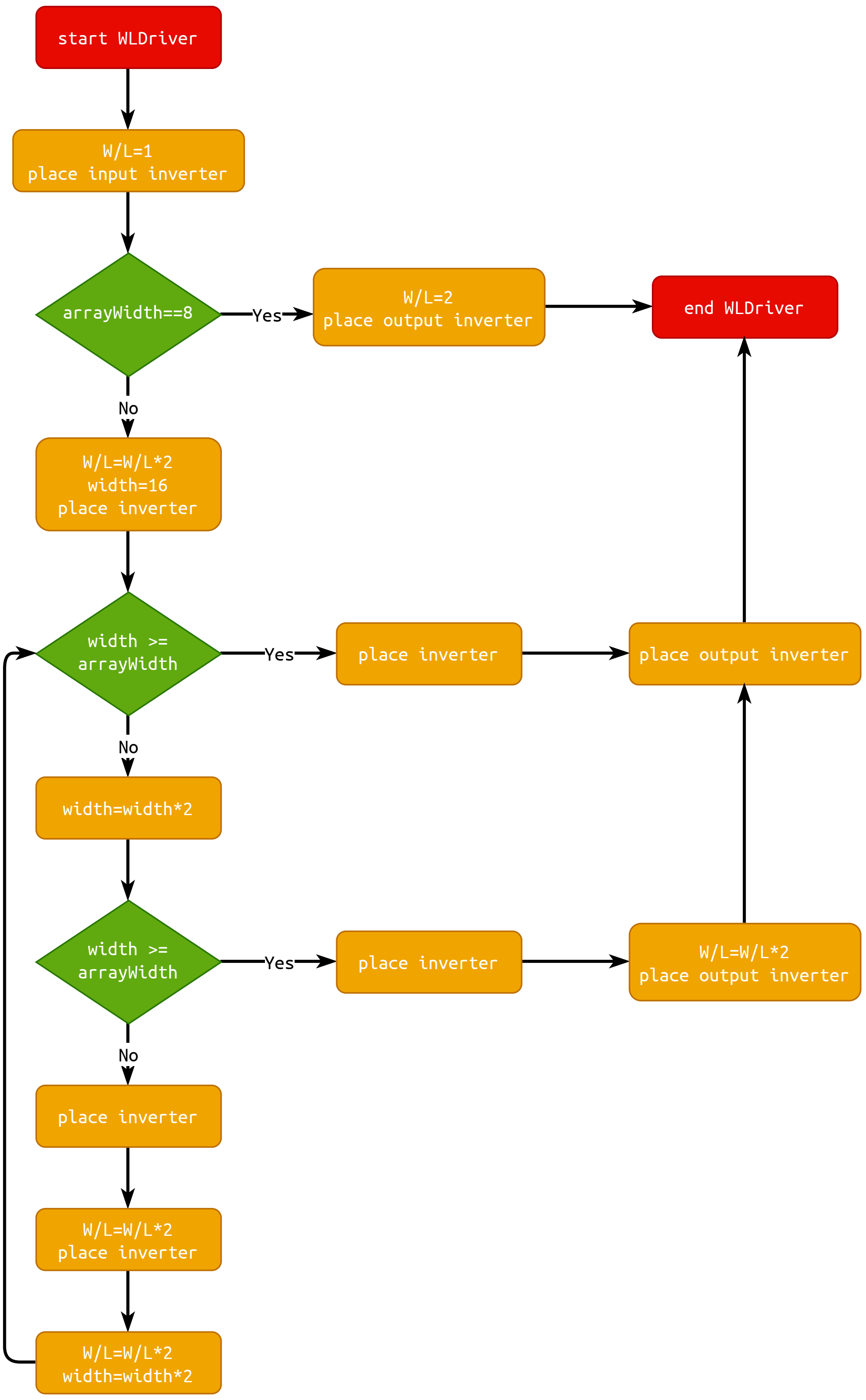


Figure 6.11: Wordline driver synthesis algorithm.

Then, the array width is checked: if this is equal to 8 (`arrayWidth == 8`, base case), the inverter aspect ratio is doubled and the output inverter is instantiated:

```
W/L=2
place output inverter
```

After this, the synthesis is ended (`end WLDriver`).

If the array width is not equal to 8, the synthesis loop is started: a parameter `width`, that represented the array hypothesized array width, is defined and initialized to 16 bits, `W/L` is doubled and an inverter is instantiated:

```
W/L=W/L*2
width=16
place inverter
```

Then, it is checked if the array width is lower than or equal to the hypothesized one (`width >= arrayWidth`): if it is, an output inverter chain is placed and the synthesis is ended:

```
place inverter
place output inverter
```

If it is not, the `width` is doubled and the previous check is repeated: if now the the array width is lower than or equal to `width`, an output inverter chain (i.e. two inverters in series) is placed, but with the last inverter characterized by and aspect ratio doubled with respect to the previous one:

```
place inverter
W/L=W/L*2
place output inverter
```

and the synthesis is ended.

If the check gives a false result (i.e. the array width is still larger than `width`), an inverter sequence is placed, in which the last inverter has an aspect ratio doubled with respect the previous one:

```
place inverter
W/L=W/L*2
place inverter
```

Then, the design aspect ratio is doubled (`W/L=W/L*2`) in order to increase the driver chain strength, together with the hypothesized array width (`width=width*2`), and the loop starts again. The synthesis process ends as soon as `width` is larger than or equal to the array actual width.

The SPECTRE syntax

Before analyzing the code, it is better to take a look at the SPECTRE syntax.

In SPECTRE, a component is called "subcircuit" and it is identified by the `subckt` keyword. The syntax to define a subcircuit is the following:

```
subckt componentName pins
        // Component body
ends componentName
```

The `subckt` is followed by the component name, decided by the user, and the pins list of the component. Then, in the component body the circuit is defined.

For example, suppose that a driver made by a chain of four equally sized inverters has to be instantiated. This driver has a `In` pin and a `Out` pin. A possible component netlist could be the following:

```
subckt Driver In Out
        Inverter_1 In net1 Inverter
        Inverter_2 net1 net2 Inverter
        Inverter_3 net2 net3 Inverter
        Inverter_4 net3 Out Inverter
ends Driver
```

In the component, four inverters are instantiated. These inverters are subcircuits previously defined somewhere in the netlist. Each line corresponds to an inverter instance, written using the following syntax:

```
instanceName pins instanceCircuit
```

First, the instance name is defined; then, the pins of this are assigned to net names that are used to interconnect the instances; then, the instance circuit (which is the kind of circuit being instantiated, like an inverter, an AND gate etc.).

Two instances are connected if they have the same label associated to one of their pins (also called "nets"). If the inverter subcircuit definition is the following:

```
subckt Inverter In Out
```

To connect the output of one inverter to the input of another one, one should write:

```
Inverter_1 In net1 Inverter
Inverter_2 net1 Out Inverter
```

Hence, the netlist reported for the `Driver` subcircuit is equivalent to the circuit shown in Figure 6.12.

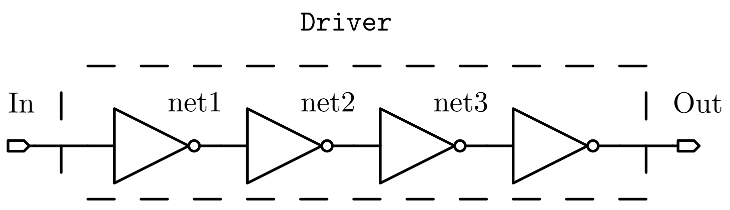


Figure 6.12: Circuit synthesized by the netlist.

WL driver generator code

In the following, the wordline generator code is presented.

```
1 #! /bin/bash
2 
3 #### FUNCTIONS ####
4 source $pathToThisProgram/netlistGenerators/Common/componentsFunctions.sh
5 
6 #### MAIN ####
7 netlistFile="$pathToResultsDir/generatedNetlists/WLDriver.scs"
8 
9 echo "subckt WLDriver In Out" > $netlistFile
```

Here, the functions used to instantiate the components are called (#4), the netlist name is defined (#7) and the `WLDriver` subcircuit is defined (#9).

```
1 # The input inverter is instantiated
2 createInverter "InputInverter" In net1 1 >> $netlistFile
3 
4 invNum=2 # Inverter instance number
```

```
5  currentSize=16 # It represents the hypotized size of the the array. As soon as this is equal to the actual size, no more inverters are added to the driver chain.
6  inverterSize=4 # The hypotized size of the output inverter. It is updated during the following while cycle.
7  finished=0 # Variable used to stop the while cycle execution.
```

Here, the input inverter is instantiated (#2) using a previously defined bash function, shown in the following:

```
1  # Function that creates the Spectre netlist of an inverter of arbitrary aspect ratio.
2  createInverter(){
3          local name="$1"
4          local inNet="$2"
5          local outNet="$3"
6          local aspectRatio="$4"
7
8          echo $name ($inNet $outNet) Inverter_xN aspectRatio=$aspectRatio //In Out
9
10         return 0
11 }
```

The function takes in input the instance name, the input and output net names and the aspect ratio of the inverter.

Then, the hypothesized size is defined (#5), the inverter aspect ratio is updated (#6) and a variable used to end the synthesis loop is defined (#7).

```
1  if [ $arrayHorSize -eq 8 ]
2  then
3          # If the array size is equal to 8, we fall in the base case (recursion...). Hence, we can instantiate the output tristate driver.
4          createInverter "OutputInverter" net1 Out 2 >> $netlistFile
5  else
6          # If not, the size of the array is at least equal to 16. Hence, at least two buffers (four inverters) are needed in the driver chain. Hence another inverter instantiated.
7          createInverter "Inv1" net1 net2 2 >> $netlistFile
8          while [  $finished -eq 0 ]
9          do
10                 if [ $currentSize -ge $arrayHorSize ]
```

```
                then
                        # If the hypotized size coincides with the real one, we
                        ↪  instantiate the last two inverters, which are
                        ↪  characterised by the same driving strength.
                        createInverter "Inv$invNum" net$invNum net$(($invNum +
                        ↪  1)) $inverterSize >> $netlistFile
                        createInverter "OutputInverter" net$(($invNum + 1)) Out
                        ↪  $inverterSize >> $netlistFile
                        finished=1
                elif [ $(($currentSize * 2)) -ge $arrayHorSize ]
                then
                        # If, instead, the real size is twice the hypotized
                        ↪  one, we instantiate the final inverter couple, in
                        ↪  which the output inverter is twice as much larger
                        ↪  than the previous one.
                        createInverter "Inv$invNum" net$invNum net$(($invNum +
                        ↪  1)) $inverterSize >> $netlistFile
                        createInverter "OutputInverter" net$(($invNum + 1)) Out
                        ↪  $(($inverterSize * 2)) >> $netlistFile
                        finished=1
                else
                        # If both the previous conditions are false, we
                        ↪  instantiate a couple of inverters which driving
                        ↪  strength is doubled from one to another, and we
                        ↪  increase the hypotized size (x4) and the inverters
                        ↪  strength (x4), and we proceed with the while loop.
                        createInverter "Inv$invNum" net$invNum net$(($invNum +
                        ↪  1)) $inverterSize >> $netlistFile
                        createInverter "Inv$(($invNum + 1))" net$(($invNum +
                        ↪  1)) net$(($invNum + 2)) $(($inverterSize * 2)) >>
                        ↪  $netlistFile
                        invNum=$(($invNum + 2))
                        currentSize=$(($currentSize * 4))
                        inverterSize=$(($inverterSize * 4))
                fi
        done
fi

echo "ends WLDriver" >> $netlistFile
```

Here, the code correspondent to synthesis loop shown in the flow chart of Figure 6.11 is presented.

The BL driver

The synthesis algorithm used for the BL driver circuit is the same presented in section 6.2.1, apart from only two differences:

- the output inverter stage of each bitline driver is a tristate one, in order for these to be disabled during the read operation, as it has been discussed in subsection 3.2.1.
- the height of the array is used as input parameter for the synthesis instead of the width, since the bitlines extension is equal to the array height.

The SA enable delay generator

In section 3.3 it has been explained how the enable signal of the sense amplifier used to read the memory content needs to be delayed, in order for the read operation to be carried out correctly. The value of this delay has to be proportional to the array dimensions, since these determine the bitlines discharge rate during a read operation.

This delay is obtained by passing the enable signal of the SA through a cascade of inverters, which number is proportional to the delay obtained.

In Figure 6.13 the algorithm used to synthesize the delay circuit is shown.

First, an equivalent size of the array is defined:

```
size = arrayWidth*0.3 + arrayHeight*0.7
```

One can notice that, in the assignment above, both width and height of the array are considered: this is because both parameters concur in the delay estimation:

- the wider is the array, the farer will be the accessed cell from the wordline driver and, so, the access delay (i.e. the time needed by the cell to be connected to the bitlines) is increased.
- the larger is the array height, the farer will be the accessed cell from the sense amplifier inputs and, so, the larger will be the capacitance that the cell has to discharge.

One can also notice that the array height has a larger influence with respect to the width in the SA delay circuit design: this is because the bitlines length determines in larger part the time needed by the cell to discharge enough the bitline for the SA to correctly sense its inputs; at the same time, the array width has to be considered since an array much larger in width than in height could be synthesized (which is not a common case, since memory arrays usually store thousands of words whose width is limited to hundreds of bits at most).

After the `size` parameter definition, an input inverter is instantiated. Then, another size variable (`hyp_size`) is initialized to 8 bits; after this, four inverters are placed in the delay chain and the synthesis loop begins.

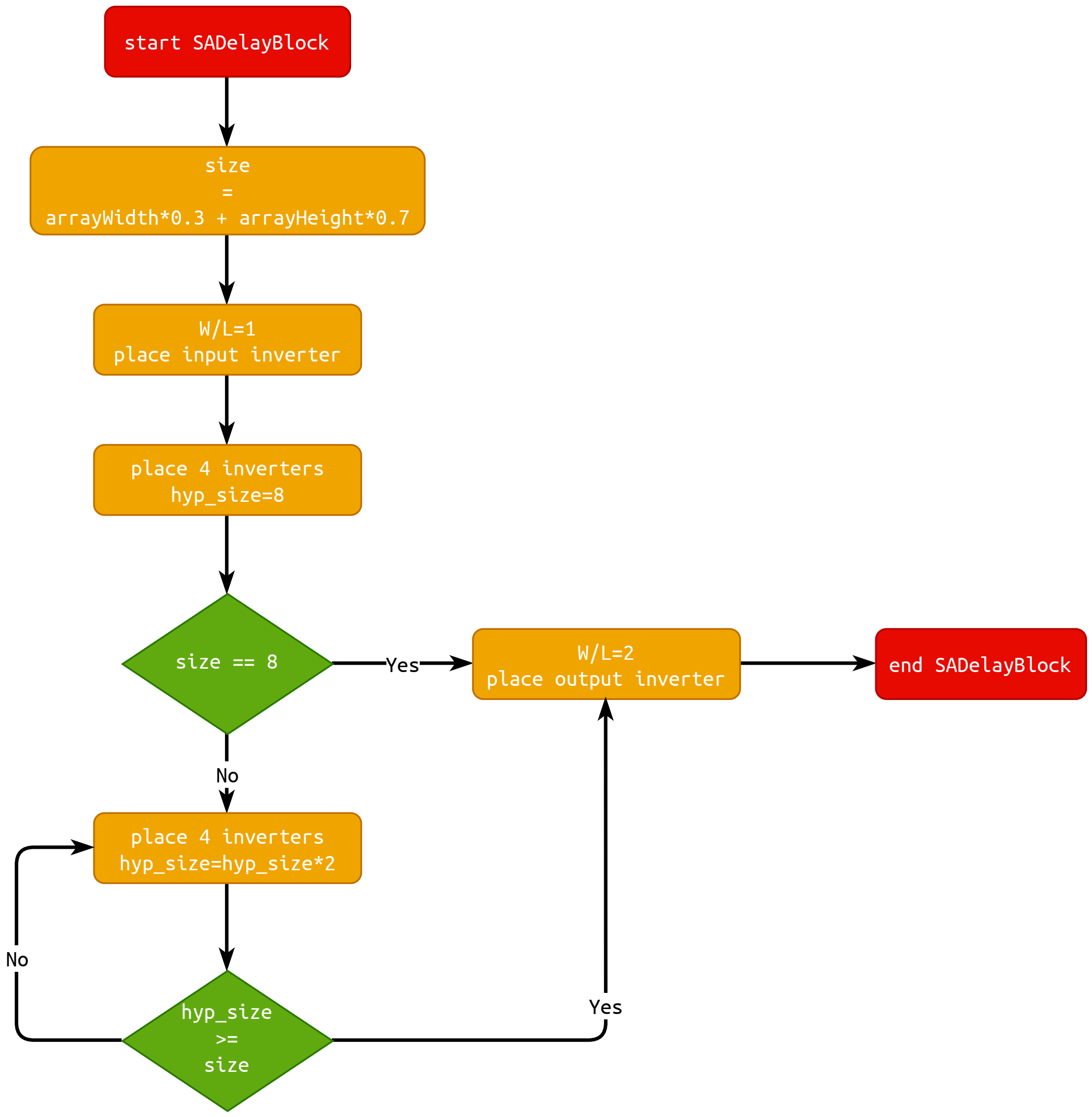


Figure 6.13: The SA delay block algorithm.

First, the base case (array size equal to 8 bits) is checked: if `size == 8`, the output inverter is placed and the synthesis is ended; if the array size is larger than 8, `hyp_size` is doubled and other 4 inverters are placed in the chain. The loop continues until `hyp_size` results to be larger than or equal to `size`.

From the synthesis loop, it may be noticed how 4 inverters are added to the delay chain every time the array size doubles. This is an heuristic approach tuned via various simulations.

MLSA and ANDSA dummy-load generator

In Figure 6.14 the synthesis algorithm used to generate the dummy load for the MLSA and the ANDSA is presented.

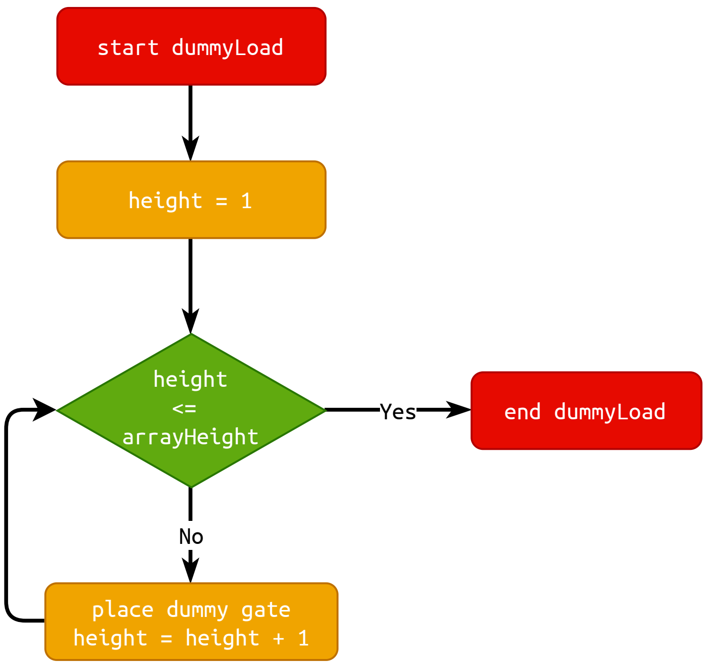


Figure 6.14: Dummy load generator algorithm.

As discussed in section 6.2.1, each sense amplifier is represented by a dummy OR gate, shown in Figure 6.9. Since in a real array a MLSA and an ANDSA is needed for each row, an OR gate is placed in the dummy load circuit for each row of the array model. Hence, the number of OR gates is equal to the array height specified by the user.

This is what is accomplished in the synthesis algorithm: a number of OR gates equal to the array height (`arrayHeight`) is instantiated in the dummy load circuit.

The array generator

In Figure 6.15, the synthesis algorithm used to generate the memory array is shown.

As discussed in section 6.2.1, one row and one column are placed in the testbench. First, a real cell is placed at the beginning of the row (first column); then, given the array width, stored in `arrayWidth`, `arrayWidth - 2` dummy cells are placed on the row; after this, a real cell is placed at the end of the row (last column), resulting in `arrayWidth` cells placed in total on the row; as last step, given the array height `arrayHeight`, `arrayHeight - 1` dummy cells are instantiated on the last column and the synthesis is ended (`arrayHeight - 1` are placed because the real cell on the top of the column also has to be counted in the array height).

The dummy-line generator

In Figure 6.16, the synthesis algorithm used to generate the dummy line, needed for the dummy-line scheme, is shown.

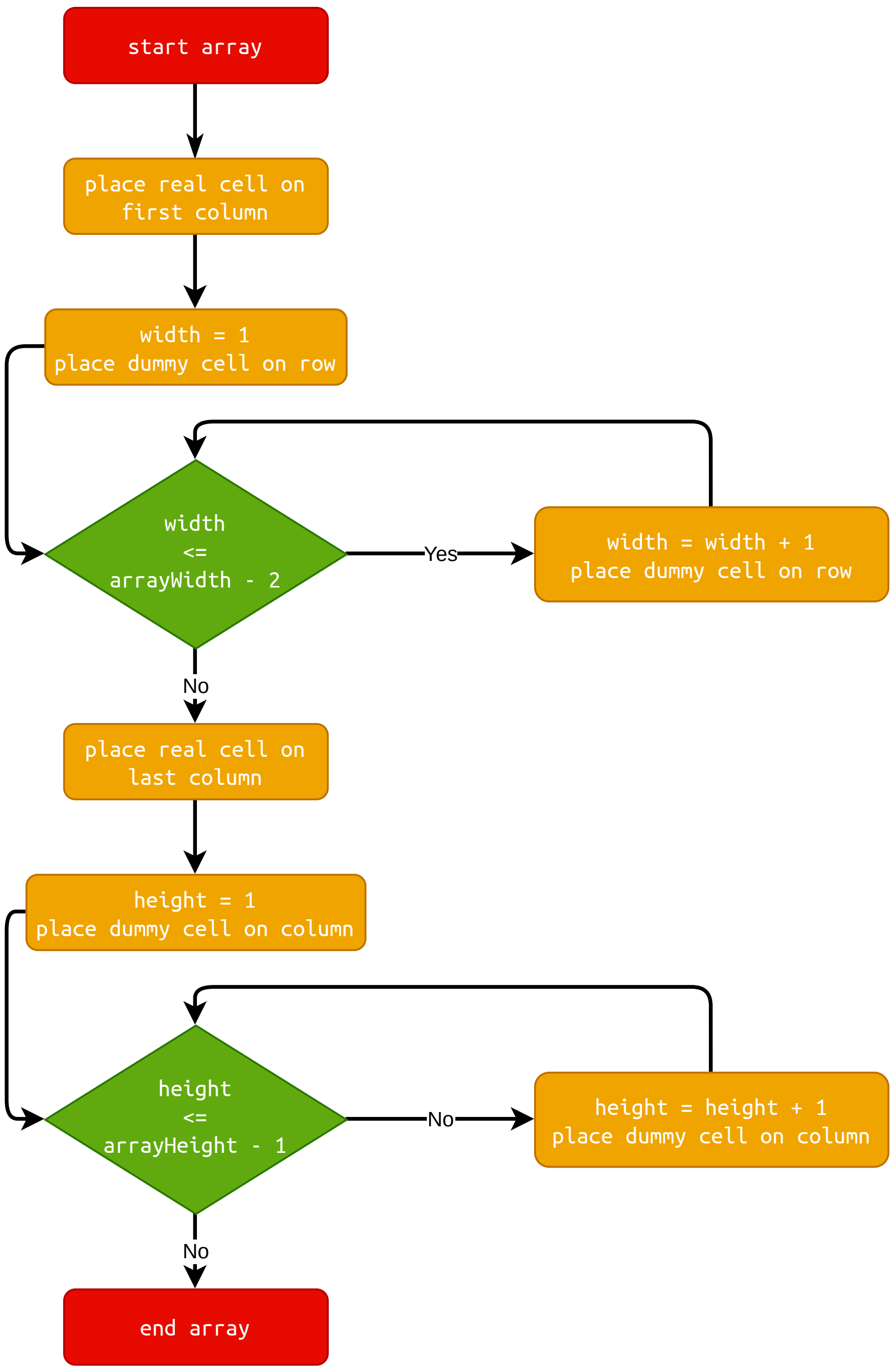


Figure 6.15: The array generator algorithm.

The synthesis loop is analogous to the one presented in Figure 6.14: a number of dummy cells equal to the array width is placed in the dummy-line.

The testbench generator

The code that generates the testbench is trivial: it simply instantiates all the components previously generated. It is more useful to analyze the testbench architecture that is

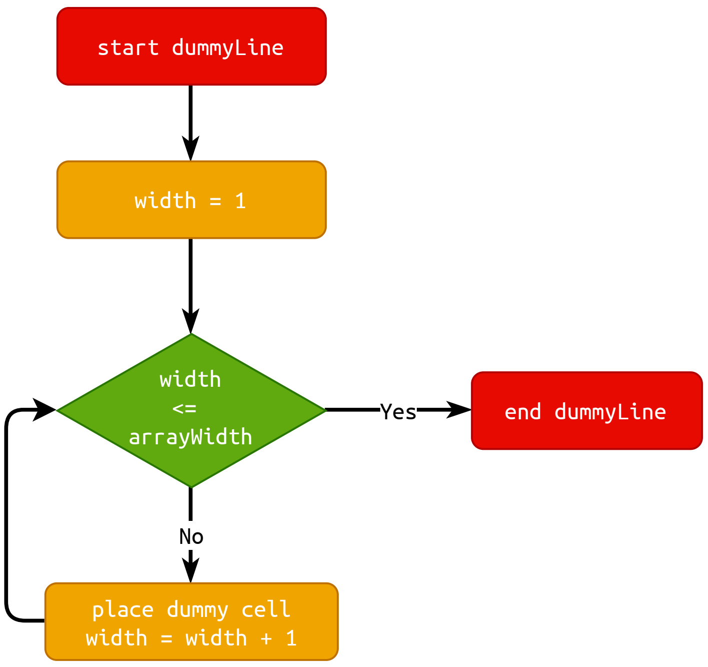


Figure 6.16: The dummy-line synthesis algorithm.

synthesized, shown in Figure 6.17.

In Figure 6.17 all the components are instantiated:

- Array: the array generated by `arrayGenerator.sh`. This takes in input the word-line signal, `WLFirstRow`, the bitline signals associated to the first (`BLFirstCol` and `BLnFirstCol`) and last (`BLLastCol` and `BLnLastCol`) columns. It provides in output the match-line (`MLOut`) and AND-line (`ANDOut`) signals.

- WLDriver: the wordline driver generated by `WLDriverGenerator.sh`. This is driven by a voltage generator, called `WLFirstRow`, that reads its input from a CSV file appositely generated.

- BLDriver: the bitline driver of the last column (the one used to test write and read operations), generated by `BLDriverGenerator.sh`. This is driven by three voltage generators: `BLLastCol` and `BLnLastCol` for the bitlines signals, and `Enable` for enabling the driver.

- PrechargheCircuit: the bitline precharge circuit, syntehsized from `PrechCircuit.scs`. This is used to precharge the bitlines of the last column before a read operation and it is driven by a voltage source `Precharge_n` that enables it.

- SA: the sense amplifier used to read the memory content, synthesized from `SA.scs`. This is enabled by the output of the delay circuit, called “Delay SA” in Figure 6.17.

- Delay SA: the delay block output used to delay the SA enable signal. This circuit

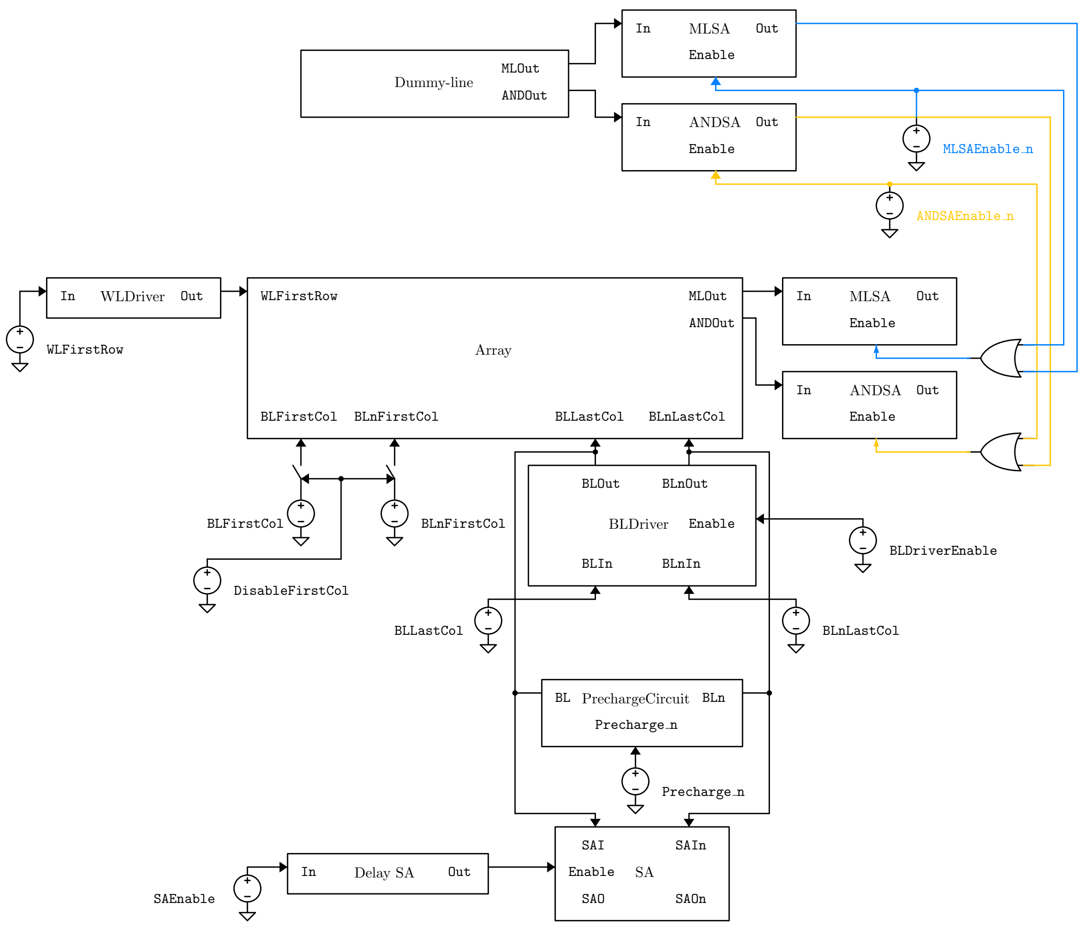


Figure 6.17: The generated testbench.

is synthesized by `SADelayGenerator.sh`, and it is driven by the voltage source `SAEnable`.

- two voltage sources, `BLFirstCol` and `BLnFirstCol`, used to drive the bitlines of the first column. These are disabled during the testing of the read operation using two switches, that are driven by the `DisableFirstCol` voltage source.

- Dummy-line: the dummy-line used to implement the dummy sensing scheme for the match-line and the AND line. This is generated by `DummyLineGenerator.sh`. The outputs of this line are sent to two dummy sense amplifiers, that in turn drive the sense amplifiers connected to the array. Both SAs, dummy and real ones, are connected to the system enable signals, `ANDSAEnable_n` and `MLSAEnable_n`, respectively.

6.2.2 Simulation script generator

In Figure 6.18, the simulation script generator organization is shown.

The simulation script is written using the OCEAN language, which is the Cadence

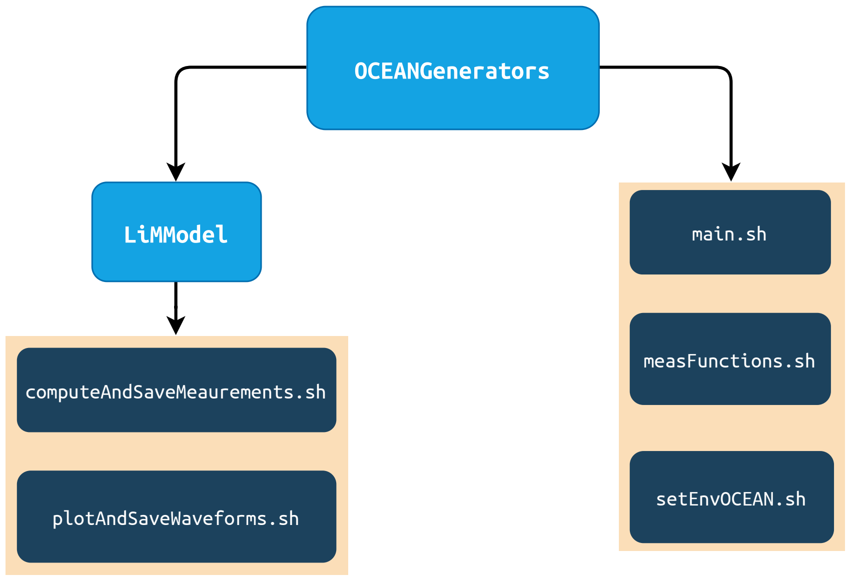


Figure 6.18: The simulation script generator organization.

proprietary scripting language that allows to automatize analog simulations in the Cadence Virtuoso environment.

For each array type, a directory is created (in this case, the `LiMModel` directory). This contains the scripts that are peculiar to a certain architecture:

- `computeAndSaveMeasurements.sh`: this script handles the OCEAN code for the measurements to be performed during the simulations, which are the delay and energy consumption associated to each memory operation.
- `plotAndSaveWaveforms.sh`: this script handles the visualization of the waveforms in the Cadence Virtuoso environment during a simulation and their conversion in CSV files to be used outside Cadence.

Then, there are the scripts that are shared by all the architecture types:

- `main.sh`: this script generates the final OCEAN script to be given to the simulator, by taking the outputs of the other scripts and combining them together.
- `measFunctions.sh`: this script contains the Bash functions that generate the corresponding OCEAN statements used to perform the measurements during a simulation.
- `setEnvOCEAN.sh`: this script sets the simulation environment in OCEAN by defining the results directory, the netlist location and the simulation parameters.

OCEAN environment setting

In the following, the code of the `setEnvOCEAN.sh` script is presented.

```bash
1  #! /bin/bash
2
3  # As first, the incipit of the OCEAN script (which is standard) is written.
4  echo "simulator('spectre)" > $fileOCEAN
5  echo "design(\"$pathToNetlist/netlist\")
6  setup(?numberNotation 'scientific)
7  modelFile('(\"$VirtuosoPath/corners.scs\" \"\"))
8  definitionFile(\"models.scs\")
9  analysis('tran ?stop $simulationTime)
10 desVar(\"R_cell_to_cell\" $parasiticRes)
11 desVar(\"C_cell_to_cell\" $parasiticCap)
12 desVar(\"C_load\" $loadCap)" >> $fileOCEAN
```

First, the simulator to be used is selected (#4); then, the simulation environment is configured (#5 to #9). After this, the design variables are assigned:

- in #10 and #11 the values of the parasitic resistance and capacitance are assigned.
- in #12, the value of the load capacitors of the sense amplifiers are assigned.

```bash
if [ $arrayType != "SRAMModel" ]
then
        biasVoltageSA="650m" #Expressed in volts.
        echo "desVar(\"Vbias\" $biasVoltageSA)" >>$fileOCEAN
fi
```

In the code above, another design parameter is assigned for the logic arrays (CAM and LiM): the bias voltage of the sense amplifier (section 4.3). This is needed for the MLSA and ANDSA operation, which are blocks instantiated in the LiM and CAM architectures.

```bash
echo "envOption('analysisOrder  list(\"tran\"))
saveOption('pwr \"all\")
temp($simulationTemp)
option( ?categ 'turboOpts 'preserveOption \"All\"  'numThreads
        \"$simulatorThreads\" 'mtOption  \"Manual\"
        'apsplus  t 'uniMode  \"APS\" )
run()
openResults(\"$pathToResultsDir/psf\")
selectResult('tran)" >> $fileOCEAN
```

At the end, other simulation options are set, such as the CPU threads assigned to the simulation.

The measurements script

In the following, the code of the `computeAndSaveMeasurements.sh` script is presented.

```bash
#! /bin/bash

#### FUNCTIONS ####
source $pathToThisProgram/OCEANGenerators/measFunctions.sh
```

In the code above, the functions used to perform the measurements are called in the script environment.

```bash
#### MAIN ####

delayList=("Write1Delay"
"Write0Delay"
"Read1Delay"
"Read0Delay"
"MatchDelay"
"ANDDelay")

energyList=("Write1Energy"
"Write0Energy"
"SRAMPrechargeEnergy"
"Read1Energy"
"Read0Energy"
"CAMPrechargeEnergy"
"MatchEnergy"
"MismatchEnergy"
"ANDPrechargeEnergy"
"AND1Energy"
"AND0Energy"
"ANDDisableEnergy"
"TotalEnergy")
```

Here, the measurements to be performed are defined: for each operation, a delay and an energy consumption variable are created.

```bash
# Then, the delay expressions are written to the OCEAN script.
delayExpressions=("$(delay 21 DataLastCol 0.46 rising)" #Write1Delay
"$(delay 25 DataLastCol "0.46" falling)" #Write0Delay
"$(delay 23 SA0n "0.46" falling)" #Read1Delay
"$(delay 27 SA0 "0.46" falling)" #Read0Delay
6  "$(delay 2 MLSAO "0.46" rising)" #MatchDelay
7  "$(delay 6 ANDSAO "0.46" rising)") #ANDDelay
8
9  numDelays=${#delayList[*]}
10
11 for ((i=0; i<$numDelays; ++i))
12 do
13     echo ${delayList[$i]}=${delayExpressions[$i]} >> $fileOCEAN
14 done
15 echo "" >> $fileOCEAN
```

Here, the delay functions are called (`#2` to `#7`) and written to the OCEAN script (`#11` to `#15`). The delay function definition is the following:

```
# Function that computes the delay of a signal with respect to a reference time.
delay(){
    local referenceTime=$1
    local signal="$2"
    local threshold="$3"
    local edgeType="$4"

    echo delay(?td1 $(($referenceTime*$ckPeriod))n ?wf2 v(\"$signal\") ?value2 $threshold ?nth2 1 ?edge2 \'$edgeType)

    return
}
```

The delay is computed with respect the beginning the of the operation cycle (`referenceTime`). For the signal, one has to provide the name, the type of edge to be sensed (falling or rising) and the threshold to which the delay computation is referred (in this case, a threshold equal to $V_{DD}/2 = 0.46\,\mathrm{V}$ is used to compute the 50% delay).

```
# The energy expressions are written to the OCEAN script.
energyExpressions=("$(energy 21 22)" #Write1Energy
"$(energy 25 26)"  #Write0Energy
"$(energy 22 23)" #SRAMPrechargeEnergy
"$(energy 23 24)" #Read1Energy
"$(energy 27 28)" #Read0Energy
"$(energy 3 4)" #CAMPrechargeEnergy
"$(energy 2 3)" #MatchEnergy
"$(energy 4 5)" #MismatchEnergy
"$(energy 7 8)" #ANDPrechargeEnergy
11 "$(energy 6 7)" #AND1Energy
12 "$(energy 17 18)" #AND0Energy
13 "$(energy 8 9)" #ANDDisableEnergy
14 "$(energy 0 28)") #TotalEnergy
15
16 numEnergies=${#energyList[*]}
17
18 for ((i=0; i<$numEnergies; ++i))
19 do
20     echo "${energyList[$i]}=${energyExpressions[i]}" >> $fileOCEAN
21 done
22 echo "" >> $fileOCEAN
```

Here, the energy functions are called and their outputs are assigned to the corresponding outputs (#2 to #14). Then, the expressions are written to the OCEAN script (#18 to #21).

The energy function definition is the following:

```
# Function that integrates the instantaneous power along a simulation cycle, obtaining the corresponding energy.
energy(){
    local startCycle=$1
    local endCycle=$2

    echo integ(getData(\":pwr\" ?result \"tran\") $(($startCycle*$ckPeriod))n $(($endCycle*$ckPeriod))n)
}
```

The function performs in OCEAN a numerical integration of the array instantaneous power along the operation cycle obtaining, in this way, the array energy consumption in the considered cycle.

```
# A file is opened in OCEAN to save the results of the simulations.
echo "fileToWrite=outfile(\"$pathToResultsDir/results$scriptName.txt\")" >> $fileOCEAN
echo "" >> $fileOCEAN

#The delay results are saved.
for ((i=0; i<$numDelays; ++i))
do
    echo fprintf(fileToWrite \"%s = %e\n\" \"${delayList[$i]}\" ${delayList[$i]}) >> $fileOCEAN
done
```

```
echo "" >> $fileOCEAN

# The energy results are saved.
for ((i=0; i<$numEnergies; ++i))
do
    echo fprintf(fileToWrite \"%s = %e\n\" \"${energyList[$i]}\"
    ↪  ${energyList[$i]}) >> $fileOCEAN
done
echo "" >> $fileOCEAN
echo "close(fileToWrite)" >> $fileOCEAN
```

At the end, the code needed to save the measurements in an external file (`results.txt` in Figure 6.2) is written to the OCEAN script.

The waveforms script

In the following, the code of the `plotAndSaveWaveforms.sh` script is presented.

```
#! /bin/bash

signalList=("WLFirstRow"
"BLFirstCol"
"BLnFirstCol"
"DataFirstCol"
"DataLastCol"
"BLLastCol"
"BLnLastCol"
"EnableBLDriver"
"Precharge_n"
"SAO"
"SAOn"
"EnableSA"
"ANDOut"
"MLOut"
"MLSAO"
"ANDSAO"
"EnableMLSAn"
"EnableANDSAn"
"dummyANDSAO"
"dummyMLSAO"
"dummyMLOut"
"dummyANDLineOut"
"DisableFirstCol"
"WLDriverIn"
"BLDriverIn"
"BLnDriverIn"
"SADelayCircuitIn")
```

Here, the testbench signals names are defined in order to select the signals in the simulation for the waveforms plot.

```
# The waveforms are exported in CSV format.
numSignals=${#signalList[*]}
for ((i=0; i<$numSignals; ++i))
do
    echo ocnPrint( ?output
    ↪  \"$pathToResultsDir/waveformCSV/${signalList[$i]}.csv\" ?numberNotation
    ↪  \'engineering v(\"${signalList[$i]}\") ?from 0 ?to $simulationTime
    ↪  ?step $simulationStep) >> $fileOCEAN
done
```

In the code above, each waveform of the testbench in exported in a CSV file in the results directory.

```
if [ $plotWf -eq 1 ]
then

        echo "newWindow()
        addTitle(\"Other signals\")" >> $fileOCEAN
        #We plot the signals.
        for ((i=0; i<$numSignals; ++i))
        do
            echo "plot(v(\"${signalList[$i]}\") ?expr \"${signalList[$i]}\")"
            ↪  >> $fileOCEAN
        done
        echo "" >> $fileOCEAN

        echo "newWindow()
        currentWindow(2)
        addTitle(\"CAM behaviour\")" >> $fileOCEAN
        CAMSignals=("MLSAO" "dummyMLOut" "dummyMLSAO" "MLOut" "EnableMLSAn"
        ↪  "DataFirstCol" "BLFirstCol" "BLnFirstCol")
        for i in ${CAMSignals[@]}
        do
            echo "plot(v(\"$i\") ?expr \"$i\")" >> $fileOCEAN
        done
        echo "" >> $fileOCEAN

        echo "newWindow()
```

```
    currentWindow(3)
    addTitle(\"AND behaviour\")" >> $fileOCEAN
    ANDSignals=("ANDSAO" "dummyANDLineOut" "dummyANDSAO" "ANDOut"
    ↪  "EnableANDSAn" "DataFirstCol" "BLFirstCol" "BLnFirstCol")
    for i in ${ANDSignals[@]}
    do
        echo "plot(v(\"$i\") ?expr \"$i\")" >> $fileOCEAN
    done
    echo "" >> $fileOCEAN

    echo "newWindow()
    currentWindow(4)
    addTitle(\"SRAM behaviour\")" >> $fileOCEAN
    ANDSignals=("SAO" "SAOn" "DataLastCol" "BLLastCol" "BLnLastCol"
    ↪  "EnableBLDriver" "EnableSA" "BLDriverIn" "BLDriverIn" "WLFirstRow")
    for i in ${ANDSignals[@]}
    do
        echo "plot(v(\"$i\") ?expr \"$i\")" >> $fileOCEAN
    done

fi
```

Here, different windows are defined in the Virtuoso environment. Each window contains the signals peculiar to a functionality of the array.

```
    echo "newWindow()
    currentWindow(2)
    addTitle(\"CAM behaviour\")" >> $fileOCEAN
    CAMSignals=("MLSAO" "dummyMLOut" "dummyMLSAO" "MLOut" "EnableMLSAn"
    ↪  "DataFirstCol" "BLFirstCol" "BLnFirstCol")
    for i in ${CAMSignals[@]}
    do
        echo "plot(v(\"$i\") ?expr \"$i\")" >> $fileOCEAN
    done
```

For example, here the signals corresponding to the CAM functionality, such as the match-line voltage `MLOut` and the MLSA output `MLSAO`, are plotted in a dedicated window.

It has to be noticed that the waveforms are plotted only if the variable `plotWf` is equal to 1, as shown in the following code line:

```
if [ $plotWf -eq 1 ]
then
```

This variable is used to allow the user to decide if the waveforms have to be plotted during the simulation or if only the final results have to be provided in the previously defined file formats (Figure 6.2).

6.2.3 Input stimuli generator

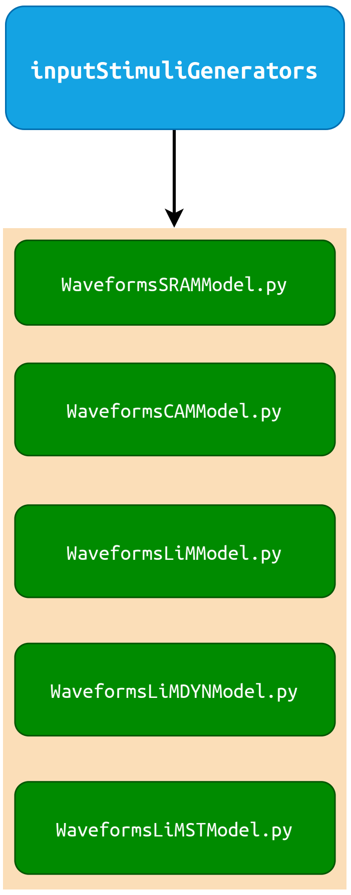


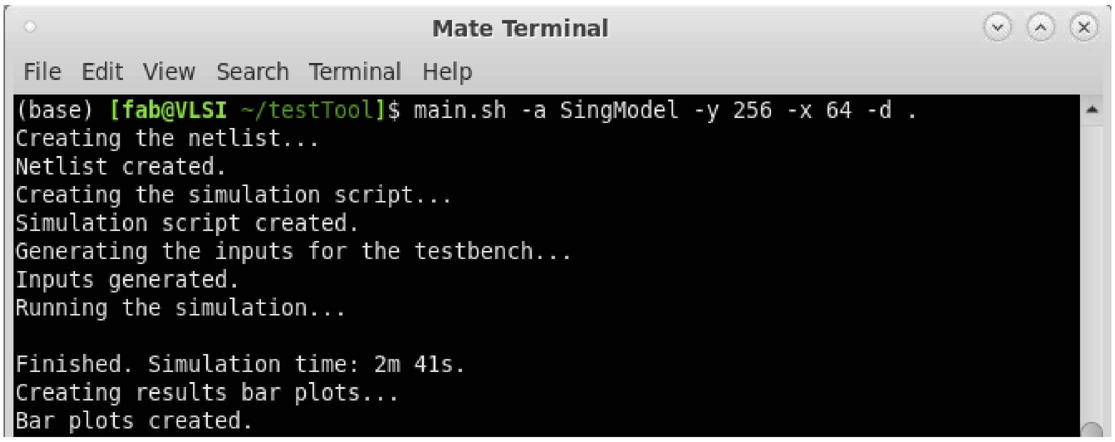
Figure 6.19: The input stimuli generator structure.

The input stimuli generator consists in a series of Python scripts, each of which is dedicated to a specific array type: SRAM, CAM, LiM with dynamic cell, LiM with static cell and LiM with special-purpose cell. These scripts generate the waveform files needed by the SPECTRE simulator to perform the simulation.

The code inside each script is identical to the ones described in the previous chapters (subsection 3.5.2, subsection 4.6.2, subsection 5.8.3).

6.3 Results provided by ALiAS

As explained in section 6.1, ALiAS provides the results of the simulation in three file formats: CSV file for the simulation waveforms, text file and bar diagrams for the array performance.



```
(base) [fab@VLSI ~/testTool]$ main.sh -a SingModel -y 256 -x 64 -d .
Creating the netlist...
Netlist created.
Creating the simulation script...
Simulation script created.
Generating the inputs for the testbench...
Inputs generated.
Running the simulation...

Finished. Simulation time: 2m 41s.
Creating results bar plots...
Bar plots created.
```

Figure 6.20: Example of ALiAS run.

In Figure 6.20, an example of ALiAS run is shown. In this case, the LiM model with the special-purpose cell, `SingModel`, is simulated. The array width is 64 bits, while the height is 256 rows.

The text file generated follows the following naming rule:

`resultsarrayTypeheightxwidth.txt`

In this case, the results file name is `resultsSingModel256x64.txt`. The file content is shown in the following:

```
Write1Delay = 1.784846e-10
Write0Delay = 1.392253e-10
Read1Delay = 2.582827e-10
Read0Delay = 2.507826e-10
MatchDelay = 3.005608e-10
ANDDelay = 2.605585e-10
Write1Energy = 1.081835e-13
Write0Energy = 9.124491e-14
SRAMPrechargeEnergy = 1.213535e-13
Read1Energy = 7.103484e-14
Read0Energy = 6.455612e-14
CAMPrechargeEnergy = 5.932526e-14
MatchEnergy = 6.704943e-14
MismatchEnergy = 6.944946e-14
SingPrechargeEnergy = 5.897603e-14
AND1Energy = 6.355464e-14
AND0Energy = 7.157836e-14
ANDDisableEnergy = 6.363723e-14
TotalEnergy = 2.333884e-12
```

In the lines from `#1` to `#6`, the delays correspondent to each memory operation are listed; in the rest of the file, the energy consumption associated to each operation, together with the total energy, are presented.

From this file, two bar diagrams are derived.

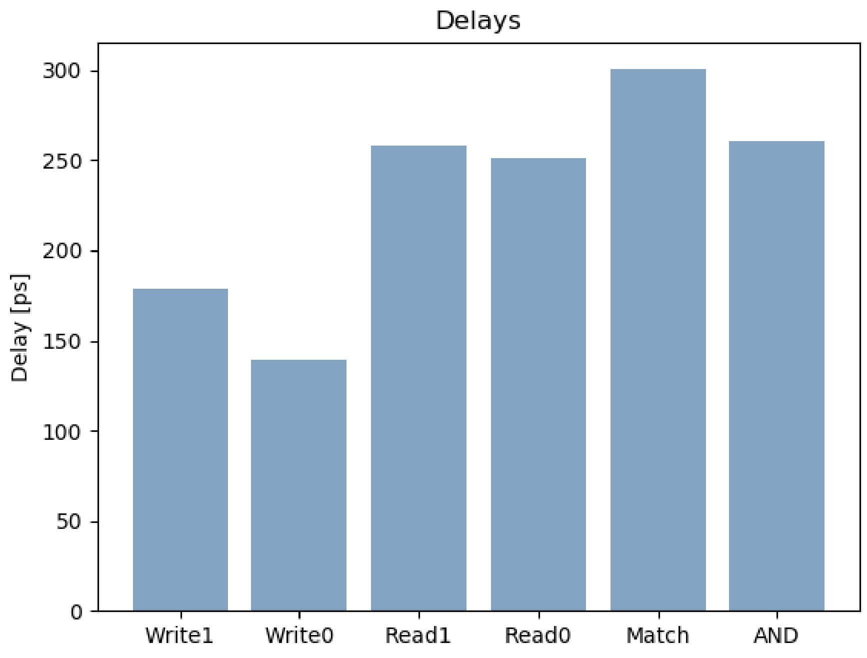


Figure 6.21: Performance bar diagram.

In Figure 6.21, the bar diagram related to the array performance is shown. For each operation, the correspondent delay is shown, in order for these to be compared.

In Figure 6.22, the diagram related to the array power consumption is presented. For each operation, the correspondent power consumption, obtained starting from the energy results stored in the results file and the clock period specified by the user for the simulation, is shown.

The CSV files generated by ALiAS can be used to plot outside Virtuoso the simulation waveforms. An example is provided in Figure 6.23 for the simulation considered above. In this, the waveforms relative to the AND operations have been extracted.

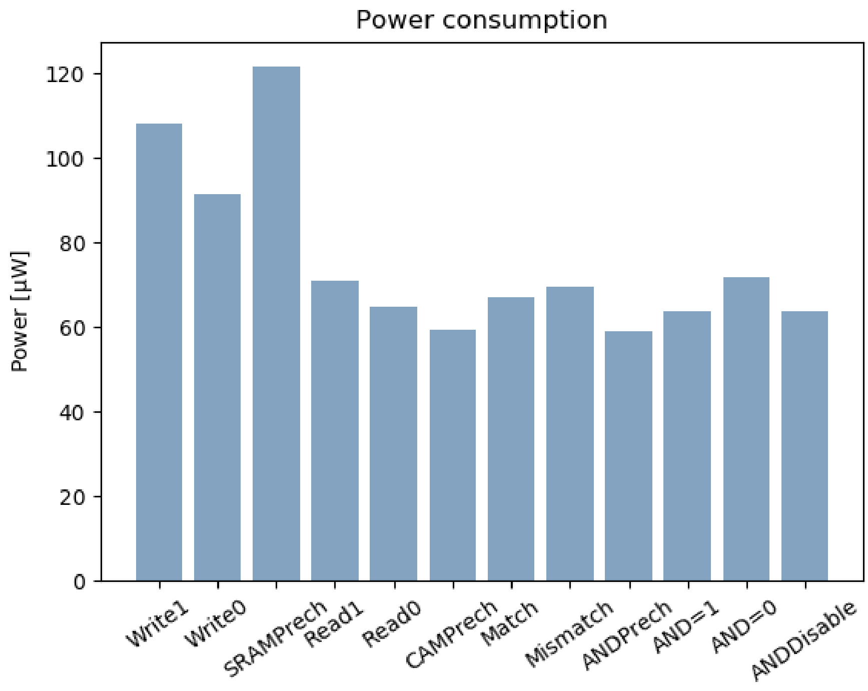


Figure 6.22: Power consumptions bar diagram.

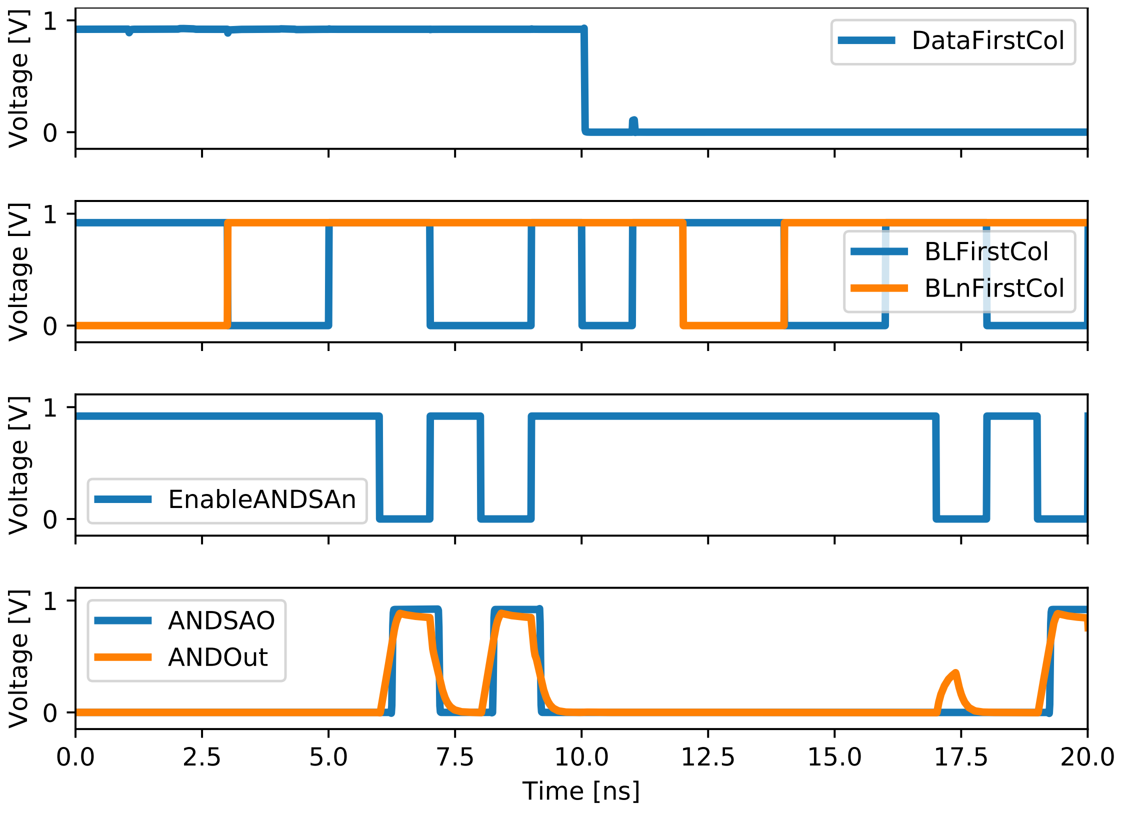


Figure 6.23: Waveforms extracted from the ALiAS run.

6.4 User guide

In this section, an user guide for ALiAS is provided.

The syntax to run a simulation in ALiAS is shown in Figure 6.24

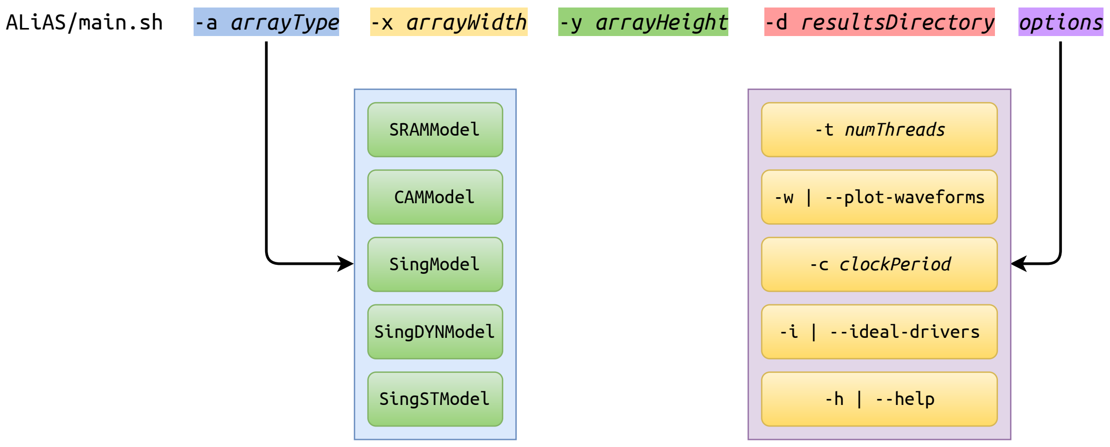


Figure 6.24: ALiAS execution syntax.

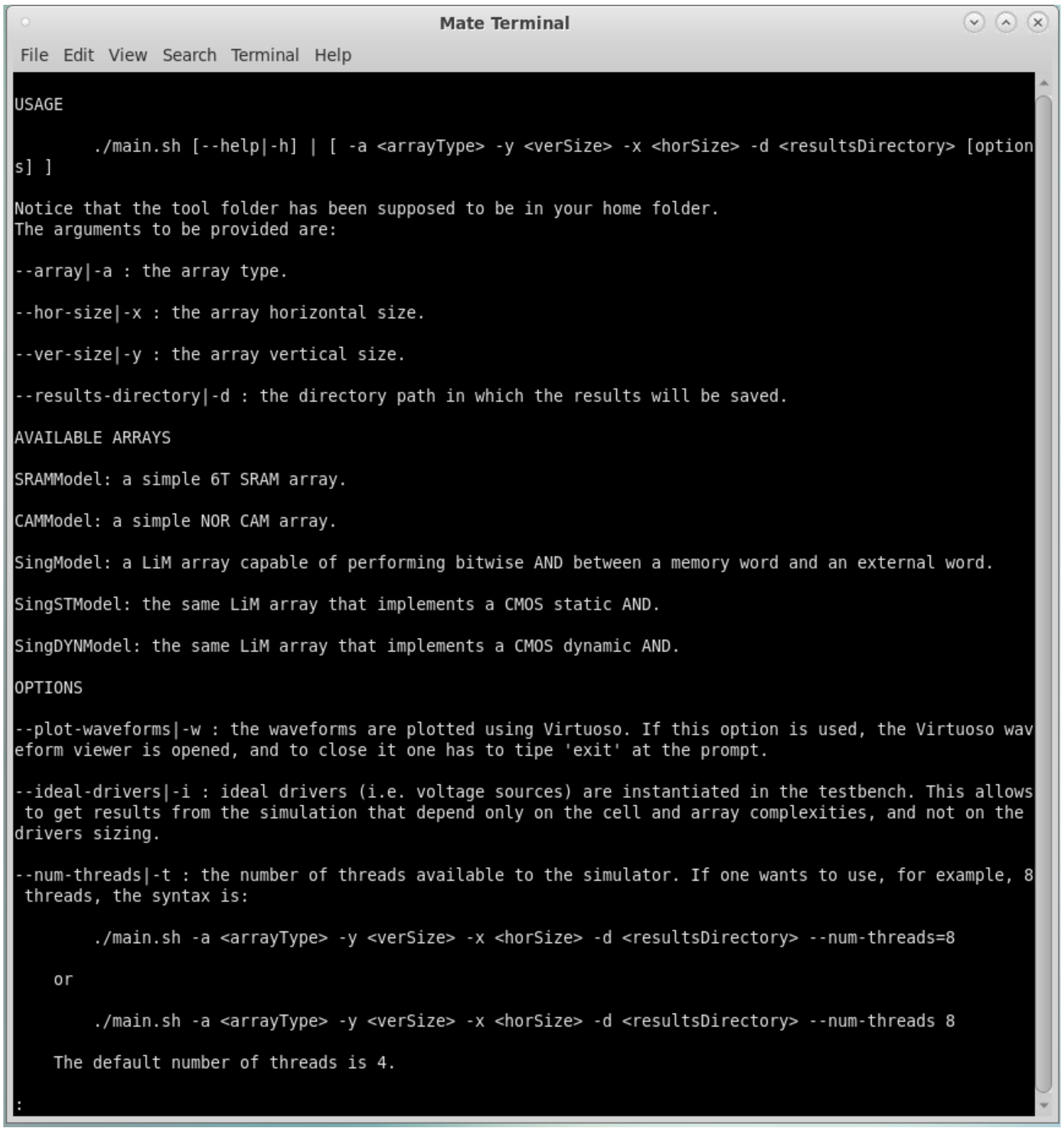
First, one has to provide the array type to be simulated, choosing between 5 arrays: `SRAMModel`, `CAMModel` and the three LiM flavors, which are `SingModel`, `SingDYNModel` and `SingSTModel`; the array name has to be preceded by the `-a` (or `--array`) flag. Second, one has to provide the array size in terms of height (number of stored words), preceded by `-y` or `--ver-size`, and width (parallelism of each word), preceded by `-x` or `--hor-size`. Third, one has to provide the path to the directory in which the results will be saved, preceded by `-d` or `--results-directory`.

Then, the user can specify some options:

- `-h`, `--help`: an help menu is displayed.
- `-i`, `--ideal-drivers`: the CMOS drivers are substituted with ideal voltage sources, so that the drivers parameters have no influence on the array performance.
- `-c`, `--ck-period`: the user can specify an integer number as clock period in nanoseconds to be used in the simulations. The minimum and default value is 1 ns. For example, if one wants to use a clock period of 3 ns:

```
ALiAS/main.sh -a SingModel -y 256 -x 64 -d . -c 3
```

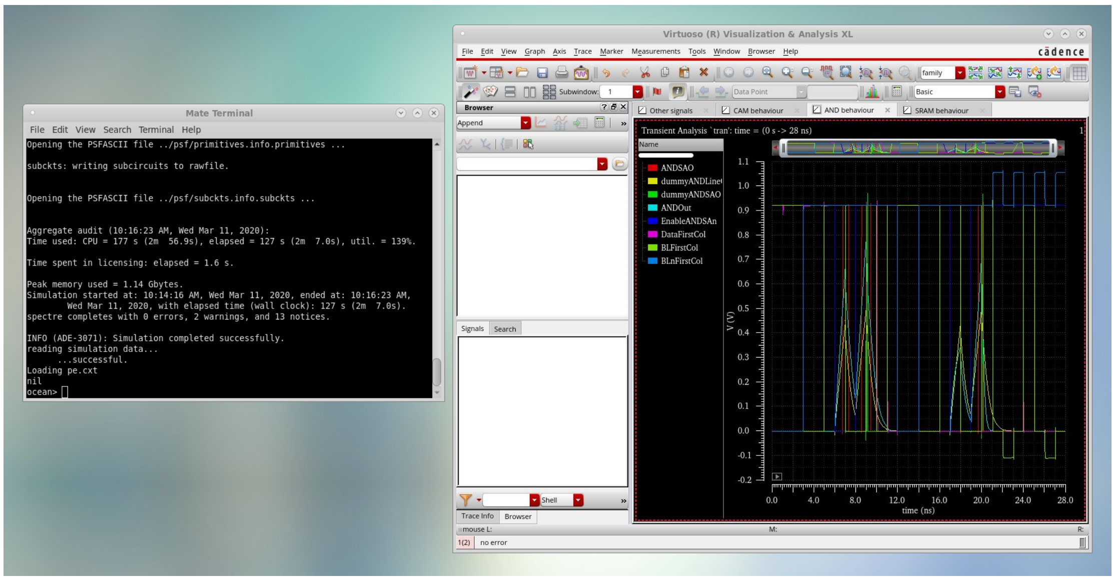
- `-w`, `--plot-waveforms`: specifying this option, the simulation waveforms are plotted in the Cadence Virtuoso environment. An example of simulation is shown in Figure 6.26. To exit the Virtuoso environment, one has to type `exit` at the terminal prompt. In this way, Virtuoso is exited and the simulation is ended.



```
USAGE

        ./main.sh [--help|-h] | [ -a <arrayType> -y <verSize> -x <horSize> -d <resultsDirectory> [option
s] ]

Notice that the tool folder has been supposed to be in your home folder.
The arguments to be provided are:

--array|-a : the array type.

--hor-size|-x : the array horizontal size.

--ver-size|-y : the array vertical size.

--results-directory|-d : the directory path in which the results will be saved.

AVAILABLE ARRAYS

SRAMModel: a simple 6T SRAM array.

CAMModel: a simple NOR CAM array.

SingModel: a LiM array capable of performing bitwise AND between a memory word and an external word.

SingSTModel: the same LiM array that implements a CMOS static AND.

SingDYNModel: the same LiM array that implements a CMOS dynamic AND.

OPTIONS

--plot-waveforms|-w : the waveforms are plotted using Virtuoso. If this option is used, the Virtuoso wav
eform viewer is opened, and to close it one has to tipe 'exit' at the prompt.

--ideal-drivers|-i : ideal drivers (i.e. voltage sources) are instantiated in the testbench. This allows
 to get results from the simulation that depend only on the cell and array complexities, and not on the
drivers sizing.

--num-threads|-t : the number of threads available to the simulator. If one wants to use, for example, 8
 threads, the syntax is:

        ./main.sh -a <arrayType> -y <verSize> -x <horSize> -d <resultsDirectory> --num-threads=8

    or

        ./main.sh -a <arrayType> -y <verSize> -x <horSize> -d <resultsDirectory> --num-threads 8

    The default number of threads is 4.

:
```

Figure 6.25: The help menu.

- `-t`, `--num-threads`: using this option, one can choose the number of CPU threads to be assigned to simulation. The default number of threads is 4. The SPECTRE simulation engine usually occupies no more than 3 threads per simulation.



```
Opening the PSFASCII file ../psf/primitives.info.primitives ...

subckts: writing subcircuits to rawfile.


Opening the PSFASCII file ../psf/subckts.info.subckts ...


Aggregate audit (10:16:23 AM, Wed Mar 11, 2020):
Time used: CPU = 177 s (2m  56.9s), elapsed = 127 s (2m  7.0s), util. = 139%.

Time spent in licensing: elapsed = 1.6 s.

Peak memory used = 1.14 Gbytes.
Simulation started at: 10:14:16 AM, Wed Mar 11, 2020, ended at: 10:16:23 AM,
        Wed Mar 11, 2020, with elapsed time (wall clock): 127 s (2m  7.0s).
spectre completes with 0 errors, 2 warnings, and 13 notices.

INFO (ADE-3071): Simulation completed successfully.
reading simulation data...
      ...successful.
Loading pe.cxt
nil
ocean> 
```

Figure 6.26: Example of simulation with the `-w` option.

Chapter 7

Results and conclusions

In this chapter, the results relative to the simulations performed in ALiAS are discussed.

The performance of each memory operation are compared among the arrays. In particular, each memory architecture has been simulated varying the array width and height, using both ideal and non-ideal drivers in the simulations, to characterize delay and energy consumption of each operation.

A first set of simulations has been performed fixing the memory height to 256 bits and varying the width from 8 to 144 bits. First, ideal drivers have been used; second, the drivers synthesized by ALiAS are included in the simulations, in order to evaluate their influence on the array performance.

Then, a second set of simulations has been performed fixing the array width to 32 bits and varying the array height from 128 to 512 bits. Also in this case, both ideal and non-ideal drivers have been used for the simulations.

For what concerns the simulation setups, the following ones have been used.

Width simulations parameters	
V_{DD}	0.92 V
$C_{parasitic}$	100 aF
$R_{parasitic}$	5 Ω
T_{ck}	1 ns

Table 7.1: Simulation parameters adopted for the array width simulations.

Height simulations parameters	
V_{DD}	0.92 V
$C_{parasitic}$	100 aF
$R_{parasitic}$	5 Ω
T_{ck}	2 ns

Table 7.2: Simulation parameters adopted for the array height simulations.

In Table 7.1 and Table 7.2, the simulation parameters adopted for the two set of simulations are presented.

A supply voltage $V_{DD} = 0.92$ V has been chosen in order to compare the results of the simulations with the ones provided by a commercial tool for VLSI synthesis that uses the same technology library (ST FD-SOI 28 nm).

For what concerns the parasitic values, used for the RC circuits instantiated in the arrays, small values have been chosen in order for their presence to be taken into account in the simulations, but also to allow the simulations results to be mainly determined by the arrays characteristics.

For the simulation clock cycle, $T_{ck} = 1\,\text{ns}$ has been chosen for the simulations in which the width is varying, while $T_{ck} = 2\,\text{ns}$ has been chosen for the height ones, in which very large arrays are instantiated. Of course, the critical path is lower than these clock cycles but, however, some room has been left in the simulations in which very large arrays are instantiated, like the height ones.

7.1 Simulations performed varying the array width

In this section, the simulations performed varying the array width from 8 to 144 bits are considered. The array height, instead, is fixed to 256 rows. In the following, each operation is analyzed in detail.

7.1.1 Read operation: ideal drivers

In this section, the reaad operation is considered, analyzing the results of the simulations performed using ideal drivers.

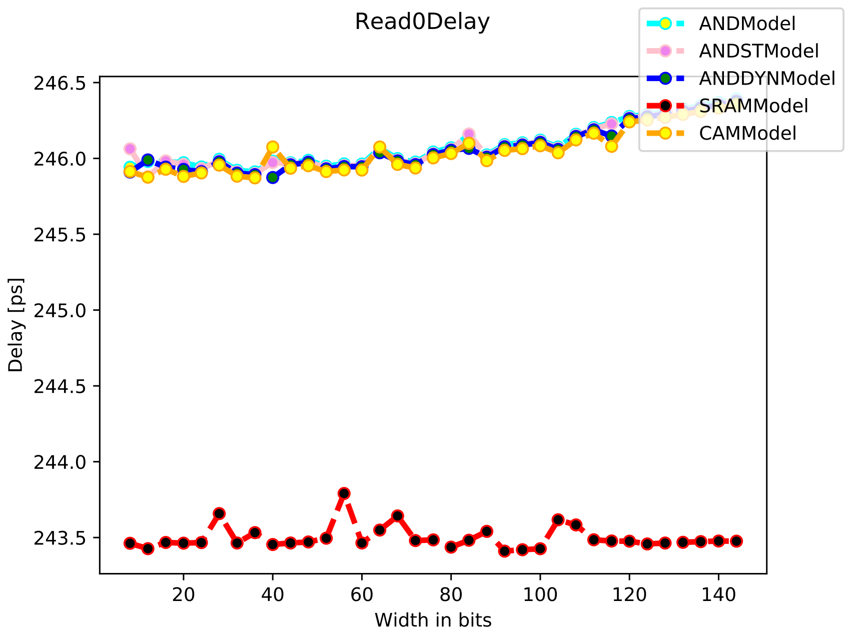


Figure 7.1: Read operation delay: cell content equal to ‘0’, ideal drivers.

In Figure 7.1, each curve represents an array type. On the vertical axis, the delay associated to the read operation of a logic ‘0’ as function of the array width, is reported.

It can be noticed how the read delay is practically constant with respect the memory width. This is due to the fact that the array height is much larger than the width, as it happens in standard memory arrays; hence, it has a stronger influence on the read delay, since the sense amplifier design is heavily influenced by the array height value, as discussed in section 6.2.1. This results in a practically constant delay associated to the read operation.

One can notice that the read delay of the SRAM array (`SRAMModel`) is lower than the one associated to the CAM (`CAMModel`) and LiM (`ANDModel`, the LiM array with the special-purpose cell discussed in section 5.6; `ANDSTModel`, the LiM array with the static cell presented in Figure 5.12; `ANDDYNModel`, the LiM array with the dynamic cell discussed in section 5.4) arrays. This is due to the fact the logic memory cells (CAM and LiM) have a larger capacitive load on the bitlines with respect the SRAM cell, as it can be verified in Figure 7.3.

However, the difference in the delay is not so large, since the sense amplifier design mostly determines the read delay value with the delay associated to its enable signal (section 6.2.1), which determines the time required by the SA to read its inputs.

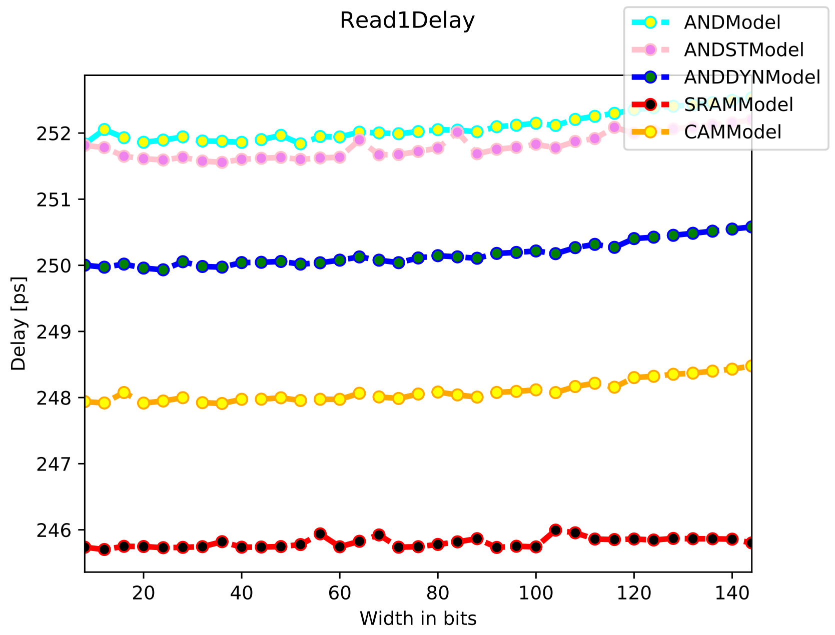


Figure 7.2: Read operation delay: cell content equal to ‘1’, ideal drivers.

In Figure 7.2, the delay associated to the read operation of a logic ‘1’ is shown.

In this case, a distinction between the arrays values is observed. The reason of this can be found referring to the graph shown in Figure 7.3, which is proposed in the following.

In Figure 7.3, it can be noticed how the capacitance associated to the bitlines is larger for all the LiM implementations (`STAT`, `DYN` and `SP`) than for the others. Hence, the resulting read delay is larger. As in the previous case, the difference is not so large, since the read delay is practically determined by the sense amplifier design.

In Figure 7.4, the energy associated to the read operation of a logic ‘0’ is reported.

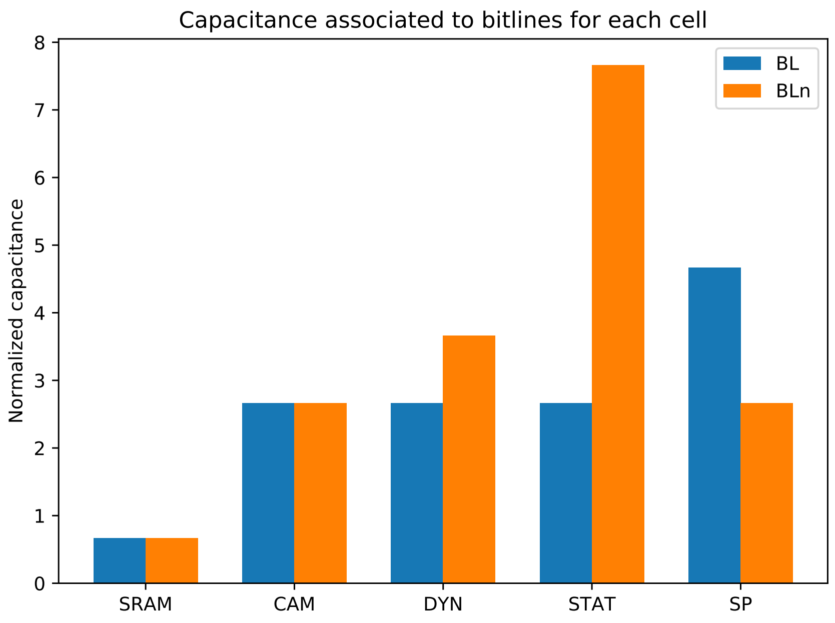


Figure 7.3: Parasitic capacitance comparison among the cells.

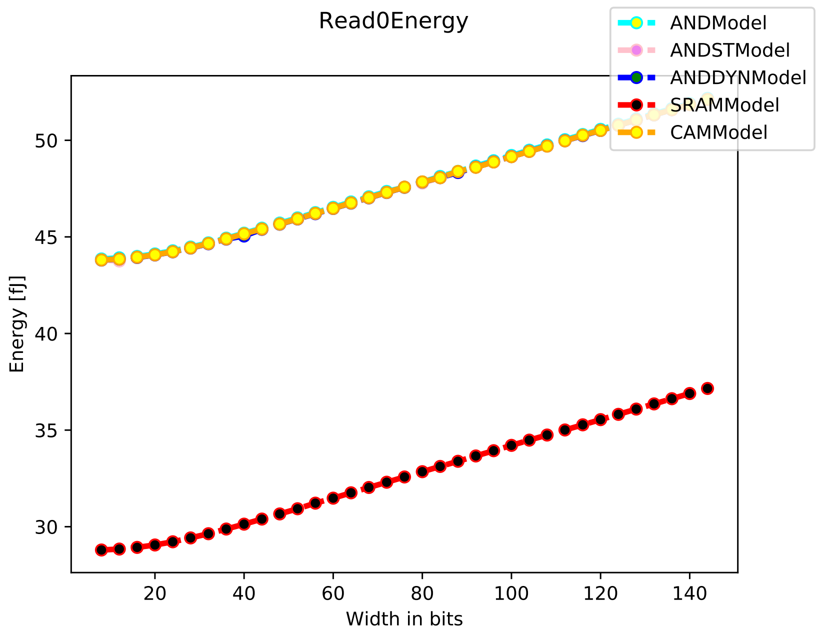


Figure 7.4: Read operation energy: cell content equal to '0', ideal drivers.

It can be noticed how the energy consumption increases linearly with the memory width, as one would expect since only one dimension of the array is varying.

One can also notice how the energy associated to the more complex arrays, CAM and LiM, is larger with respect to the SRAM implementation. Being more complex the cell and the architecture, larger parasitic capacitances are involved and, so, a larger energy consumption results from the simulations.

Also, one can notice that the energy consumption of the CAM and LiM arrays are very similar. This is due to the fact that, during the reading of a logic ‘0’, the BL is discharged, while the $\overline{\text{BL}}$ is kept at V_{DD}. Since the LiM cells have most of their capacitive load concentrated on $\overline{\text{BL}}$ (Figure 7.3 and Figure 5.21), the difference in complexity between CAM and LiM arrays is not exploited.

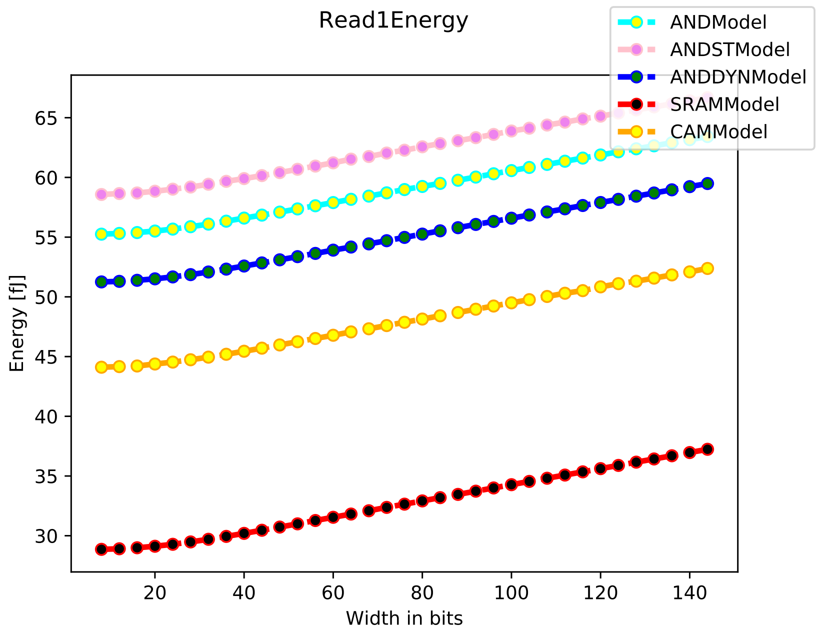


Figure 7.5: Read operation energy: cell content equal to ‘1’, ideal drivers.

In Figure 7.5, the energy associated to the read operation of a logic ‘1’ is reported. As in the read operation case (Figure 7.2), a distinction can be observed between the logic arrays (CAM and LiM). In fact, the most complex cell (the static LiM one, `ANDSTModel`) is the one with the largest energy consumption associated. This is due to the fact that during the reading of a logic ‘1’, $\overline{\text{BL}}$ is discharged and, so, the difference in the cells complexity (i.e. capacitive load on bitlines) is exploited in the results.

7.1.2 Read operation: non-ideal drivers

In this section, the same quantities examined in subsection 7.1.1 are considered, but using non-ideal drivers in the simulations.

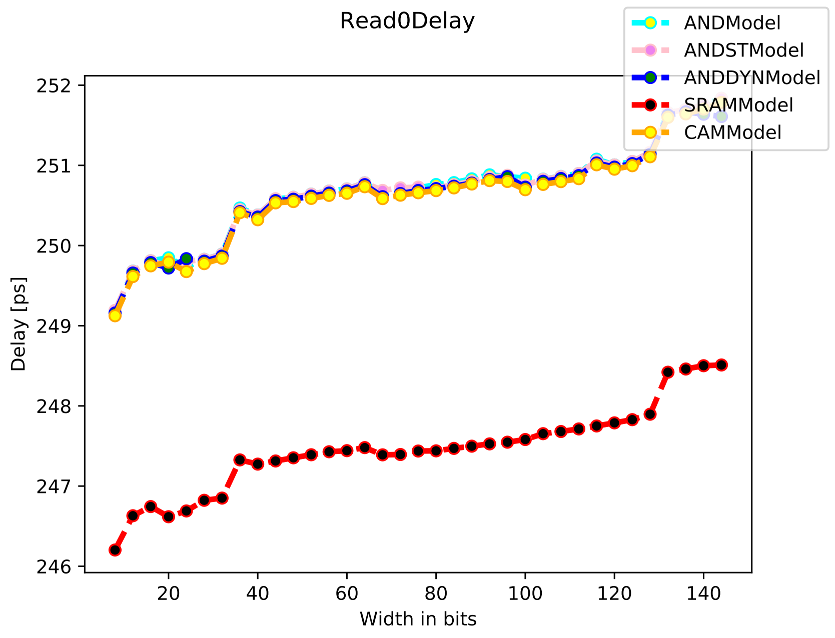


Figure 7.6: Read operation delay: cell content equal to ‘0’, non-ideal drivers.

In Figure 7.6, the read operation delay with respect to the memory width variation is considered for each array type.

It can be noticed how a step-like behavior is obtained in the simulations, differently from what is shown in Figure 7.1. This is due to the non-ideality of the wordline driver, that is synthesized by ALiAS according to the memory width value (section 6.2.1). To each step in the curve, the addition of inverter stages is associated, that leads to an increase in the delay value. The addition of driving stage allows to keep almost constant the value of the read delay, that would otherwise increase much more if the wordline driver strength was not adjusted accordingly to the row width value.

As in Figure 7.4, the delay associated to the SRAM architecture is lower than for the logic arrays, because of its higher simplicity. It can also be noticed how the row width has not a large influence on the read delay value, thanks to the fact that this is practically determined by the sense amplifier design.

In Figure 7.7, the read operation of a logic ‘1’ is analyzed.

It can be noticed how all the delays are very similar while, on the contrary, there is a clear distinction between them in Figure 7.2. This is due to the fact that real drivers have been instantiated in the testbench and that the synthesis algorithm is rough. A better tuning of the driver synthesis algorithm would allow to obtains results similar to the ones in Figure 7.5.

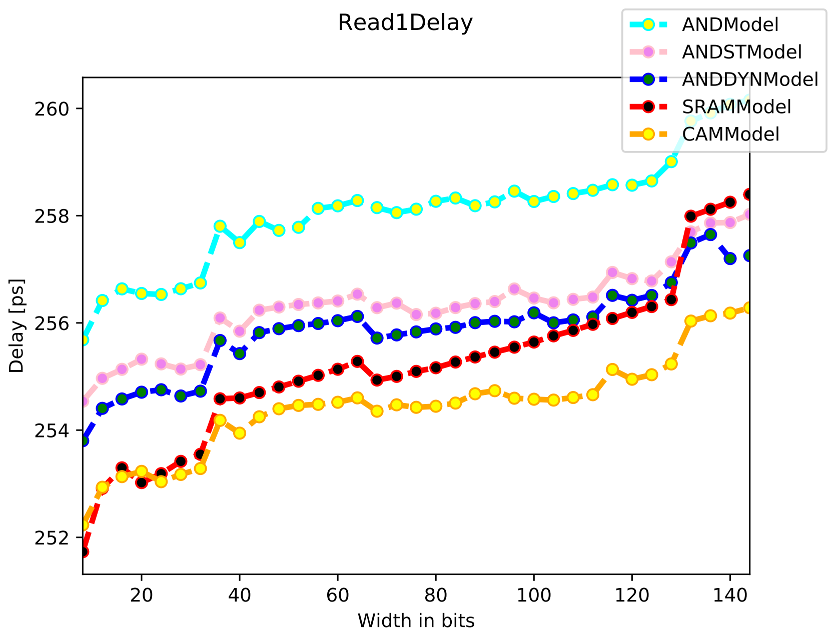


Figure 7.7: Read operation delay: cell content equal to ‘1’, non-ideal drivers.

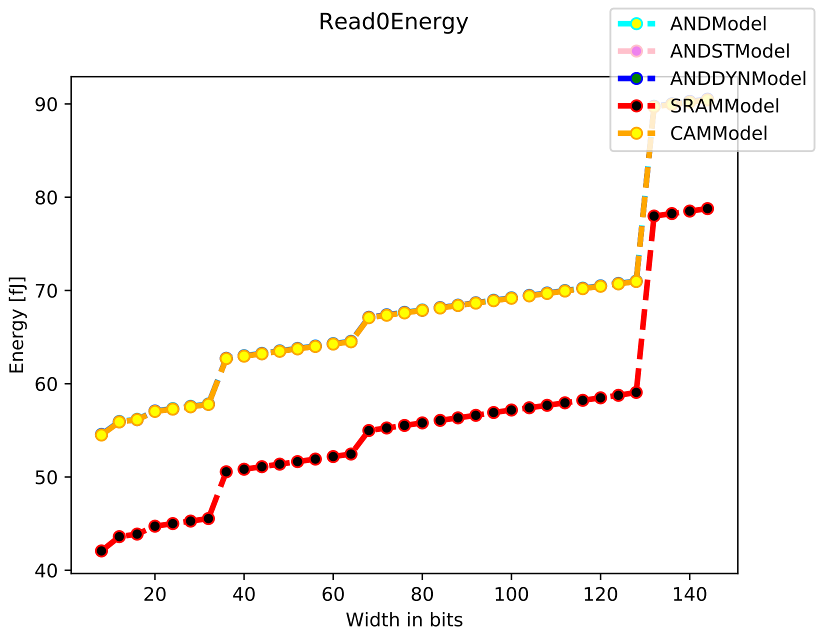


Figure 7.8: Read operation energy: cell content equal to ‘0’, non-ideal drivers.

In Figure 7.8, the energy consumption associated to the read operation of a logic ‘0’ is shown.

Also in this case, a step-behavior for the energy consumption can be observed. These steps are due to the addition of inverter stages in the driver chain, that leads to a sharp increase in the energy consumption of the array as the drivers strength is adjusted.

In Figure 7.9, a distinction in the energy values among the logic arrays can be

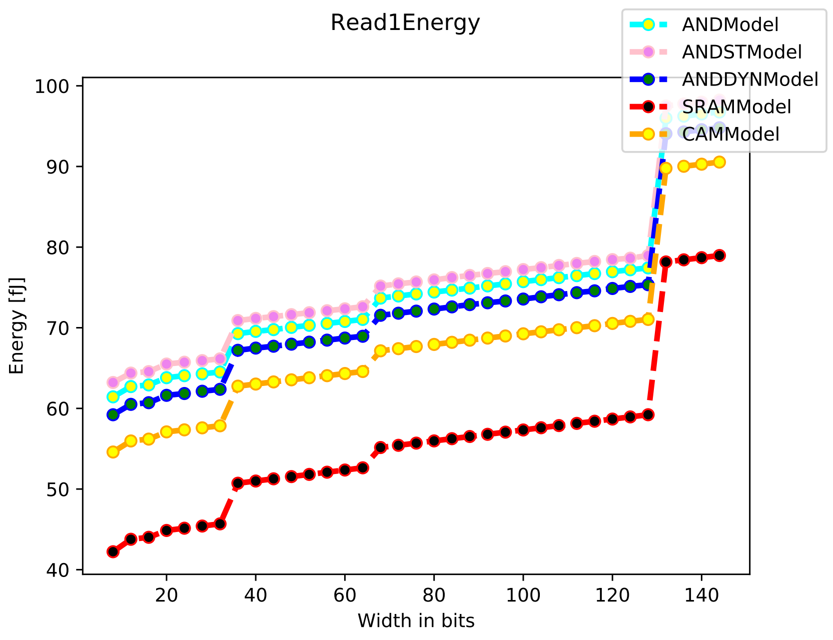


Figure 7.9: Read operation energy: cell content equal to ‘1’, non-ideal drivers.

observed, differently from what happens in Figure 7.8. As discussed for Figure 7.5, in this case the difference in the cells complexities is exploited because the negated bitline, $\overline{\mathrm{BL}}$, is discharged during the read operation and, so, the difference in the capacitive load on this bitline can be observed in the simulation results.

7.1.3 Write operation: ideal drivers

In this section the write operation is considered, analyzing the results of the simulations performed using ideal drivers.

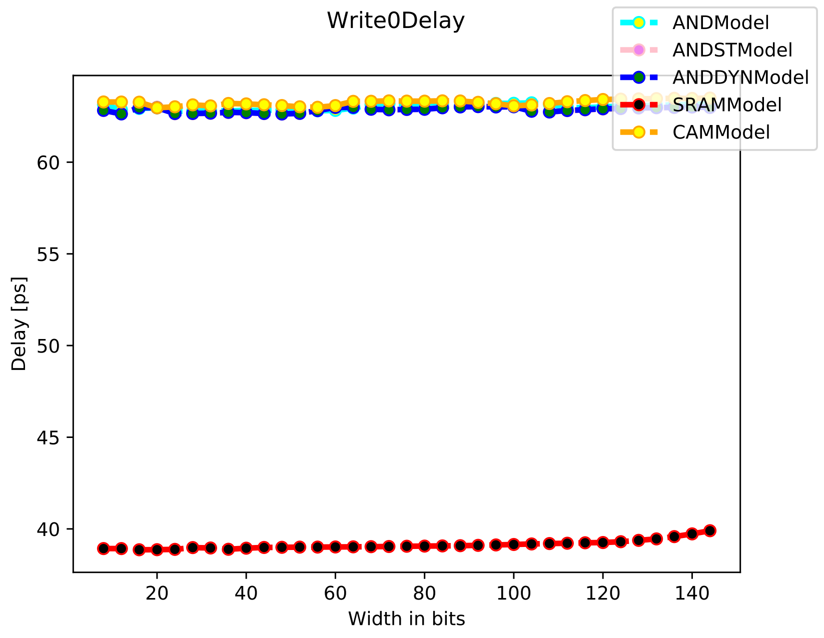


Figure 7.10: Write operation delay: cell content written to '0', ideal drivers.

In Figure 7.10, the write operation delay of a logic '0' is considered.

It can be noticed how the delay associated to the logic arrays (CAM and LiM) is larger than the SRAM one. This is due to the higher cell and array complexities of the logic arrays, that result in a slower operation: in fact, while in the read operation the performance are determined by the sense amplifier design, in the write operation the cell complexity (i.e. its capacitive load on the bitlines) has a large influence on the the write performance.

It can also be noticed how all the logic arrays have similar performance. This is due to the fact that a '0' is being written to the cell: since most of the capacitive load of the LiM cell is on $\overline{\text{BL}}$, as shown in Figure 7.3, and this bitline remains charged at V_{DD} during the writing of a logic '0', no difference can be observed between CAM and LiM arrays.

In Figure 7.10, the write operation delay of a logic '1' is considered.

In this case, a clear distinction between CAM and LiM arrays can be observed. In fact, writing a logic '1', the negated bitline $\overline{\text{BL}}$ is discharged to '0'; hence, since this is the most capacitively loaded line in the LiM arrays (Figure 7.3), these memory types perform worse than the CAM memory.

It can also be noticed how the delay associated to the writing of a logic '1' is larger than the one of a logic '0': in fact, a larger capacitive load on one of the bitlines leads

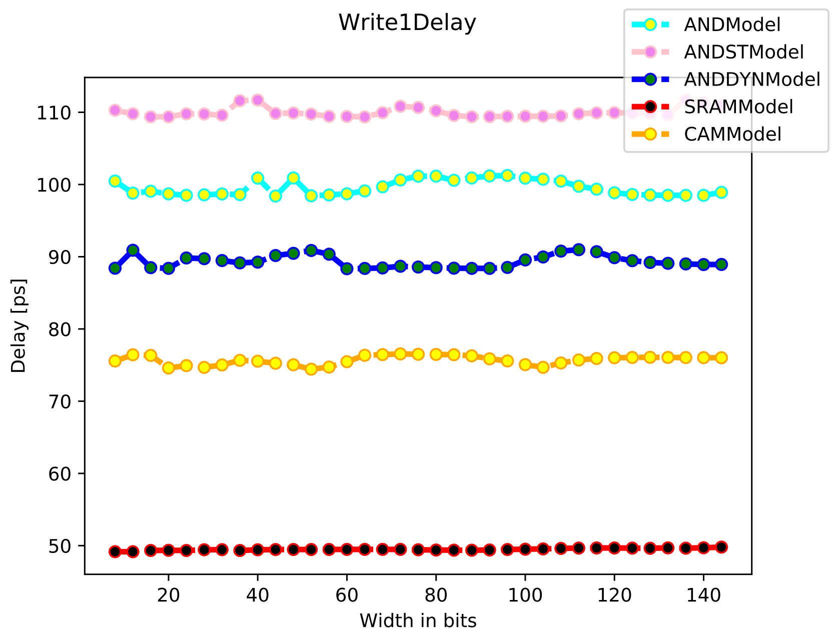


Figure 7.11: Write operation delay: cell content written to ‘1’, ideal drivers.

to an unbalance in the performance, which means that, depending on the data to the be written, delay and energy consumption change.

Between the LiM arrays, the one with the poorest performance (i.e. larger delay associated) is the static cell array `ANDSTModel`, which is also the most complex one. One can notice that the dynamic cell `ANDDYNModel` performs better than the special-purpose one `ANDModel`, even if the latter is simpler from a circuital point of view. This is due to the fact that the special-purpose cell has a transistor connected to $\overline{\text{BL}}$ that is larger than the one of dynamic cell one (Figure 5.21), which results in a larger capacitive load for the special-purpose array than the dynamic one (Figure 7.3).

In Figure 7.12, the write operation energy of a logic ‘0’ is considered.

As in Figure 7.10, one can notice that the logic arrays perform in a similar way. The same considerations discussed for the delay can be made: since during the writing of a logic ‘0’ the negated bitline $\overline{\text{BL}}$ is not discharged, no difference is observed in the measurements.

Of course, the SRAM array, which is the most simple one, is characterized by the lowest energy consumption.

In Figure 7.13, the write operation energy of a logic ‘1’ is considered.

As in Figure 7.11, it can be noticed how the energy consumptions of the CAM and LiM arrays differ. Also, the largest energy consumption is associated to the most complex array (`ANDSTModel`), and `ANDModel` performs worse than `ANDDYNModel`, having a larger energy consumption associated, because of its higher capacitive load (Figure 7.3).

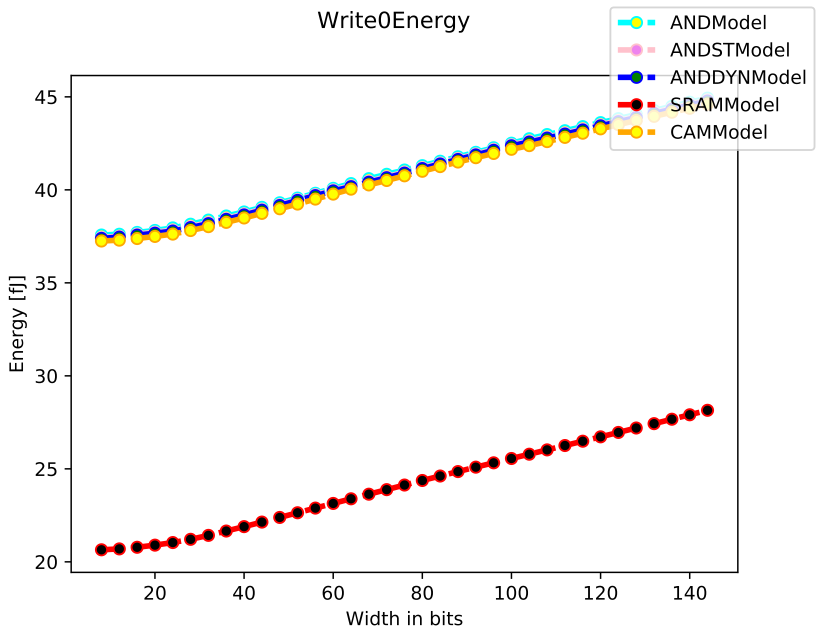


Figure 7.12: Write operation energy: cell content written to ‘0’, ideal drivers.

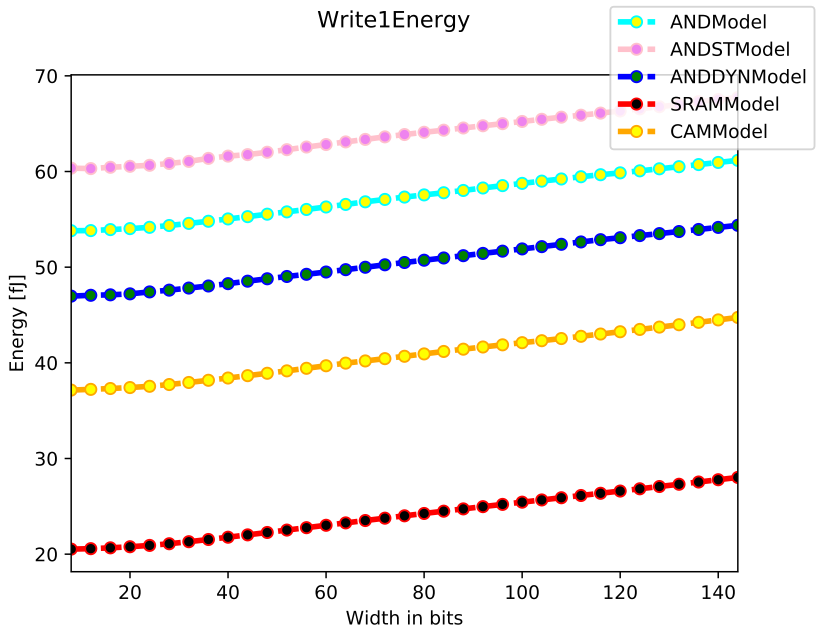


Figure 7.13: Write operation energy: cell content written to ‘1’, ideal drivers.

7.1.4 Write operation: non-ideal drivers

In this section, the write operation is considered, analyzing the results of the simulations performed the drivers synthesized by ALiAS.

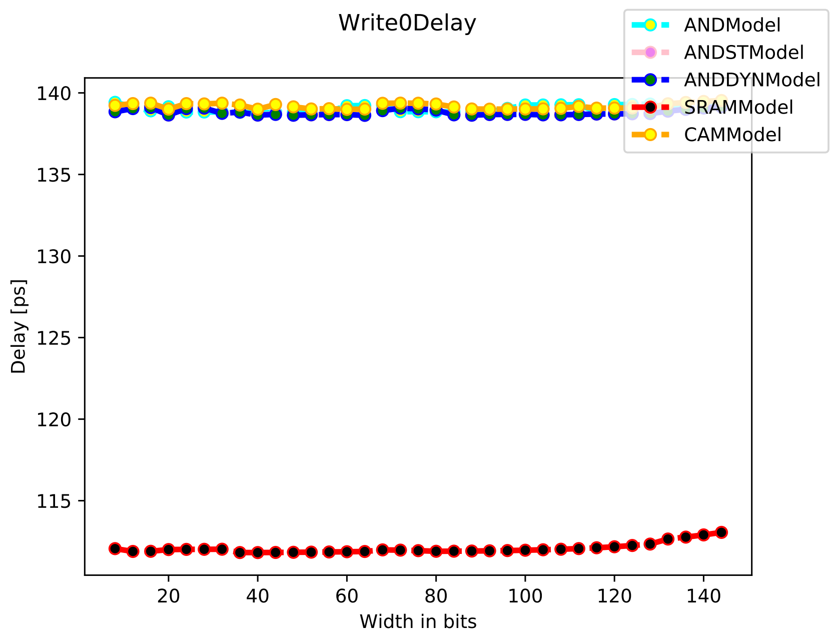


Figure 7.14: Write operation delay: cell content written to ‘0’, non ideal drivers.

In Figure 7.14, the write delay of a logic ‘0’ is considered.

As in the ideal drivers case, shown in Figure 7.10, the logic arrays present a much larger write delay with respect to the SRAM memory, because of their higher complexity. It can also be noticed how using non-ideal drivers the value of the write delay is much larger than in the ideal case: while in Figure 7.14 the value of the delay is around 140 ps for the logic arrays, in Figure 7.10 the delay value results to be around 65 ps.

One can also observe that the delay is kept constant thanks to the adaptive synthesis performed by ALiAS: in fact, the drivers strength is adjusted accordingly the memory width; furthermore, no variations are observed in the delay value since the array height (256 rows) is still much larger than the width, that ranges between 8 and 144 bits.

In Figure 7.15, the write delay of a logic ‘1’ is considered.

It can be noticed how the logic arrays values are now well distinguishable, thanks to their different capacitive load on the bitlines, as discussed for Figure 7.11. Also in this case, the delay values are much larger than in the ideal case, because of the non-idealities of the bitline and wordline drivers.

In Figure 7.16, the write energy of a logic ‘0’ is considered.

It can be noticed a step-like behavior in the energy function, due to the fact that more and larger inverter stages are added to the bitline and wordline drivers as the

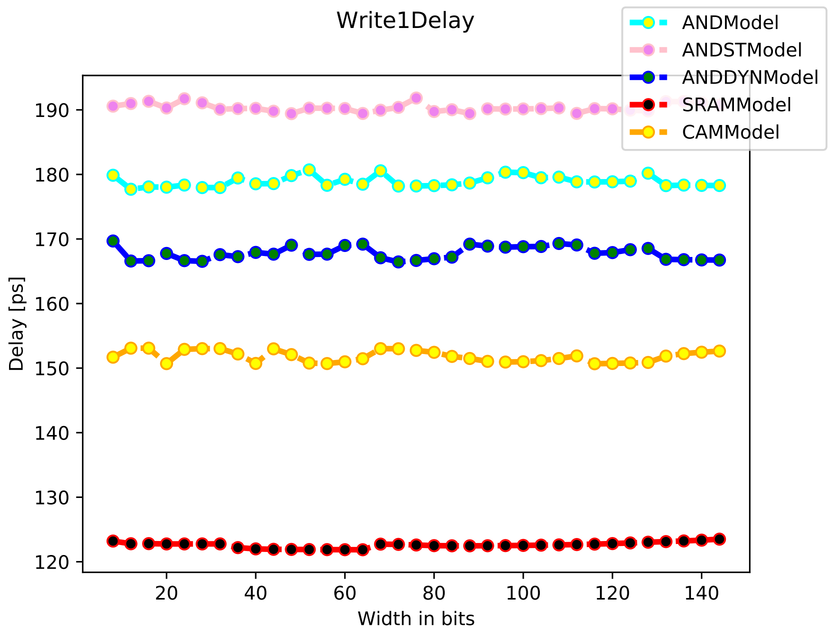


Figure 7.15: Write operation delay: cell content written to ‘1’, non ideal drivers.

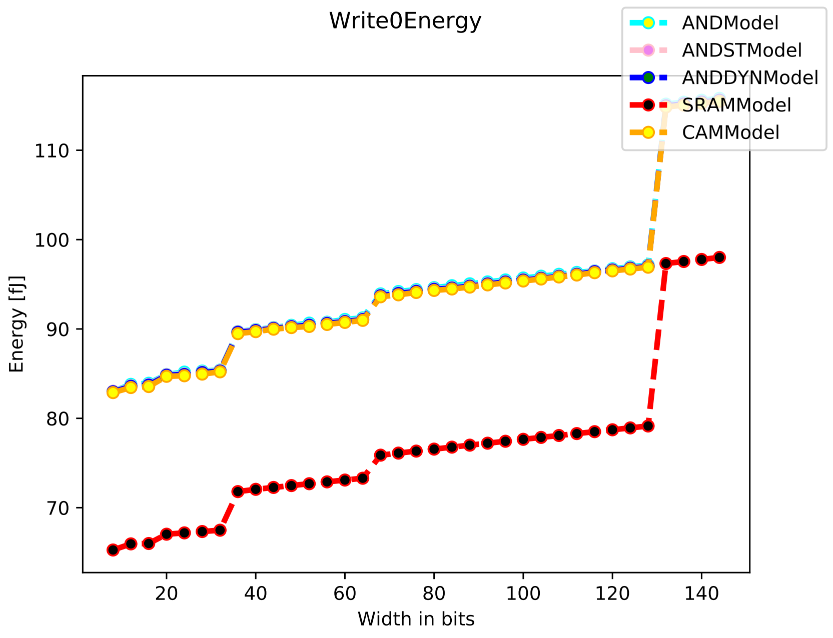


Figure 7.16: Write operation energy: cell content written to ‘0’, non ideal drivers.

memory width increases. The addition of each inverter stage leads to a sharp increase in the energy consumption.

As in Figure 7.14, also here it can be noticed how the LiM and CAM arrays have similar values of energy consumption, since a ‘0’ is written to the cell and, hence, the difference in the cell complexities (that results in a difference in the bitline capacitive load) is not exploited in the simulations.

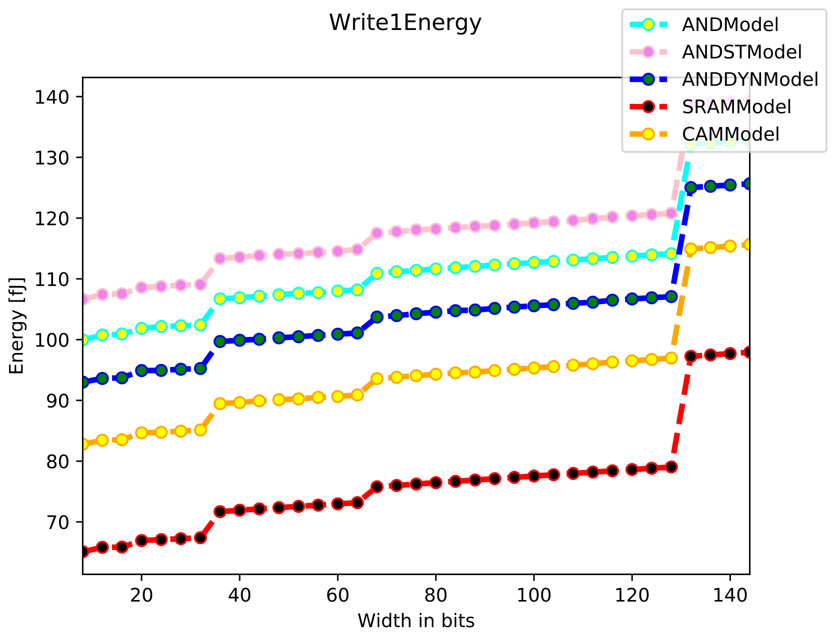


Figure 7.17: Write operation energy: cell content written to ‘1’, non ideal drivers.

In Figure 7.16, the write energy of a logic ‘1’ is considered.

It can be observed how, as in Figure 7.15, the energy values for the logic arrays are well separated. The most complex array, `ANDSTModel`, is the one with the largest energy consumption associated. `ANDModel` is characterized by an energy lower than `ANDDYNModel`, even if its cell circuit is simpler. This is due to its higher bitline capacitive load (Figure 7.3).

It can also be noticed how the simplest array, `SRAMModel`, is also the one with the lowest energy consumption associated.

7.1.5 Search operation: ideal drivers

In this section the search operations is considered, analyzing the results provided by the simulation performed using ideal drivers.

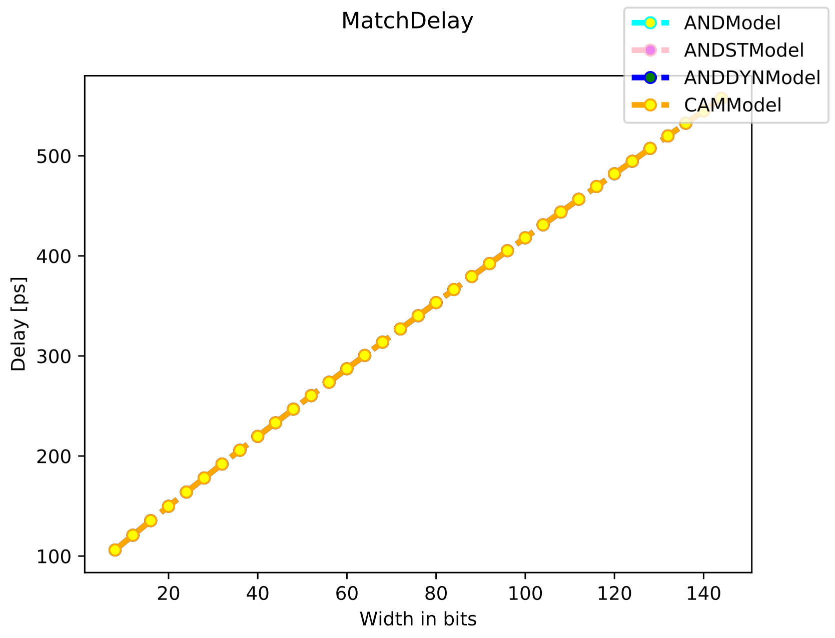


Figure 7.18: Search operation delay: ideal drivers.

In Figure 7.18, the search delay value with respect to the array width is analyzed.

A linear dependence on the row width can be observed, which is due to the fact that not all the cell parasitics are considered (as they would by realizing a proper memory layout) and that the search operation is performed on a row, which width is varied in the simulations.

It can be noticed how there is no difference in the search delay values for LiM and CAM arrays, since the CAM cell and match-line are completely separated from the LiM part of the cell; hence, since the simulations have been conducted at schematic level and not all the parasitics between AND and match lines are taken into account, a complete independence between the two operations performance is found. For this reason, the CAM and LiM arrays perform in the same way for the search operation, from both delay and energy points of view.

Of course, the SRAM array is excluded from the simulations, since no search operation can be performed in this architecture.

Since the line is discharged only if it has been previously charged, that happens only in the match case, one has to take into account this energy to estimate the total one associated to the match result. In the latter case, the line is charged during the search operation and then discharged; in the mismatch case, instead, the line is not charged and, so, no energy is wasted for a mismatch.

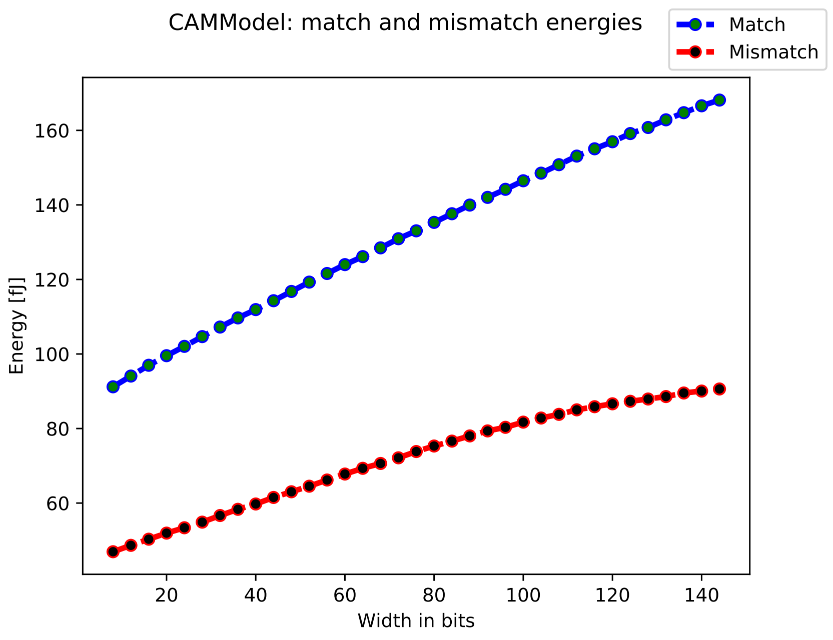


Figure 7.19: CAM architecture, search operation energy: ideal drivers.

For this reason, in estimating the energy associated to the match result, the energies of the pre-discharge cycle and search cycle have been summed up, obtaining the results shown in Figure 7.19.

In this graph, the match and mismatch energies are compared for the CAM architecture. It can be noticed how the energy associated to the match result is always larger than the one associated to the mismatch, as it has been described in Figure 2.27.

In the following, the same results are presented for the LiM architectures: the special-purpose in Figure 7.20, the dynamic in Figure 7.21 and the static in Figure 7.22. One can notice how the energy values obtained are equal to the ones reported for the CAM architecture, as previously explained.

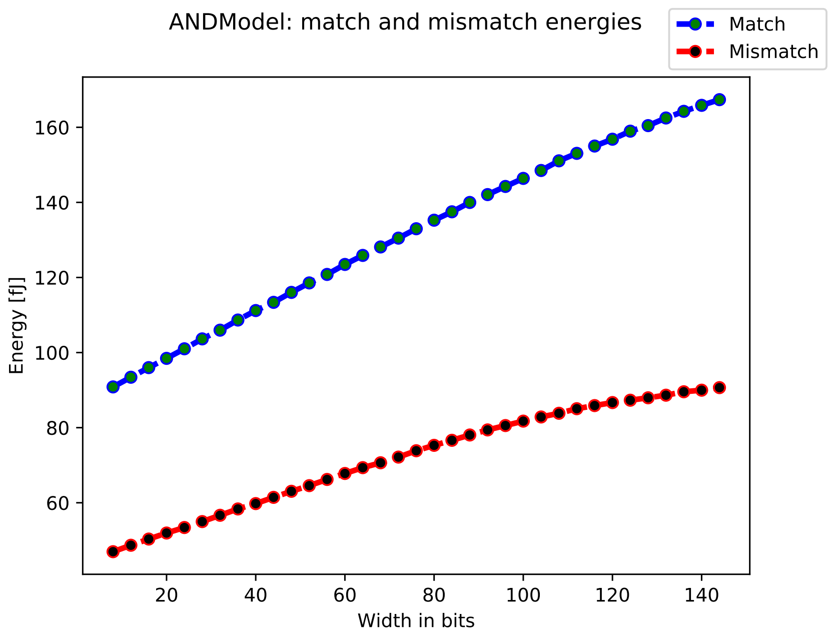


Figure 7.20: LiM architecture with special-purpose cell, search operation energy: ideal drivers.

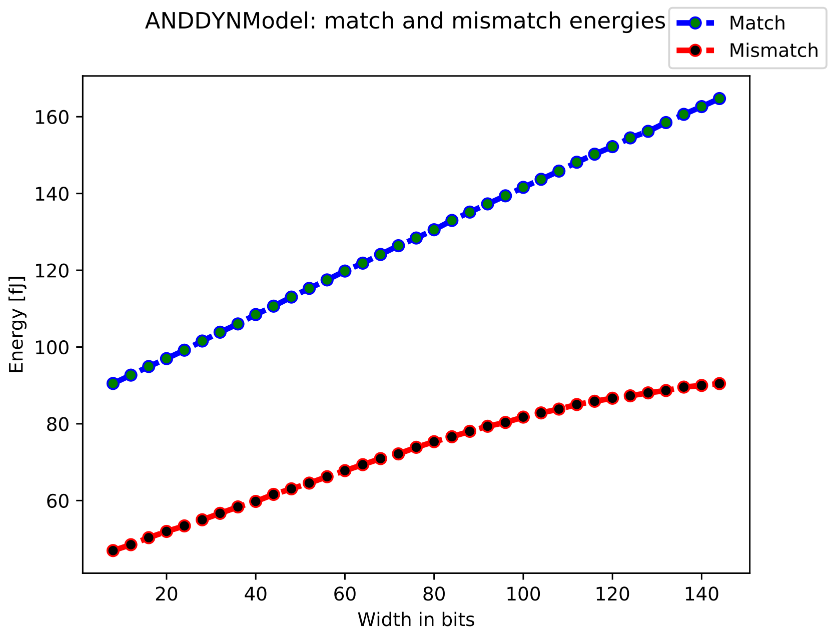


Figure 7.21: LiM architecture with dynamic cell, search operation energy: ideal drivers.

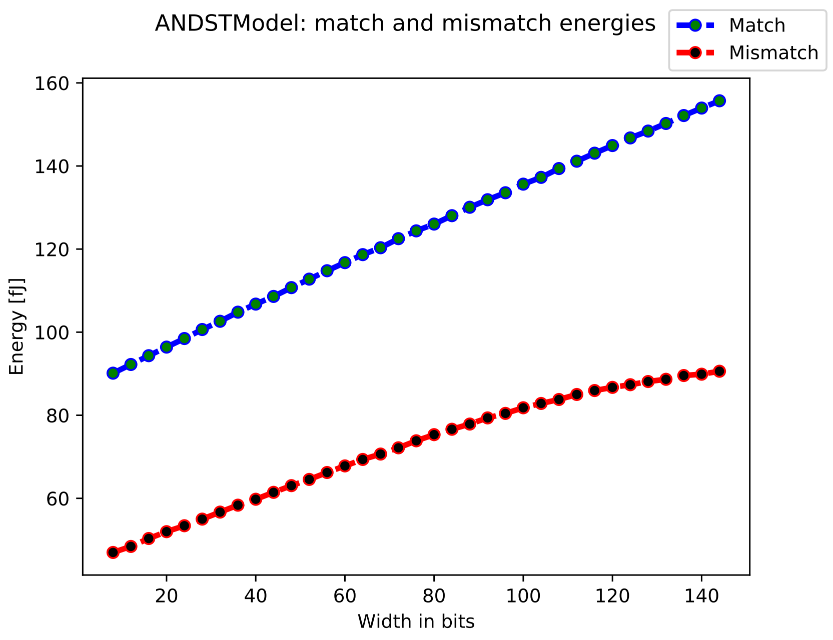


Figure 7.22: LiM architecture with static cell, search operation energy: ideal drivers.

7.1.6 Search operation: non-ideal drivers

In this section the search operations is considered, analyzing the results provided by the simulations performed using non-ideal drivers.

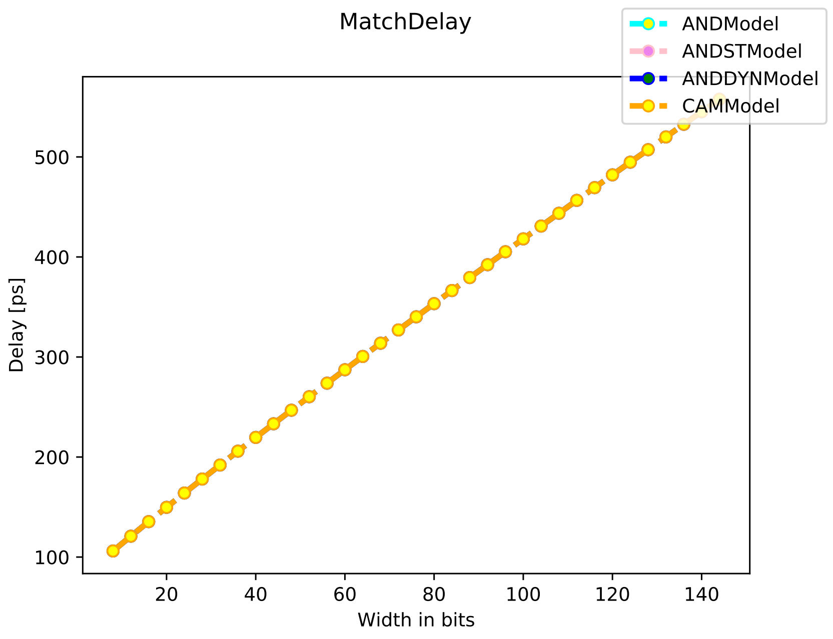


Figure 7.23: Search operation delay: non-ideal drivers.

In Figure 7.23, the search delay value with respect to the array width is analyzed. A linear dependence on the row width can be observed, as in Figure 7.18. In fact, the drivers have no influence on the search performance thanks to the sensing scheme implemented, as discussed in section 4.3. For this reason, the same delay values as in Figure 7.18 are obtained.

In Figure 7.24, the match and mismatch energies are compared. Slightly larger values are obtained for the match energy with respect to Figure 7.19, since during the pre-discharge phase the bitline drivers are used to load the search data on the bitlines.

As in the ideal drivers case, the same results are obtained also for the LiM arrays, thank to the independence between the AND and search operations, as previously explained.

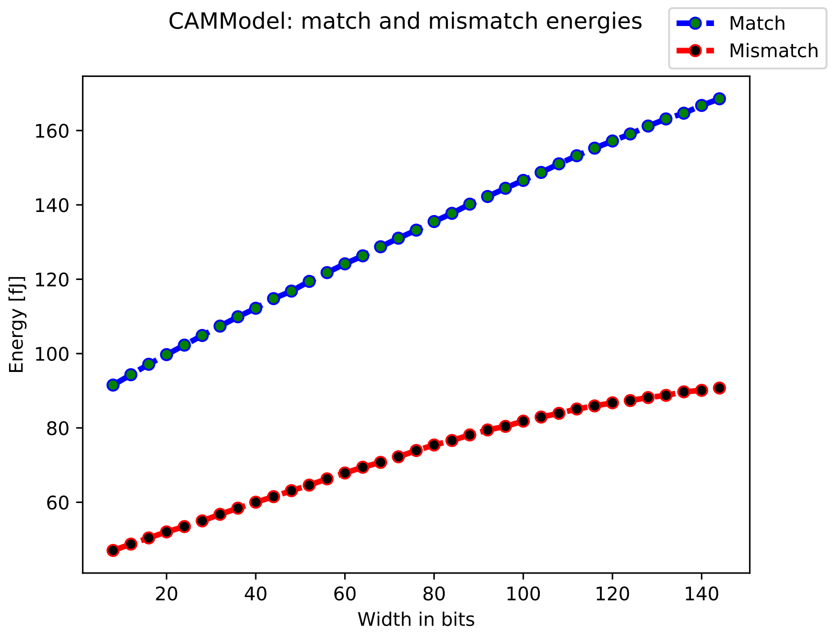


Figure 7.24: CAM architecture, search operation energy: non-ideal drivers.

7.1.7 AND operation: ideal drivers

In this section the AND operations is considered, analyzing the results provided by the simulations performed using non-ideal drivers.

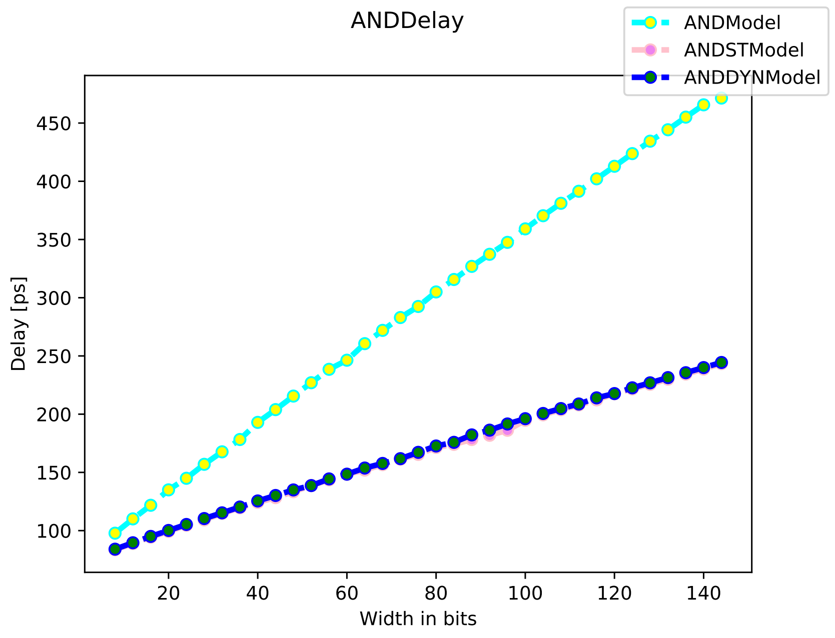


Figure 7.25: AND operation delay: ideal drivers.

In Figure 7.25, the AND operation delay is considered as function of the row width.

One can notice how the cell that performs worst is the special-purpose one, denoted by `ANDModel`. This is due to the fact that this cell has to transistors connected in series to the AND line, while in the other cells only one transistor is present (Figure 5.18). Since the AND operation is performed on the line, `ANDModel` results to perform worse than the other cells.

It can be noticed how the dynamic and static cells, `ANDDYNModel` and `ANDSTModel` respectively, have the same performance. This is due to the fact that they are characterized by the same output circuit (i.e. the transistors connected to the AND line). The difference in the cells complexities can be exploited considering the cells energy consumptions.

In Figure 7.26, the energy consumption associated to an AND operation whose result is '1', is shown.

The first thing that can be noticed is that `ANDModel` is characterized by the highest energy consumption. This can be explained considering the sensing scheme that is implemented in the architecture.

In section 4.4, it has been explained that the sense amplifiers are turned off as soon as the dummy SA switches its output. While the amplifiers are enabled, they inject current in the line and, so, they consume energy. For this reason, since `ANDModel` is characterized

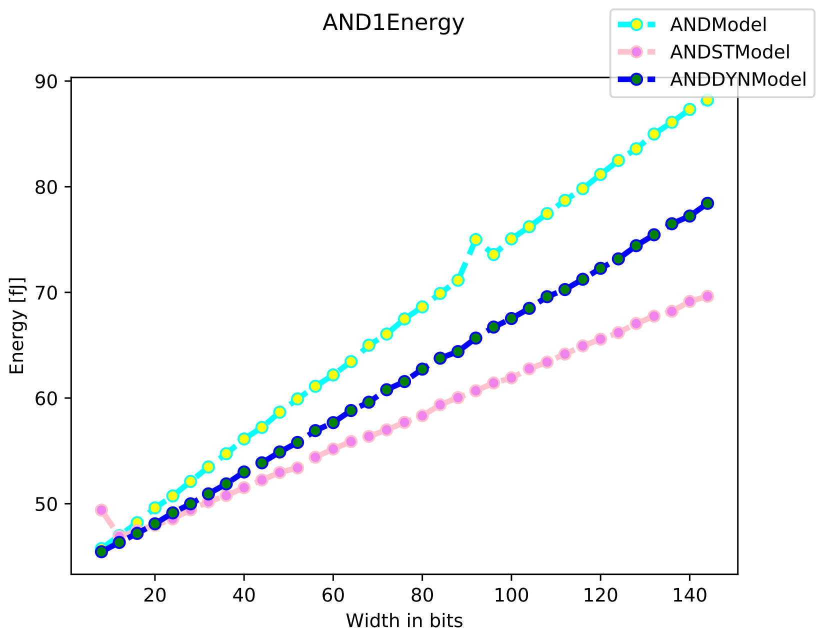


Figure 7.26: AND operation energy: AND='1' case, ideal drivers.

by the largest delay, to it is associated also the largest energy consumption: in fact, the larger is the delay, the larger will be the conduction time of the sense amplifiers and, so, the energy involved in the AND operation. In this way, it can be explained how the special-purpose cell has the largest energy consumption even if it is the simplest architecture among the LiM arrays.

In Figure 7.26, it can be noticed how the static cell, `ANDSTModel`, is characterized by a lower energy consumption than `ANDDYNModel`. This is due to the internal implementation of the cells: while `ANDSTModel` is implemented in standard static CMOS logic, `ANDDYNModel` utilizes dynamic CMOS logic, which is characterized by a larger energy consumption.

In Figure 7.27, the dynamic cell behavior is shown.

One can observe how the dynamic AND inputs are evaluated as soon as $\overline{\text{PRE}}$ is deactivated, which happens at the end of the pre-discharge cycle. Hence, it takes some time for the AND output to be discharged and, so, for the transistor connected to the AND-line to be disabled. During this small time interval additional energy is wasted, since the pull-down transistor connected to the line conducts the current provided by the sense amplifier to ground, not allowing the line to be charged.

This does not happen in the static cell where, since static logic is implemented, the pull-down transistor is disabled during the pre-discharge cycle and, so, energy is preserved.

In Figure 7.28, the energy associated to the pre-discharge operation is shown.

It can be noticed how the largest energy consumption is associated to `ANDModel`. This is due to its complex output circuit, as previously explained. Also in this case, the

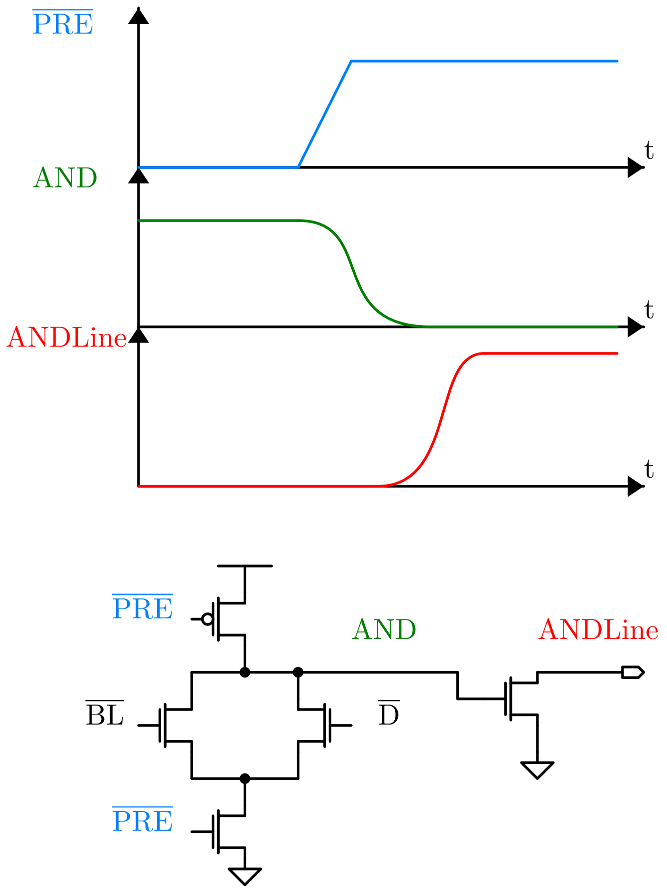


Figure 7.27: The dynamic cell behavior.

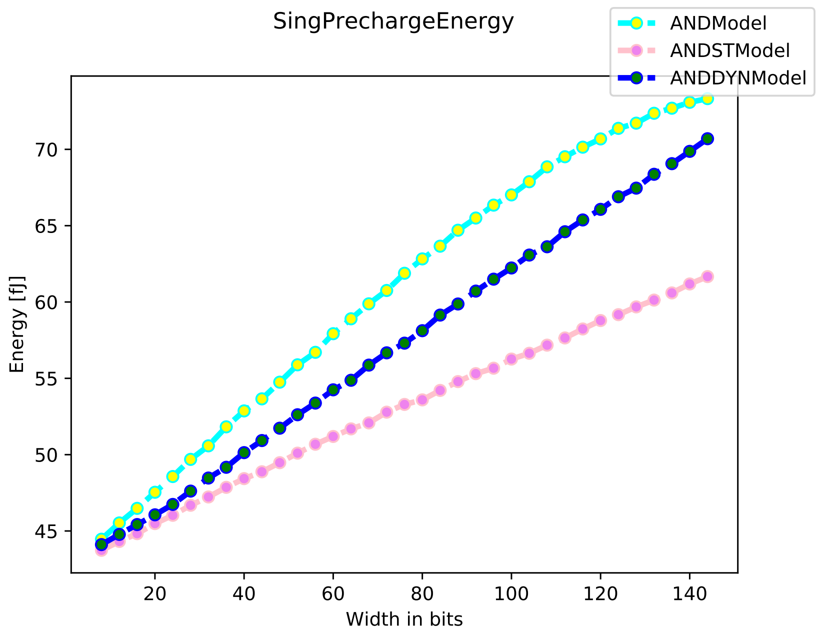


Figure 7.28: AND operation energy: pre-discharge cycle, ideal drivers.

dynamic cell is characterized by an energy consumption larger than the static one. This is due to the fact that during the pre-discharge phase, the internal AND nodes of the cells have to be precharged in the dynamic implementation, while this does not happen in the static one.

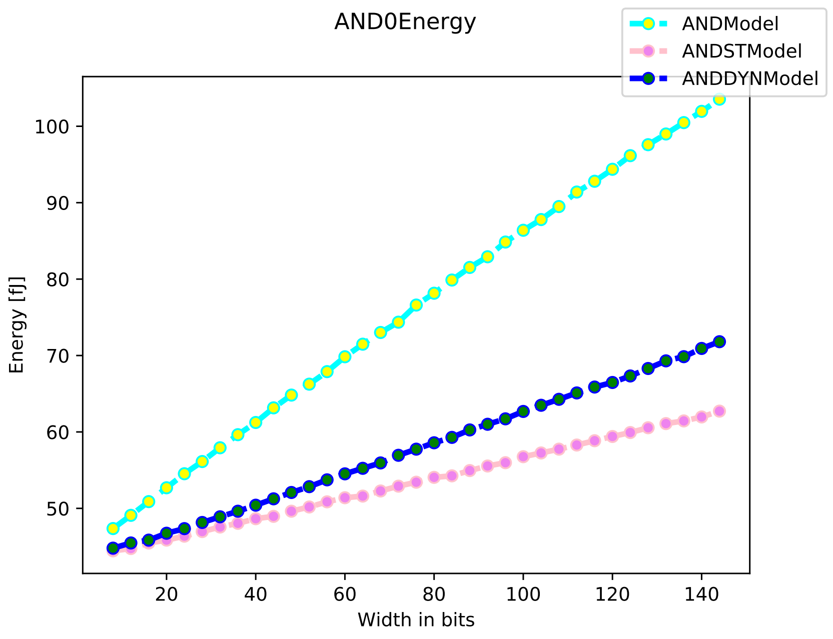


Figure 7.29: AND operation energy: AND='0' case, ideal drivers.

In Figure 7.29, the energy consumption associated to an AND operation whose result is '0', is shown.

The same observations made before for the energy values hold true.

In Figure 7.30, the energies associated to the two results of an AND operation for the `ANDModel` architecture are shown.

If the result of the operation is '1', the line has to be discharged afterwards. This implies that, when considering the energy consumption associated to a result equal to '1', one has to take into account also the pre-discharge energy. This is what is done in Figure 7.30. In fact, it can be observed how the largest energy consumption is associated to the result equal to '1'.

This can lead to reduced energy consumption in the memory operation if the memory words contain a lot of zeros: in fact, in this case, the most frequent result of the AND operation would be '0', which is also the one with the lowest energy consumption associated.

This is true only for the `ANDModel` architecture: for the other two arrays, the opposite happens, since the result of the AND operation is inverted on the AND-line, as it has been discussed in section 5.3. For the `ANDDYNModel` and `ANDSTModel` arrays, hence, the result of the AND operation with the lowest energy consumption associated is '1'.

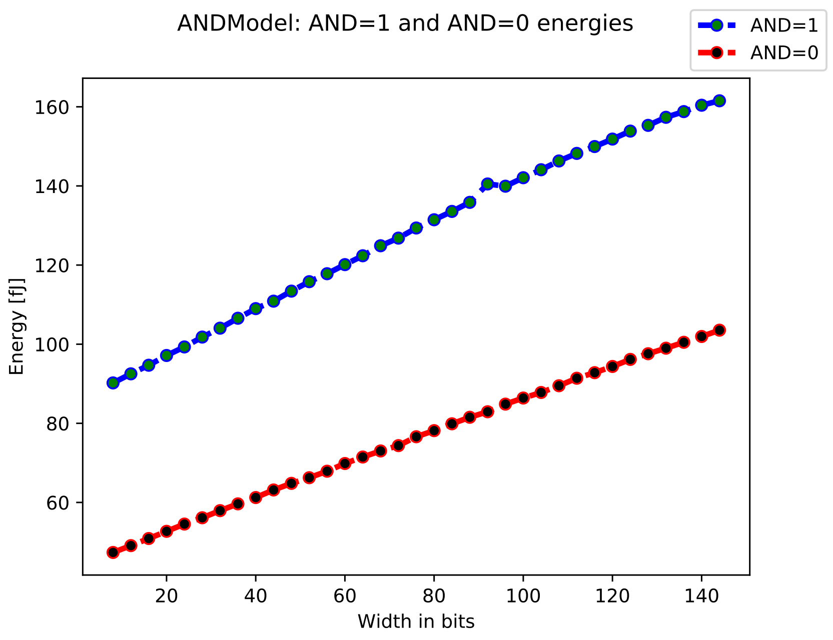


Figure 7.30: Comparison between AND=1 and AND=0 energies for `ANDModel`.

7.1.8 AND operation: non-ideal drivers

In this section, the AND operation is studied, analyzing the results of the simulations performed using non-ideal drivers.

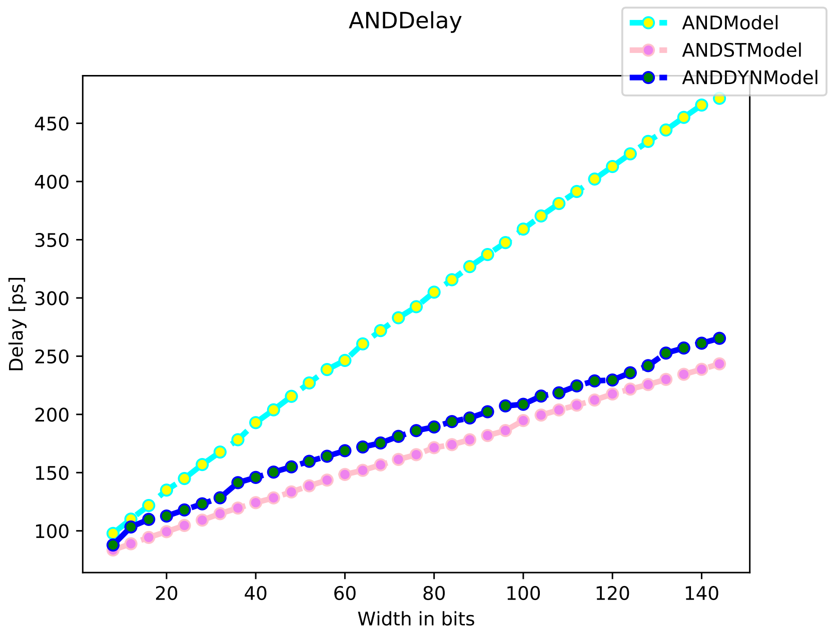


Figure 7.31: AND operation delay: non-ideal drivers.

In Figure 7.31, the delay of the AND operation is analyzed.

It can be noticed that the delay associated to `ANDDYNModel` is larger than the one of `ANDSTModel`, differently from what is shown in Figure 7.25, where ideal drivers are instantiated in the simulations. This is due to the fact that the precharge signal of the cells ($\overline{\text{PRE}}$ in Figure 7.27) has to be distributed on the row: since the tested cell is put in the farthest position with respect the driver of $\overline{\text{PRE}}$, this has an influence on the performance of the array.

In the static architecture, instead, no precharge signal is distributed on the the row. Hence, the performance are maximized.

In Figure 7.32, the energy consumption of an AND operation whose result is ‘1’, is considered.

One can observe that the array type to which the largest energy consumption is associated, is the one with the dynamic cell: `ANDDYNModel`. The reasons of this are the same examined before: since in these simulation a real driver is used to distribute the precharge signal in the row, the energy consumption of the array is increased during an AND operation. Referring to Figure 7.33, the conduction time of the pull-down transistor is increased, since the precharge signal is not ideal anymore, being generated by a real driver. For this reason, the AND-line transistor is enabled for a larger time interval and, so, more energy is wasted in the operation.

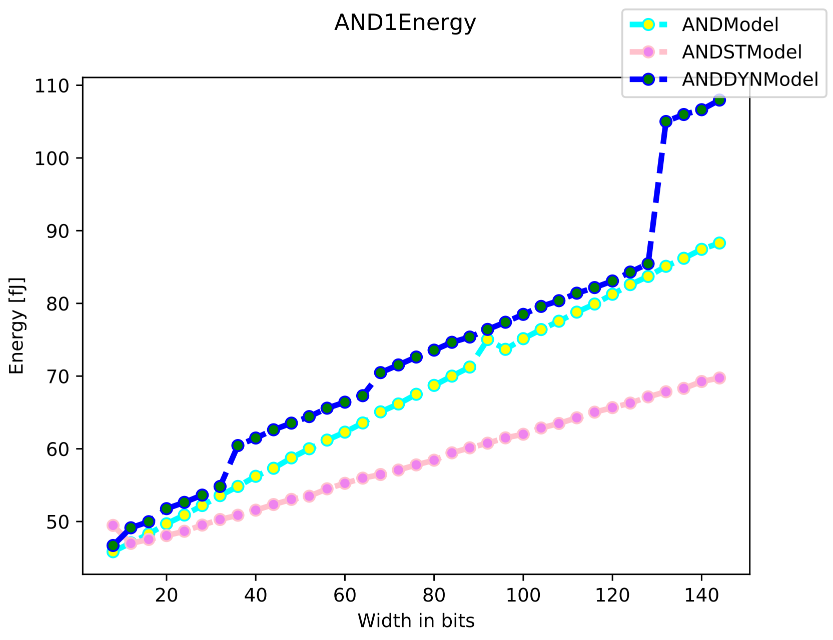


Figure 7.32: AND operation energy: AND=‘1’ case, non-ideal drivers.

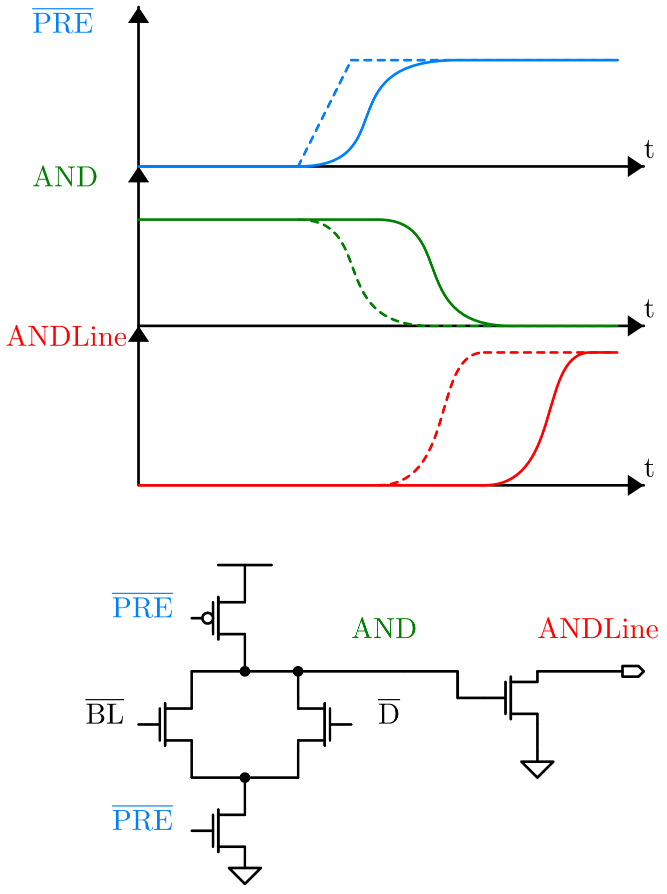


Figure 7.33: The dynamic cell behavior with non-ideal drivers.

In this energy increase also the non-ideality of the driver has to be taken into account, that results in an additional energy consumption during the AND operation.

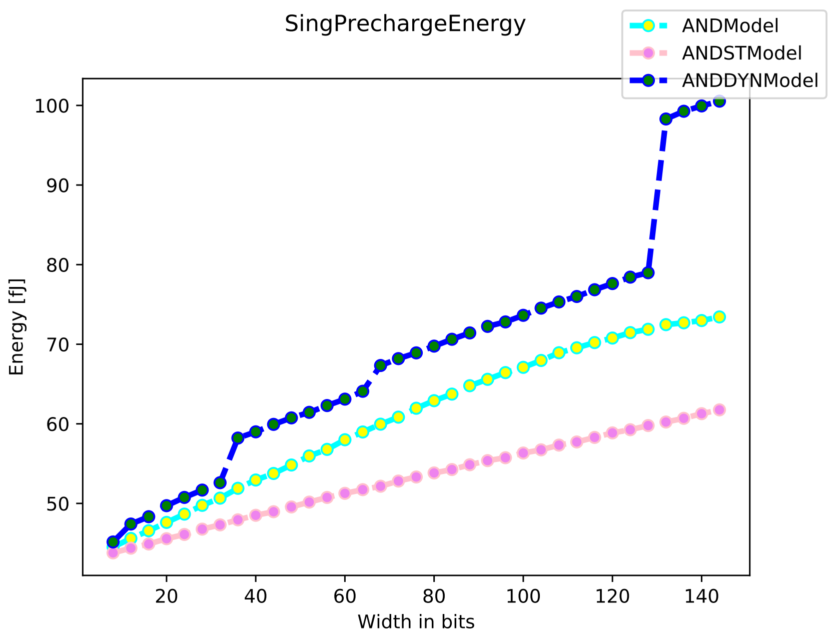


Figure 7.34: AND operation energy: pre-discharge cycle, non-ideal drivers.

In Figure 7.34, the energy associated to the pre-discharge cycle is shown.

Also in this case, the largest energy consumption is associated to `ANDDYNModel`, for the reasons discussed before: since the precharge signal is not ideal in these simulations, a larger energy consumption is obtained. In fact, a step-like behavior can be observed for `ANDDYNModel`, since the precharge signal driver is redesigned as the row width is increased.

In Figure 7.35, the energy consumption associated to an AND operation whose result is ‘0’, is shown.

When the result of the AND operation is ‘0’, it means that the AND-line transistor is always enabled. Hence, the delay in the precharge signal has no effect on the performance, since the internal AND node of the gate is not discharged. However, the resulting energy consumption is increased with respect the ideal case of Figure 7.29, since a non-ideal driver is generating $\overline{\mathrm{PRE}}$ and, so, a larger energy consumption is obtained.

In Figure 7.36, the two results of the AND operation are compared in terms of energy consumption for the `ANDModel` architecture. As in Figure 7.30, the pre-discharge energy is added to the one associated to the result ‘1’ for the reasons previously discussed, resulting in a larger energy consumption associated to the AND operation whose result is a logic ‘1’.

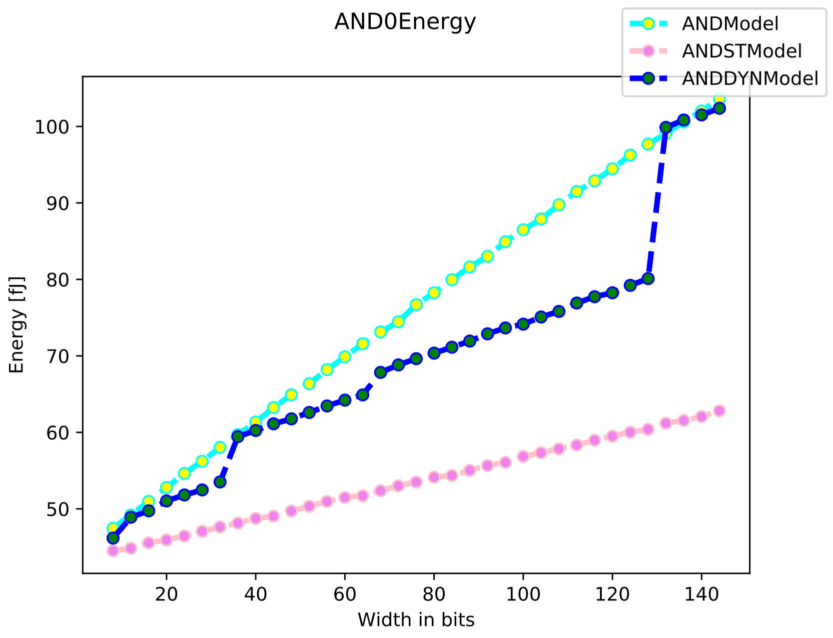


Figure 7.35: AND operation energy: AND=‘0’ case, non-ideal drivers.

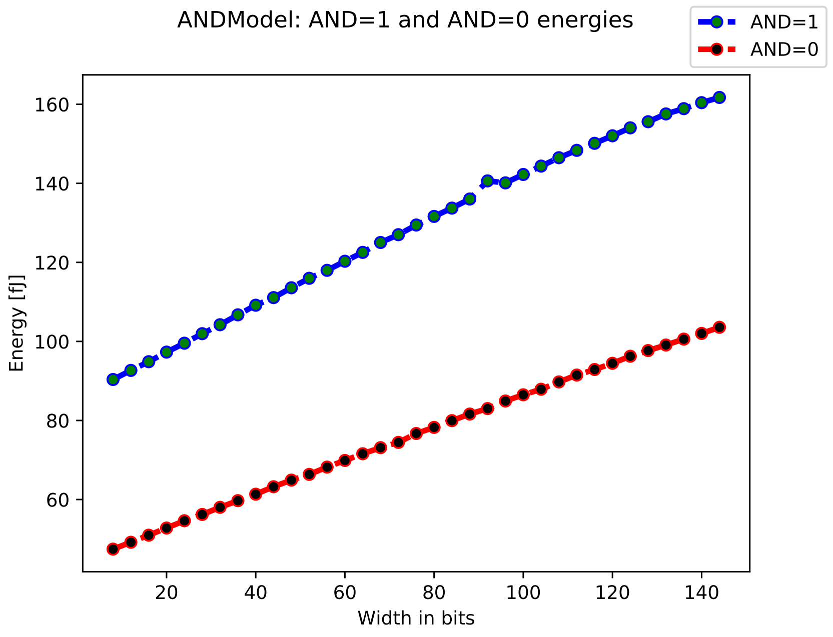


Figure 7.36: Comparison between AND=1 and AND=0 energies for `ANDModel`.

7.2 Simulations performed varying the array height

In this section, the simulations performed varying the array height from 128 to 512 rows, while the width is fixed to 32 bits, are considered. In the following, each operation is analyzed in detail.

7.2.1 Read operation: ideal drivers

In this section, the read operation as function of the array height is analyzed, using the results provided by simulations in which ideal drivers have been instantiated.

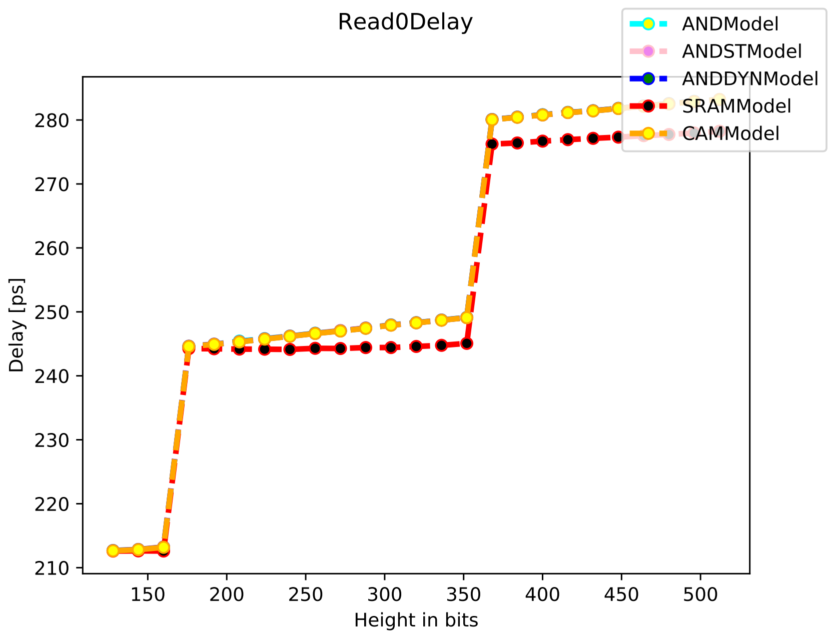


Figure 7.37: Read operation delay: cell content equal to ‘0’, ideal drivers.

In Figure 7.37, the read delay of a logic ‘0’ is shown.

It can be noticed a step-like behavior of the delay value, which is due to the re-design of the delay circuit of the sense amplifiers to the array height: as the number of rows enlarges, the delay associated to the enable signal of the SA is increased (section 6.2.1) in order for the read operation to be properly carried out.

One can observe that `SRAMModel` performs always better than the logic arrays, because of its higher simplicity: being lower the capacitive load on the bitlines (Figure 7.3), the discharge rate of the bitlines results to be higher and, so, a lower delay is registered for the read operation.

In Figure 7.38, the read delay of a logic ‘1’ is shown.

It can be noticed how the logic arrays values are now well separated. This is due to the fact that when reading a logic ‘1’, the negated bitline $\overline{\text{BL}}$ is discharged; since this line is the most capacitively loaded line among the LiM memory cells, the difference in their complexities is registered in the arrays performance.

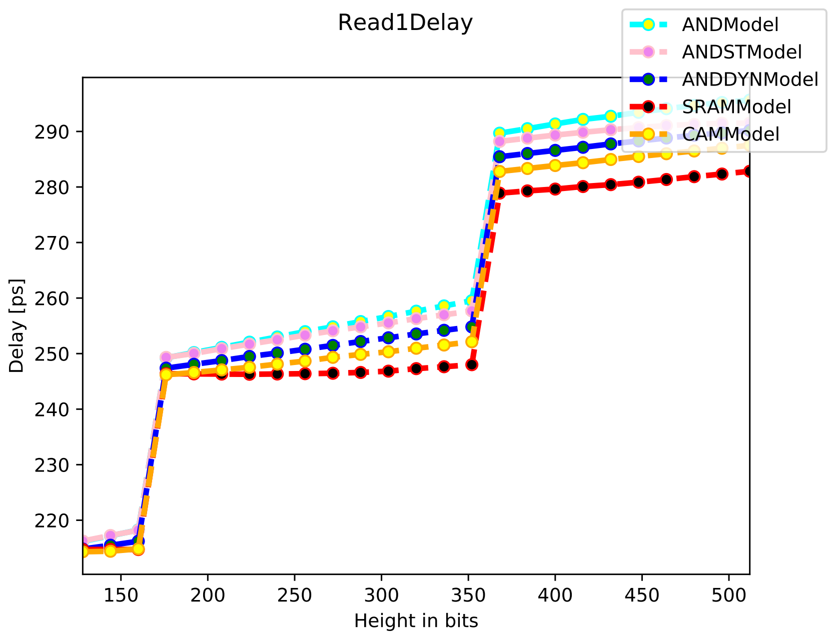


Figure 7.38: Read operation delay: cell content equal to ‘1’, ideal drivers.

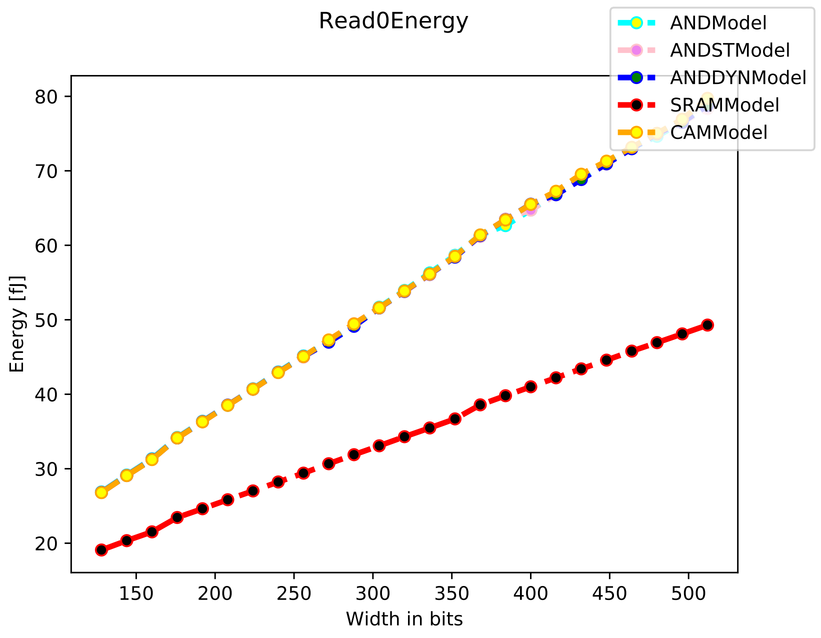


Figure 7.39: Read operation energy: cell content equal to ‘0’, ideal drivers.

In Figure 7.39, the energy consumption associated to the reading of a logic ‘0’ is shown.

One can notice that the SRAM array is the less energy consuming one; this is due to its higher simplicity, that results in lower energy consumption. No difference in the performance can be observed for the logic arrays (CAM and LiM) since during the reading of a logic ‘0’, the bitline BL is not discharged and, so, the difference in the capacitive loads (Figure 7.3) is not exploited in the measurements.

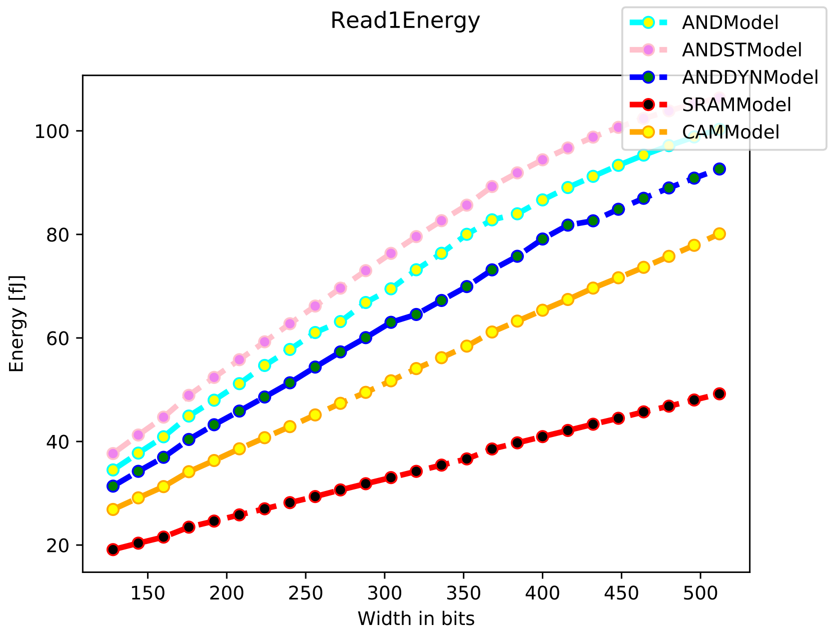


Figure 7.40: Read operation energy: cell content equal to ‘1’, ideal drivers.

In Figure 7.40, the energy consumption associated to the reading of a logic ‘1’ is shown.

In this case, a clear distinction between the logic arrays can be observed: the most complex arrays (referring to their bitline capacitive load) are the ones to which the largest energy consumptions are associated. This is particularly true for this operation, since during the reading of a logic ‘1’ the capacitive load difference between the cells (Figure 7.3) is exploited, as explained before.

7.2.2 Read operation: non-ideal drivers

In this section, the read operation as function of the array height is analyzed, using the results provided by simulations in which non-ideal drivers have been instantiated.

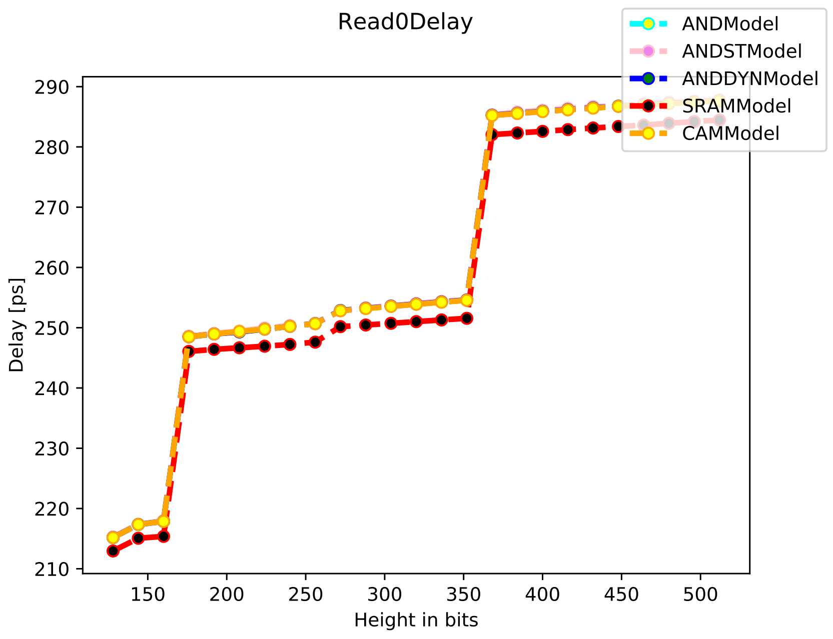


Figure 7.41: Read operation delay: cell content equal to ‘0’, non-ideal drivers.

In Figure 7.41, the read delay associated to a logic ‘0’ is shown.

Also in this case, a step-like behavior can be observed, which is due to the re-design of the sense amplifier delay circuit as the array height is increased.

It can be noticed how the SRAM delay is not so much lower than the logic arrays one. This is due to the fact that the read delay is largely determined by the delay circuit of the sense amplifier, and that the same algorithm for the SA delay generation is used for all the architectures.

In Figure 7.42, the read delay associated to a logic ‘1’ is shown.

In this graph, the difference in the cells complexities is exploited since a logic ‘1’ is being read, as explained for the ideal drivers case.

In Figure 7.43, the read energy associated to a logic ‘0’ is shown.

One can observe that, as expected, the most simple array, `SRAMModel`, is the one with the lowest energy consumption associated. Also, it can be noticed how the energy values are larger than in the ideal case: this is due to the non-idealities of the drivers that generate an additional contribution to the operation energy consumption.

In Figure 7.44, the read energy associated to a logic ‘1’ is shown.

A clear distinction can be observed between the logic arrays in this case, since a logic ‘1’ is being read and, so, the different cells complexities are registered in the results.

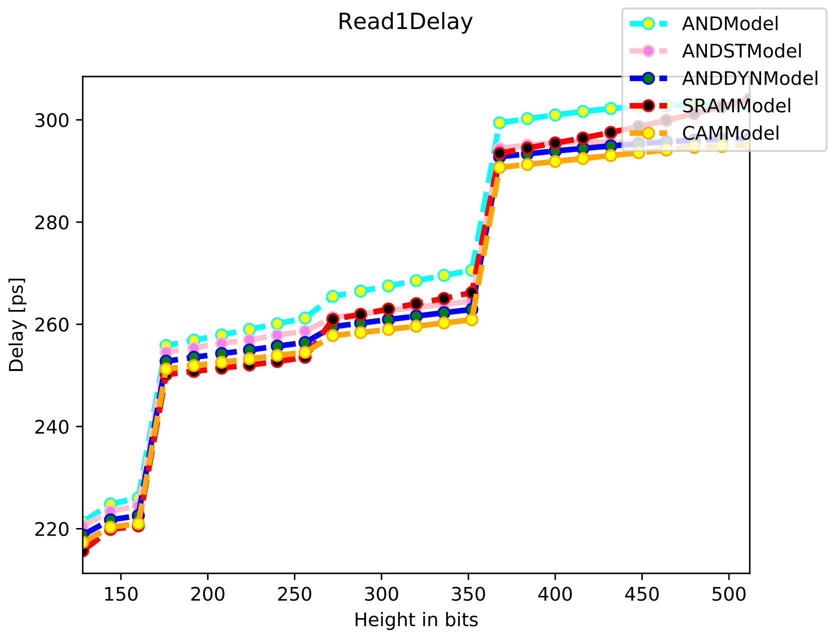


Figure 7.42: Read operation delay: cell content equal to ‘1’, non-ideal drivers.

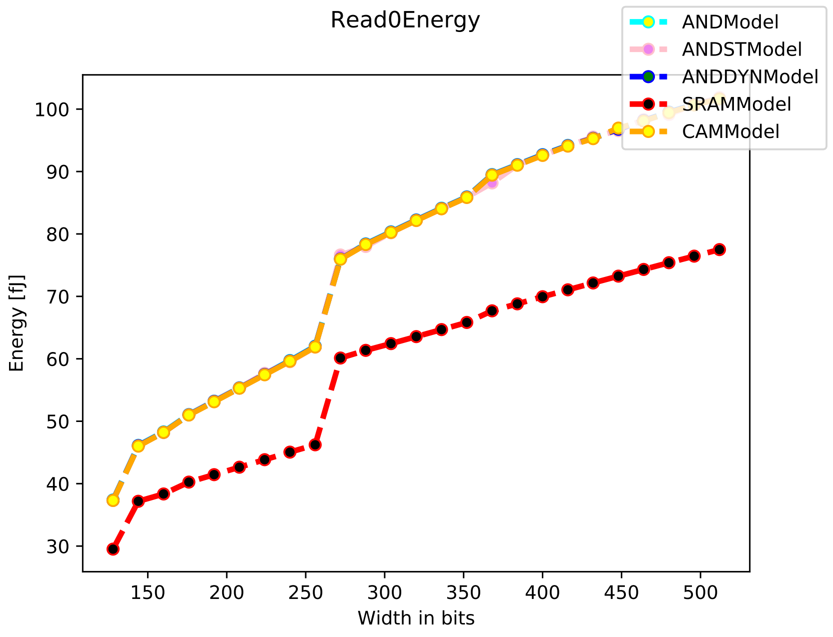


Figure 7.43: Read operation energy: cell content equal to ‘0’, non-ideal drivers.

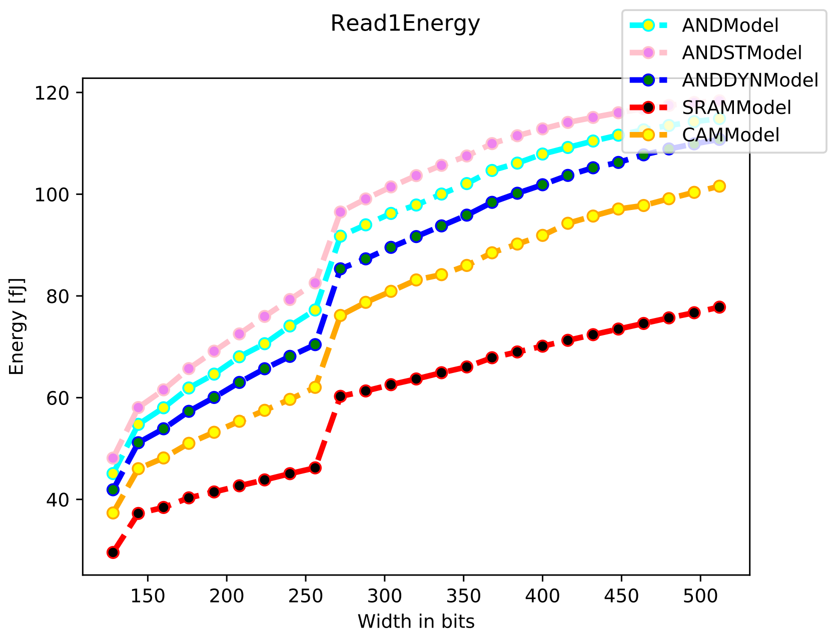


Figure 7.44: Read operation energy: cell content equal to ‘1’, non-ideal drivers.

7.2.3 Write operation: ideal drivers

In this section, the write operation as function of the array height is analyzed, using the results provided by simulations in which ideal drivers have been instantiated.

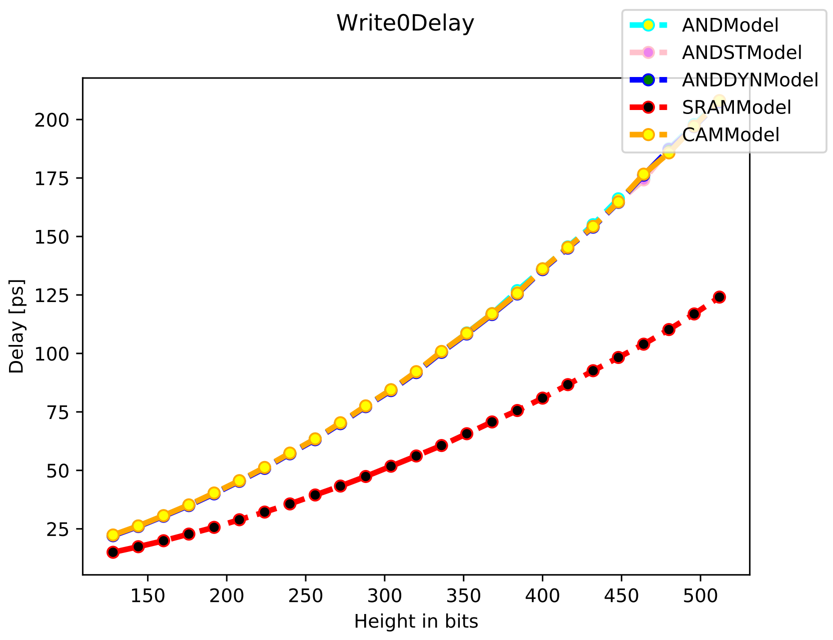


Figure 7.45: Write operation delay: cell content written to '0', ideal drivers.

In Figure 7.45 the write delay associated to a logic '0' is shown.

It can be noticed a super-linear dependence of the write delay on the array height. Also, as the array size is increased, the difference in the delay value between logic and SRAM arrays enlarges. This is due to the fact that the SRAM is characterized by a low capacitive load associated to the bitlines (Figure 7.3) and, so, its write delay increases in a way slower than the one of the logic arrays that, on the contrary, have a large capacitive load associated.

In Figure 7.46, the write delay associated to a logic '1' is shown.

One can observe that, in this case, the delay values associated to the logic arrays are well separated. The most complex cells (i.e. the LiM ones) have the largest delays associated. This is due to their large capacitive load and to the fact that a logic '1' is being written to the cell and, so, the negated bitline $\overline{\text{BL}}$ has to be discharged (in fact, $\overline{\text{BL}}$ is the most loaded bitline in the LiM arrays).

In Figure 7.47, the write energy associated to a logic '0' is shown.

In this case, a linear dependence on the array height can be noticed. This was expected since, as previously explained, not all the non idealities of the array are taken into account in the simulations and, as the array height is increased, the number of cells per row increases proportionally, leading to a linear growth in the energy consumption associated to the write operation.

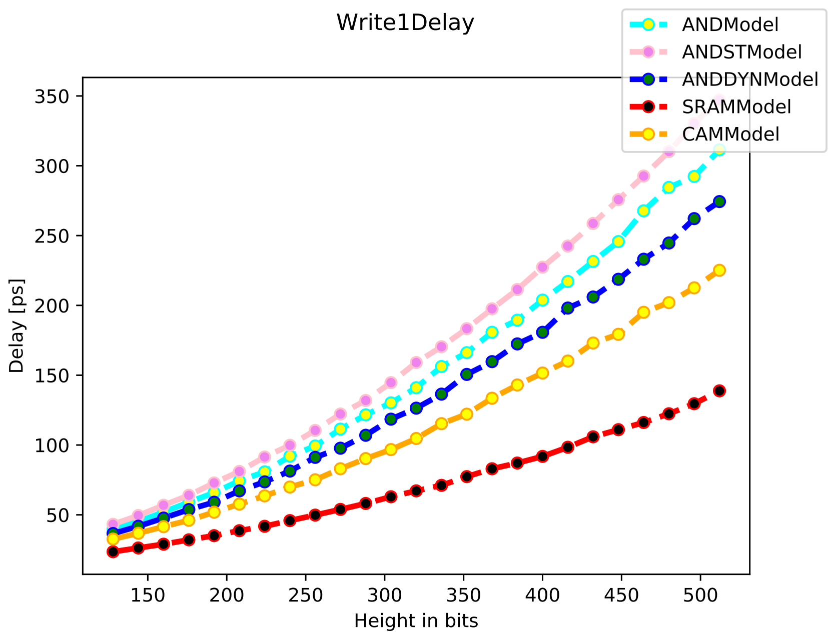


Figure 7.46: Write operation delay: cell content written to ‘1’, ideal drivers.

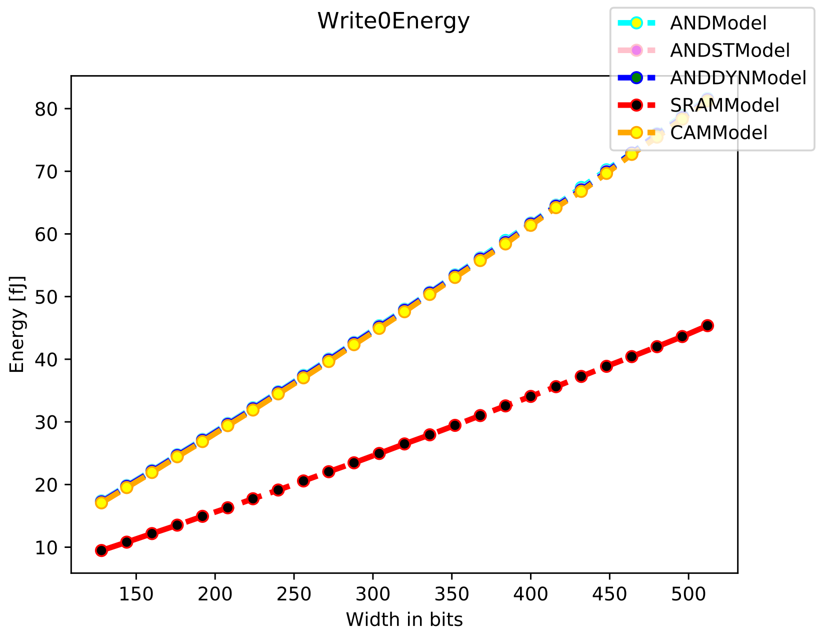


Figure 7.47: Write operation energy: cell content written to ‘0’, ideal drivers.

As in Figure 7.45, no difference can be observed among the logic arrays, since a logic ‘0’ is being read and the LiM cells capacitive load is associated to $\overline{\mathrm{BL}}$.

In Figure 7.48, the write energy associated to a logic ‘1’ is shown.

Since a logic ‘1’ is being written and, so, the $\overline{\mathrm{BL}}$ has to be brought from V_{DD} to 0 V, different energy values can be observed among the logic arrays, with the most complex one, `ANDSTModel`, being the array with the largest energy consumption associated.

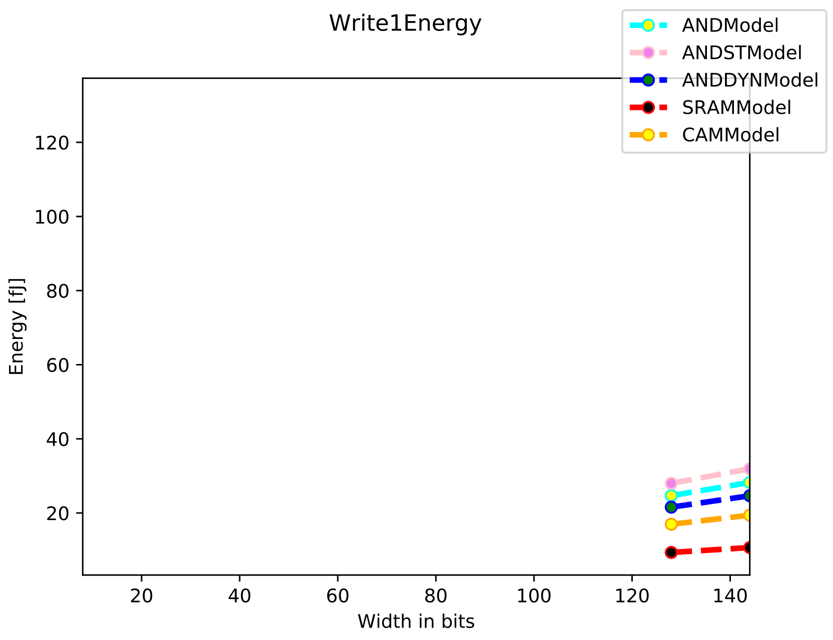


Figure 7.48: Write operation energy: cell content written to ‘1’, ideal drivers.

7.2.4 Write operation: non-ideal drivers

In this section, the write operation as function of the array height is analyzed, using the results provided by simulations in which non-ideal drivers have been instantiated.

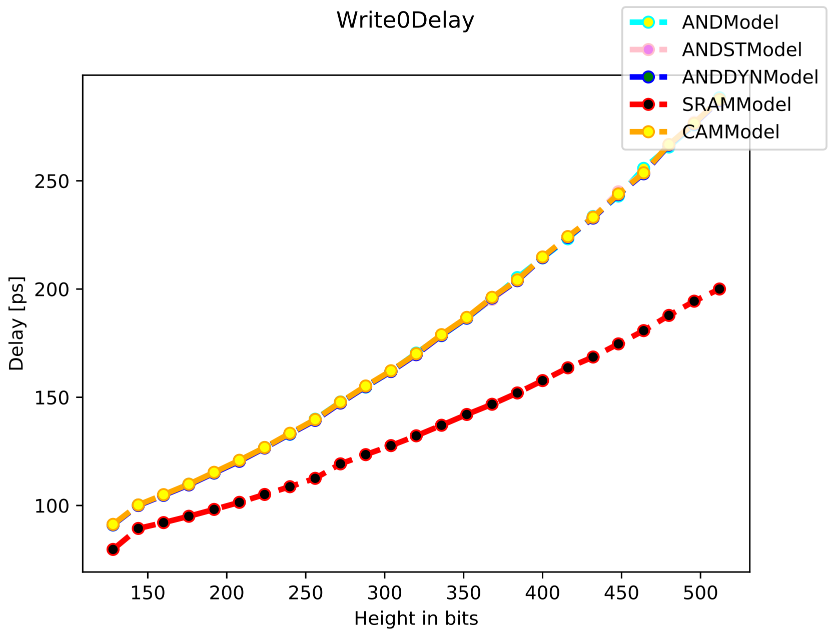


Figure 7.49: Write operation delay: cell content written to ‘0’, non ideal drivers.

In Figure 7.49, the write delay associated to a logic ‘0’ is shown.

A behavior similar to the one shown in the ideal case (Figure 7.45) can be observed. This is due to the fact that the bitline drivers strength is adjusted by ALiAS as the array height is increased.

As in the ideal case, the SRAM array presents the best performance, being the less complex architecture among all the arrays.

In Figure 7.50, the write delay associated to a logic ‘1’ is shown.

As in the ideal case, the delay values are now well separated among the arrays, thanks to the fact that a logic ‘1’ is being written to the cell. Also, a super-linear behavior can be observed, as in the ideal case (Figure 7.46).

In Figure 7.51, the write energy associated to a logic ‘0’ is shown.

A step-like behavior can be observed. This is due to the fact that inverter stages are added to the bitline drivers as the array height is increased, in order for their driving strength to be adapted to the array size.

As in the delay case (Figure 7.49), the less complex array, `SRAMModel`, is also the one with the lowest energy consumption associated.

In Figure 7.52, the write energy associated to a logic ‘1’ is shown.

Also in this case, a step-like behavior can be noticed, which is due to the bitline drivers re-design as the array size is increased. Furthermore, since a logic ‘1’ is being

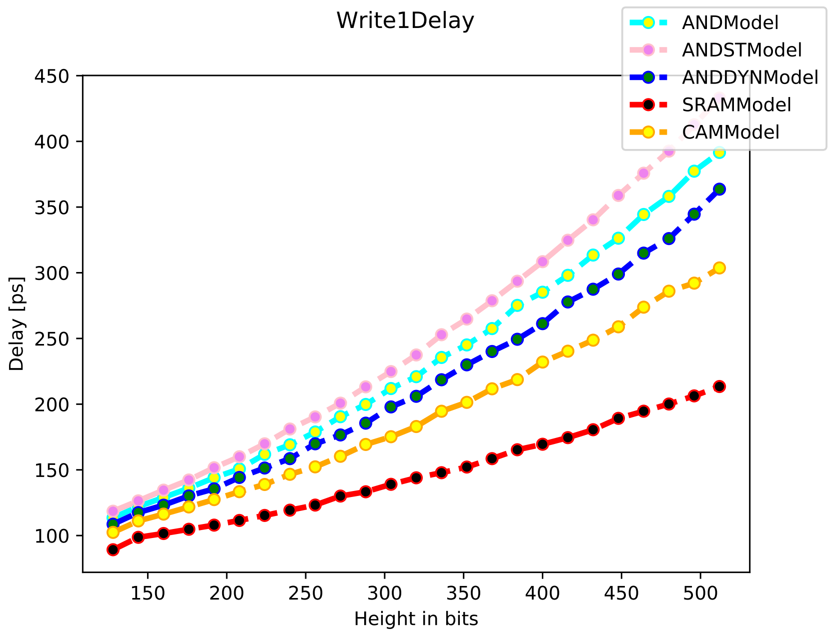


Figure 7.50: Write operation delay: cell content written to ‘1’, non ideal drivers.

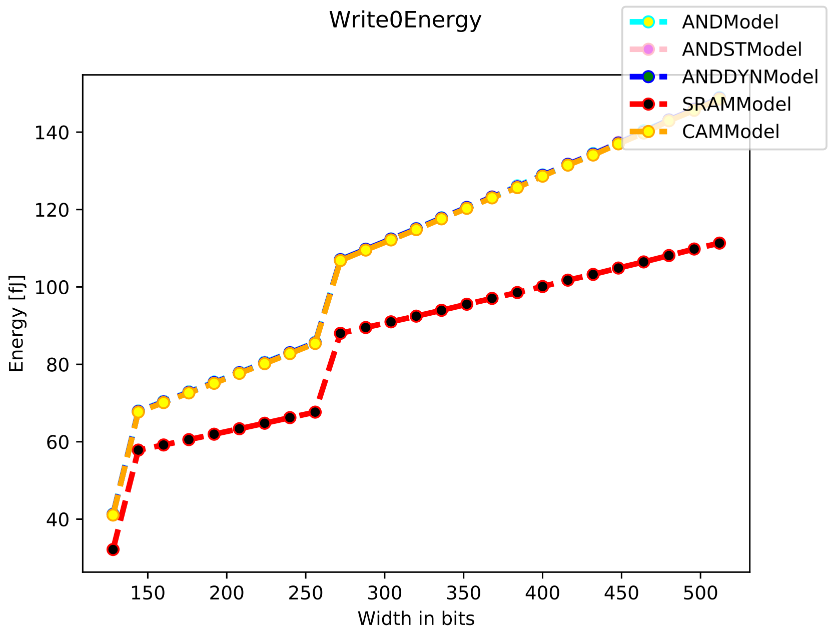


Figure 7.51: Write operation energy: cell content written to ‘0’, non ideal drivers.

written to the cell, different energy values can be observed among the arrays, with the most complex one, `ANDSTModel`, having the largest energy consumption associated.

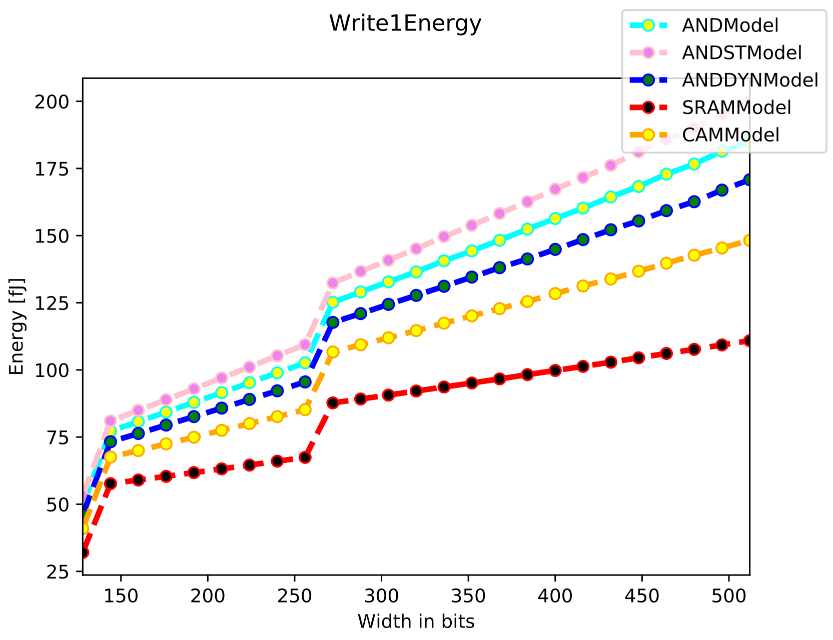


Figure 7.52: Write operation energy: cell content written to ‘1’, non ideal drivers.

7.2.5 Search operation: ideal drivers

In this section, the search operation as function of the array height is analyzed, using the results provided by simulations in which ideal drivers have been instantiated.

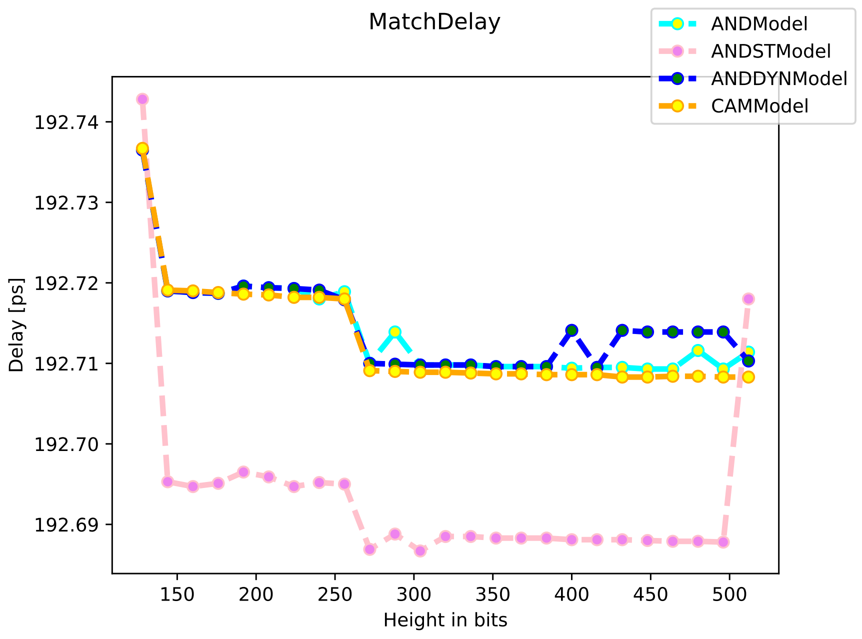


Figure 7.53: Search operation delay: ideal drivers.

In Figure 7.53, the delay associated to the search operation is shown.

It can be noticed that the search delay is practically constant as the array height is varied. This was expected since the search operation is carried out on a row: being constant the row width, no variation is registered in the performance.

In Figure 7.54, the match and mismatch energies are compared.

Also in this case, the energy associated to the match result is larger than the mismatch one. This is due to the fact that to a match-result corresponds a variation in the match-line potential (i.e. the match-line is charged to V_{DD}).

As in subsection 7.1.5, the same energy results are obtained for the LiM arrays, since in these the match-line circuitry is completely separated from the LiM one.

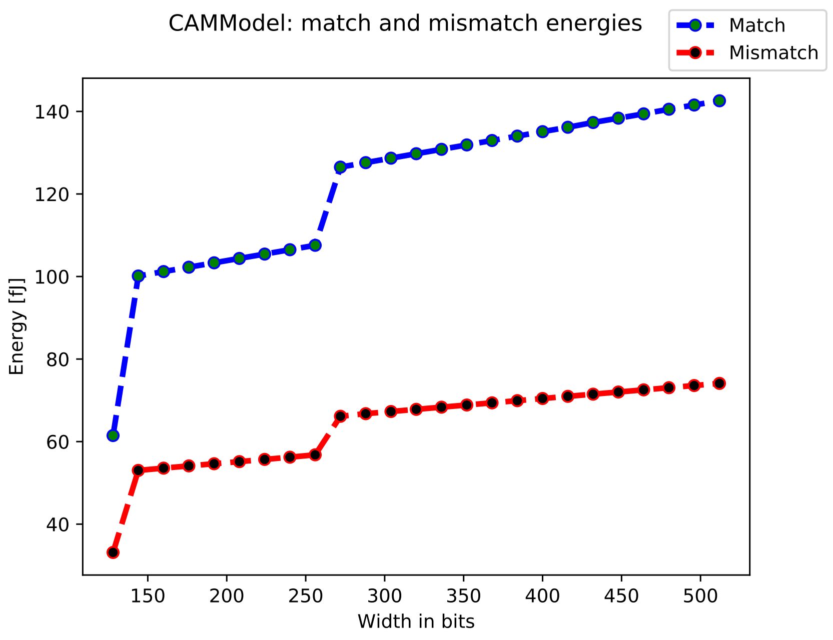


Figure 7.54: CAM architecture, search operation energy: ideal drivers.

7.2.6 Search operation: non-ideal drivers

In this section, the search operation as function of the array height is analyzed, using the results provided by simulations in which non-ideal drivers have been instantiated.

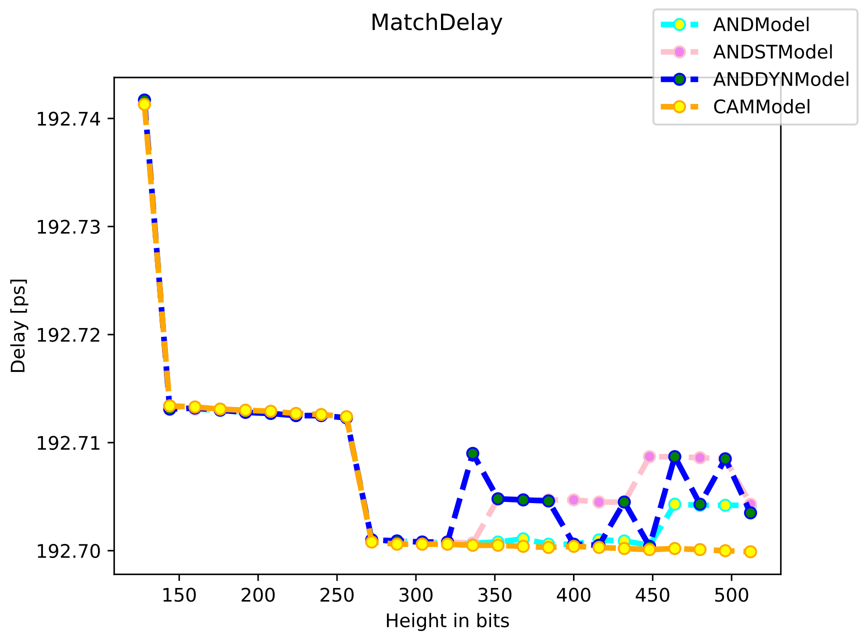


Figure 7.55: Search operation delay: non-ideal drivers.

In Figure 7.55, the search delay as function of the memory height is shown.

As in the ideal case, a practically constant value for the match delay is found, thanks to the fact that the search operation is performed on the memory row and, so, its delay is practically independent from the array height.

In Figure 7.56, the match and mismatch energies are compared.

As in the ideal case, an energy larger for the match result than for the mismatch one can be observed. Differently from Figure 7.54, a larger energy consumption is associated to the match case, because of the non-ideal drivers used to drive the bitlines during the pre-discharge phase.

One can notice that the energy increases as the array height is varied, while the delay remains constant. This is due to the fact that the energy consumed during a search operation is related to the time needed by the dummy MLSA to switch its output.

In section 6.2.1 it has been explained that a dummy load is placed on the output of the dummy MLSA to emulate the presence of the MLSAs associated to the remaining rows of the array. Hence, as the array height is increased, more loads are attached to the dummy MLSA output. This results in a larger delay associated to it, that leads to an increased conduction time for the actual MLSA; this increase in the conduction time, hence, results in an enlargement in the energy consumption associated to the search operation.

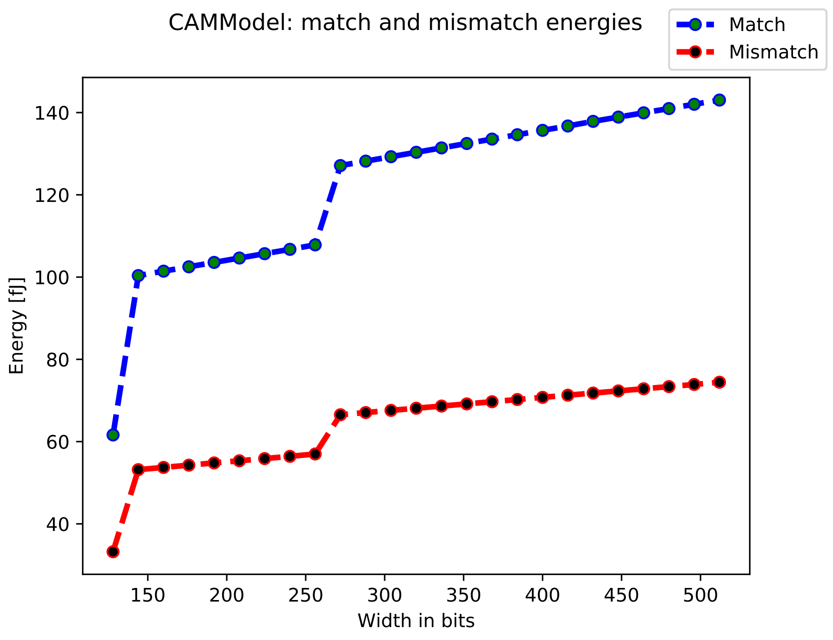


Figure 7.56: CAM architecture, search operation energy: non-ideal drivers.

Hence, as result, as the array height is increased, the energy consumption associated to the search operation enlarges.

7.2.7 AND operation: ideal drivers

In this section, the AND operation as function of the array height is analyzed, using the results provided by simulations in which ideal drivers have been instantiated.

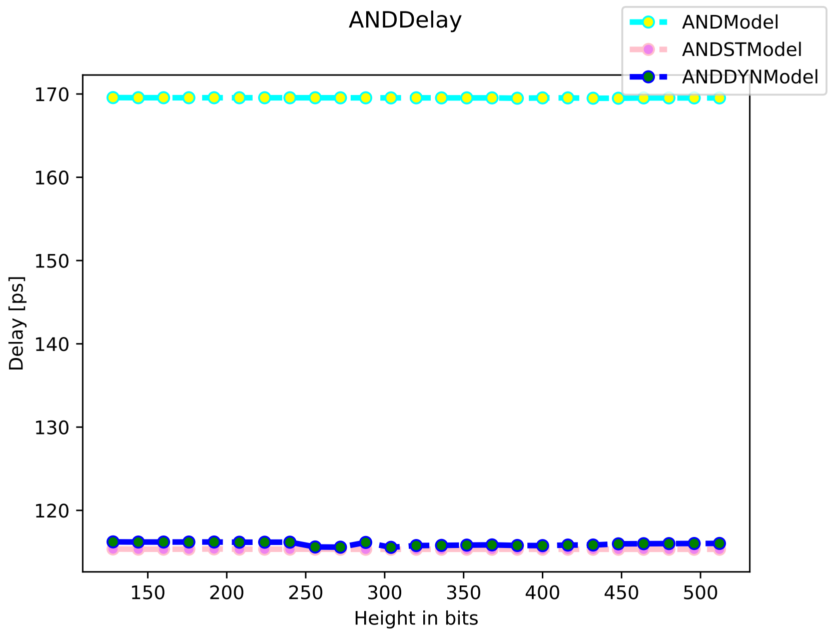


Figure 7.57: AND operation delay: ideal drivers.

In Figure 7.57, the AND delay as function of the array height is shown.

One can observe that the delay remains constant as the array height is varied. This is due to the fact that the AND operation, as the search one, is performed on the row and, so, is independent from the array height.

It can be noticed that the largest delay is associated to `ANDModel`, since in the cell has the most complex AND output circuit (i.e. the cell transistors connected to the AND-line). The dynamic and static arrays, instead, are characterized by the same performance.

In Figure 7.58, the energy consumption associated to an AND operation whose result is '1', is shown.

As in Figure 7.26, the most consuming architecture results to be `ANDModel`, because of its complex AND-line circuit. An increase in the energy with the array height can be observed too. This is due to the same reasons explained in subsection 7.2.5: as the array height increases, the conduction time of the AND-line sense amplifier and, so, the associated energy consumption are increased.

In Figure 7.59, the energy consumption associated to the pre-discharge cycle is shown.

As in Figure 7.58, the largest energy consumption is associated to `ANDModel`, because of its complex AND-line circuit.

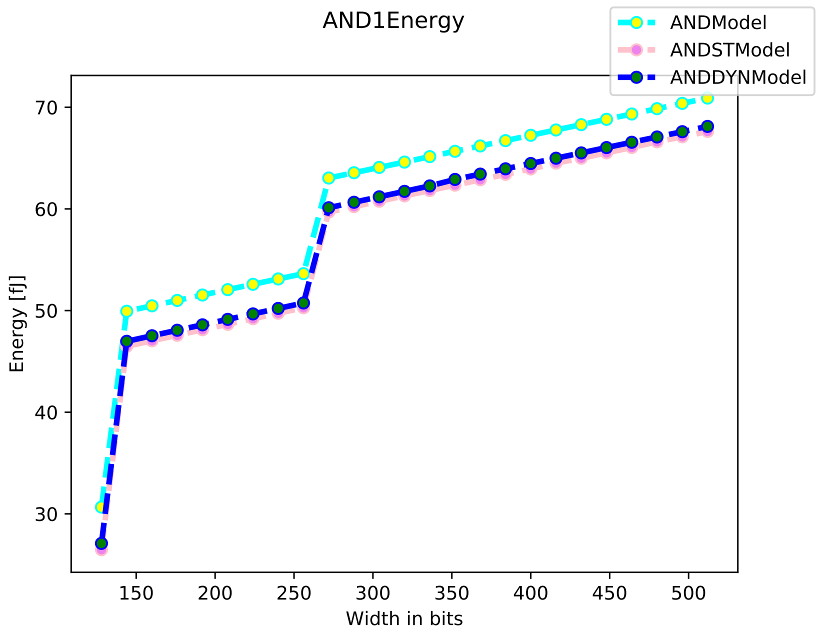


Figure 7.58: AND operation energy: AND=‘1’ case, ideal drivers.

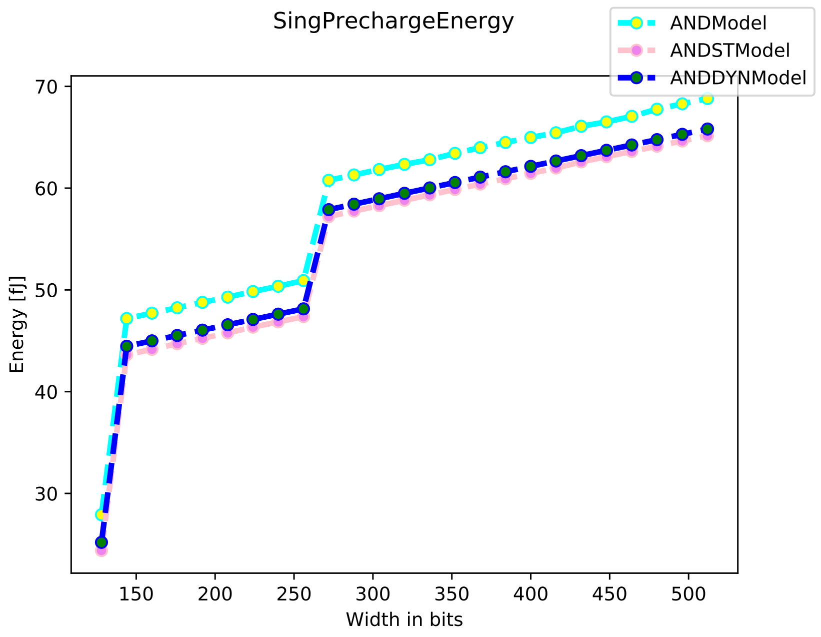


Figure 7.59: AND operation energy: pre-discharge cycle, ideal drivers.

In Figure 7.60, the energy consumption associated to an AND operation whose result is ‘0’, is shown.

As in Figure 7.58, the largest energy consumption is associated to `ANDModel`, for the same reasons discussed before.

In Figure 7.61, a comparison between the energies associated to the possible results of an AND operation, is shown.

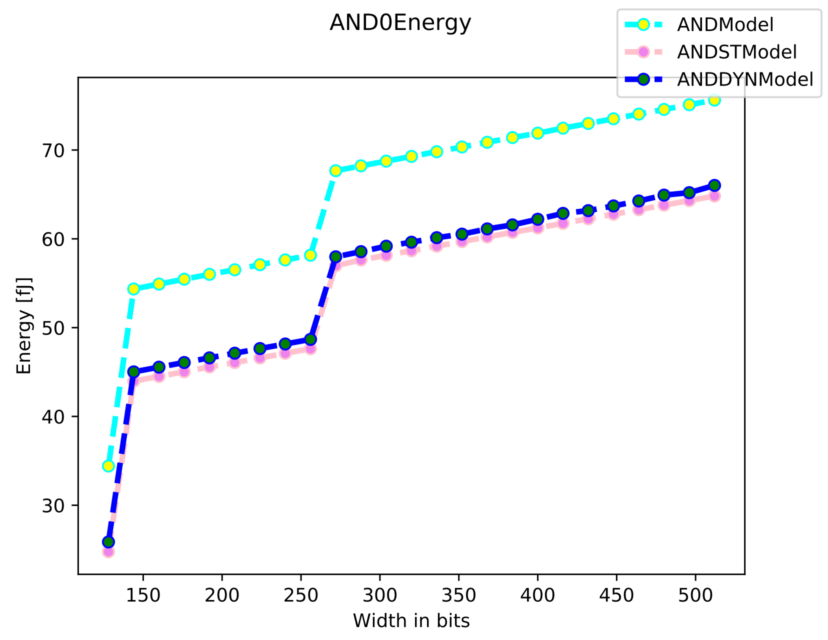


Figure 7.60: AND operation energy: AND=‘0’ case, ideal drivers.

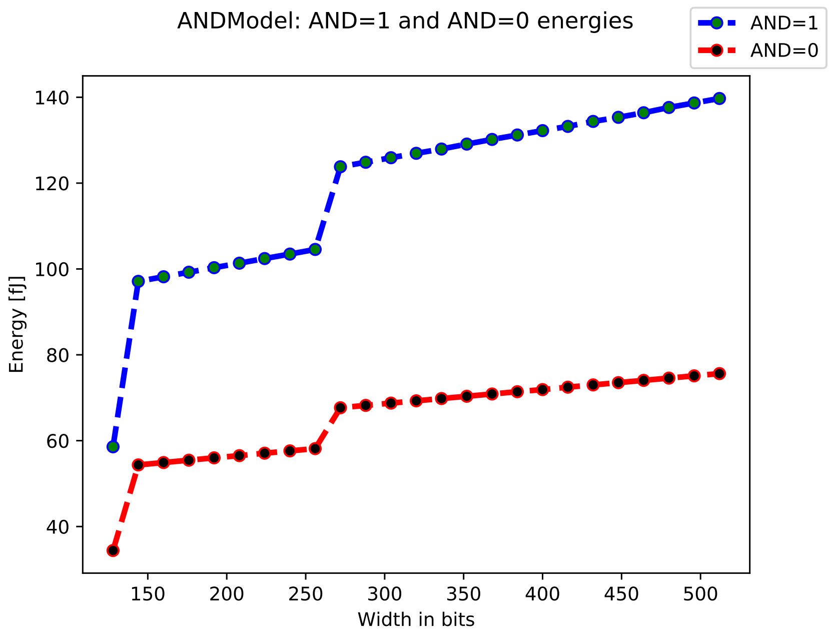


Figure 7.61: Comparison between AND=1 and AND=0 energies for `ANDModel`.

As in subsection 7.1.7, for `ANDModel` a larger energy consumption for the ‘1’ than the ‘0’ is registered. The same considerations made in subsection 7.1.7 hold true.

7.2.8 AND operation: non-ideal drivers

In this section, the AND operation as function of the array height is analyzed, using the results provided by simulations in which non-ideal drivers have been instantiated.

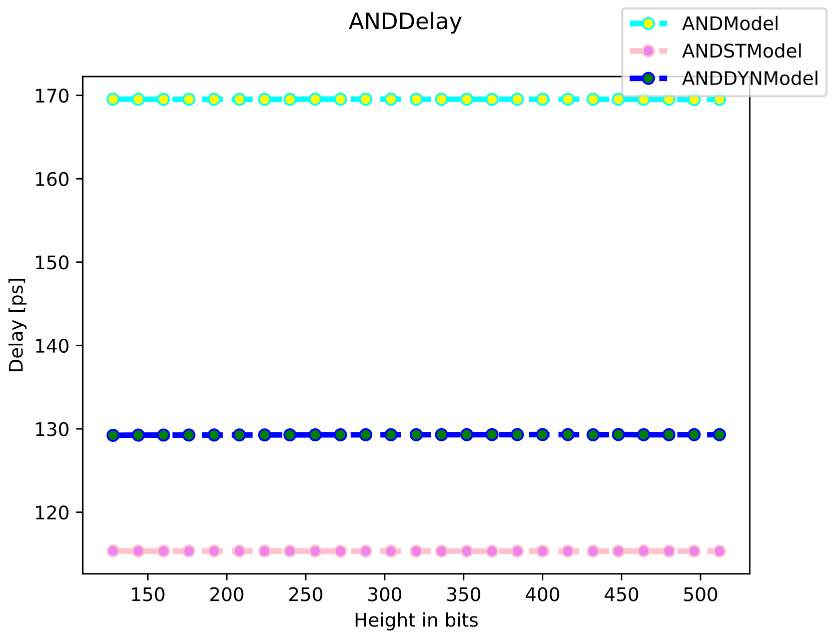


Figure 7.62: AND operation delay: non-ideal drivers.

In Figure 7.62, the AND operation delay as function of the array height is shown.

One can notice that, differently from Figure 7.57, the `ANDDYNModel` array presents a delay larger than `ANDSTModel`. This is due to the fact that a real driver is used to distribute the precharge signal of the dynamic cells on the row and, so, worse performance are obtained for the dynamic LiM array. This has already been discussed in subsection 7.1.8.

In Figure 7.63 the energy consumption associated to an AND result equal to ‘1’, is shown.

Differently from the ideal case (Figure 7.58), the largest energy consumption is now associated to the dynamic array, `ANDDYNModel`. This is due to the presence of the precharge signal driver, that causes an enlargement in the conduction time of the sense amplifier and, so, to an increase in the energy consumption associated to the AND operation, as discussed in subsection 7.1.8.

Since the array height is being increased, the enlargement in the energy consumption is even higher, since the the dummy ANDSA requires a larger time to switch its output and, so, to disable the real sense amplifier.

In Figure 7.64, the energy consumption associated to the pre-discharge cycle is shown.

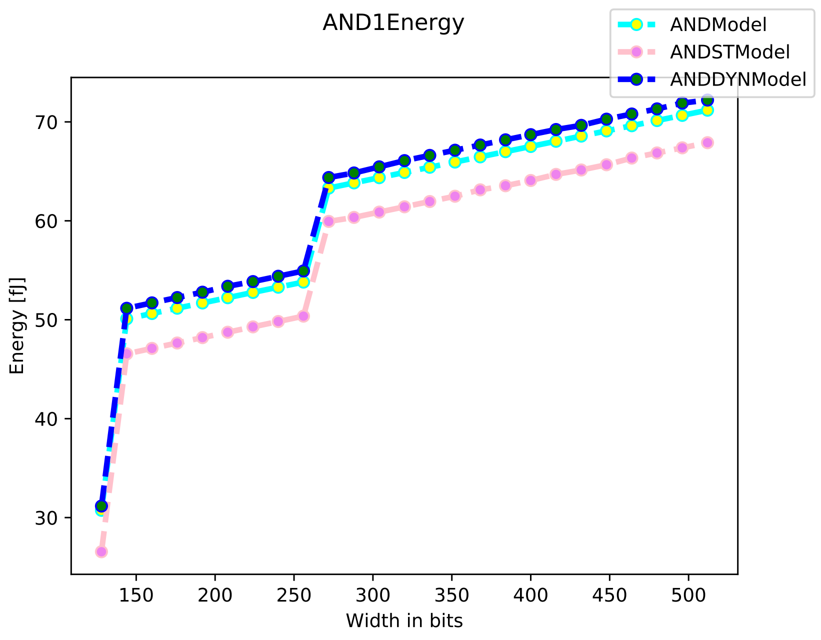


Figure 7.63: AND operation energy: AND='1' case, non-ideal drivers.

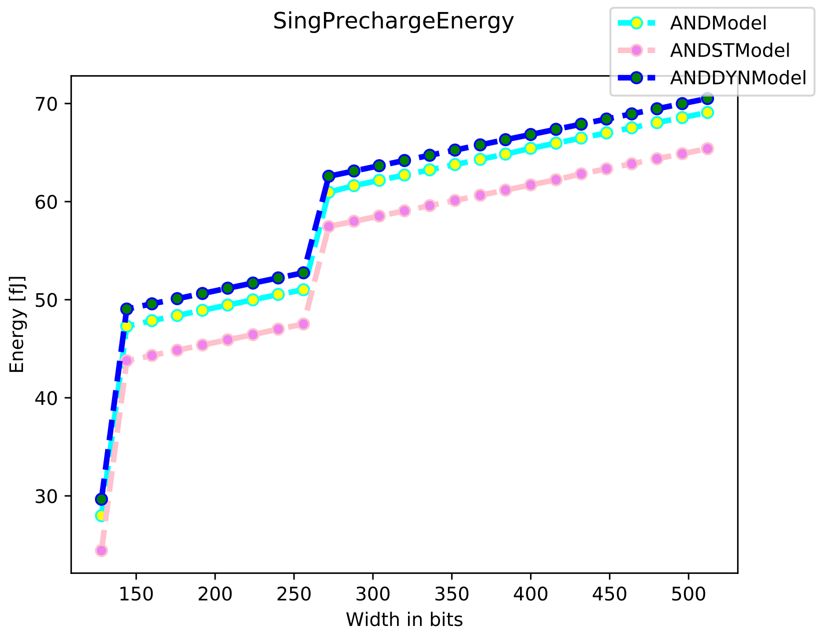


Figure 7.64: AND operation energy: pre-discharge cycle, non-ideal drivers.

Also in this case, the most consuming architecture results to be `ANDDYNModel`, for the same reasons discussed before: the non-ideality of the driver that distributes the precharge signal to the dynamic cells on the row leads to an increase in the energy consumption and to a decrease in the performance.

In Figure 7.65, the energy associated to an AND operation whose result is '0', is shown.

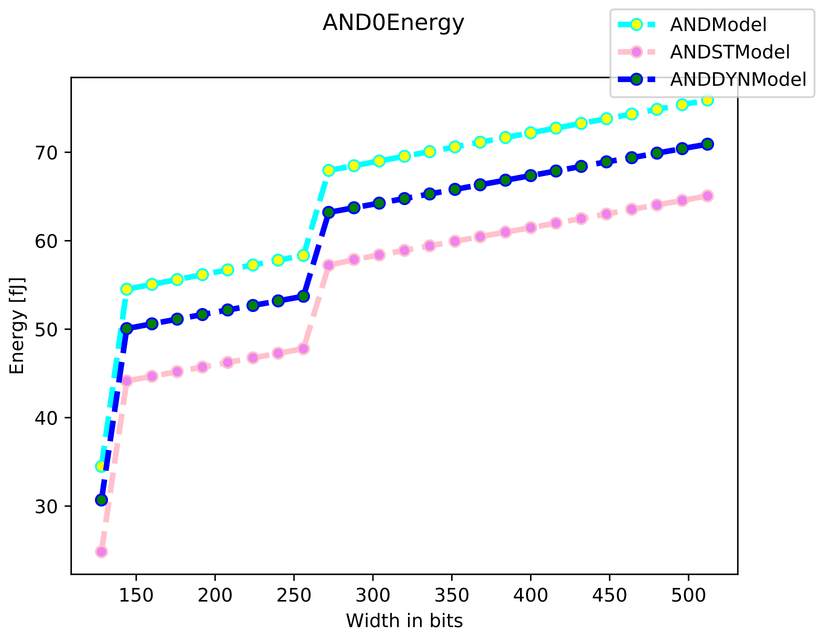


Figure 7.65: AND operation energy: AND='0' case, non-ideal drivers.

In this case, the `ANDDYNModel` is the second most consuming architecture. As in Figure 7.35, this is due to the fact that, since the internal AND node is not discharged (Figure 7.33), the AND-line transistor does not need to be disabled and, so, no increase in the energy consumption is registered. However, since the precharge signal is still being distributed by a non-ideal driver, the dynamic architecture results to be more energy consuming than the static one.

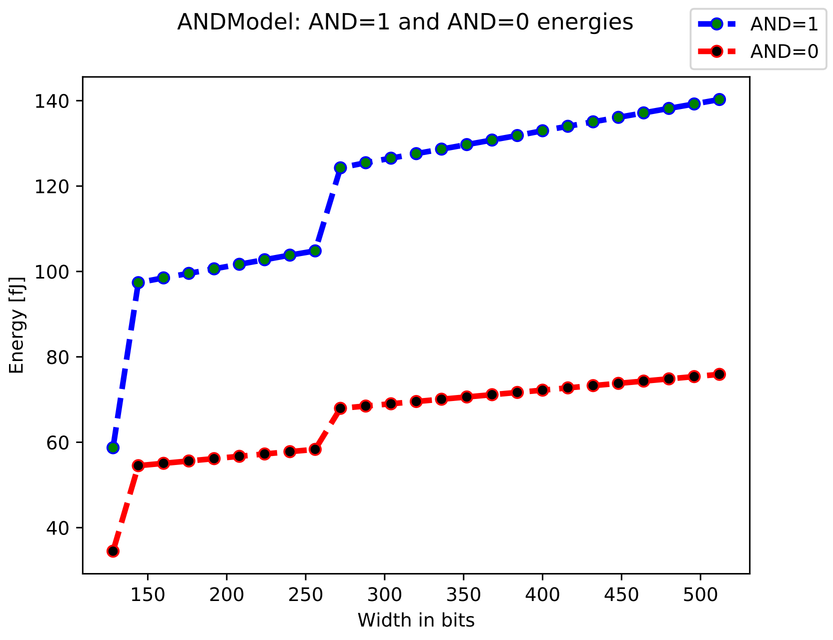


Figure 7.66: Comparison between AND=1 and AND=0 energies for `ANDModel`.

In Figure 7.66, a comparison between the energies associated to the possible results of an AND operation, is shown.

As in subsection 7.1.8, for `ANDModel` a larger energy consumption for the '1' than the '0' is registered. The same considerations made in subsection 7.1.8 hold true.

7.3 Conclusions and future works

A Logic-In-Memory array for maximum/minimum value has been realized in this thesis.

First, standard memory architectures, such as CAM and SRAM, have been studied and realized, in order to obtain reference architecture to which the LiM one could be compared.

Then, the LiM array has been realized, adding analog and digital circuits to the memory array in order to perform a bitwise AND operation, which is the basis of the proposed maximum/minimum algorithm [1]. This array has been designed starting from the cell topology, of which three variants have been proposed, and the sensing scheme, that has been inspired by the CAM architecture.

The design, then, has been generalized realizing a software tool named ALiAS, that allows to synthesize and simulate an array of arbitrary dimensions, allowing to characterize its performance and consumptions as functions of the array size.

From the results presented in section 7.1 and section 7.2, it can be noticed how the AND operation performs way better than the search one. This suggests that is more convenient to perform a maximum/minimum search in memory using a LiM array instead of a CAM one (or another standard memory architecture). The possibility to perform the research in memory surely involves a lower power consumption than a standard system made by a CPU and a memory that exchange data continuously; however, in this thesis it has been demonstrated that it is even more convenient from a performance point of view.

Many characteristics of ALiAS can be improved:

- the synthesis algorithm for the wordline and bitline drivers is very rough. By refining this algorithm, one would be able to obtain more realistic results.
- the cell and some other array parts layout could be realized, in order to get realistic results from the simulation and to choose proper values for the parasitic circuits that have been included in the architecture.
- part of the around-memory logic proposed in [1] could be realized to take into account its presence in the performance estimations.
- different sensing schemes can be implemented to obtain better performance. In fact, the sensing scheme adopted in this work is particularly tuned for the CAM architecture, and not for the LiM operation; however, it has been used to better compare the CAM and LiM performance.

That's all Folks!

Bibliography

[1] A. Chattopadhyay M. Vacca, Y. Tavva and A. Calimera. Logic-In-Memory Architecture For Min/Max Search. *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS).*

[2] Kevin Zhang and Yamauchi Hiroyuki. Embedded memories for nano-scale VLSIs, Embedded SRAM design in nanometer-scale technologies. pages 39–69, 2009.

[3] Kevin Zhang, Naveen Verma, and Anantha P. Chandrakasan. Embedded memories for nano-scale VLSIs, Ultra Low Voltage SRAM Design. pages 89–124, 2009.

[4] E. Seevinck, F.j. List, and J. Lohstroh. Static-noise margin analysis of MOS SRAM cells. *IEEE Journal of Solid-State Circuits*, 22(5):748–754, 1987.

[5] Hiroyuki Yamauchi. Embedded SRAM circuit design technologies for a 45nm and beyond. *2007 7th International Conference on ASIC*, 2007.

[6] Hiroyuki Yamauchi. Embedded SRAM trend in nano-scale CMOS. *2007 IEEE International Workshop on Memory Technology, Design and Testing*, 2007.

[7] Yeonbae Chung and Sang-Won Shim. An Experimental 0.8 V 256-kbit SRAM Macro with Boosted Cell Array Scheme. *ETRI Journal*, 29(4):457–462, Mar 2007.

[8] Yasuhiro Morita, Hidehiro Fujiwara, Hiroki Noguchi, Yusuke Iguchi, Koji Nii, Hiroshi Kawaguchi, and Masahiko Yoshimoto. An Area-Conscious Low-Voltage-Oriented 8T-SRAM Design under DVS Environment. *2007 IEEE Symposium on VLSI Circuits*, 2007.

[9] Y. Tsukamoto, K. Nii, S. Imaoka, Y. Oda, S. Ohbayashi, T. Yoshizawa, H. Makino, K. Ishibashi, and H. Shinohara. Worst-case analysis to obtain stable read/write DC margin of high density 6T-SRAM-array with local Vth variability. *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*

[10] Makoto Yabuuchi, Koji Nii, Yasumasa Tsukamoto, Shigeki Ohbayashi, Susumu Imaoka, Hiroshi Makino, Yoshinobu Yamagami, Satoshi Ishikura, Toshio Terano, Toshiyuki Oashi, and et al. A 45nm Low-Standby-Power Embedded SRAM with Improved Immunity Against Process and Temperature Variations. *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, 2007.

[11] K. Zhang, U. Bhattacharya, Z. Chen, F. Hamzaoglu, D. Murray, N. Vallepalli, Y. Wang, B. Zheng, and M. Bohr. A 3-GHz 70-Mb SRAM in 65-nm CMOS Technology With Integrated Column-Based Dynamic Power Supply. *IEEE Journal of Solid-State Circuits*, 41(1):146–151, 2006.

[12] Fatih Hamzaoglu, Kevin Zhang, Yih Wang, Hong Jo Ahn, Uddalak Bhattacharya, Zhanping Chen, Yong-Gee Ng, Andrei Pavlov, Ken Smits, Mark Bohr, and et al. A 153Mb-SRAM Design with Dynamic Stability Enhancement and Leakage Reduction in 45nm High-Â¿ Metal-Gate CMOS Technology. *2008 IEEE International*

Solid-State Circuits Conference - Digest of Technical Papers, 2008.

[13] K. Osada, Y. Saitoh, E. Ibe, and K. Ishibashi. 16.7 fA/cell tunnel-leakage-suppressed 16 Mb SRAM for handling cosmic-ray-induced multi-errors. *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*

[14] Masanao Yamaoka, Noriaki Maeda, Yasuhisa Shimazaki, and Kenichi Osada. 65nm Low-Power High-Density SRAM Operable at 1.0V under 3σ Systematic Variation Using Separate Vth Monitoring and Body Bias for NMOS and PMOS. *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008.

[15] L. Chang, D.m. Fried, J. Hergenrother, J.w. Sleight, R.h. Dennard, R.k. Montoye, L. Sekaric, S.j. Mcnab, A.w. Topol, C.d. Adams, and et al. Stable SRAM cell design for the 32 nm node and beyond. *Digest of Technical Papers. 2005 Symposium on VLSI Technology, 2005.*

[16] Benton Highsmith Calhoun and Anantha P. Chandrakasan. A 256-kb 65-nm Subthreshold SRAM Design for Ultra-Low-Voltage Operation. *IEEE Journal of Solid-State Circuits*, 42(3):680–688, 2007.

[17] Ik Joon Chang, Jae-Joon Kim, Sang Phill Park, and Kaushik Roy. A 32kb 10T Subthreshold SRAM Array with Bit-Interleaving and Differential Read Scheme in 90nm CMOS. *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008.

[18] Tae-Hyoung Kim, Jason Liu, John Keane, and Chris H. Kim. A High-Density Subthreshold SRAM with Data-Independent Bitline Leakage and Virtual Ground Replica Scheme. *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, 2007.

[19] K. Pagiamtzis and A. Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006.

[20] Teuvo Kohonen. Logic Principles of Content-Addressable Memories. *Content-Addressable Memories Springer Series in Information Sciences*, pages 125–189, 1987.

[21] L. Chisvin and R.j. Duckworth. Content-addressable and associative memory: alternatives to the ubiquitous RAM. *Computer*, 22(7):51–64, 1989.

[22] K.e. Grosspietsch. Associative processors and memories: a survey. *IEEE Micro*, 12(3):12–19, 1992.

[23] S. Stas. Associative processing with CAMs. *Proceedings of NORTHCON93 Electrical and Electronics Convention.*

[24] I.n. Robinson. Pattern-addressable memory. *IEEE Micro*, 12(3):20–30, 1992.

[25] T.-B. Pei and C. Zukowski. VLSI implementation of routing tables: tries and CAMs. *IEEE INFCOM 91. The conference on Computer Communications. Tenth Annual Joint Comference of the IEEE Computer and Communications Societies*

Proceedings, 1991.

[26] A.j. Mcauley and P. Francis. Fast routing table lookup using CAMs. *IEEE INFOCOM 93 The Conference on Computer Communications, Proceedings.*

[27] Jinn-Shyan Wang, Hung-Yu Li, Chia-Cheng Chen, and Chingwei Yeh. An AND-type match-line scheme for energy-efficient content addressable memories. *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*

[28] Sungdae Choi, K. Sohn, and Hoi-Jun Yoo. A 0.7-fJ/bit/search 2.2-ns search time hybrid-type TCAM architecture. *IEEE Journal of Solid-State Circuits*, 40(1):254–260, 2005.

[29] G. Kasai, Y. Takarabe, K. Furumi, and M. Yoneda. 200MHz/200MSPS 3.2W at 1.5V Vdd, 9.4Mbits ternary CAM with new charge injection match detect circuits and bank selection scheme. *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.*

[30] M.m. Khellah and M.i. Elmasry. Use of charge sharing to reduce energy consumption in wide fan-in gates. *ISCAS 98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No.98CH36187).*

[31] A. Igor, C. Trevis, and A. Sheikholeslami. A ternary content-addressable memory (TCAM) based on 4T static storage and including a current-race sensing scheme. *IEEE Journal of Solid-State Circuits*, 38(1):155–158, 2003.

[32] C.a. Zukowski and Shao-Yi Wang. Use of selective precharge for low-power content-addressable memories. *Proceedings of 1997 IEEE International Symposium on Circuits and Systems. Circuits and Systems in the Information Age ISCAS 97.*

[33] A. Roth, D. Foss, R. Mckenzie, and D. Perry. Advanced ternary CAM circuits on 0.13 Î¼m logic process technology. *Proceedings of the IEEE 2004 Custom Integrated Circuits Conference (IEEE Cat. No.04CH37571).*

[34] Ilion Yi-Liang Hsiao, Ding-Hao Wang, and Chein-Wei Jen. Power modeling and low-power design of content addressable memories. *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196).*

[35] A. Efthymiou and J.d. Garside. An adaptive serial-parallel CAM architecture for low-power cache blocks. *Proceedings of the International Symposium on Low Power Electronics and Design*, 2002.

[36] A. Efthymiou and J.d. Garside. A CAM with mixed serial-parallel comparison for use in low energy caches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):325–329, 2004.

[37] N. Mohan and M. Sachdev. Low power dual matchline ternary content addressable memory.

[38] Kuo-Hsing Cheng, Chia-Hung Wei, and Shu-Yu Jiang. Static divided word matching line for low-power Content Addressable Memory design. *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512).*

[39] K. Pagiamtzis and A. Sheikholeslami. A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme. *IEEE Journal of Solid-State Circuits*, 39(9):1512–1519, 2004.

[40] K. Pagiamtzis and A. Sheikholeslami. Pipelined match-lines and hierarchical search-lines for low-power content-addressable memories. *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.*

[41] I.m. Hyjazie and Chunyan Wang. An approach for improving the speed of content addressable memories. *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS 03.*

[42] I. Arsovski and A. Sheikholeslami. A current-saving match-line sensing scheme for content-addressable memories. *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*

[43] I. Arsovski and A. Sheikholeslami. A mismatch-dependent power allocation technique for match-line sensing in content-addressable memories. *IEEE Journal of Solid-State Circuits*, 38(11):1958–1966, 2003.

[44] H. Noda, K. Inoue, M. Kuroiwa, F. Igaue, K. Yamamoto, H.j. Mattausch, T. Koide, A. Amo, A. Hachisuka, S. Soeda, and et al. A cost-efficient high-performance dynamic TCAM with pipelined hierarchical searching and shift redundancy architecture. *IEEE Journal of Solid-State Circuits*, 40(1):245–253, 2005.

[45] H. Noda, K. Inoue, M. Kuroiwa, A. Amo, A. Hachisuka, H.j. Mattausch, T. Koide, S. Soeda, K. Dosaka, K. Arinnoto, and et al. A 143MHz 1.1W 4.5Mb dynamic TCAM with hierarchical searching and shift redundancy architecture. *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No.04CH37519).*

[46] B. Wicht, T. Nirschl, and D. Schmitt-Landsiedel. Yield and speed optimization of a latch-type voltage sense amplifier. *IEEE Journal of Solid-State Circuits*, 39(7):1148–1158, 2004.