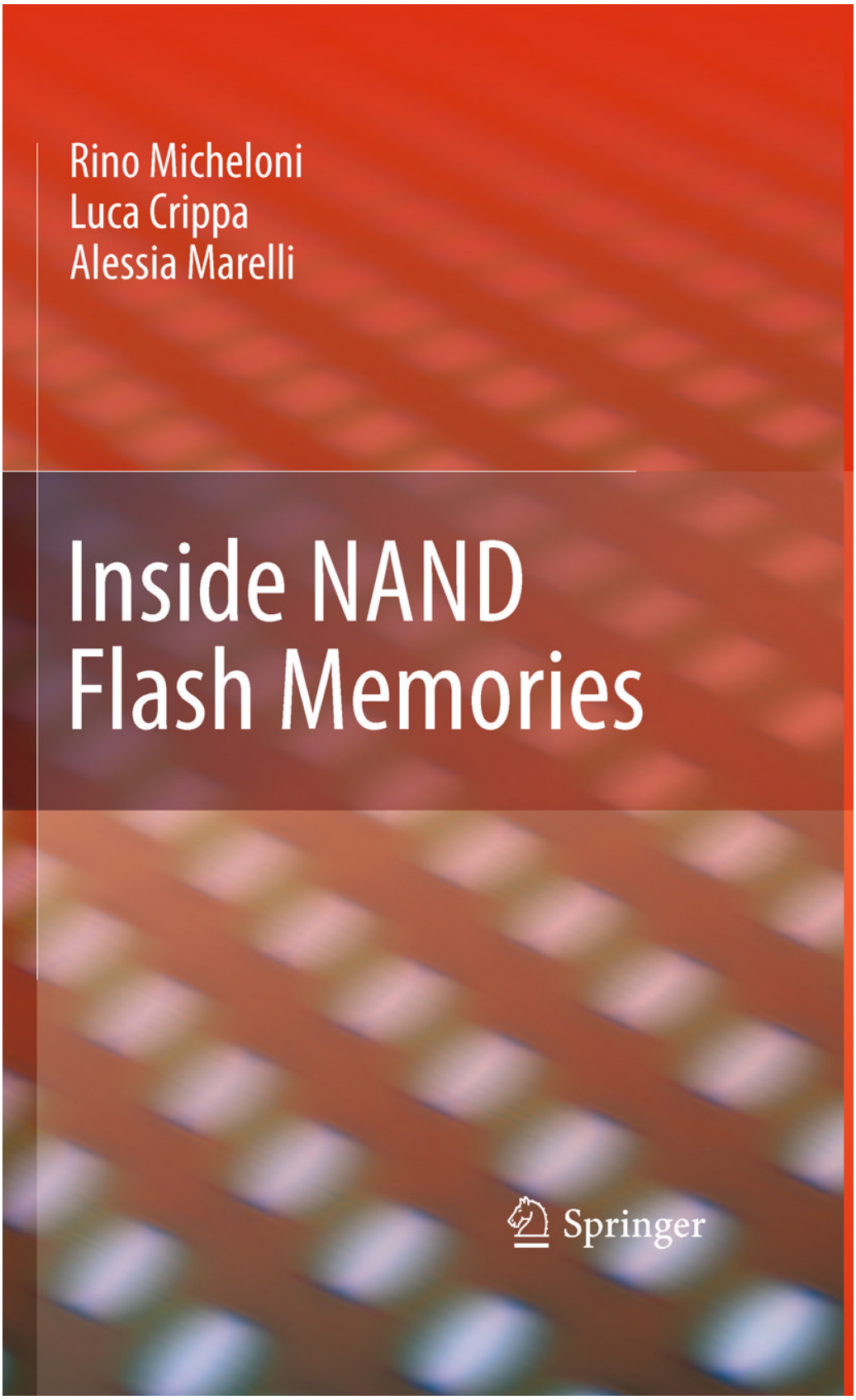

Rino Micheloni
Luca Crippa
Alessia Marelli
Inside NAND
Flash Memories
Springer

Inside NAND Flash Memories

Rino Micheloni • Luca Crippa • Alessia Marelli

Inside NAND Flash Memories

Springer

Rino Micheloni
Integrated Device Technology
Agrate Brianza
Italy
rino.micheloni@ieee.org

Luca Crippa
Forward Insights
North York
Canada
luca.crippa@ieee.org

Alessia Marelli
Integrated Device Technology
Agrate Brianza
Italy
alessiamarelli@gmail.com

ISBN 978-90-481-9430-8 e-ISBN 978-90-481-9431-5
DOI 10.1007/978-90-481-9431-5
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2010931597

© Springer Science+Business Media B.V. 2010
No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Cover design: eStudio Calamar S.L.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

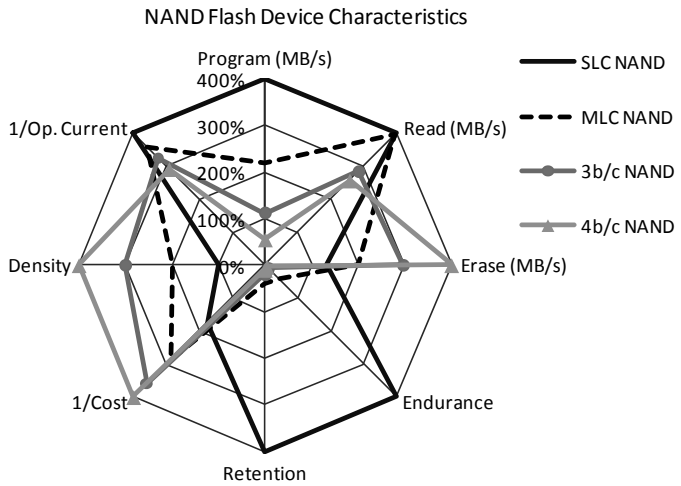


Fig. 1.10. NAND device characteristics (Source: Forward Insights)

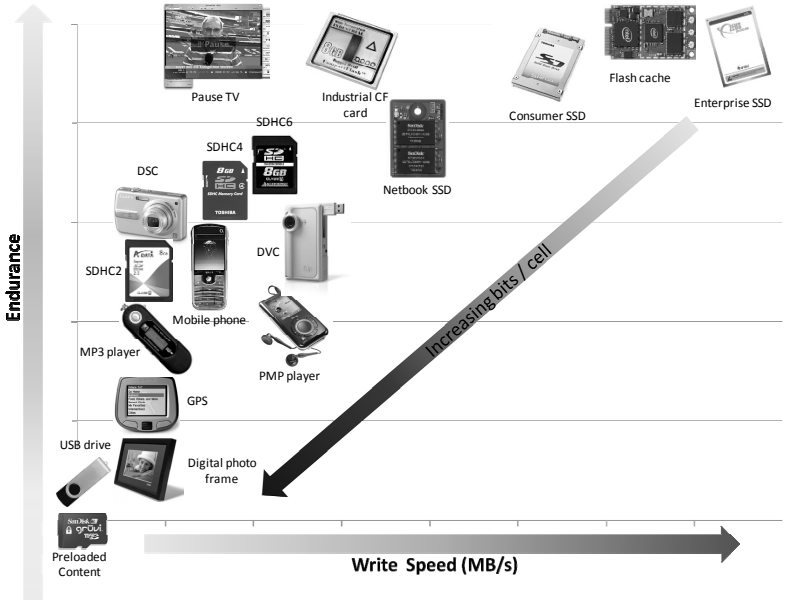


Fig. 1.11. Applications for multi-bit per cell NAND Flash memories (Source: Forward Insights)

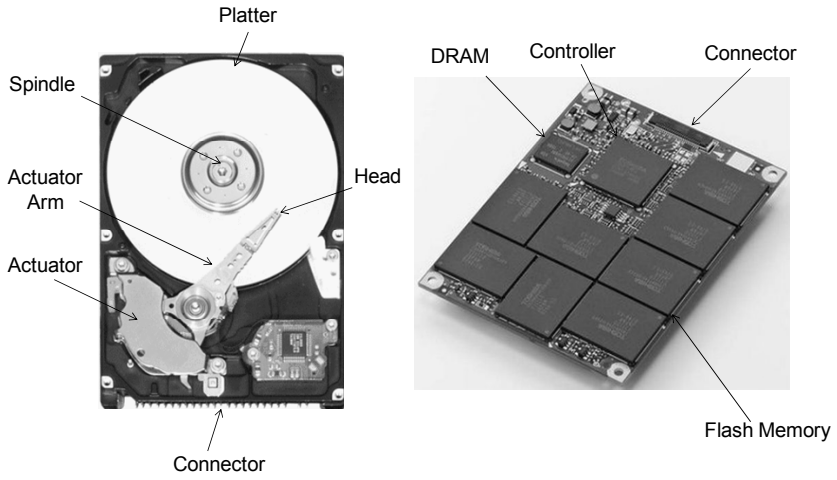


Fig. 1.16. Solid state drive versus hard disk drive (Source: Toshiba Corp., Forward Insights)

Figure 1.16 details the major components of an SSD and HDD. The HDD, being based on storage in a spinning magnetic platter, requires an actuator and actuator arm to move the head to the appropriate sector to be read or written. This movement of the read head results in extremely long latencies in the milliseconds and a DRAM buffer is used to hide the seek time. The HDD controller, particularly if it is a system-on-chip, incorporates the processor, servo control logic, interface, error correction code, disk sequencer and buffer controller.

In contrast, the SSD contains no mechanical parts and consists of a few major components: NAND Flash memory, SSD controller, connector, DRAM, PCB and passives. In addition, because of the small size of NAND Flash memory, the SSD form factor is not limited to standard 1.8, 2.5 or 3.5″ HDD form factors but can also come in module form.

NAND-based SSDs started out in industrial and military where ruggedness and reliability are a priority. In 2006, the first SSD for personal computers was introduced followed by SSDs for enterprise computing.

Table 1.1 summarizes the NAND Flash/SSD usage models in the different market segments. There are several ways NAND Flash or SSDs may utilized. Firstly, it can act as an alternate storage device, in many cases replacing a HDD for latency or reliability improvement and power consumption reduction. An optimized OS or file-system is required to obtain the full benefits of the SSD.

Secondly, the SSD may act as an I/O accelerator. I/O acceleration is primarily used in the enterprise computing environment. A software driver is required for this approach.

Thirdly, the SSD/NAND Flash may be part of the tiered system memory and replace part of the DRAM. Due to the better economics of NAND Flash versus DRAM, this allows an increase in system memory storage capacity with the

The most popular Flash memory cell is based on the *Floating Gate* (FG) technology, whose cross section is shown in Fig. 2.1. A MOS transistor is built with two overlapping gates rather than a single one: the first one is completely surrounded by oxide, while the second one is contacted to form the gate terminal. The isolated gate constitutes an excellent "trap" for electrons, which guarantees charge retention for years. The operations performed to inject and remove electrons from the isolated gate are called program and erase, respectively. These operations modify the threshold voltage V_{TH} of the memory cell, which is a special type of MOS transistor. Applying a fixed voltage to cell's terminals, it is then possible to discriminate two storage levels: when the gate voltage is higher than the cell's V_{TH}, the cell is on ("1"), otherwise it is off ("0").

It is worth mentioning that, due to floating gate scalability reasons, charge trap memories are gaining more and more attention and they are described in Chap. 5, together with their 3D evolution.

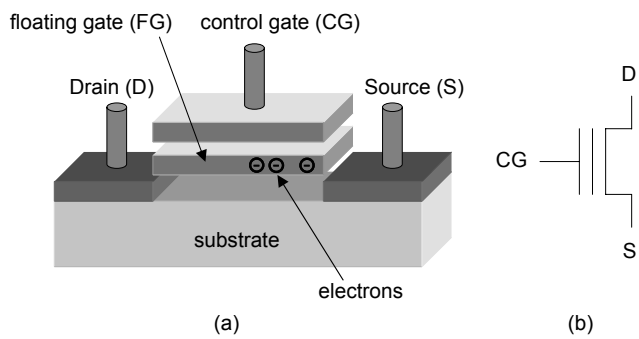


Fig. 2.1. (a) Floating gate memory cell and (b) its schematic symbol

2.2 NAND memory

2.2.1 Array

The memory cells are packed to form a matrix in order to optimize silicon area occupation. Depending on how the cells are organized in the matrix, it is possible to distinguish between NAND and NOR Flash memories. This book is about NAND memories as they are the most widespread in the storage systems. NOR architecture is described in great details in [1].

In the NAND string, the cells are connected in series, in groups of 32 or 64, as shown in Fig. 2.2. Two selection transistors are placed at the edges of the string, to ensure the connections to the source line (through M_{SL}) and to the bitline (through M_{DL}). Each NAND string shares the bitline contact with another string. Control gates are connected through wordlines (WLs).

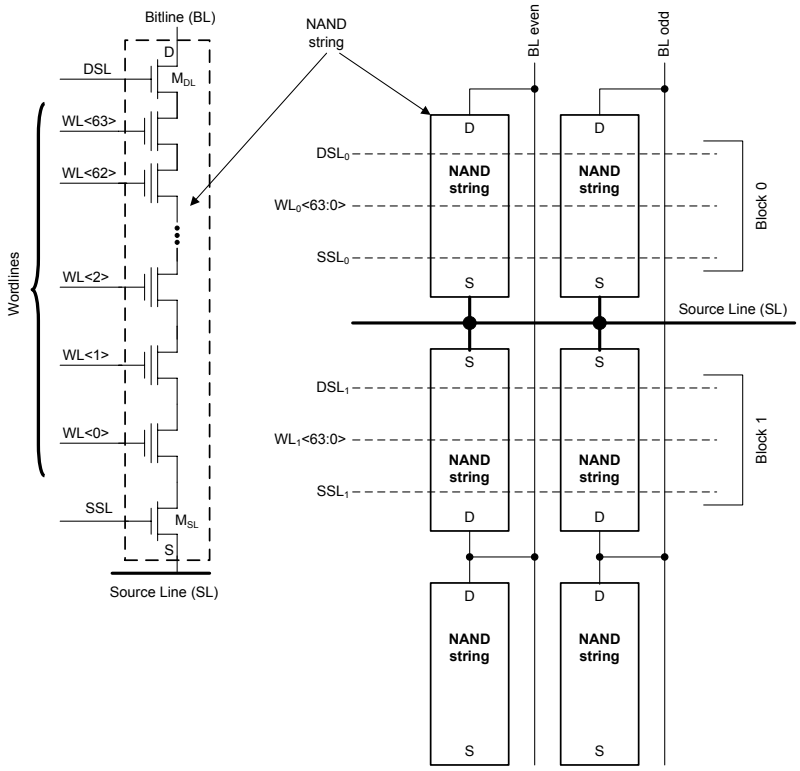


Fig. 2.2. NAND string (left) and NAND array (right)

Logical pages are made up by cells belonging to the same wordline. The number of pages per wordline is related to the storage capabilities of the memory cell. Depending on the number of storage levels, Flash memories are referred to in different ways: SLC memories store 1 bit per cell, MLC memories (Chap. 10) store 2 bits per cell, 8LC memories (Chap. 16) store 3 bits per cell and 16LC memories (Chap. 16) store 4 bits per cell.

If we consider the SLC case with interleaved architecture (Chap. 8), even and odd cells form two different pages. For example, a SLC device with 4 kB page has a wordline of 65,536 cells.

Of course, in the MLC case there are four pages as each cell stores one *Least Significant Bit* (LSB) and one *Most Significant Bit* (MSB). Therefore, we have:

- MSB and LSB pages on even bitlines
- MSB and LSB pages on odd bitlines

where V_{THN} indicates the threshold voltage value of the NMSO M_N.

At this point, the M_N and M_P transistors are switched off. C_{BL} is free to discharge. After a time T_{VAL}, the gate of M_N is biased at $V_2 < V_1$, usually 1.6–1.4 V.

If a T_{VAL} time has elapsed long enough to discharge the bitline voltage under the value:

$$V_{BL} < V_2 - V_{THN} \tag{2.2}$$

M_N turns on and the voltage of node OUT (V_{OUT}) becomes equal to the one of the bitline. Finally, the analog voltage V_{OUT} is converted into a digital format by using simple latches.

A more detailed analysis of read techniques is presented in Chap. 8.

Program

Programming of NAND memories exploits the quantum-effect of electron tunneling in the presence of a strong electric field (Fowler–Nordheim tunneling [2]). In particular, depending on the polarity of the electric field applied, program or erase take place. Please refer to Chap. 3 for more details.

During programming, the number of electrons crossing the oxide is a function of the electric field: in fact, the greater such field is, the greater the injection probability is. Thus, in order to improve the program performances, it is essential to have high electric fields available and therefore high voltages (Chaps. 11 and 12). This requirement is one of the main drawbacks of this program method, since the oxide degradation is impacted by these voltages.

The main advantage is the current required, which is definitely low, in the range of nanoamperes per cell. This is what makes the Fowler–Nordheim mechanism suitable for a parallel programming of many cells as required by NAND page sizes.

The algorithm used to program the cells of a NAND memory is a *Program & Verify* algorithm (Chaps. 3 and 12): verify is used to check whether the cell has reached the target distribution or not.

In order to trigger the injection of electrons into the floating gate, the following voltages are applied, as shown in Fig. 2.5:

- V_{DD} on the gate of the drain selector
- $V_{PASS,P}$ (8–10 V) on the unselected gates
- V_{PGM} (20–25 V) on the selected gate (to be programmed)
- GND on the gate of the source selector
- GND on the bitlines to be programmed
- V_{DD} on other bitlines

The so-called *self-boosting* mechanism (Chap. 3) prevents the cells sharing the same gate with the programmed one from undergoing an undesired program. The basic idea is to exploit the high potentials involved during programming through

the parasitic capacitors of the cell, in order to increase the potential of the region underneath the tunnel oxide (inset of Fig. 2.5).

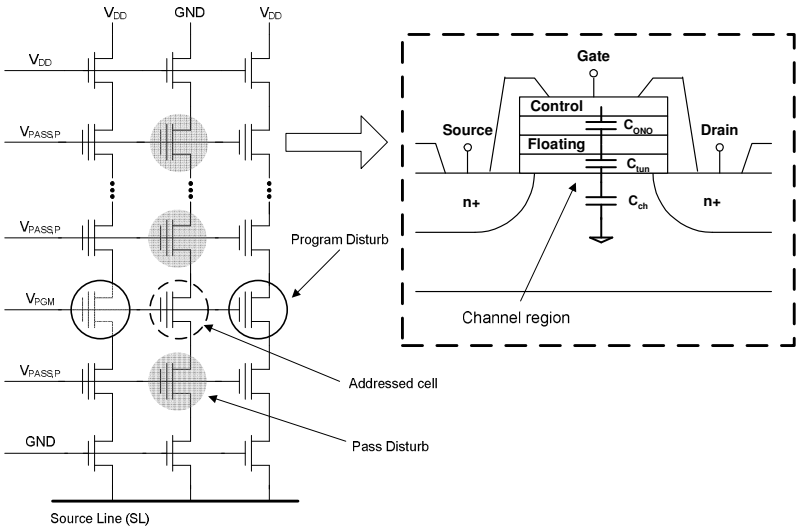


Fig. 2.5. NAND string biasing during program and parasitic capacitors contributing to the self-boosting inhibit

When the bitlines are driven to V_{DD}, drain transistors are diode-connected and the corresponding bitlines are floating. When $V_{PASS,P}$ is applied to the unselected wordlines, parasitic capacitors boost the potential of the channel, reducing the voltage drop across the tunnel oxide and, hence, inhibiting the tunneling phenomena.

As the memory cells are organized in a matrix, all the cells along the wordline are biased at the same voltage even if they are not intended to be programmed, i.e. they are "disturbed". Two important typologies of disturbs are related to the program operation: the *Pass disturb* and the *Program disturb,* as shown in Fig. 2.5: their impact on reliability is described in Chap. 4.

Erase

NAND memory is placed in a triple-well structure, as shown in Fig. 2.6a. Usually, each plane has its own triple-well. The source terminal is shared by all the blocks: in this way the matrix is more compact and the multiplicity of the structures which bias the iP-well is drastically reduced.

The electrical erase is achieved by biasing the iP-well with a high voltage and keeping grounded the wordlines of the sector to be erased (Fig. 2.6c). Therefore, NAND technologies don't need negative voltages. Again, the physical mechanism is the Fowler–Nordheim tunneling. As the iP-well is common to all the blocks,

erase of unselected blocks is prevented leaving their wordlines floating. In this way, when the iP-well is charged, the potential of the floating wordlines raises thanks to the capacitive coupling between the control gates and the iP-well (Fowler–Nordheim tunneling inhibited).

Figure 2.6b sketches the erase algorithm phases. NAND specifications are quite aggressive in terms of erase time. Therefore, Flash vendors try to erase the block content in few erase steps (possibly one). As a consequence, a very high electric field is applied to the matrix during the *Electrical Erase* phase. As a matter of fact, erased distribution is deeply shifted towards negative V_{TH} values. In order to minimize floating gate coupling (Chap. 12), a *Program After Erase* (PAE) phase is introduced, with the goal of placing the distribution near the 0 V limit (of course, guaranteeing the appropriate read margin).

Typical erase time of a SLC block is about 1–1.5 ms; electrical erase pulse lasts 700–800 µs.

Technology shrink will ask for more sophisticated erase algorithms, especially for 3–4 bit/cell devices. In fact, reliability margins are usually shrinking with the technology node: a more precise, and therefore time consuming, PAE will be introduced, in order to contain the erased distribution width. In summary, erase time is going to increase to 4–5 ms in the new NAND generations.

Chapter 12 contains a detailed description of the whole erase algorithm.

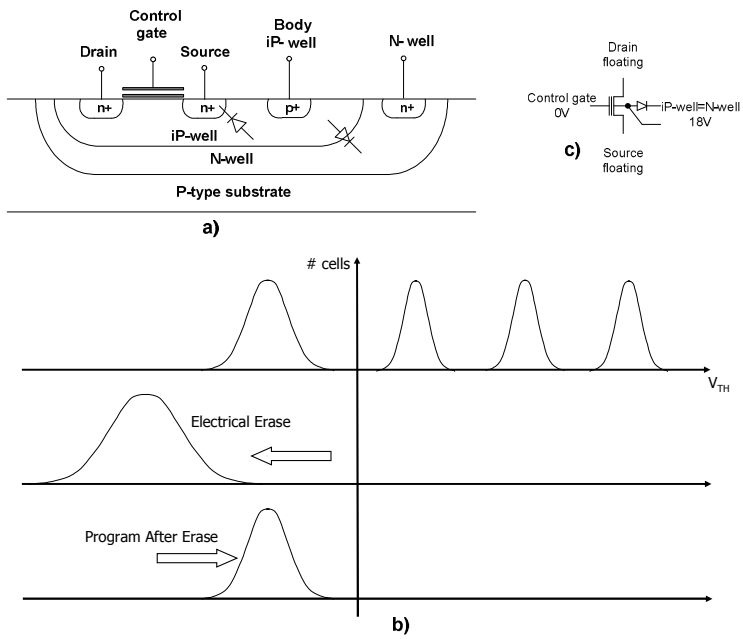


Fig. 2.6. (a) NAND matrix in triple-well; (b) erase algorithm; (c) erase biasing on the selected block

2.2.3 Logic organization

NAND memory contains information organized in a specified way. Looking at Fig. 2.7, a memory is divided in pages and blocks. A block is the smallest erasable unit. Generally, there are a power of two blocks within any device. Each block contains multiple pages. The number of pages within a block is typically a multiple of 16 (e.g. 64, 128). A page is the smallest addressable unit for reading and writing. Each page is composed of main area and spare area (Fig. 2.8). Main area can range from 4 to 8 kB or even 16 kB. Spare area can be used for ECC (Chap. 14) and system pointers (Chap. 17) and it is in the order of a couple of hundreds bytes every 4 kB of main area.

Every time we want to execute an operation on a NAND device, we must issue the address where we want to act. The address is divided in row address and column address (Fig. 2.9). Row address identifies the addressed page, while column address is used to identify the bytes inside the page. When both row and column addresses are required, column address is given first, 8 bits per address cycle. The first cycle contains the least significant bits. Row and column addresses cannot share the same address cycle.

The row address identifies the block and the page involved in the operation. Page address occupies the least significant bits.

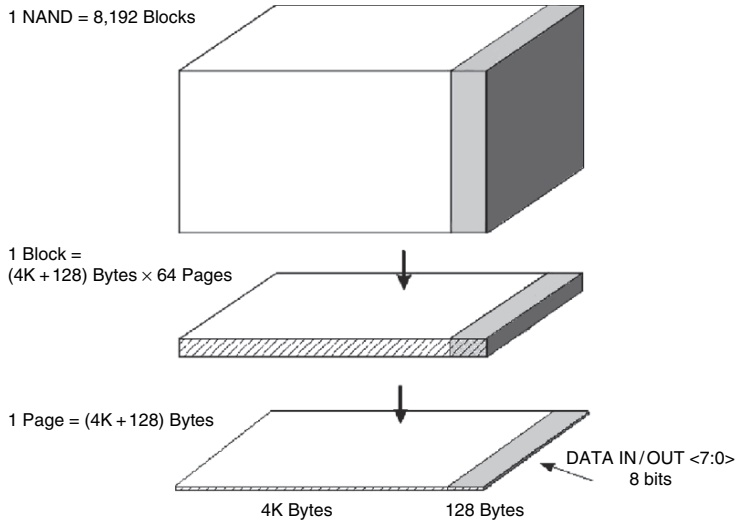


Fig. 2.7. NAND memory logic organization

Fig. 2.8. NAND page structure

All these issues cause a severe limitation to the maximum capacity of the card; in addition external components, like voltage regulators and quartz, cannot be used. In other words, the memory controller of the card has to implement all the required functions.

The assembly stress for small form factors is quite high and, therefore, system testing is at the end of the production. Hence, production cost is higher (Chap. 15).

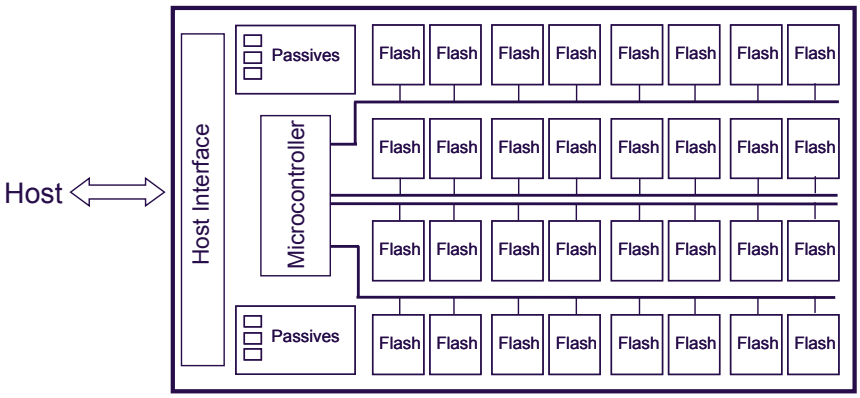


Fig. 2.32. Block diagram of a SSD

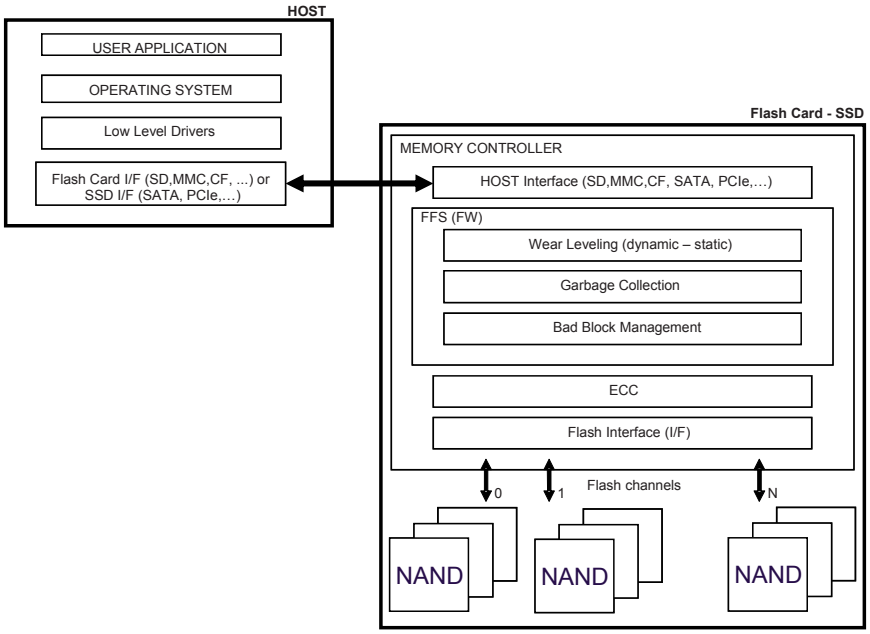


Fig. 2.33. Functional representation of a Flash card (or SSD)

For a more detailed description of Flash cards, please, refer to Chap. 17. SSDs are described in Chap. 18.

Figure 2.33 shows a functional representation of a memory card or SSD: two types of components can be identified: the memory controller and the Flash memory components. Actual implementation may vary, but the functions described in the next sections are always present.

2.4.1 Memory controller

The aim of the memory controller is twofold:

1. To provide the most suitable interface and protocol towards both the host and the Flash memories
2. To efficiently handle data, maximizing transfer speed, data integrity and information retention

In order to carry out such tasks, an application specific device is designed, embedding a standard processor – usually 8–16 bits – together with dedicated hardware to handle timing-critical tasks.

For the sake of discussion, the memory controller can be divided into four parts, which are implemented either in hardware or in firmware. Proceeding from the host to the Flash, the first part is the host interface, which implements the required industry-standard protocol (MMC, SD, CF, etc.), thus ensuring both logical and electrical interoperability between Flash cards and hosts. This block is a mix of hardware – buffers, drivers, etc. – and firmware – command decoding performed by the embedded processor – which decodes the command sequence invoked by the host and handles the data flow to/from the Flash memories.

The second part is the Flash File System (FFS) [6]: that is, the file system which enables the use of Flash cards, SSDs and USB sticks like magnetic disks. For instance, sequential memory access on a multitude of sub-sectors which constitute a file is organized by linked lists (stored on the Flash card itself) which are used by the host to build the File Allocation Table (FAT).

The FFS is usually implemented in form of firmware inside the controller, each sub-layer performing a specific function. The main functions are: *Wear leveling Management*, *Garbage Collection* and *Bad Block Management*. For all these functions, tables are widely used in order to map sectors and pages from logical to physical (Flash Translation Layer or FTL) [7, 8], as shown in Fig. 2.34.

The upper block row is the logical view of the memory, while the lower row is the physical one. From the host perspective, data are transparently written and overwritten inside a given logical sector: due to Flash limitations, overwrite on the same page is not possible, therefore a new page (sector) must be allocated in the physical block and the previous one is marked as invalid. It is clear that, at some point in time, the current physical block becomes full and therefore a second one (Buffer) is assigned to the same logical block.

The required translation tables are always stored on the memory card itself, thus reducing the overall card capacity.

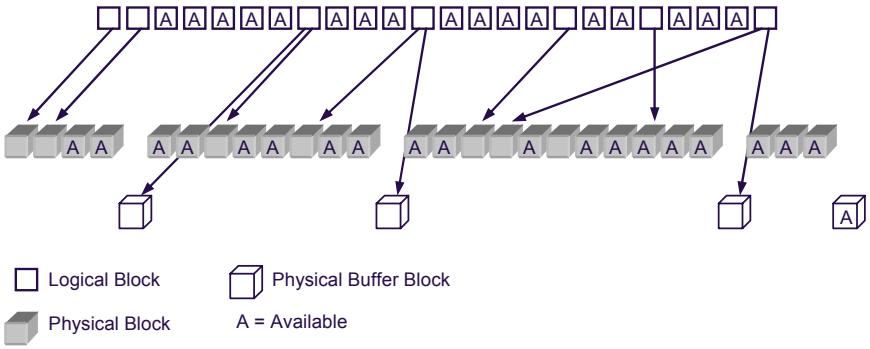


Fig. 2.34. Logical to physical block management

Wear leveling

Usually, not all the information stored within the same memory location change with the same frequency: some data are often updated while others remain always the same for a very long time – in the extreme case, for the whole life of the device. It's clear that the blocks containing frequently-updated information are stressed with a large number of write/erase cycles, while the blocks containing information updated very rarely are much less stressed.

In order to mitigate disturbs, it is important to keep the aging of each page/block as minimum and as uniform as possible: that is, the number of both read and program cycles applied to each page must be monitored. Furthermore, the maximum number of allowed program/erase cycles for a block (i.e. its endurance) should be considered: in case SLC NAND memories are used, this number is in the order of 100 k cycles, which is reduced to 10 k when MLC NAND memories are used.

Wear Leveling techniques rely on the concept of logical to physical translation: that is, each time the host application requires updates to the same (logical) sector, the memory controller dynamically maps the sector onto a different (physical) sector, keeping track of the mapping either in a specific table or with pointers. The out-of-date copy of the sector is tagged as both invalid and eligible for erase. In this way, all the physical sectors are evenly used, thus keeping the aging under a reasonable value.

Two kinds of approaches are possible: Dynamic Wear Leveling is normally used to follow up a user's request of update for a sector; Static Wear Leveling can also be implemented, where every sector, even the least modified, is eligible for re-mapping as soon as its aging deviates from the average value.

Garbage collection

Both wear leveling techniques rely on the availability of free sectors that can be filled up with the updates: as soon as the number of free sectors falls below a given threshold, sectors are "compacted" and multiple, obsolete copies are deleted.

This operation is performed by the Garbage Collection module, which selects the blocks containing the invalid sectors, copies the latest valid copy into free sectors and erases such blocks (Fig. 2.35).

In order to minimize the impact on performance, garbage collection can be performed in background. The equilibrium generated by the wear leveling distributes wear out stress over the array rather than on single hot spots. Hence, the bigger the memory density, the lower the wear out per cell is.

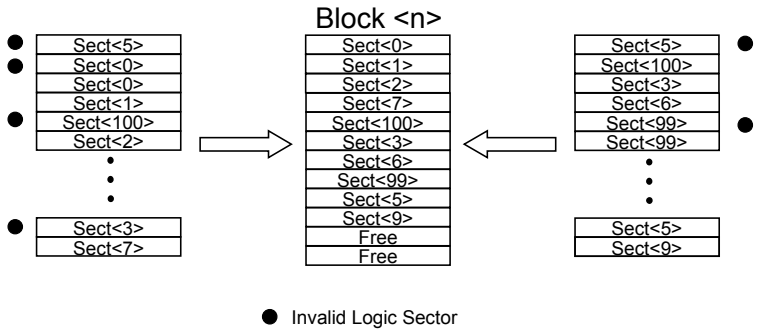


Fig. 2.35. Garbage collection

Bad block management

No matter how smart the Wear Leveling algorithm is, an intrinsic limitation of NAND Flash memories is represented by the presence of so-called *Bad Blocks* (BB), i.e. blocks which contain one or more locations whose reliability is not guaranteed.

The *Bad Block Management* (BBM) module creates and maintains a map of bad blocks, as shown in Fig. 2.36: this map is created during factory initialization of the memory card, thus containing the list of the bad blocks already present during the factory testing of the NAND Flash memory modules. Then it is updated during device lifetime whenever a block becomes bad.

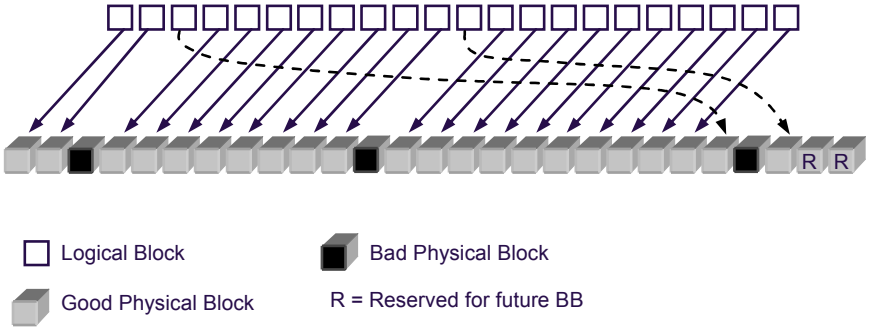


Fig. 2.36. Bad Block Management (BBM)

Toggle mode adds DQS data strobe signals; RE is used to trigger the read cycle as done in asynchronous interface; DQS is used to strobe the data on both edges.

7.1.5 High density systems: density, power and performance

Today, SSDs are offered in densities up to 64–128 GB and higher. To satisfy Enterprise requirements of endurance and quality, producers are forced to use SLC devices or MLC with trailing-edge technology. As of today, a 32 GB SSD can be built using eight packages (e.g. TSOP), each containing four 16 Gbit-sub 50 nm generation SLC NAND. The system can be built using four independent data channels. This means that two TSOP packages are connected to a single channel with a total of eight dies hanging on the same data bus which is loaded roughly by 50 pF for each data line.

Some considerations are fundamental when designing a system:

- Which is the minimum needed configuration in terms of channels number and which is the operating frequency to achieve the required throughput?
- How much power is dissipated in the system? Is it sustainable?

Due to the channel capacitance, each time a single NAND die is being written or read, the entire capacitance (that can be as high has 50 pF in the example above) of the data lines is driven.

I/O power consumption in a DDR system can be derived as [26]:

$$P = 9 \cdot \eta \cdot f \cdot C \cdot V^2 \tag{7.1}$$

where η is the bit activity ratio, f is the DDR frequency, C is the capacitance of a single line and V is the supply voltage of the interface.

For the system described $\eta = 0.5$, $C = 50$ pF and $V = 3.3$ V. We can derive the power consumption as a direct function of throughput (Fig. 7.8).

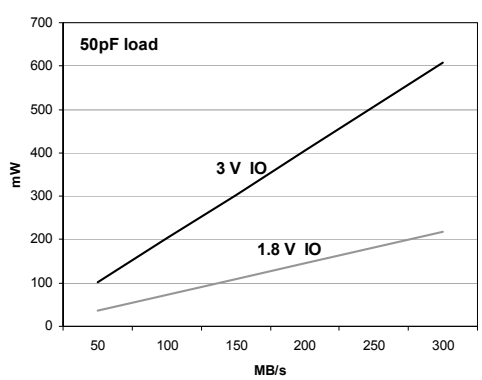


Fig. 7.8. I/O power as a function of channel throughput

Of course, the latch I1–I2 can also be used without exploiting the three-state option. In this case, M_{EN}, M_{SA} and the latch must be correctly sized so that the latch unbalancing always takes place when V_{SO} goes below $V_{DD} - V_{THP}$.

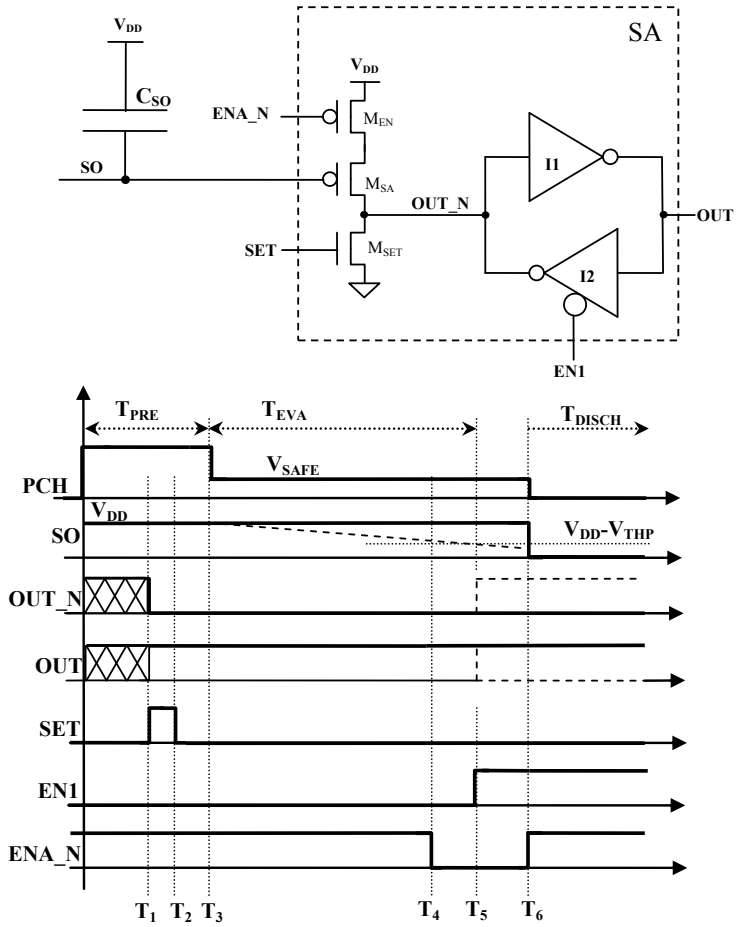


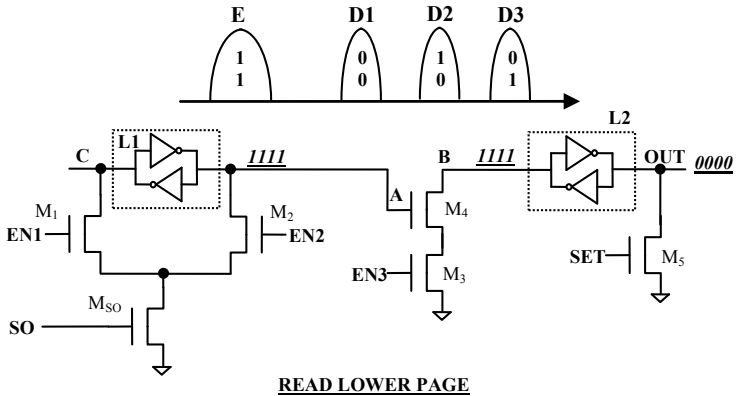
Fig. 8.28. ABL sense amplifier and its timing diagram

8.4 ABL versus interleaving architecture

In this section a comparison between ABL and the interleaving architecture is presented [25].

Such sensing circuit can also be used to implement the read operation for other types of coding, like the one shown in Fig. 10.15. As described in the table of Fig. 10.16 (ENX column indicates the enabling signal), the read of the upper-page is done by two SROs: the former using V_{READ1} and the latter using V_{READ2}. Read of the lower-page needs three SROs, at V_{READ1}, V_{READ2} and V_{READ3}. In this case as well, latch L1 is busy doing the read operation while latch L2 performs the data-out, which means that even this structure belongs to the cache read category.

The implementation of the read operation for the type of coding shown in Fig. 10.7 can be easily derived: since the coding of the upper-page and the coding of the lower-page are swapped, it is sufficient to swap the labels between the upper-page and the lower-page in the table of Fig. 10.16.



READ LOWER PAGE

	SO	ENX	A	C	B	OUT
Start	xxxx	x	1111	0000	1111	0000
V_{READ1}	0111	EN2	1000	0111	1111	0000
V_{READ3}	0001	EN1	1001	0110	1111	0000
L1 TO L2 Transfer	xxxx	EN3	1001	0110	0110	1001

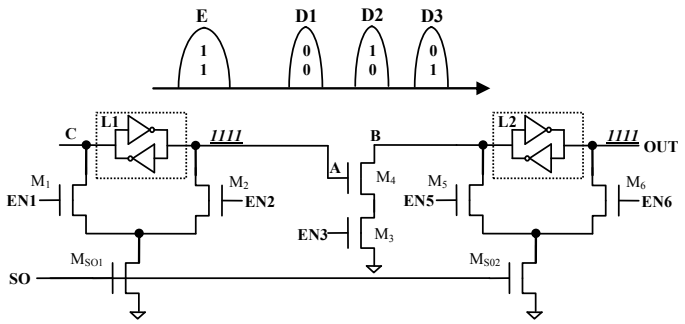
READ UPPER PAGE

	SO	ENX	A	C	B	OUT
Start	xxxx	x	1111	0000	1111	0000
V_{READ1}	0111	EN2	1000	0111	1111	0000
V_{READ2}	0011	EN1	1011	0100	1111	0000
V_{READ3}	0001	EN2	1010	0101	1111	0000
L1 TO L2 Transfer	xxxx	EN3	1010	0101	0101	1010

Fig. 10.16. MLC with different distribution coding and cache read

It should be noted that using the solution shown in Fig. 10.13 it is possible to read both pages using only three SROs, while the example shown in Fig. 10.16 requires five SROs.

It should always be possible to discriminate the position of the threshold voltage using only three SROs. For instance, the solution shown in Fig. 10.17 implements the simultaneous read of the two pages: the two latches, L1 and L2, work in parallel during SROs. At the end of three SROs, the lower-page is on node A, and the upper page is on node OUT.



READ LOWER AND UPPER PAGE

	SO	ENX	A	C	B	OUT
Start	xxxx	x	1111	0000	0000	1111
V_{READ1}	0111	EN2/EN6	1000	0111	0111	1000
V_{READ2}	0011	EN1	1011	0100	0111	1000
V_{READ3}	0001	EN2/EN5	1010	0101	0110	1001

LOWER PAGE UPPER PAGE

Fig. 10.17. MLC page buffer for reading two pages

10.2.3 MLC program/verify operations

During program operation, the same program pulse (at a gate voltage equal to V_{PGM}) is applied to all the cells belonging to the same wordline. It is responsibility of the sensing circuit to verify and, if necessary, inhibit those cells which have reached the desired distribution. In particular, the cell is inhibited and it does not change its threshold V_{THR} any longer if during the program step the corresponding bitline is biased to a voltage which is high enough to switch off selection transistor M_{BLS} (Fig. 10.6). The voltage applied to the bitline is typically equal to either V_{DD} or $V_{DD}-V_{THN}$. On the other hand, the cell increments its threshold voltage if during the program step the bitline is biased to a voltage which is low enough to

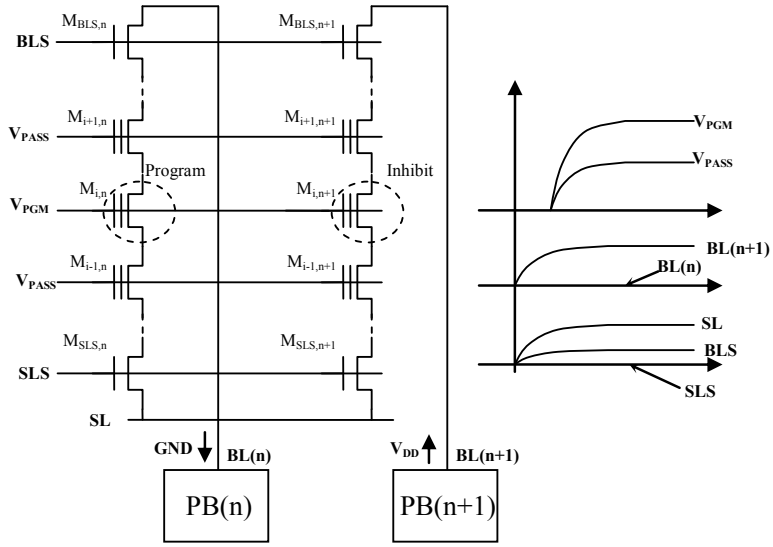
switch on M_{BSL} so that it can act as a pass transistor, Typically, the bitline is biased to ground and the situation just described is shown in Fig. 10.18.

In order to switch-off the transistor M_{BSL}, its gate is biased to the lowest possible voltage, considering that M_{BSL} has to be on when the bitline is forced to ground: typical values lie between 1 and 2 V.

Program of the upper-page is the most complex operation and its analysis allows us to understand the fundamental elements which must be present inside the sensing circuit. First of all, in all the coding cases previously seen, two information are needed in order to program the upper-page properly:

- Which is the initial distribution: the state of the lower-page must be known and stored in the sensing circuit. If the lower-page has already been programmed, the information must be recovered from the cell.
- Which is the target distribution: this information, given by the value of the bit of the upper-page, is provided by the user and it must be stored in the sensing circuit.

Since two pieces of information must be stored, it is evident that at least two latches are needed inside the sensing circuit. Furthermore, the circuit has to be interfaced to a logic-unit capable of loading the information that program needs (the data-load described in Sect. 10.3).



- PB(n) discharge BL(n) to ground: cell $M_{i,n}$ programmed

- PB(n+1) charge BL(n+1) to V_{DD}: cell $M_{i,n+1}$ inhibited

Fig. 10.18. Cells program and inhibit

The high voltage section of the device (comprising charge pumps and regulators) has to provide precise voltages necessary for reliable array operations during the whole device's life.

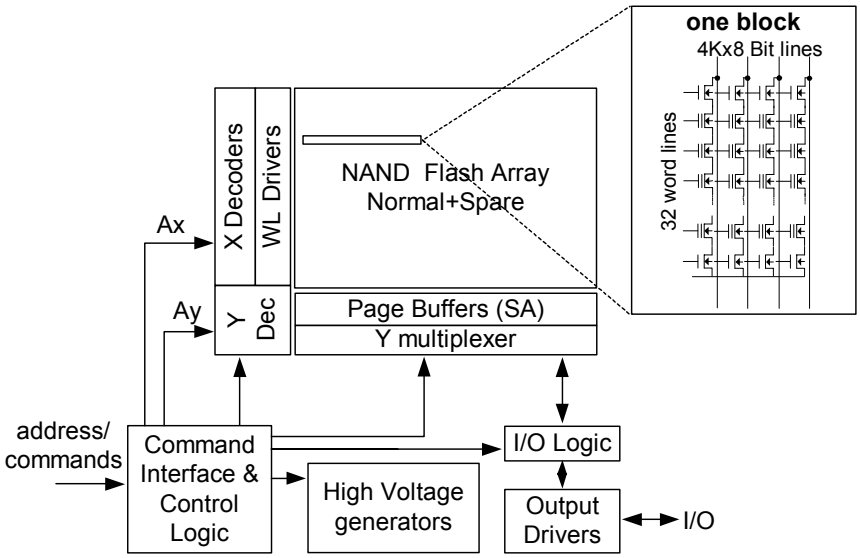


Fig. 15.1. NAND Flash memory block diagram

If we consider a write/erase cycling specification for one block of the array equal to 10 k cycles, we must be aware that the charge pumps, providing the voltages necessary for the cycling, operate in the worst (unlikely) case for 10 k cycles multiplied by the number of blocks in the array. Even if those reliability aspects are taken into account during the design phase, the manufacturer must guarantee that the production is able to maintain the reliability targets from lot to lot.

Moreover, a control logic section is used to manage all the circuits involved in memory operations. This section is frequently made up by a microcontroller plus a mask ROM and a RAM for executing code and storing data. Other implementations can be based on programmable logic arrays [3]. The digital circuitry in the chip has the same testability issues as in digital ASICs; testability can be secured by proper testing tools and accessibility such as digital scan chains in order to enable the highest quality levels and minimum testing time.

The aim of the following sections is to provide an introductory view of the test issues related to array architecture and peripheral circuits. For detailed descriptions of the circuits, it is recommended to refer to specific chapters in this book and bibliography. Specific test mode descriptions will be given in Sect. 15.4.

In fact, as is the case for a 4LC device, the highest verification level must be low enough to prevent bit failures such as program disturb and read disturb. The more states a memory cell must be able to store, the more finely divided its program threshold window is. Therefore, it is mandatory to reduce further the width of programmed distributions (Sect. 16.2).

A reduction in area occupation of the matrix has a drawback in an increment of the area occupied by the peripheral circuits: in particular, the area of the sensing circuits. In fact, the number of latches inside a sensing circuit is (as a minimum) equal to the number *n* of bits stored inside the cell (Sects. 16.3 and 16.4).

A number *n* of bits corresponds to a number $(2^n - 1)$ of levels of read and verify (see Fig. 16.2). Therefore, a device with *n* bits per cell needs $(2^n - 1)$ SROs to read the content of the cell. Moreover, the number of verify operations after a single program pulse is equal to $(2^n - 1)$.

Taking into account the last item, we could come to the conclusion that a read operation of a 16LC device is 15 times slower than a SLC device. Indeed, 15 SROs can read four pages, and therefore the *average page access time* T_R

$$T_R = \frac{(2^n - 1)}{n} \cdot T_{SRO} \tag{16.1}$$

is equal to 3.75 T_{SRO}, where T_{SRO} is the time spent by one SRO.

Therefore, T_R ratio between a 16LC and a SLC device is 3.75 and not 15. This ratio is as low as 2.5 if a 4LC device is considered ($T_R = 1.5 \cdot T_{SRO}$).

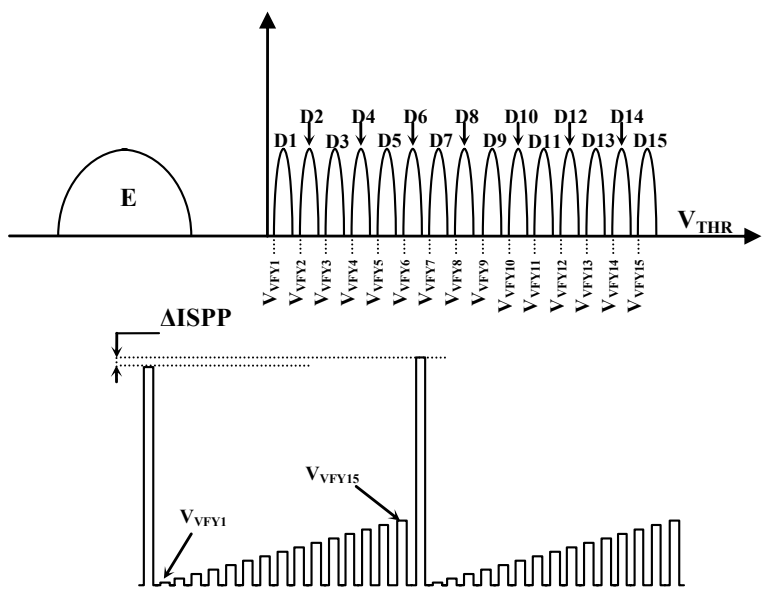


Fig. 16.2. 16LC ISPP and its 15 verify levels

The *Driver Stage Register* selects the power strength of the output buffer. This permits to adapt the card's output stage to the transfer rate and to the bus capacitance (which varies for instance according to the bus length and the number of connected cards).

17.4 Flash translation layer

Operating systems usually write data in sectors (e.g. 512 bytes in size). Unfortunately, Flash memories lack the so-called *update-in-place* capability. This means they cannot be altered on a sector (page in NAND nomenclature) basis, because the erase operation involves a whole block (made up of several sectors/pages). Therefore, file systems cannot write ones back to zeros in a single addressed sector. This is awkward for a high-level file system to manage.

In order to alleviate the workload of the operating system, software routines can be implemented to emulate a NAND Flash behavior where single sectors (or NAND pages) can be modified independently. Such software is called *Flash Translation Layer* (*FTL*), which is an intermediate layer between the file system and the storage media (the NAND Flash). The operating system can then write the NAND memory on a page basis without worrying about the details of its physical implementation. Usually it is the Controller in Fig. 17.1 taking care of the FTL.

The FTL achieves its task through a *logical to physical re-mapping*. Hence, when the file system sends the command to write or update a certain logical page, the FTL actually writes it in a different available physical page and updates a table storing the logical to physical mapping (*mapping table*). The page containing the older data is marked as invalid; Fig. 17.6 shows an example.

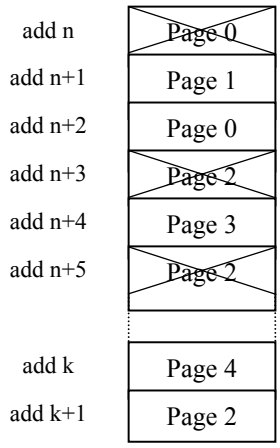


Fig. 17.6. Logical to physical re-mapping

Here page 0 has been written first at address n, followed by page 1. Then page 0 has been updated, but instead of overwriting it at address n, the FTL writes it in another location (n + 2) and marks address n as invalid.

As it is apparent from Fig. 17.6, after a while a great portion of the memory will be filled with invalid data taking precious unusable space. Eventually, all these locations will have to be erased to make this memory space available to new data. This procedure is not trivial, because the erase operation involves a whole block, comprising several pages, some with valid data, while others with invalid data. Therefore, valid data must be copied first to a new location, before the erase operation takes place. All this procedure is called *garbage collection*. An example is shown in Fig. 17.7.

All valid pages are copied into new locations (the order does not matter because the mapping table takes care of the correct correspondence between the logical and physical address). Then all the memory locations containing the old data (valid and invalid) are erased, thus creating space accessible to new data.

Garbage collection is very critical for many reasons. Let us start to better illustrate this point by introducing the notion of garbage collection efficiency.

Garbage collection efficiency is defined as the ratio of the number of invalid pages in a block to the total number of pages in that block. The higher the invalid pages within a block, the higher the efficiency. Indeed, when the number of invalid pages within a block is high we have a twofold benefit. First, only a few pages must be copied into new locations, thus limiting the program time and reducing the cycling efforts (greater lifetime) of the memory cells. Moreover, when erasing such block, a lot of space will be available for new data, because only few data were valid and still taking memory space.

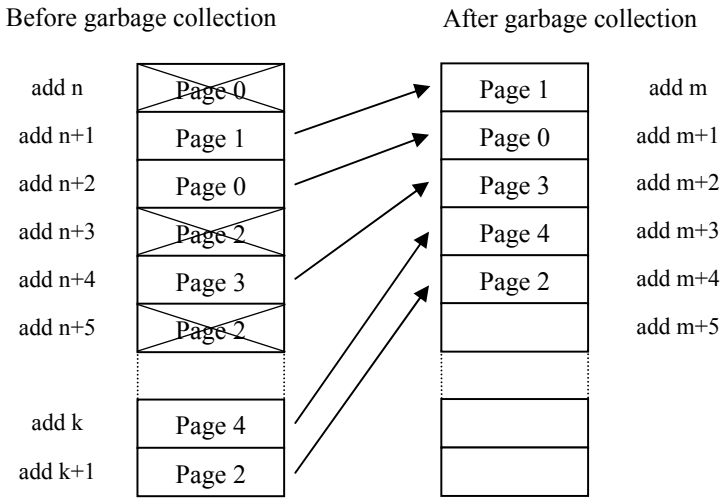


Fig. 17.7. Garbage collection

When should the garbage collection be run? A high program throughput calls for a lot of free space. If the host requires more space than readily available, it will have to halt its operation and wait for the garbage collection to kick in and erase enough invalid pages. This will limit the sustainable program throughput. So apparently, the garbage collection should be run as often as possible. However, this in turn will limit the garbage collection efficiency. Therefore, an accurate trade-off will have to be determined to establish the best performance of garbage collection.

Flash memories' blocks can only be programmed and erased a limited number of times. This is the reason why a good FTL should also implement a *wear leveling* algorithm. A wear leveling software module makes sure all NAND blocks are cycled an identical number of times, avoiding some blocks ending their cycle life while others being cycled only a few times. Were not a wear leveling algorithm present, blocks with frequently changed data would endure much more cycles than blocks containing long-lived data.

There are two levels of wear leveling. In the *first level wear leveling* new data is programmed into a free block with the fewest write/erase cycles. Two techniques to achieve this are the chain method and the array method.

In the *chain method,* the pool of available blocks is organized in a chain (Fig. 17.8).

A simple round robin algorithm selects one block after another. When a block is erased, it is added to the chain.

In the *array method* an age vector stores the number of each block's program/erase cycles. This information can be used by the wear leveling algorithm to ensure a uniform exploitation of all the blocks.

In the *second level wear leveling* from time to time blocks containing long-lived data are moved (copied) into other blocks even though they never received an update command. The original block can then be used to store new incoming data. The second level wear leveling is triggered when the difference between the maximum and the minimum number of cycles per block reaches a predetermined threshold.

Devising an efficient FTL is a taxing job. Two main concerns are the clustering and the cleaning policy. A few examples of both will be reported, but let start off by introducing the concepts of hot and cold data.

Hot data are the most frequently updated data. *Cold data* (or non-hot data) are seldom updated data. In a given block, it is advantageous to gather (cluster) all hot data or all cold data, because blocks with hot data will soon become garbage. Cleaning such blocks is beneficial because they have high garbage collection efficiency.

How to effectively cluster hot data then? One method is the *Dynamic dAta Clustering* (*DAC*) depicted in Fig. 17.9.

Fig. 17.8. Chain method

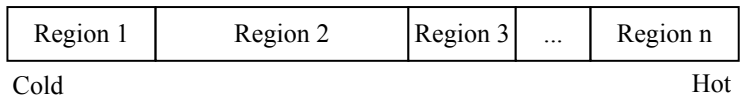


Fig. 17.9. Dynamic dAta Clustering

In the DAC method, the Flash memory is divided into *n* regions, whose size is not necessarily identical. Each region is not even necessarily made up of contiguous memory locations. Region 1 contains the coldest data, Region n the hottest. The hotter a block becomes, the more it shifts towards Region n. Conversely, the colder a block becomes, the more it shifts towards Region 1. When a block is written for the first time, it is put in Region 1. When a block is updated, if it has been into its Region k for a time shorter than a $t_{threshold}$ it is written in Region k + 1, otherwise it is written again in Region k. The $t_{threshold}$ is introduced because a block's hotness degrades as it ages. Upon the cleaning request of a block, if its valid data have been in their Region k longer than $t_{threshold-2}$ they are copied in Region k − 1, otherwise they are still copied in Region k. Each block has associated with it the number of the Region it belongs to.

The real advantage of this technique is twofold. Firstly, the algorithm does not require complex calculations and secondly the classification is fine grained (which in turn improves the performance).

Another clustering method is the *Dynamic Striping*, illustrated in Fig. 17.10. In the Dynamic Striping methodology, two registers called LRU (least recently used) are utilized. Only logical block addresses belonging to the hot list are hot, all the others are cold. The candidate list contains the logical block addresses recently written. If a logical block address belongs to the candidate list and is updated after a short while, it is promoted to the hot list. The last element of the hot list is demoted to the candidate list.

As a further improvement of the Dynamic Striping, hot data are written in the bank with the lowest number of erase cycles to enhance the wear leveling.

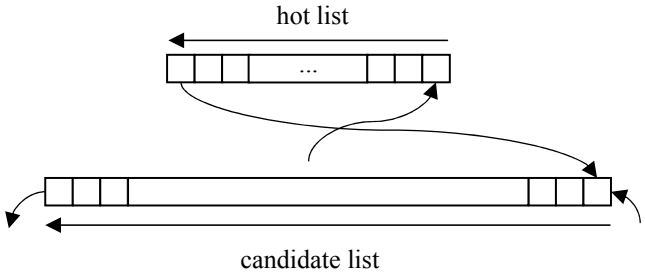


Fig. 17.10. Dynamic striping

Cold data are written in the bank with the lowest capacity utilization (i.e. the ratio of the number of live pages to the number of total pages) in order to make the capacity utilization more uniform (cold data will remain longer in their locations).

Cleaning policies relate to the criterion used to pick up the block to be erased for garbage collection purposes.

The *greedy policy* is one of the simplest cleaning policies, in that it selects for erasing the block with the greatest number of invalid data. Though simple, this policy is not very efficient because it does not take into account the cycling the blocks have been subjected to. It has also been proved ineffective in case of localities of reference (i.e. the fact that if a certain memory location is referenced at a given time, it is probable that the neighboring memory locations will be referenced in the near future).

A better cleaning policy is the *cost-benefit policy*. The block chosen for erase is the one with the greatest value of the following expression:

$$\frac{age \cdot (1 - valid)}{2 \cdot valid} \tag{17.1}$$

where *valid* is the percentage of valid data in the block, and *age* is the time elapsed since the last modification. Note that (1-*valid*) is the percentage of invalid data, i.e. the space that ultimately will be freed. The term 2·*valid* can be conceptually regarded as the cost of reading the valid data and to copy them in another location (2·*valid* = *valid* + *valid* and the first term is proportional to the time required to read the valid data while the second term is proportional to the time required to copy them in a new location). Therefore, the ratio (1-*valid*)/2·*valid* in the above expression can be considered as a benefit over cost ratio. This policy is better than greedy's because it takes into account the age of the block.

Yet another cleaning policy is the *Cost Age Time* (*CAT*) *policy*. The block chosen for erase is the one with the minimum value of the following expression:

$$\frac{valid}{1 - valid} \cdot \frac{1}{age} \cdot n_C \tag{17.2}$$

where *valid* and *age* are defined as in the previous case and n_C is the number of cleaning, i.e. the number of times a block has been erased. This cleaning policy is endowed even with a sort of wear leveling thanks to the term n_C.

A good FTL must also take care of *bad blocks*. Bad blocks contain not-working or unreliable bits and therefore cannot be used. They may be present since the manufacturing of the device or may develop during its lifetime. The FTL must store all the necessary information about which blocks can be used and which cannot. When a program operation in a certain block fails, the FTL acknowledges that through the NAND Status Register and acts copying the valid data of the block into a new good block; the block where the fail occurred is then be marked as bad.

Finally, a FTL must ensure the reliability of data in the event of sudden power loss. In order to guarantee this, the FTL must wait for the program operation to

complete successfully before marking the old page as invalid or before erasing a block. In such a way, if the new data is lost due to a power failure, the old data can still be recovered.

17.5 Cryptography

Security has become an important part of cards' characteristics. New protocols implement commands handling confidential data with the utmost caution. Let us mention the standard JESD84-A44 (eMMC), which states that every card complying with it must implement a special memory area called *Replay Protected Memory Block* (*RPMB*), which must be protected against replay attacks. In a *replay attack* a valid data transmission is intercepted first, recorded and then played back again later on by an attacker. Hence, cryptography has become fundamental even in card design and usage. To protect a portion of the memory against replay attacks the eMMC protocol employs a key (MAC) and a nonce. To better understand these concepts, let us briefly review a few pivotal definitions and notions about cryptography.

A *cipher* is an algorithm to *encrypt* and *decrypt* a *plaintext* into and from a *ciphertext* (Fig. 17.11).

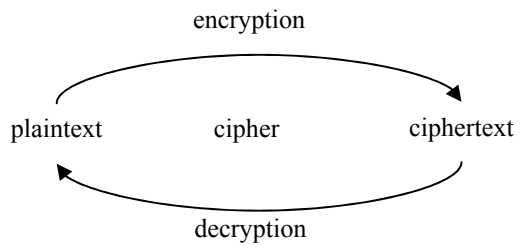


Fig. 17.11. Cipher

Note that in the everyday vernacular language sometimes the terms cipher and code are used interchangeably. However, in cryptography a *code* has a *codebook* associating each word or phrase of the plaintext with one or more codewords. Therefore, the plaintext set is somewhat limited and must be anticipated a priori. Codes operate according to the meaning while ciphers operate according to a single letter or bit.

Block ciphers work on blocks whereas *stream ciphers* work on continuous streams of data. Each cipher needs a *key* or *cryptovariable*. *Symmetric key algorithms* have the same key for encryption and decryption. *Asymmetric key algorithms* have different keys for encryption and decryption. In symmetric key algorithms, the key must be known only by the sender and by the recipient.

A *cryptographic hash function* is a function mapping a *message* into a message *digest* (a.k.a. *hash value*) as depicted in Fig. 17.12.

The memory, in which user data can be stored and the added memory necessary to implement the code, can be managed through the microcontroller, as shown in Fig. 17.21.

The microcontroller, apart from managing the two types of memories (they can be of different types and therefore having dedicated interfaces), is also requested to manage coding and decoding of the selected code.

The memory which stores the added information of the code, "Auxiliary Memory" in Fig. 17.21, can be used with different methods. A first approach is to evaluate the error correction capability needed for the total size of the memory we need to protect, the minimum codeword length (chunk), type and capability of the error correction code. Once the choices are made, it is possible to find the minimal additional memory compatible with the requests.

Using this method it is possible to link a chunk of the memory to be protected to a portion of the additional memory; this solution allows a fixed error correction capability for every chunk.

A second approach is based on a "dynamic" error correction capability (i.e. the error correction capability for a single chunk is increased when an error occurs); in this way it is possible to read correctly even in the case of further errors.

Specification to be fixed for implementation are, beyond of the size of the additional memory, the minimum and the maximum error correction capabilities (Nmin and Nmax).

At the beginning, every chunk of the primary memory is linked to a portion of the "Auxiliary Memory" needed to correct a minimum number of errors. Multiplying the total number of chunks by the parity bits needed to correct Nmin errors, we can calculate the required portion of the memory.

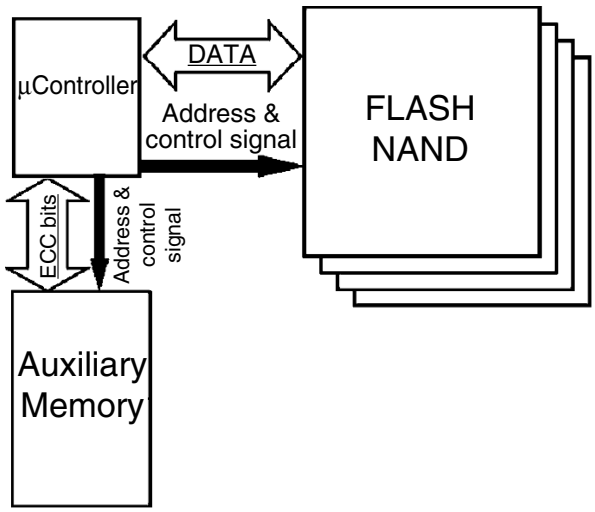


Fig. 17.21. Auxiliary memory scheme used to store code bits

Table 17.2. NAND Flash: SLC versus MLC

	SLC	MLC
Sequential read access time	25 ns	25 ns
Random read access time	25 µs	60 µs
Program time	200 µs	800 µs
Erase time	2 ms	2.5 ms
Power supply – Vdd	1.8/3.0 V	3.0 V
Program/Erase cycles	100 k	10 k
Data retention	10 years	10 years

Many market applications require both SLC features to reliably store system code and data and MLC features, to save high density data while requiring lower reliability

A new solution, we'll call it flexible, is realized thanks to the microcontroller and consists on the partitioning of the single NAND cell in two sub sections, the first of them to be used as SLC, the other one as MLC (see Fig. 17.23).

The partitioning allows the microcontroller to use different algorithms and settings in memory usage depending on the features of the selected portion. The partition is dynamic; the space assigned to the code can be modified by the microcontroller. In the SLC partition program (PV) and read (RV) references are more relaxed and the pulses can have a wider size due to a single distribution in the positive side of the threshold voltage values. This minor precision in Vth values allows speeding up program and read operations, getting closer to native SLC memories. Additionally, in SLC partition, there is not the need of a high ECC capability, affecting as well performances and data area to store parity bits.

The same approach can be applied to 3 or more bits per cell devices, where a portion is reserved for a SLC-mode usage to get better performances in terms of throughput and reliability.

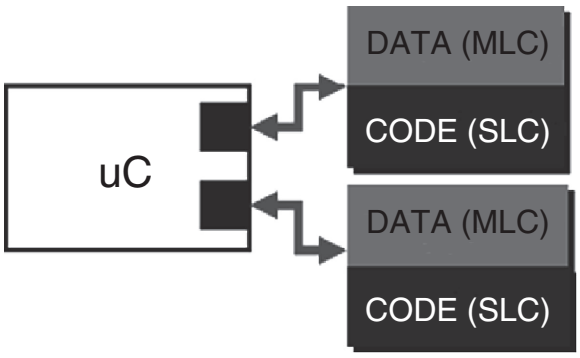


Fig. 17.23. Flexible solution scheme