2 NAND overview: from memory to systems

R. Micheloni¹, A. Marelli² and S. Commodaro³

2.1 Introduction

It was in 1965, just after the invention of the bipolar transistor by W. Shockley, W. Brattain and J. Bardeen, that Gordon Moore, co-founder of Intel, observed that the number of transistors per square centimeter in a microchip doubled every year. Moore thought that such trend would have proven true for the years to come as well, and indeed in the following years the density of active components in an integrated circuit kept on doubling every 18 months. For example, in the 18 months that elapsed between the Pentium processor 1.3 and the Pentium-4, the number of transistors grew from 28 to 55 million.

Today, a standard desktop PC has processors whose operating frequency is in the order of some gigahertz, while its memory can store as much information as terabytes.

In this scenario, a meaningful portion of the devices produced is represented by memories, one of the key components of any electronic systems.

Semiconductor memories can be divided into two major categories: RAM, acronym for Random Access Memories, and ROM, acronym for Read Only Memories: RAM loses its content when power supply is switched off, while ROM virtually holds it forever. A third category lies in between, i.e. NVM, acronym for Non-Volatile Memories, whose content can be electrically altered but it is also preserved when power supply is switched off. These are more flexible than the original ROM, whose content is defined during manufacturing and cannot be changed by the consumer anymore.

The history of non-volatile memories began in the 1970s, with the introduction of the first EPROM memory (Erasable Programmable Read Only Memory). Since then, non-volatile memories have always been considered one of the most important families of semiconductors and, up to the 1990s, their interest was tied up more to their role as a product of development for new technologies than to their economic value. Since the early 1990s, with the introduction of non-volatile Flash memories into portable products like mobile phones, palmtop, camcorders, digital cameras and so on, the market of these memories has experienced a stunning increase.

¹ Integrated Device Technology, rino.micheloni@ieee.org
² Integrated Device Technology, alessiamarelli@gmail.com
³ Pegasus MicroDesign, stefano_commodaro@yahoo.it
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](image)

### 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).
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
All the NAND strings sharing the same group of wordlines are erased together, thus forming a Flash block. In Fig. 2.2 two blocks are shown: using a bus representation, one block is made up by WL<63:0> while the other one includes WL<63:0>.

NAND Flash device is mainly composed by the memory array. Anyway, in order to perform read, program, and erase additional circuits are needed. Since the NAND die must be inserted in a package with a well-defined size, it is important to organize all the circuits and the array in the early design phase, i.e. it is important to define a floorplan.

In Fig. 2.3 an example of a floorplan is given. The Memory Array can be split in different planes (two planes in Fig. 2.3). On the horizontal direction a Wordline is highlighted, while a Bitline is shown in the vertical direction.

The Row Decoder is located between the planes: this circuit has the task of properly biasing all the wordlines belonging to the selected NAND string (Sect. 2.2.2). All the bitlines are connected to sense amplifiers (Sense Amp). There could be one or more bitlines per sense amplifier; for details, please, refer to Chap. 8. The purpose of sense amplifiers is to convert the current sunk by the memory cell to a digital value. In the peripheral area there are charge pumps and voltage regulators (Chap. 11), logic circuits (Chap. 6), and redundancy structures (Chap. 13). PADs are used to communicate with the external world.

**Fig. 2.3.** NAND Flash memory floorplan

### 2.2.2 Basic operations

This section briefly describes the basic NAND functionalities: read, program, and erase.
Read

When we read a cell (Fig. 2.4), its gate is driven at $V_{\text{READ}}$ (0 V), while the other cells are biased at $V_{\text{PASS,R}}$ (usually 4–5 V), so that they can act as pass-transistors, regardless the value of their threshold voltages. In fact, an erased Flash cell has a $V_{\text{TH}}$ smaller than 0 V; vice versa, a written cell has a positive $V_{\text{TH}}$ but, however, smaller than 4 V. In practice, biasing the gate of the selected cell with a voltage equal to 0 V, the series of all the cells will conduct current only if the addressed cell is erased.

String current is usually in the range of 100–200 nA. The read technique is based on charge integration, exploiting the bitline parasitic capacitor. This capacitor is precharged at a fixed value (usually 1–1.2 V): only if the cell is erased and sinks current, then the capacitor is discharged. Several circuits exist to detect the bitline parasitic capacitor state: the structure depicted in the inset of Fig. 2.4 is present in almost all solutions. The bitline parasitic capacitor is indicated with $C_{BL}$ while the NAND string is equivalent to a current generator.

During the charge of the bitline, the gate of the PMOS transistor $M_P$ is kept grounded, while the gate of the NMOS transistor $M_N$ is kept at a fixed value $V_1$. Typical value for $V_1$ is around 2 V. At the end of the charge transient the bitline will have a voltage $V_{BL}$:

$$V_{BL} = V_1 - V_{THN} \quad (2.1)$$
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:

$$V_{BL} < V_2 - V_{THN}$$

$M_N$ 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).

![Diagram of NAND memory with parasitic capacitors](image)

**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.
NAND devices communicate with the external user by means of pins (Fig. 2.10). These pins are monitored by the Command Interface (Chap. 6) which has the task to understand the functionality required to the memory in that moment.

Pins shown in the Fig. 2.10 are listed below.

- **CE#**: it is the Chip Enable signal. This input signal is “1” when the device is in stand-by mode, otherwise it is always “0”.
- **R/B#**: it is the Ready/Busy signal. This output signal is used to indicate the target status. When low, the target has an operation in progress.
- **RE#**: it is the Read Enable signal. This input signal is used to enable serial data output.
- **CLE**: it is the Command Latch Enable. This input is used by the host to indicate that the bus cycle is used to input the command.
- **ALE**: it is the Address Latch Enable. This input is used by the host to indicate that the bus cycle is used to input the addresses.
- **WE#**: it is the Write Enable. This input signal controls the latching of input data. Data, command and address are latched on the rising edge of WE#.
- **WP#**: it is the Write Protect. This input signal is used to disable Flash array program and erase operations.
- **DQ<7:0>**: these input/output signals represent the data bus.
2.3 Command set

2.3.1 Read operation

The read function reads the data stored at a specified address. In order to accomplish this goal, NAND device must recognize when a read operation is issued and the related addresses. After a busy time needed to execute the read algorithm, NAND device outputs the data sequence. Based on the device pin signals, the NAND Command Interface (CI, Chap. 6) is able to understand when a command is issued, when an address is issued or when it must perform data out.

Figure 2.11 shows a command cycle (“Cmd”). CI recognizes a “Cmd” cycle if CLE is high. In that case, the 8-bit value on DQs represents the command code.

Figure 2.12 shows address cycles. Generally, all the operations need the addresses where they have to act. The address length depends on the operation and on the capacity of the NAND; anyway, N cycles are needed to input column addresses and M cycles are needed to input row addresses. CI recognized an address cycle if ALE is high. In the meanwhile, all other input signals are low and the DQs value is the address.

The last command phase used by the read operation is the data out, shown in Fig. 2.13. Data out is performed by toggling signal RE#: at every cycle a new data is available on DQs.

---

Command Phase

**Cmd**

CLE

DQ<7:0> **Code**

**Fig. 2.11.** Command cycle (“Cmd”): CLE is high, all other input signals are low

Command Phase

Add ... Add Add ... Add

**Fig. 2.12.** Address cycle: ALE is high, all other inputs signals are low
These basic cycles are used by the NAND to decode and perform every operation. Figure 2.14 shows the command sequence for a read operation. The first cycle is used to issue the read command “RD” (e.g. 00h). After the command cycle a number of cycles is used to provide the addresses. As described in Sect. 2.2 the column addresses are given first, followed by the row addresses. All the pins (ALE, CLE, RE#) not present in the figure must be driven as described above. Code “RDC” (Read Command Confirm, e.g. 30h) is used to confirm the read command. Finally, the device goes busy and the read operation starts. When NAND returns ready, the data output cycles start.

The above described read command outputs the entire Flash page, regardless the number of bytes we want to read. In some cases, a small move of data is required or we may want to read randomly inside a page. Command Change Read Column, also known as Random Data Output, is able to change the column address we are reading.

Figure 2.15 shows the Change Read Column sequence. After the usual read command is executed, it is possible to change the column address during data out. A command cycle “CRC” (Change Read Column, e.g. 05h) is issued, followed by the addresses of the locations we want to output from. Only fewer cycles are required with respect to the usual read command, since only the column addresses are needed. A confirm command cycle “CRCC” (Change Read Column Confirm, e.g. E0h) is used to enable the data out. It is worth noting that no additional busy time is necessary, because data are already stored in the page buffers.

Generally, the busy time in a read operation lasts for about 25–30 μs. One way to improve read throughput is the Read Cache Command (when available). With this command, it is possible to download data from the Flash memory, while page buffers are reading another page from the Flash array.
Sequential Read Cache Command sequence is shown in Fig. 2.16. A Read Command must be issued before Read Cache Command. After the device returns ready, the command code “RC” (Read Cache, e.g. 31h) is used to initiate data download from the matrix to page buffers. RB# goes low for a while and then N Dout cycles are used to output first page. Since no other addresses are input, the next sequential page is automatically read inside the NAND. When we don’t need to read other pages, the last page is copied into the page buffers by using command “RCE” (Read Cache End, e.g. 3Fh).

Random Cache Read sequence is shown in Fig. 2.17: with this command it is possible to select the address of the page we want to cache.
On multi-plane device, it is possible to issue a read command on multiple planes simultaneously. Figure 2.18 shows the command sequence for Multi-plane Read.

After the standard Read Command cycle “RD”, the addresses of the page we want to read on plane 0 are issued. The command code “MR” (Multi-plane read, e.g. 32h) is used in the next command cycle so that the device is ready to receive the addresses belonging to plane 1. Once the new addresses and the Read Command Confirm Code “RDC” are given, the device goes busy to perform the read algorithm on both planes simultaneously. When the device returns ready, the command cycle CC (Choose Column, e.g. 06h) is used to select the address of the page we want to output, followed by a number of address cycles. The command code “CCC” (Choose Column Confirm, e.g. E0h) is a command confirm. Finally the Dout cycles are used to output the read data.

Since both the Read Cache command and the Multi-plane read have been introduced to increase performances, it is interesting to compare them. Figure 2.19 shows a comparison among Read, Cache Read and Multi-plane Read.

![Diagram of Multi-plane read command](image)

**Fig. 2.18.** Multi-plane read command

![Performance comparison diagram](image)

**Fig. 2.19.** Performance comparison among Read, Double-plane Read and Cache Read
Given $T_{ALGO}$ the time needed to read from the NAND array, and $T_{OUT}$ the time needed to download the page, the total time to perform the read of two pages with a standard read is $T = 2T_{ALGO} + 2T_{OUT}$. If a multi-plane command is used, $T_{ALGO}$ runs simultaneously on both planes and $T = T_{ALGO} + 2T_{OUT}$. Evaluating $T$ in the Cache Read case is more difficult, because we have to consider the ratio between $T_{ALGO}$ and $T_{OUT}$. If $T_{OUT}$ is longer than $T_{ALGO}$, then $T = T_{ALGO} + 2T_{OUT}$. It follows that the performances of Cache Read and Double Plane Read are the same. On the contrary, if $T_{ALGO}$ is longer than $T_{OUT}$ (Chap. 16), it won’t be possible anymore to mask $T_{ALGO}$ with a single page data out (Fig. 2.20). In this case, Double-plane Read performs better than Cache Read.

![Fig. 2.20. Performance comparison among Read, Double-plane Read and Cache Read with a NAND array read time longer than page data output](image)

### 2.3.2 Program operation

Purpose of program operation is to write a data sequence at a specified address. The basic cycles are those already described for read operation, such as Command cycle and Address cycle. The only added cycle is the “Din” cycle, represented in Fig. 2.21. Data in is performed by toggling signal WE#: at every cycle a new byte shall be made available on DQs.

![Fig. 2.21. “Din” cycle: WE# is low, all other inputs signals are low](image)
Program sequence is shown in Fig. 2.22. A command cycle to input “PM” code (Program, e.g. 80h) is followed by a number of address cycles to input the addresses where we want to write. Once the location is set, N “Din” cycles are used to input data into the page buffers. Finally a “PMC” (Program Confirm, e.g. 10h) command is issued to start the algorithm.

As already described for read operation, also in the program case there could be the need to move a small amount of data. Change Write Column is used to change the column address where we want to load the data.

Program busy time can be very long: 150–200 μs. Program cache command or double-plane program are used to increase write throughput. Figure 2.23 shows the sequence for Cache Program and Double Plane Program.

The first cycles (“PM” cycle, address cycles and “Din” cycles) are the same as in the standard program. Instead of “PMC” a “C/M” command cycle is issued. “C/M” can be the Cache Program Code (e.g. 15h) or a Double Plane Command (e.g. 11h). Once another “PM” command is given, followed by the new addresses and the “PMC” command, the device goes busy and the program algorithm is performed simultaneously on both pages. It is worth noting that the above described Double plane program is generally known as Concurrent double-plane Program, because the program algorithm works simultaneously on both planes.
Overlapped Double-Plane Program is also available, where the program on the first plane starts as soon as data are loaded in the page buffers. In this case the NAND architecture must be able to perform the algorithm in an independent way on both planes.

The comparison between the above mentioned program commands is shown in Fig. 2.24.

2.3.3 Erase operation

The Erase operation is used to delete data from the Flash array. As already described (Sect. 2.2.3), the smallest erasable unit for the NAND memory is the block. Figure 2.25 shows the Erase Command sequence.

The Erase command is very simple: a “ER” code is issued (Erase Command, e.g. 60h), followed by the block address and the “ERC” code (Erase Command Confirm, e.g. D0h). After that, the device goes busy to perform the algorithm.

![Fig. 2.25. Erase command](image)
Since erase is the longest operation (about 1 ms), the Double-Plane Erase command has been introduced to erase two blocks at the same time. Figure 2.26 shows the command sequence for the Double-Plane Erase.

The standard erase cycles (“ER” command and row address cycles) are followed by a “MER” command (Multi-plane erase, e.g. D1h). Once both the plane 1 addresses and the “ERC” code are given, the device goes busy, erasing both blocks simultaneously.

![Fig. 2.26. Double-Plane Erase command](image)

### 2.3.4 Synchronous operations

NAND read throughput is determined by array access time and data transfer across the DQ bus. The data transfer is limited to 40 MB/s by the asynchronous interface. As technology shrinks, page size increases and data transfer takes longer; as a consequence, NAND read throughput decreases, totally unbalancing the ratio between array access time and data transfer on the DQ bus. DDR interface (Chap. 7) has been introduced to balance this ratio.

Nowadays two possible solutions are available on the market. The first one, **Source Synchronous Interface** (SSI), is driven by the ONFI (Open NAND Flash Interface) organization established in 2006 with the purpose of standardizing the NAND interface. Other NAND vendors use the **Toggle-Mode interface**.

Figure 2.27 shows the NAND pinout for SSI. Compared to the **Asynchronous Interface** (ASI, Sect. 2.2), there are three main differences:

- RE# becomes W/R# which is the Write/Read direction pin.
- WE# becomes CLK which is the clock signal.
- DQS is an additional pin acting as the data strobe, i.e. it indicates the data valid window.

![Fig. 2.27. Pinout of a NAND Flash supporting Source Synchronous Interface](image)
Hence, the clock is used to indicate where command and addresses should be latched, while a data strobe signal is used to indicate where data should be latched. DQS is a bi-directional bus and is driven with the same frequency as the clock.

Obviously, the basic command cycles described in the previous sections must be modified according to the new interface.

Figure 2.28 shows a “Cmd” sequence, followed by “Dout” cycles for SSI. Toggle-Mode DDR interface uses the pinout shown in Fig. 2.29.

![Fig. 2.29. Pinout of a NAND Flash supporting Toggle-Mode Interface](image)

---

Fig. 2.28. Source Synchronous Interface DDR sequence

Fig. 2.30. Toggle-Mode DDR sequence
It can be noted that only the DQS pin has been added to the standard ASI. In this case, higher speeds are achieved increasing the toggling frequency of \( \text{RE\#} \).

Figure 2.30 shows a “Cmd” sequence, followed by “Dout” cycles for Toggle-Mode interface.

### 2.4 NAND-based systems

Flash cards, USB sticks and Solid State Disks (SSDs) are definitely the most known examples of electronic systems based on NAND Flash.

Several types of memory cards are available on the market [3–5], with different user interfaces and form factors, depending on the needs of the target application: e.g. mobile phones need very small-sized removable media like \( \mu\text{SD} \). On the other hand, digital cameras can accept larger sizes as memory capacity is more important (CF, SD, MMC). Figure 2.31 shows different types of Flash cards.

The interfaces of the Flash cards (including USB sticks) support several protocols: parallel or serial, synchronous or asynchronous. Moreover, the Flash cards support hot insertion and hot extraction procedures, which require the ability to manage sudden loss of power supply while guaranteeing the validity of stored data.

For the larger form factors, the card is a complete, small system where every component is soldered on a PCB and is independently packaged. For example, the NAND Flash memories are usually available in TSOP packages. It is also possible to include some additional components: for instance, an external DC-DC converter can be added in order to derive an internal power supply (CompactFlash cards can work at either 5 or 3.3 V), or a quartz can be used for a better clock precision. Usually, reasonable filter capacitors are inserted for stabilizing the power supply. Same considerations apply to SSDs; the main difference with Flash cards is the system capacity as shown in Fig. 2.32 where multiple NANDs are organized in different channels in order to improve performances.

For small form factors like \( \mu\text{SD} \), the size of the card is comparable to that of the NAND die. Therefore, the memory chip is mounted as bare die on a small substrate. Moreover, the die thickness has to be reduced in order to comply with the thickness of \( \mu\text{SD} \), considering that several dies are stacked, i.e. mounted one on top of each other.
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.
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.

![Garbage collection diagram](image)

**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.

![Bad Block Management (BBM) diagram](image)
This task is typically executed by a specific hardware inside the memory controller. Examples of memories with embedded ECC are also reported [9–11]. Most popular ECC codes, correcting more than one error, are Reed–Solomon and BCH [12]. While the encoding takes few controller cycles of latency, the decoding phase can take a large number of cycles and visibly reduce read performance as well as the memory response time at random access.

There are different reasons why the read operation may fail (with a certain probability):

- Noise (e.g. at the power rails)
- \( V_{TH} \) disturbances (read/write of neighbor cells)
- Retention (leakage problems)

The allowed probability of failed reads after correction is dependent on the use case of the application. Price sensitive consumer application, with a relative low number of read accesses during the product life time, can tolerate a higher probability of read failures as compared to high-end applications with a high number of memory accesses. The most demanding applications are cache modules for processors.

The reliability that a memory can offer is its intrinsic error probability. This probability could not be the one that the user wishes. Through ECC it is possible to fill the discrepancy between the desired error probability and the error probability offered by the memory (Chap.14).

The object of the theory of error correction codes is the addition of redundant terms to the message, such that, on reading, it is possible to detect the errors and to recover the message that has most probably been written.

Methods of error correction are applied for purpose of data restoration at read access. Block code error correction is applied on sub-sectors of data. Depending on the used error correcting schemes, different amount of redundant bits called parity bits are needed.

Between the length \( n \) of the code words, the number \( k \) of information bits and the number \( t \) of correctable errors, a relationship known as Hamming inequality exists, from which it is possible to compute the minimum number of parity bits:

\[
\sum_{i=0}^{t} \binom{n}{i} \leq 2^{n-k} \tag{2.3}
\]

It is not always possible to reach this minimum number: the number of parity bits for a good code must be as near as possible to this number. On the other hand, the bigger the size of the sub-sector is, the lower the relative amount of spare area (for parity bits) is. Hence, there is an impact in Flash die size.

BCH and Reed–Solomon codes have a very similar structure, but BCH codes require less parity bits and this is one of the reasons why they were preferred for an ECC embedded in the NAND memory [11].
2.4.2 Multi-die systems

A typical memory system is composed by several NAND memories. Typically, an 8-bit bus, usually called channel, is used to connect different memories to the controller (Fig. 2.32). It is important to underline that multiple Flash memories in a system are both a means for increasing storage density and read/write performance [13].

Operations on a channel can be interleaved which means that another memory access can be launched on an idle memory while the first one is still busy (e.g. writing or erasing). For instance, a sequence of multiple write accesses can be directed to a channel, addressing different NANDs, as shown in Fig. 2.37: in this way, the channel utilization is maximized by pipelining data load, while the program operation takes place without requiring channel occupation. A system typically has two to eight channels operating in parallel (or even more).

As shown in Fig. 2.38, using multiple memory components is an efficient way to improve data throughput while having the same page programming time.

The memory controller is responsible for scheduling the distributed accesses at the memory channels. The controller uses dedicated engines for the low level communication protocol with the Flash.

Moreover it is clear that the data load phase is not negligible compared to the program operation (the same comment is valid for data output); therefore increasing I/O interface speed is another smart way to improve general performance: high-speed interfaces, like DDR, have already been reported [14] and they are discussed in more details in Chap. 7. Figure 2.39 shows the impact of DDR frequency on program throughput. As the speed increases, more NAND can be operated in parallel before saturating the channel. For instance, assuming a target of 30 MB/s, 2 NANDs are needed with a minimum DDR frequency of about 50 MHz. Given a page program time of 200 μs, at 50 MHz four NANDs can operate in interleaved mode, doubling the write throughput. Of course, power consumption has then to be considered.

There are also hybrid architectures which combine different types of memory. Most common is usage of DRAM as memory cache. During write access the cache is used for storing data before transfer to the Flash. The benefit is that data updating, e.g. in tables, is faster and does not wear out the Flash.

![Interleaved operations on one Flash channel](image-url)

Fig. 2.37. Interleaved operations on one Flash channel
Another architecture uses a companion NOR Flash for purpose of “in-place execution” of software without pre-fetch latency. For hybrid solutions, a multi-die approach, where different memories are packaged in the same chip, is a possibility to reduce both area and power consumption.

### 2.4.3 Die stacking

Reduced form factor has been one of the main drivers for the success of the memory cards; on the other hand, capacity requirement has grown dramatically to the extent that standard packaging (and design) techniques are no longer able to
sustain the pace. In order to solve this issue, two approaches are possible: advanced die stacking and 3D technologies.

The standard way to increase capacity is to implement a multi-chip solution, where several dies are stacked together. The advantage of this approach is that it can be applied to existing bare die, as shown in Fig. 2.40: die are separated by means of a so-called interposer, so that there is enough space for the bonding wires to be connected to the pads.

On the other hand, the use of the interposer has the immediate drawback of increasing the height of the multi-chip, and height is one of the most relevant limiting factors for memory cards.

One way to overcome this issue is to exploit both sides of the PCB, as shown in Fig. 2.41: in this way, the PCB acts as interposer, and components are evenly flipped on the two sides of the PCB. Height is reduced, but there is an additional constraint on design: in fact, since the lower die is flipped, its pads are no longer matching those of the upper die. The only way to have corresponding pads facing one another is to design the pad section in such a way that pad to signal correspondence can be scrambled: that is, when a die is used as the bottom one, it is configured with mirrored-pads. Such a solution is achievable, but chip design is more complex (signals must be multiplexed in order to perform the scramble) and chip area is increased, since it might be necessary to have additional pads to ensure symmetry when flipping.

**Fig. 2.40.** Standard die stacking

**Fig. 2.41.** Flipped die stacking
The real breakthrough is achieved by completely removing the interposer, thus using all the available height for silicon (apart from a minimum overhead due to the die-to-die glue). Figure 2.42 shows an implementation, where a staircase arrangement of the dies is used: any overhead is reduced to the minimum, bonding does not pose any particular issue and chip mechanical reliability is maintained (the disoverlap between dies is small compared to the die length, and therefore the overall stability is not compromised, since the upmost die does not go beyond the overall center of mass).

The drawback is that such a solution has a heavy impact on chip design, since all the pads must be located on the same side of the die. In a traditional memory component, pads are arranged along two sides of the device: circuitry is then evenly located next to the two pad rows and the array occupies the majority of the central area. Figure 2.43 shows the floorplan of a memory device whose pads lie on two opposite sides.
If all pads lie on one side, as shown in Fig. 2.44, chip floorplan is heavily impacted [15]: most of the circuits are moved next to the pads in order to minimize the length of the connections and to optimize circuit placement. But some of the circuits still reside on the opposite side of the die (for instance, part of the decoding logic of the array and part of the page buffers, i.e. the latches where data are stored, either to be written to the memory or when they are read from the memory to be provided to the external world).

Of course, such circuits must be connected to the rest of the chip, both from a functional and from a power supply point of view. Since all pads are on the opposite side, including power supply ones, it is necessary to re-design the power rail distribution inside the chip, making sure that the size and geometry of the rails is designed properly, in order to avoid IR drops issues (i.e. the voltage at the end of the rail is reduced due to the resistive nature of the metal line).

One of the main disadvantages of staircase stacking is the increased size in the direction opposite to the pad row. Of course, this fact limits the number of dies, given a specific package.
Once the design effort has been done, a possible alternative is the “snake” stacking as shown in Fig. 2.45. In this case, thanks to the double side bonding, the overall size can be clearly reduced.

Recently, another stacking option came into the game: Through Silicon Via (TSV) [16–19]. With this technology, dies are directly connected without asking for a wire bonding, as depicted in Fig. 2.46. A 3-D view of one TSV connection is shown in Fig. 2.47. One of the main advantages is the reduction of the interconnection length and the associated parasitic $RC$. As a result, data transfer rate can be definitely improved, as well as power consumption.
In fact, DRAM are the main drivers for TSV technology as, with the standard bonding technology, they cannot stack more than two dies in the same package.

The solutions presented so far exploit advances in stacking techniques, eventually requiring changes in chip design floorplan. Quite recently, advanced design and manufacturing solutions have been presented, where the 3D integration [20] is performed directly at chip level.

### 2.4.4 3D memories and XLC storage

The 3D concept is simple: instead of stacking several dies, each of which being a fully-functional memory component, it is possible to embed in the same silicon die more than one memory array. In this way, all the control logic, analog circuits and the pads can be shared by the different memory arrays. In order to keep the area to the minimum, the memory arrays are grown one on top of the other, exploiting the most recent breakthroughs in silicon manufacturing technology.

Two different solutions have been recently presented for NAND Flash memories: in one case [21, 22], the topology of the memory array is the usual one, and another array is diffused on top of it, as shown in Fig. 2.48, so that two layers exist. Therefore the NAND strings (i.e. the series of Flash memory cells which is the basic building block of the array) are diffused on the X–Y plane. Around the arrays, all the peripheral circuitry is placed in the first (i.e. lower) layer. The only exception is the wordline (WL) decoder. To avoid Fowler–Nordheim (FN) erasing of the unselected layer, all WLs in that layer must be floating, just like the WLs of unselected blocks in the selected layer. This function is performed by a layer-dedicated WL decoder.

![Fig. 2.48. Three-dimensional horizontal memory array](image-url)
The second approach [23, 24] is shown in Figs. 2.49 and 5.9: in this case the NAND strings are orthogonal to the die (along the Z direction). NAND string is on the plugs located vertically in the holes punched through whole stack of the gate plates. Each plate acts as control gate except the lowest plate which takes a role of the lower select gate. 3-D charge trap memories are described in Sect. 5.3.

The density of the memory can also be increased acting at cell level: in its simplest form, a non-volatile memory cell stores one bit of information: ‘1’ when the cell is erased and ‘0’ when it is programmed: Sensing techniques for measuring the amount of electrical charge stored in the floating gate are described in Chap. 8. This kind of storage is referred to as Single Level Cell (SLC).

The concept can be extended by having four different charge levels (corresponding to logic values 00, 01, 10, 11) inside the floating gate, thus leading to the so-called Multi Level Cell (MLC) approach, i.e. 2 bit/cell. Chapter 10 is entirely dedicated to MLC NAND devices.

Several devices implementing the 2 bit/cell technology are commercially available, and indeed MLC has become a synonym for 2 bits/cell. Almost all the Flash cards contain MLC devices as they are cheaper than SLC.

The circuitry required to read multiple bits out of a cell is of course more complex than in the case of single bit, but the saving in term of area (and the increase in density) is worth the complexity. The real disadvantage lies in reduced endurance and reliability.

In terms of endurance, as already mentioned previously, a SLC solution can withstand up to 100,000 program/erase cycles for each block, while a MLC solution is usually limited to 10,000. For this reason, wear leveling algorithms must be used, as already outlined in Sect. 2.4.1. In terms of reliability, it is clear that the more levels are used, the more read disturbs can happen, and therefore the ECC capability must be strengthened.

![Fig. 2.49. Three dimensional vertical memory array: (a) top down view of 3-D vertical memory array; (b) equivalent circuit of the vertical NAND string](image-url)
The concept has been extended recently to 3 bit/cell and 4 bit/cell, where 8 and 16 different charge levels are stored inside the same cell. This storage approach is known as XLC and it is described in Chap. 16.

References

3. www.mmca.org
4. www.compactflash.org
5. www.sdcard.com
Inside NAND Flash Memories
Micheloni, R.; Crippa, L.; Marelli, A.
2010, X, 582 p., Hardcover
ISBN: 978-90-481-9430-8