UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Design and development of an embedded flash memory integrated simulator for the automotive microcontroller firmware validation

Laureando: **Roberto Guerra**

Relatore: **Prof. Andrea Bevilacqua**

Correlatori Aziendali:

Ing. Massimo Atti

Ing. Alessandro Di Marzio

Ing. Angelo De Poli

Anno Accademico 2012 / 2013

to Angela

"Immagination is more important than knowledge"

Albert Einstein

Padua, 15th october 2013

Contents

Abstract

Electronic components for automotive purposes have to guarantee higher persistent reliability than electronic consumer ones. Compare, for example, a device which controls the airbag system in a vehicle with a MP3 reader: both have a flash memory embedded, but if a bit corruption occurs in the MP3, in the worst case the customer will have to waste the device with a little bit of disappoint. Much more damages occur, instead, if a bit corruption induces an erroneously airbag activation in a vehicle.

In order to guarantee the necessary reliability and persistence in time of the electronic components for automotive purposes, the manufacturers have to spend many resources during and after the production process. Several researches have been carried out to characterize the components, in order to be able to predict their behavior for a long time, avoiding eventual malfunctioning.

In the automotive microcontrollers' scenario, the main effort required to guarantee the reliability is the test of the embedded flash memory. To accomplish this task, the devices are tested by programming the flash with data apparently with nonsense, just to the aim of verifying the correct operation. The way for achieving this goal consists in writing a particular firmware, denoted as "testware", which is downloaded and executed on the microcontroller. Developing the testware is a complex operation, since the device is usually not available during the firmware designing. In this context, the availability of testing the firmware with a simulator would be helpful to verify the correctness while the device is not yet concretely produced. In this way, the testware engineers' team would be able to provide the test application to the production centre promptly.

The main goal of this thesis is the creation of a simulator of the TC27x Infineon® microcontroller's unit, which embeds the flash memory. The simulator is dedicated to simplify the work of the engineer who has to design and perform the tests after the production and before the sale of the device.

SOMMARIO

I componenti elettronici per uso auto motive devono garantire un'elevata rispetto ai componenti rivolti ad applicazioni di elettronica di consumo. Paragonando un lettore MP3 con un dispositivo dedito al controllo degli airbag di un veicolo, ad esempio, si osserva che entrambi sono dotati di una memoria flash, ma se un errore si verifica nel primo per un difetto della memoria del dispositivo stesso, nel peggiore dei casi si dovrà sostituire l'oggetto con uno nuovo, senz'altro con disappunto; se, invece, un errore si verifica anche su un singolo bit della memoria del secondo dispositivo potrebbero verificarsi danni estremamente gravi, come l'esplosione del'airbag in condizioni di normale funzionamento del veicolo.

Per garantire la necessaria affidabilità dei componenti per applicazioni auto motive, i costruttori devono impiegare molte risorse durante e dopo il processo di produzione. Molti studi compiuti sulla caratterizzazione dei dispositivi hanno permesso di prevederne il funzionamento anche a lungo termine, evitando così malfunzionamenti dovuti a difetti di fabbricazione.

Per i microcontrollori per uso auto motive, lo sforzo maggiore è richiesto per garantire l'affidabilità della memoria flash in essi integrata, per la quale è necessario un test accurato effettuato programmandola con dati che, di fatto, non hanno nessun senso se non quello di verificarne il corretto funzionamento. Tale programmazione è detta "testware" ed è scaricata ed eseguita nel microcontrollore in oggetto. Lo sviluppo del testware è un'operazione complessa poiché il dispositivo solitamente non è disponibile durante la progettazione del firmware. In questo contesto, la disponibilità di testare il firmware con un simulatore sarebbe molto d'aiuto per verificarne la correttezza, dato che il dispositivo non è ancora realizzato. In questo modo, la squadra degli ingegneri che sviluppano il testware sarebbero in grado di fornire tempestivamente l'applicazione di test al centro di produzione.

L'obiettivo principale di questa tesi consiste nella creazione di un simulatore dell'unita che include la memoria flash dei microcontrollori Infineon® della famiglia TC27x, rivolto a semplificare il lavoro degli ingegneri che devono fornire il test da effettuare prima della vendita del dispositivo.

Chapter 1 Introduction

This thesis has been realized in collaboration with the Product Test Engineering's team of Infineon Technologies® Italia, located in Padua.

The main goal of this work is the creation of an Infineon TriCore 27x microcontroller simulator dedicated to simplify the work of the engineer who has to design and perform the tests on the controller after the production and before the sale.

1.1 Infineon Technologies®

Infineon Technologies® is a German semiconductor manufacturer founded in 1999, when the semiconductor operations of the parent company Siemens were spun off to form a separate legal entity. In the last years, Infineon Technologies has become a leader company in the design and manufacturing of electronic components. Its research activities and business have always been focused on the three central challenges facing modern society: energy efficiency, mobility and security, offering semiconductors and system solutions for automotive and industrial electronic and security applications [***1***].

The Infineon's automotive division supplies the automotive industry with sensors, microcontrollers, power semiconductors and power modules that contribute to a more sustainable mobility in terms of reduced fuel consumption and emissions, improved safety and affordability.

Automotive products require much more reliability than consumer's ones; therefore a great effort is made by all manufacturers of these devices, such as Infineon Technologies, to guarantee the proper operation in any condition. In this context, tests have a key role during the components' production process, since the proper operation of a device must be verified under different kind of stresses (e.g. temperature, high voltage, etc.) to be imposed both for short and long terms.

Since the correct design of the test phase has remarkable effects on the device final cost, Infineon Technologies spends many resources in the optimization

of its own test processes. In particular, the microcontroller's (testware) team in Padua is focused on the design and improvement of the test concerning the flash memory of microcontrollers.

The goal of this thesis is the creation of a tool which allows the testware's team to validate the firmware they usually write to test the flash module, normally known as "testware". This tool is based on software which imitates the behavior of a flash module of the microcontrollers belonging to a specific family.

1.2 Introduction to Microcontrollers

A microcontroller is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of "NOR" flash or "OTP ROM" is often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computer or other general purpose applications.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

Some microcontrollers are optimized for low power consumption (single-digit milliwatts or microwatts). They will generally have the ability to retain functionality while waiting for an event such as a button pressed or other interrupts; power consumption while sleeping (CPU clock and most peripherals off) may be just nanowatts, making many of them well suited for long lasting battery applications.

Other microcontrollers may serve performance-critical roles, where they may need to act more like a digital signal processor (DSP), with higher clock speeds and power consumption.

1.2.1 RISC Architecture

The most common microcontrollers' CPU design strategy is based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A

computer based on this strategy is a reduced instruction set computer, also called RISC [***2***].

Various suggestions have been made regarding a precise definition of RISC, but the general concept is that of a system that uses a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures. Another common trait is that RISC systems use the load/store architecture, where memory is normally accessed only through specific instructions, rather than accessed as part of other instructions.

Although a number of systems from the 1960s and 70s have been identified as being forerunners of RISC, the modern version of the design dates to the 1980s. In particular, two projects at Stanford University and University of California, Berkeley are most associated with the popularization of the concept. Stanford's design would go on to be commercialized as the successful MIPS architecture, while Berkeley's RISC gave its name to the entire concept, commercialized as the SPARC. Another success from this era was IBM's efforts that eventually led to the Power Architecture. As these projects matured, a wide variety of similar designs flourished in the late 1980s and especially the early 1990s, representing a major force in the UNIX workstation market as well as embedded processors in laser printers, routers and similar products.

1.2.2 CISC Architecture

The opposing architecture is known as “Complex Instruction Set Computing” (CISC). In computers designed following concepts of CISC architecture, single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) and/or are capable of multi-step operations or addressing modes within single instructions [***2***].

Before the RISC philosophy became prominent, many computer architects tried to bridge the so-called semantic gap, i.e. to design instruction sets that directly supported high-level programming constructs such as procedure calls, loop control, and complex addressing modes, allowing data structure and array accesses to be combined into single instructions. Instructions are also typically highly encoded in order to further enhance the code density. The compact nature of such instruction sets results in smaller program sizes and fewer (slow) main memory accesses, which at the time (early 1960s and onwards) resulted in a tremendous savings on the cost of computer memory and disc storage, as well as faster execution. It also meant good programming productivity even in assembly language, as high level languages were not always available or appropriate

(microprocessors in this category are sometimes still programmed in assembly language for certain types of critical applications).

1.2.3 Infineon TriCore microcontrollers

The Infineon's product portfolio includes microcontrollers based on the TriCore architecture, which usually finds application in automotive and security fields.

TriCore is a 32-bit microcontroller/DSP architecture optimized for "integrated real-time systems". It combines the outstanding characteristics of three different areas: the signal processing of DSP, real-time microcontrollers, and RISC processing power; and it allows the implementation of RISC load-store architectures. Fig. 1.1 shows a diagram summarizing the TriCore architecture, which offers an ideal system requiring fewer modules because more functions are integrated on the chip. This family of controllers is equipped with an optimal range of powerful peripherals for a wide spectrum of applications in power train, safety, and vehicle dynamics, driver information and entertainment electronics, and for body and convenience applications.

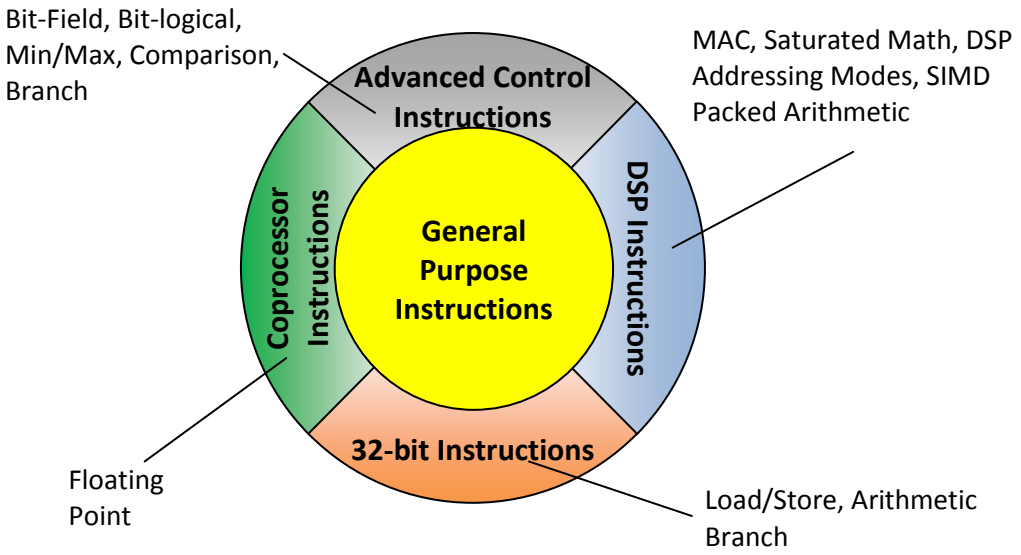

Fig. 1.1: TriCore: one architecture with a modular instruction set.

The instruction set architecture (ISA) supports a global linear 32-bit address space with memory-oriented I/O. The operation of the core is superscalar, i.e. it can execute simultaneously up to three instructions with up four operations. Furthermore, the ISA can work in conjunction with different system architectures,

also with multi-processing architectures. This flexibility at the implementation and system level permits different cost/performance combinations to be created whenever required.

TriCore contains a mixed 16-bit and 32-bit instruction set. Instructions with different instruction lengths can be used alongside each other without changing the operating mode. This substantially reduces the volume of code, so that even faster execution is combined with a reduction in the memory space requirement, system costs and energy consumption.

The real-time capability is essentially determined by the interrupt wait and context switching times. Here, the high performance architecture reduces response times to a minimum by avoiding long multi-cycle instructions and by providing a flexible hardware-supported interrupt scheme. In addition, the architecture supports rapid context switching. [***3***]

The basic features of the TriCore architecture are summarized in the following list:

- 32-bit architecture;
- Unified 4-Gbyte data, program, and input/output address space;
- 16-bit/32-bit instructions to reduce code volume;
- Low interrupt response times;
- Fast, automatic HW context switching;
- Multiplication-accumulation unit;
- Saturation integer arithmetic;
- Bit-operations and bit addressing supported by the architecture and instruction set;
- Packed data operations (single instruction multiple data, SIMD);
- Zero overhead loop for DSP applications;
- Flexible power management;
- Byte and bit addressing;
- Little endian byte order;
- Support for big and little endian byte ordering on the bus interface;
- Precise exception states;
- Flexible, configurable interrupt managements with up to 256 levels.

In 1999, Infineon launched the first generation of AUDO (Automotive unified processor) which was based on TriCore architecture. The fourth generation of AUDO is called AUDO MAX and it has been the spearhead up to 2011.

Actually, the newest Infineon's microcontrollers belong to AURIX generation which guarantee more relevant features than AUDOMAX, inasmuch are realized with newer CMOS technology.

1.3 Motivations

In Infineon's microcontrollers the module which contains the flash memory is the largest unit of the device. It involves both digital and analog sub-circuits which allow every memory cell to be read, programmed and erased.

Since 2002, the Infineon microcontrollers have adopted the System-On-Chip (SoC) technique to test the embedded flash memory. This technique makes use of the resources of the device under test to detect eventual breakdown of the device itself. In general, this is accomplished by loading a dedicated firmware on the microcontroller (testware), which has the only purpose of executing particular operations on the flash array. Any unexpected reaction of the microcontroller detects some non-compliance. If the faulty so revealed cannot be resolved, the microcontroller will not be sold.

Typically, microcontroller flash memories are used to store both the program instructions that will be executed by the CPU, and the generic data which have to be maintained during the lack of power supply. Especially for the first one, it is very important to guarantee the proper functionality of each single bit. Consider, for example, an instruction which has to decide whether or not the command to an air-bag's explosion has to be given: any malfunction is not eligible.

In this context, testing the flash memory is very important for each device produced, since the test phase is expensive inasmuch it takes a lot of time for any device. Hence characterization, validation and test have an important role to balance the quality and the cost of the provided device.

The simulator developed in this thesis is intended to validate the firmware used during the test phase, thus facilitating the process of the test's design: direct validation on the microcontroller is more expensive than using a simulator, because it requires hardware instruments that are not always available. Furthermore, some devices are still in the design stage while the engineers are writing the testware which could not be run on the device inasmuch is not physically available. Another reason of using the simulator to test the firmware is the easier way to monitor the flash module status during the programming and erase operation, which, instead, could be challenging if performed directly on the microcontroller.

This simulator is intended to be interfaced with an existed CPU simulator which is provided by the Bristol team of Infineon. Therefore, the thesis's output is a

whole microcontroller simulator which will be able to run some firmware which in turn will be able to compute operations on the flash module. Currently, the CPU simulator is already there, whereas the flash module will be implemented in this work. Next improvements will provide the enhancement of the simulator by adding other peripherals existing in hardware.

1.4 Thesis navigation

In Chapter 2, the flash memory technology will be introduced. Infineon makes use of the NOR-based architecture because it is more reliable than the AND one.

Unlike other solid state non-volatile memories, a finite state machine is needed to handle a flash array. In Infineon's microcontrollers the module which manages the flash array, through the combination of digital and analog sub-circuits, is called Program Memory Unit. The Chapter 3 will provide a brief description of the Infineon's microcontroller family focusing on the Program Memory Unit whose function is imitated by the simulator object of this thesis.

In Chapter 4, advantages and disadvantages of using a simulator will be described, and the technology used by the developed simulator will be explored. The union of two different kind of programming language has been adopted: Verilog and C++. The chapter will compare the features of each one.

In Chapter 5, the implementation techniques will be explained. Each part of the simulator will be described, providing information about the algorithms which compute the reactions of the microcontroller.

In Chapter 6, some test examples will be given for the PMU simulator and its interfacing with the CPU simulator, providing some tests computed by the whole microcontroller simulator which return the respective results.

Finally, Chapter 7 will present and describe the possible next improvements and enrichments of the microcontroller's simulator.

Chapter 2 Semiconductor Memories

A memory is a device that is able to store information for more or less time, making it available, at least to be read. Since the 80's the technology used to realize electronic memories is the complementary metal–oxide–semiconductor (CMOS) technology, which is also used to build most of the integrated circuits, such as microprocessors, microcontrollers, static RAM, and other digital and analog circuits or sub-circuits belonging to a more complex system.

This technology is based on two networks, called pull-up and pull-down. They provide a path to either ground or VDD which can be complementarily active. The fundamental unit of CMOS is the MOSFET (metal-oxide-semiconductor-field-effect-transistor), a field effect transistor which in general, in digital circuits is used as a switch, whereas in analog circuits is used as a transconductor.

In order to get a mechanism able to store electric charge, the structure of the common MOSFET is modified by adding a floating gate. The device so obtained is the fundamental component to realize memories.

The purpose of this chapter is to describe how the memories work. Description of different kinds of memories and the related issue to capacity, speed and reliability will be provided, focusing on the Flash memories. Flash technology, in fact, nowadays is the most common to realize memories that have to store data even if the power supply is missing.

The two largest categories the memories are divided in are:

- Volatile memories;
- Non-volatile memories.

The first ones are able to keep data as long as the power supply is provided; indeed, the data are lost as soon as the power supply gives out.

The Non-volatile memories are able to keep data during the absence of the power supply for a long time. However, in general after ten years that the device is not powered, the data is lost. The flash memories belong to this category.

2.1 Volatile Memories

Most types of random access memory (RAM) belong to volatile memories' category. Random access memories can be divided into Static-RAM (SRAM) and Dynamic-RAM (DRAM).

2.1.1 Static Random Access Memories

The basic Static-RAM structure is a cell composed by six transistor connected to each other like shown in Fig. 2.1:

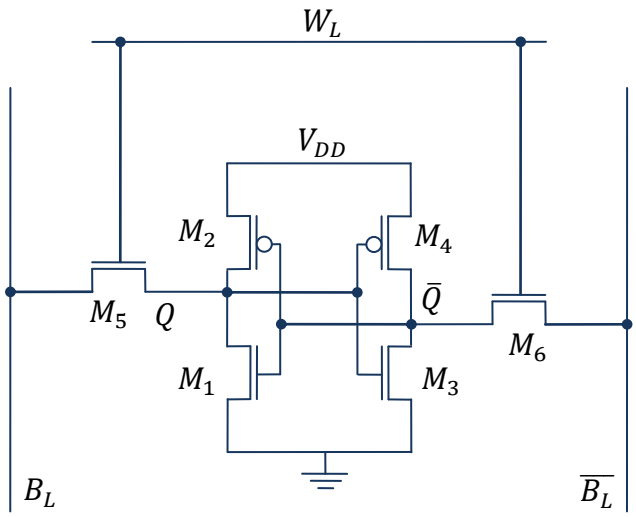

Fig. 2.1: Static RAM 6T cell.

The cell stores just one bit in transistors M_1, M_2, M_3 and M_4. These form two cross-coupled inverters. This storage cell has two stable states which are used to store "0" and "1". Two additional access transistors control the access to a storage cell during read and write operations. In addition to such six-transistors SRAM, other kinds of SRAM chips use 4T (as shown in Fig. 2.2), 8T, 10T, or more transistors per bit. Four-transistors SRAM is quite common in stand-alone SRAM devices (as opposed to SRAM used for CPU caches), implemented in special processes with an extra layer of poly-silicon, allowing for very high-resistance pull-up resistors.

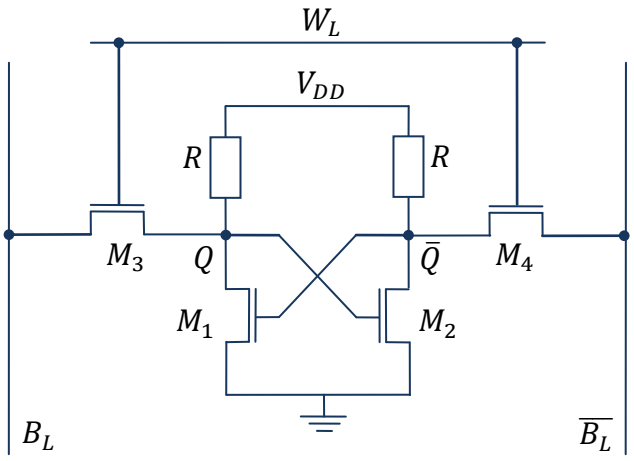

Fig. 2.2: Static-RAM 4T cell.

Access to the cell is enabled by the word line which controls the two access transistors M_5 and M_6 which, in turn, control whether the cell should be connected to the bit lines. They are used to transfer data for both read and write operations. Although it is not strictly necessary to have two bit lines, both the signal and its inverse are typically provided in order to improve noise margins.

During read accesses, the bit lines are actively driven high and low by the inverters in the SRAM cell. This improves SRAM bandwidth compared to DRAMs. The symmetric structure of SRAMs also allows for differential signaling, which makes small voltage swings more easily detectable. Another difference with DRAM that contributes to making SRAM faster is that commercial chips accept all address bits at a time. By comparison, commodity DRAMs have the address multiplexed in two halves, i.e. higher bits followed by lower bits, over the same package pins in order to keep their size and cost down.

2.1.2 Dynamic Random Access Memories

A dynamic-RAM stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called “0” and “1”. Since capacitors leak charge, the information eventually fades unless the capacitor charge is periodically refreshed. Because of this refresh requirement, it is a dynamic memory as opposed to SRAM and others static memories.

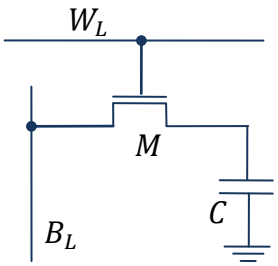

Fig. 2.3: Dynamic-RAM cell.

The advantage of DRAM is its structural simplicity: just one transistor and one capacitor are required per bit, compared to four or six transistors in SRAM. This allows DRAM to reach very high densities. Unlike flash memory, DRAM is a volatile memory, since it loses its data quickly when power is removed. The transistors and capacitors used are extremely small; billions can fit on a single memory chip.

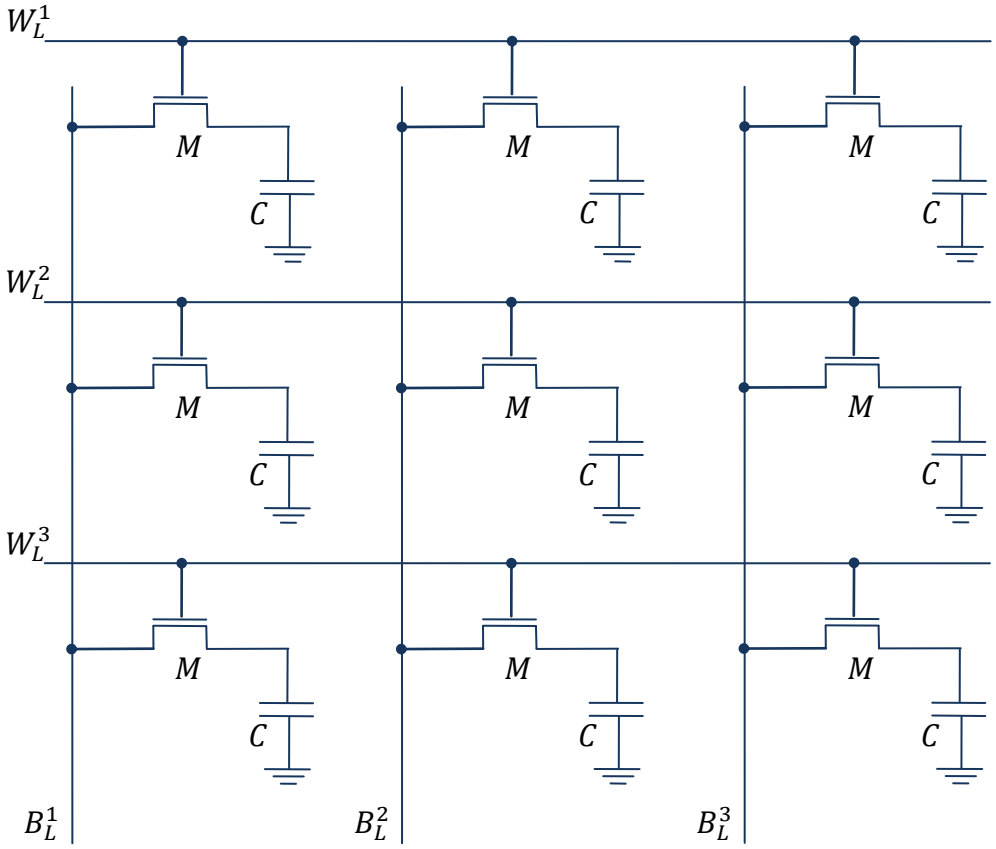

Fig. 2.4: DRAM Matrix.

DRAM is usually arranged in a rectangular array of charge storage cells consisting of one capacitor and transistor per data bit. The Fig. 2.4 shows a simple example with a 3 by 3 cell matrix. Modern DRAM matrices are many thousands of cells in height and width.

The long horizontal lines connecting each row are known as word-lines. Each column of cells is composed of two bit-lines, each connected to every other storage cell in the column, generally known as the bit lines.

Typically, manufacturers specify that each row must have its storage cell capacitors refreshed every 64 ms or less, as defined by the JEDEC (Foundation for developing Semiconductor Standards) standard. Refresh logic is provided in a DRAM controller which automates the periodic refresh, that is no software or other hardware has to perform it. This makes the controller's logic circuit more complicated, but this drawback is outweighed by the fact that DRAM is much cheaper per storage cell and because each storage cell is very simple, DRAM has much greater capacity per unit of surface than SRAM.

2.2 Non-Volatile Memories

A non-volatile memory (NVM) is a computer memory able to store information even when not powered. Examples of non-volatile memory include:

- Read-Only Memory (ROM);
- Programmable Read Only Memory (PROM);
- Erasable Programmable Read Only Memory (EPROM);
- Electrically Erasable Programmable Read Only Memory (EEPROM)
- Flash memory;
- Most types of magnetic computer storage devices (e.g. hard disks, floppy disks, and magnetic tape);
- Optical discs.

Non-volatile memories are typically used for the task of secondary storage, or long-term persistent storage. Unfortunately, most forms of them have too many limitations to be used as primary storage. Typically, they either cost more or have a poorer performance than volatile RAMs.

Read Only Memories (ROMs) are NVMs implemented by writing permanently the data in the memory array during manufacturing. Their drawback is that they cannot be programmed, so information is stored permanently. To solve, at least partially this issue, Programmable-ROMs (PROMs) were developed around the 1970s. They offer the possibility to write just once the content data of the memory

by blowing fusible links or anti-fuses, changing permanently the cell content. The disadvantage of both ROM and PROM is that they cannot be erased, making them suitable for a limited set of applications.

Over the years, non-volatiles memories have been improved, introducing the possibility to erase the stored data. The EPROMs (Erasable Programmable Read Only Memories) were the first memories which could be erased by exposing to Ultra-Violet (UV) radiation for about 20 minutes, whereas they could be electrically programmed. The long lead times required for erasure pushed the researchers to completely change the technology used to erase the memory content. In 1980s, thanks to the intuition of George Perlegos of using a charge pump to supply the high voltages necessary for programming, the EPROM memories were improved to offer the possibility to be electrically erased, giving rise to the "Electrically Erasable Programmable Read Only Memories", EEPROMs. Their drawback was the larger used areas then EPROMs; therefore incrementing the cost and reducing the density.

Flash memories – which nowadays are largely used in most of applications – combine the electrical in-system erase-ability of EEPROMs with the high density of EPROMs and an access time comparable to DRAMs. For this reason they are the most common technology used to store data permanently in different applications.

2.2.1 The Floating Gate Device

In order to obtain a memory cell having two logical stable states which is able to keep stored information independently of external conditions, the storage element must be a device having conductivity which can be altered in a non-destructive way.

Most of the solutions adopted over the years consist in a transistor with a programmable threshold voltage; i.e. an element able to change its state from a high to low impedance and vice versa. These states correspond to the states of a memory cell "erased" and "programmed" [***4***].

Considering a MOSFET, the threshold voltage can be derived using the following equation:

$$V_{th} = K - \frac{Q}{C_{ox}} \tag{2.1}$$

Where K is a constant which depends on the material of the gate and bulk, channel doping and gate oxide capacitance, Q is the charge in the gate oxide and C_{ox} is the gate oxide capacitance.

It is clear that the threshold voltage can be altered by changing the amount of charge between the gate and the channel. The two most common ways to achieve this goal are:

- Charge Trapping: the charge is stored in traps that are into the insulator or at interface between two dielectric materials. The most commonly used interface is the silicon oxide/nitride interface as shown in Fig. 2.5(b). Devices that store charge in this way are called MNOS (Metal-Nitride-Oxide-Silicon) [***5***] - [***6***];
- Floating Gate Device: the charge is stored in a conductive layer that is between the gate and the channel and is completely surrounded by insulator as shown in Fig. 2.5(a). This workaround is called Floating Gate (FG).

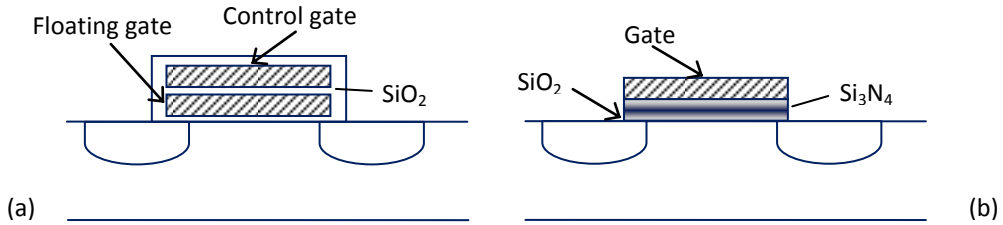

Fig. 2.5: Two classes of non-volatile semiconductor memory devices: (a) floating gate devices; (b) charge-trapping devices (MNOS device).

MNOS devices are not used anymore in consumer electronics due to their low endurance and retention. FG devices are at the basis of every modern NVM, particularly for Flash applications.

2.2.1.1 Floating Gate Device's operation

The basic concepts and the functionality of a floating gate device are easily understood if it is possible to determine the floating gate potential. The schematic cross section of a generic floating gate device is shown in Fig. 2.6; the upper gate is the control gate and the lower gate, completely isolated within the gate dielectric, is the floating gate (FG). The FG acts as a potential well. If a charge is forced into the well, it cannot move from there without applying an external force, so the FG stores charge [***7***].

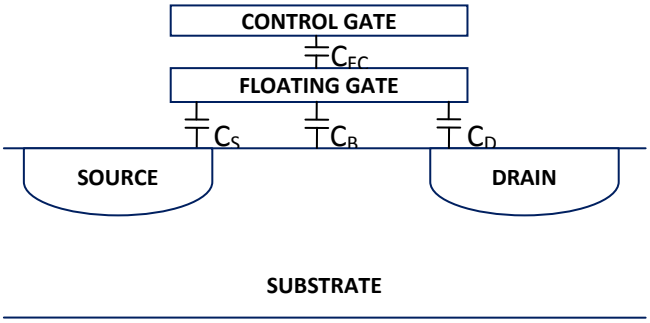

Fig. 2.6: Schematic cross section of a floating gate device.

The simple model shown in Fig. 2.6 helps in understanding the electrical behavior of a FG device. C_{FC}, C_S, C_D and C_B are the capacitances between the FG and control gate, source, drain, and substrate regions, respectively. Consider the case when no charge is stored in the FG, i.e. $Q = 0$.

$$Q = 0 = C_{FC}(V_{FG} - V_{CG}) + C_S(V_{FG} - V_S) + C_D(V_{FG} - V_D) + C_B(V_{FG} - V_B) \quad (2.2)$$

Where V_{FG} is the potential on the FG, V_{CG} is the potential on the control gate and V_s, V_D, and V_B are potential on source, drain and substrate respectively. If we define $C_T = C_{FC} + C_D + C_S + C_B$ the total capacitance of the FG, and $\alpha_J = C_J/C_T$ the coupling coefficient relative to the electrode J, where J can be one among G, D, S and B, the potential on the FG due to capacitive coupling will be given by:

$$V_{FG} = \alpha_G \cdot V_{GS} + \alpha_D \cdot V_{DS} + \alpha_S \cdot V_S + \alpha_B \cdot V_B \quad (2.3)$$

It should be pointed out that (2.3) shows that the FG potential does not depend only on the control gate voltage but also on the source, drain, and bulk potentials. If the source and bulk are both grounded, (2.3) will be able to be rearranged as:

$$V_{FG} = \alpha_G \left(V_{GS} + \frac{\alpha_D}{\alpha_G} \cdot V_{DS}\right) = \alpha_G(V_{GS} + f \cdot V_{DS}) \quad (2.4)$$

Where:

$$f = \frac{\alpha_D}{\alpha_G} = \frac{C_D}{C_{FC}} \tag{2.5}$$

Device equations for the FG MOS transistor can be obtained from the conventional MOS transistor equations by replacing MOS gate voltage V_{GS} with FG voltage and transforming the device parameters, such as threshold voltage and conductivity factor β, to values measured with respect to the control gate. Supposing to define for $V_{DS} = 0$:

$$V_T^{FG} = \alpha_G \cdot V_T^{CG} \tag{2.6}$$

$$\beta^{FG} = \frac{1}{\alpha_G}\beta^{CG} \tag{2.7}$$

It will be possible to compare the current–voltage (I–V) equations of a conventional and a FG MOS transistor in the triode region and in the saturation region [***8***].

- **Conventional MOS transistor**
  - **Triode Region**: $|V_{DS}| < |V_{GS} - V_T|$

$$I_{DS} = \beta\left[(V_{GS} - V_T) \cdot V_{DS} - \frac{1}{2} \cdot V_{DS}^2\right] \tag{2.8}$$

  - **Saturation Region:** $|V_{DS}| \geq |V_{GS} - V_T|$

$$I_{DS} = \frac{\beta}{2}(V_{GS} - V_T)^2 \tag{2.9}$$

- **Floating gate MOS transistor**
  - **Triode Region:** $|V_{DS}| < \alpha \cdot V_{GS} + f \cdot V_{DS} - V_T$

$$I_{DS} = \beta\left[(V_{GS} - V_T) \cdot V_{DS} - \left(f - \frac{1}{2 \cdot \alpha_G}\right) \cdot V_{DS}^2\right] \tag{2.10}$$

  - **Saturation Region:** $|V_{DS}| \geq \alpha \cdot V_{GS} + f \cdot V_{DS} - V_T$

$$I_{DS} = \frac{\beta}{2} \cdot \alpha_G (V_{GS} + f \cdot V_{DS} - V_T)^2 \tag{2.11}$$

Where β and V_T of (2.10) and (2.11) are measured with respect to the control gate rather than with respect to the FG of the stacked gate structure. They are to be read as $\beta(control\ gate) = \beta^{CG}$ and $V_T(control\ gate) = V_T^{CG}$. Several effects can be observed from these equations, many of them due to the capacitive coupling between the drain and the FG, which modifies the I–V characteristics of FG MOS transistors with respect to conventional MOS transistors.

- The floating gate transistor can go into depletion-mode operation and can conduct current even when $|V_{GS}| < |V_T|$. This is because the channel can be turned on by the drain voltage through the $f \cdot V_{DS}$ term. This effect is usually referred to as “drain turn-on.”
- The saturation region for the conventional MOS transistor is where I_{DS} is essentially independent on the drain voltage. This is no longer true for the floating gate transistor, in which the drain current will continue to rise as the drain voltage increases and saturation will not occur.
- The boundary between the triode and saturation regions for the FG transistor is expressed by:

$$|V_{DS}| = \alpha_G \cdot |V_{GS} + f \cdot V_{DS} - V_T| \tag{2.12}$$

  Compared to the conditions valid for the conventional transistor, $|V_{DS}| = |V_{GS} - V_T|$.
- The transconductance in SR is given by:

$$g_m = \left.\frac{\partial I_{DS}}{\partial V_{GS}}\right|_{V_{DS}=const.} = \alpha_G \cdot \beta \cdot (V_{GS} + f \cdot V_{DS} - V_T) \tag{2.13}$$

  Where $g_m$increases with V_{DS} in the floating gate transistor in contrast to the conventional transistor, where g_m is relatively independent of the drain voltage in the saturation region.

- The capacitive coupling ratio f depends on C_D and C_{FC} only $\left(f = {\alpha_D}/{\alpha_G} = \right.$ $CDCFC$, and its value can be verified by:

$$f = -\frac{\partial V_{GS}}{\partial V_{DS}}\bigg|_{I_{DS}\ const.} \tag{2.14}$$

in the saturation region.

Many techniques have been presented to extract the capacitive coupling ratios from simple dc measurements [***9***]–[***10***]. The most widely used methods [***11***], [***12***] are:

- Linear threshold voltage technique
- Subthreshold slope method
- Transconductance technique

These methods require the measurement of the electrical parameter in both a memory cell and in a "dummy cell," i.e., a device identical to the memory cell, but with floating and control gates connected. By comparing the results, the coupling coefficient can be determined. Other methods have been proposed to extract coupling coefficients directly from the memory cell without using a "dummy" one, but they need a more complex extraction procedure [***13***]–[***14***].

2.2.1.2 The Reading Operation

In this section the case with charge stored in the FG will be considered, i.e. $Q \neq 0$. All the hypotheses made in 2.2.1.1 hold true, and the following modifications need to be included. Equations (2.4), (2.6) and (2.10) become:

$$V_{FG} = \alpha_G \cdot V_{GS} + \alpha_D \cdot V_{DS} + \frac{Q}{C_T} \tag{2.15}$$

$$V_T^{CG} = \frac{1}{\alpha_G} \cdot V_T^{FG} - \frac{Q}{C_T \cdot \alpha_G} = \frac{1}{\alpha_G} V_T^{FG} - \frac{Q}{C_{FC}} \tag{2.16}$$

$$I_{DS} = \beta\left[\left(V_{GS} - V_T - \left(1 - \frac{1}{\alpha_G}\right)\cdot\frac{Q}{C_T}\right)\cdot V_{DS} + \left(f - \frac{1}{2\cdot\alpha_G}\right)\cdot V_{DS}^2\right] \quad (2.17)$$

Equation (2.16) shows the V_T dependence on Q. In particular, the threshold voltage shift ΔV_T is derived as:

$$\Delta V_T = V_T - V_{T0} = -\frac{Q}{C_{FC}} \quad (2.18)$$

Where V_{T0} is the threshold voltage when $Q = 0$. This is the key result explaining the success of the FG device as the basic cell for nonvolatile memories applications.

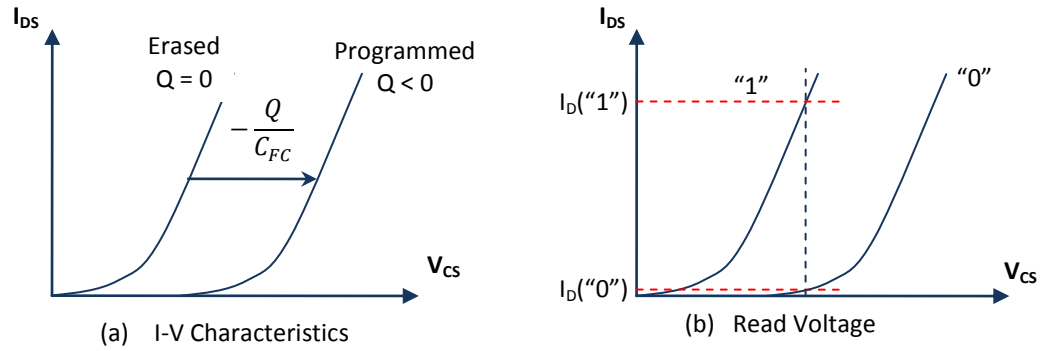

Fig. 2.7: Trans-characteristic of a floating-gate device.

Equation (2.17) shows that the role of injected charge is to shift the I-V curve of the cell. If the reading biases are fixed (usually $V_{GS} \cong 5V, V_{DS} \cong 1V$) the presence of charge will greatly affect the current level used to sense the cell state. Fig. 2.7(a) [***15***] shows two curves:

- The one the left side represents the "Erased" state: it is assumed when no charge is trapped in floating gate.
- The curve on the right side represents the same cell in the "Programmed" state: in figure is shown a case with $Q < 0$. If the charge trapped in floating gate is greater than zero the curve will be shift on the left side.

In the defined reading condition (Fig. 2.7(b)) an erased cell draws a significant current whereas the programmed cell does not draw any current. In the depicted case it will be assumed that an "erased" cell stores the "1" information, vice versa

the “programmed” cell stores the “0” information. This assumption can be change in different applications.

2.2.1.3 Charge Injection Mechanism

There are many solutions used to transfer electric charge from and into the FG. For both erase and program, the problem is making the charge pass through a layer of insulating material.

The channel hot-electron injection (CHE) mechanism generally is used in Flash memories, where a lateral electric field (between source and drain) “heats” the electrons and a transversal electric field (between channel and control gate) injects the carriers through the oxide.

The Fowler–Nordheim (FN) tunneling mechanism takes place when there is a high electric field through a thin oxide. In these conditions, the energy band diagram of the oxide region is very steep; therefore, there is a high probability that electrons pass through the energy barrier.

These two mechanisms have been deeply investigated for MOS transistors in order to avoid unwanted degradation effects. In Flash cells, they are exploited to become efficient program/erase mechanisms.

2.2.1.3.1 Channel Hot Electron Injection

The physical mechanism of CHE is relatively simple to understand qualitatively. An electron traveling from the source to the drain is powered by the lateral electric field and loses energy due to lattice vibrations (acoustic and optical phonons). At low fields, this is a dynamic equilibrium condition, which holds until the field strength reaches approximately $100\ {}^{kV}/_{cm}$ [***16***]. For fields exceeding this value, electrons are no longer in equilibrium with the lattice, and their energy relative to the conduction band edge begins to increase. Electrons are “heated” by the high lateral electric field, and a small fraction of them have enough energy to surmount the barrier between oxide and silicon conduction band edges. For an electron to overcome this potential barrier, three conditions must hold [***17***]:

- Its kinetic energy has to be higher than the potential barrier;
- It must be directed toward the barrier;
- The field in the oxide should be collecting it.

To evaluate how many electrons will actually cross the barrier, it should be known the energy distribution $f_E(\mathcal{E}, x, y)$ as a function of lateral field $\mathcal{E}$, the momentum distribution $f_K(E, x, y)$ as a function of electron energy E (i.e., how many electrons

are directed toward the oxide), the shape and height of the potential barrier, and the probability that an electron with energy E, wave vector k, and distance d from the Si/SiO_2 interface will cross the barrier. Each of these functions needs to be specified in each point of the channel (see Fig. 2.8).

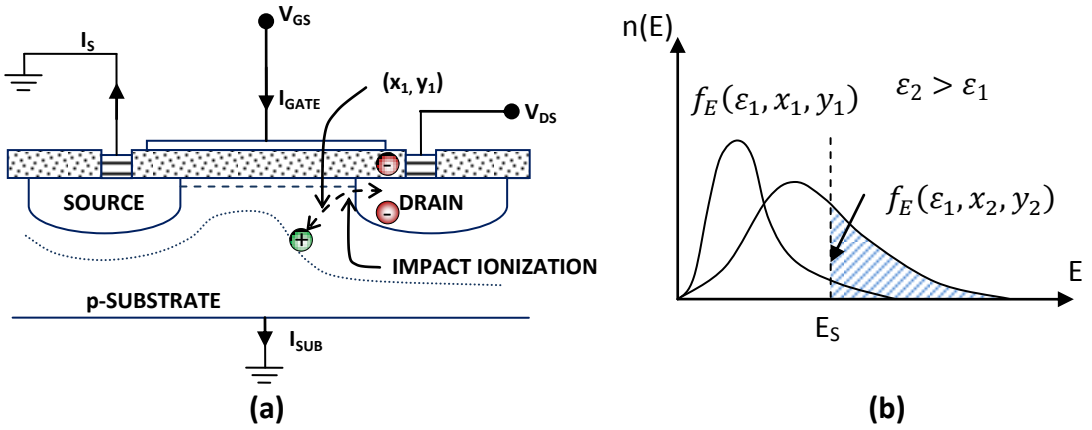

Fig. 2.8: (a) Schematic cross section of a MOSFET. (b) The energy-distribution function at point X_1, Y_1.

A quantitative model, therefore, is very heavy to handle. Moreover, when the energy gained by the electron reaches a threshold, impact ionization becomes a second important energy-loss mechanism [***18***], which needs to be included in models.

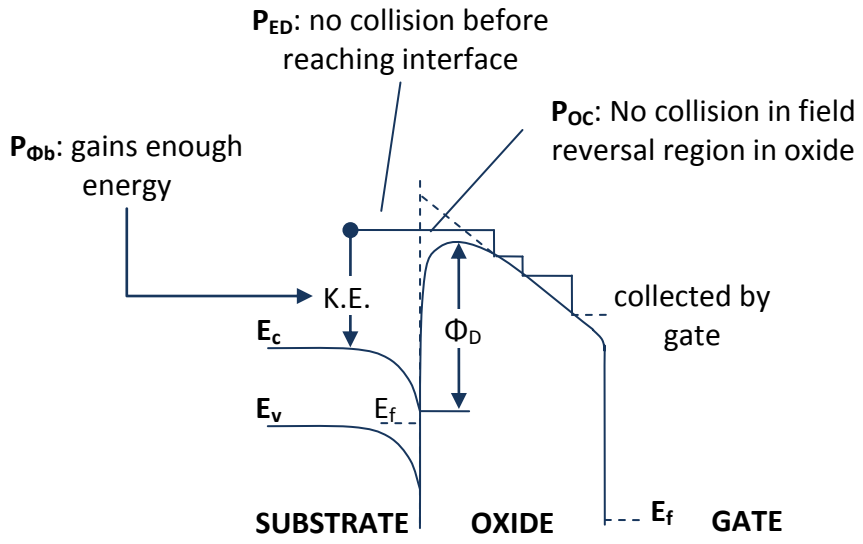

Fig. 2.9: A schematic energy band diagram describing the three processes involved in electron injection.

Nevertheless, a description of the injection conditions can be accomplished with two different approaches. The CHE current is often explained and simulated following the "lucky electron" model [***19***]. This model is based on the probability of an electron is being "lucky" enough to travel with a ballistic conduction phenomena in the field $\mathcal{E}$ for a distance several times the mean free path without scattering, eventually acquiring enough energy to cross the potential barrier if a collision pushes it toward the Si/SiO_2 interface. Consequently, the probability of injection is the lumped probability of the following events [***20***], which are depicted in Fig. 2.9.

- The carrier has to be "lucky" enough to acquire enough energy from the lateral electric field to overcome the oxide barrier and to retain its energy after the collision that redirects the electron toward the interface ($P_{\Phi b}$).
- The carrier follows a collision-free path from the redirection point to the interface (P_{ED}).
- The carrier can surmount the repulsive oxide field at the injection point, due to the Schottky barrier lowering effect, without suffering an energy-robbing collision in the oxide (P_{OC}).

Although this simple model does not fit precisely with some experiments, it allows a straightforward and quite successful simulation of the gate current. A more rigorous model is based on the quasi-thermal equilibrium approach [***21***], [***22***]. It assumes that the electron can be treated as a gas in quasi-thermal equilibrium with the electric field. This electron gas is characterized by an "effective temperature," which is different from the lattice temperature. The model establishes a non-local relation between the effective electron temperature and the drift field. Thus, the carrier probability to acquire certain energies depends on the complete profile of the electric field in the channel region [***23***].

2.2.1.3.2 Fowler-Nordheim Tunneling

In the framework of quantum mechanics, the solutions of the Schrödinger equation represent a particle. The continuous nonzero nature of these solutions, even in classically forbidden regions of negative energy, implies an ability to penetrate these forbidden regions and a probability of tunneling from one classically allowed region to another [***24***]. The concept of tunneling through a potential barrier applies well to MOS structures with thin oxide. Fig. 2.10 shows the energy-band diagram of a MOS structure with negative bias applied to the metal electrode with respect to the p-doped silicon substrate. The probability of electron

tunneling depends on either the distribution of occupied states in the injecting material or the shape, height, and width of the barrier.

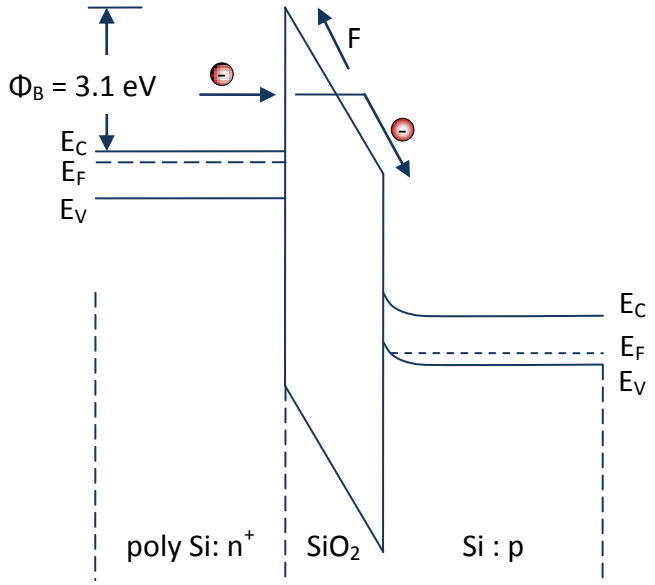

Fig. 2.10: FN tunneling through a potential barrier in a MOS structure.

Using a free-electron gas model for the metal and the Wentzel–Kramers–Brillouin (WKB) approximation for the tunneling probability [***25***], it is obtained the following expression for current density [***26***]:

$$J = \frac{q^3 \cdot F^2}{16\pi^2 h^2 \cdot \Phi_B} exp\left[-4\frac{(2 \cdot m_{ox}^*)^{\frac{1}{2}} \cdot \Phi_B^{\frac{3}{2}}}{3 \cdot \bar{h} \cdot q \cdot F}\right] \quad (2.19)$$

where Φ_B is the barrier height, m_{ox}^* is the effective mass of the electron in the forbidden gap of the dielectric, h is the Planck's constant, q is the electronic charge, and F is the electric field through the oxide.

2.2.1.4 EPROM Floating Gate Devices

In the following sections a historical review of transistors based on floating gate used for EPROM applications will be provided.

2.2.1.4.1 The floating gate avalanche-injection MOS transistor (FAMOS) Cell

In 1967 D. Khang and S. M. Sze at Bell laboratories [***27***] proposed the first non-volatile memory device based on a floating gate in a Metal-Insulator-Metal-Insulator-Semiconductor (MIMIS) structure. The lower dielectric had to be enough thin (< 5 nm) to allow quantum-mechanical tunneling of electrons from the substrate to the floating gate and vice versa. The MIMIS cell structure could not be manufactured reliably at that time because achieving a so thin oxide layer without defects was challenging.

For this reason, the tunneling mechanism was initially abandoned and the first operating floating gate device was developed at Intel in 1971 by Frohman-Bentchowsky [***28***]. This cell, without control gate, was programmed by applying a highly negative voltage at the drain to create a group of highly energetic electrons under the gate. To program the cell, the electrons were injected into the oxide reaching the floating gate. But, since the cell was devoid of control gate very high voltage was required, moreover the operation was extremely inefficient. In order to inject electrons in the floating gate, p-channel devices had to be used.

Erasure was performed by providing externally the required energy to re-emit electrons from the floating gate. To achieve this goal, exposing the cell to ultra-violet (UV) was needed. The FAMOS was developed into a double poly-silicon stacked gate n-channel device that constituted the basic cell of an EPROM. The cell was programmed by injection of channel hot-electrons into the floating gate and was erased using UV radiations (see Fig. 2.11).

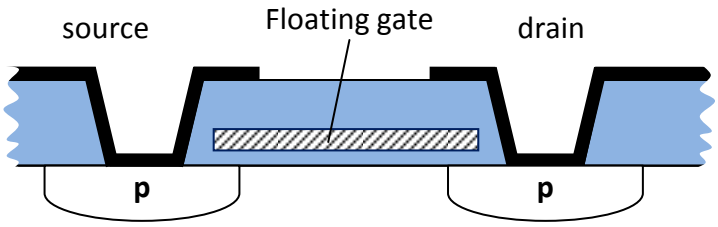

Fig. 2.11: First operating floating gate device: the FAMOS (Floating gate Avalanche injection MOS device).

Nonetheless some problems existed:

1. Hot electrons programming process was very inefficient, because needs both high voltage and high current;

2. Only the cell with control gate and drain at high voltage was programmed: it was a bit-selective operation;
3. The bit selective programming mechanism and the UV erasure process were self-limiting: first of all, UV erasure could produce over-erased cells because it cannot remove all electrons from the floating gate. An over-erased cell is a cell with excessive source-drain leakage current when unselected due to its threshold which is lower than the applied control gate voltage.

Moreover, if reprogramming was needed, an EPROM would have to be removed from the circuit board, UV erased and then reprogrammed. On the other hand, EPROMs could be implemented as a one-transistor memory cell, with the possibility to create extremely compact structures.

2.2.1.4.2 Metal-Nitride-Oxide Semiconductor (MNOS)

In 1967 Wegener introduced the MNOS cell [***29***], a standard MOS transistor wherein the oxide was been replaced by a nitride-oxide stacked layer (Fig. 2.12). The nitride is a charge storage element. Programming was achieved by inducing quantum-mechanical tunneling of electrons from the channel into the nitride traps and erasure by tunneling of holes from the semiconductor to the nitride traps when the gate potential was enough negative.

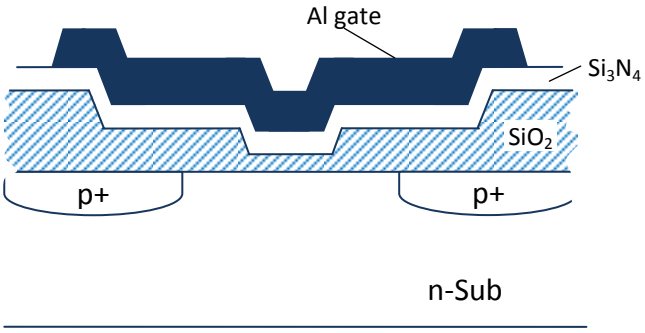

Fig. 2.12: Cross section of the p-channel tri-gate MNOS device.

2.2.1.4.3 Silicon-Nitride-Oxide Semiconductor (SNOS)

In order to improve the charge retention of MNOS, the SNOS (Silicon Nitride Oxide Semiconductor) devices were developed (Fig. 2.13). These kinds of memories had an improved retention and a reduced thickness of the nitride. A top oxide layer

was used between the gate and the nitride to prevent holes injection to the gate, giving rise to SONOS (Silicon Oxide Nitride Oxide Semiconductor) structure.

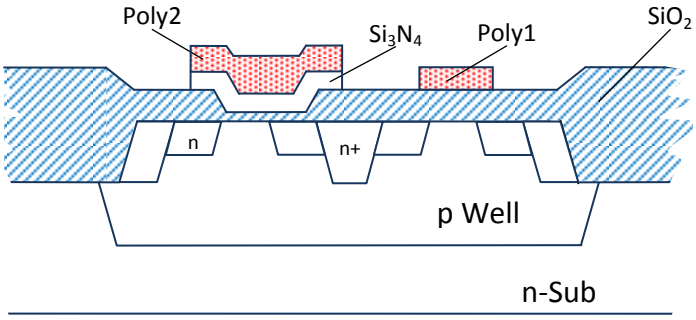

Fig. 2.13: Cross sections of the two transistors n-channel SNOS memory cell consisting of a MOS select transistor and a SNOS memory transistor.

2.2.1.5 EEPROM Floating Gate Devices

In the following sections a historical review of transistors based on floating gate used for EEPROM applications will be provided.

The EEPROM cell is composed of two transistors connected as shown in Fig. 2.14(a). A fulfillment on silicon is shown in Fig. 2.14(b), where the bit line metal layer is also depicted. The storage transistor has a floating gate which traps electrons. In addition, there is an access transistor, required for the erase operation. This is the main reason because the EEPROM cell takes a large area.

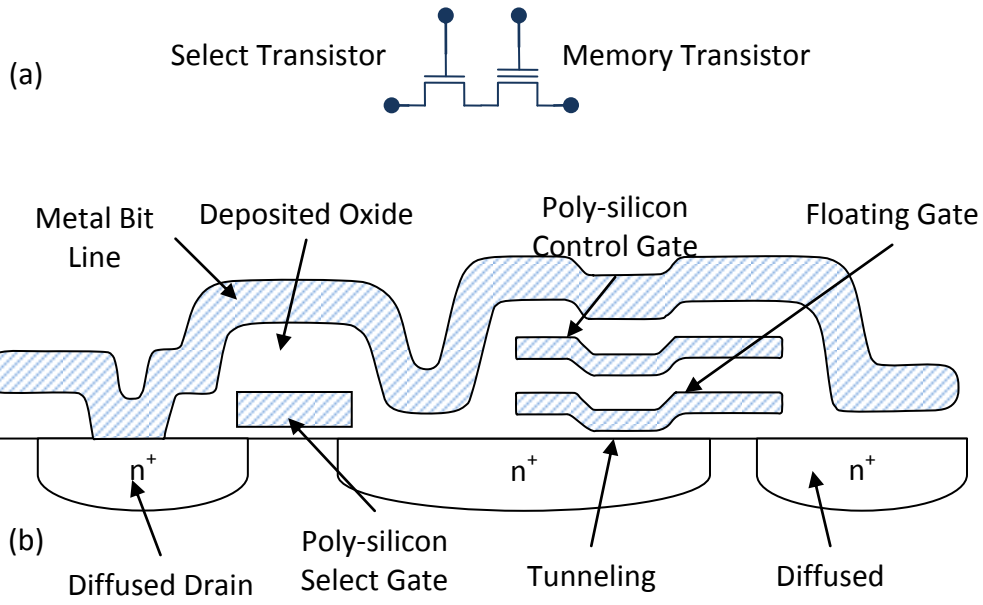

Fig. 2.14: EEPROM cell (a) schematic; (b) layout.

2.2.1.5.1 The FLOating gate Thin Oxide (FLOTOX) Memory Cell

Harari E. et al. proposed a non-volatile memory device called FLOTOX [***30***]: an electrically erasable and programmable cell which made use of Fowler-Nordheim tunneling mechanism [***31***]. It requires a "selection" transistor due to the non-selectivity of the tunneling process. Fig. 2.15 shows the cross section of the whole cell, including the selection transistor.

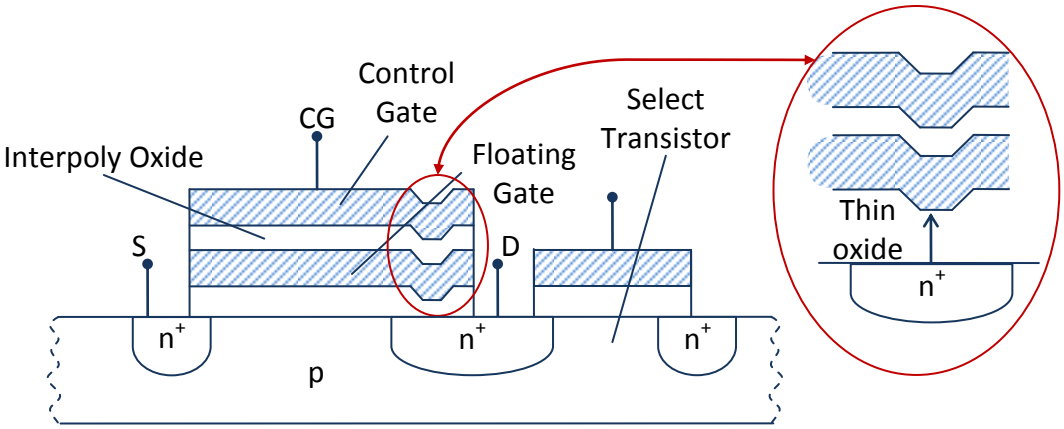

Fig. 2.15: Schematic section of a FLOTOX cell including the select transistor.

Programming is performed by applying a high voltage to the control gate, with the drain at low bias. By capacitive coupling, the voltage on the floating gate is also increased, and tunneling of electrons from the drain to the floating gate is initiated through the thin oxide grown on top of the drain. Erasing occurs when the drain is raised to a high voltage, and the control gate is grounded; the floating gate is capacitively coupled to a low voltage, and electrons tunnel from the floating gate into the drain. The drain bias is controlled by the select transistor.

2.2.1.5.2 Textured Poly-silicon Cell

A different way of using tunneling technique consists of employing the tunneling itself through oxides thermally grown on poly-silicon, instead of through the oxide between floating gate and drain. In this cell structure an enhancement of electric field exists although the used oxide layer is thick. This is due to the rough surface of poly-silicon. The cell consists in three transistors in series having poly gates partially overlapped as shown in Fig. 2.16. The floating gate is in the middle part of the structure (poly 2). Programming consists in injecting electrons from poly 1 to the floating gate, whereas the erasing injects electrons from the floating gate to poly 3. The voltage applied on poly 3 is always high, so erasing or programming is selected with the voltage applied to the drain. The cell structure results compact

and vertical and occupies less area than FLOTOX EEPROM cell, increasing the density. Furthermore, unlike the FLOTOX, this cell does not need to be programmed before selective erasure.

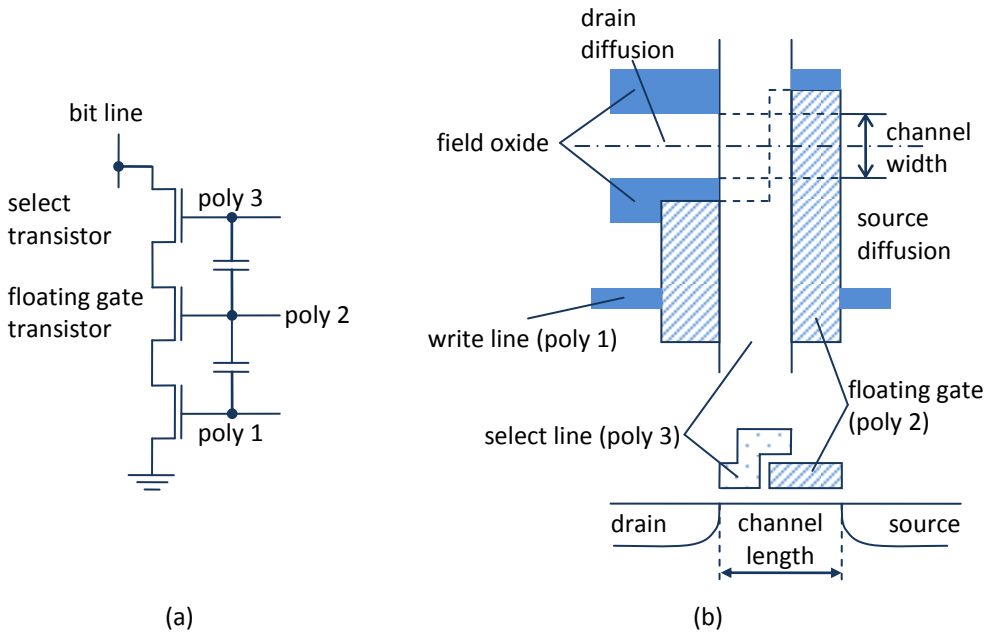

Fig. 2.16: Equivalent circuit (a), layout and schematic cross-section (b) of a textured poly EEPROM cell.

On the other hand, quality and reliability of this technology strongly depends on the features of both the poly-silicon and the oxide thermally grown on it. Trapping in the poly oxide may result in device wear-out.

2.2.1.5.3 Ferroelectric Memories

In order to obtain a non-volatile memory element, some approaches adopting ferroelectric materials have been demonstrated.

For non-volatile memories, ferroelectric materials serve not just a capacitors but as the memory element itself. Their main advantages are:

- Low voltage (1.0 V) operation;
- Small size (about 20% of a conventional EEPROM cell) and cost is proportional to size once high-yield production is achieved;
- Radiation hardness (not just for military applications, but for satellite communication systems)

- Very high speed (60 ns access time in commercial devices, sub-nanoseconds in laboratory tests on single cells).

Ferroelectrics are pyroelectric[1] crystals (include fine-grained ceramics) whose spontaneous electric polarization can be reversed by application of an external electric field that is smaller than the breakdown field [***32***].

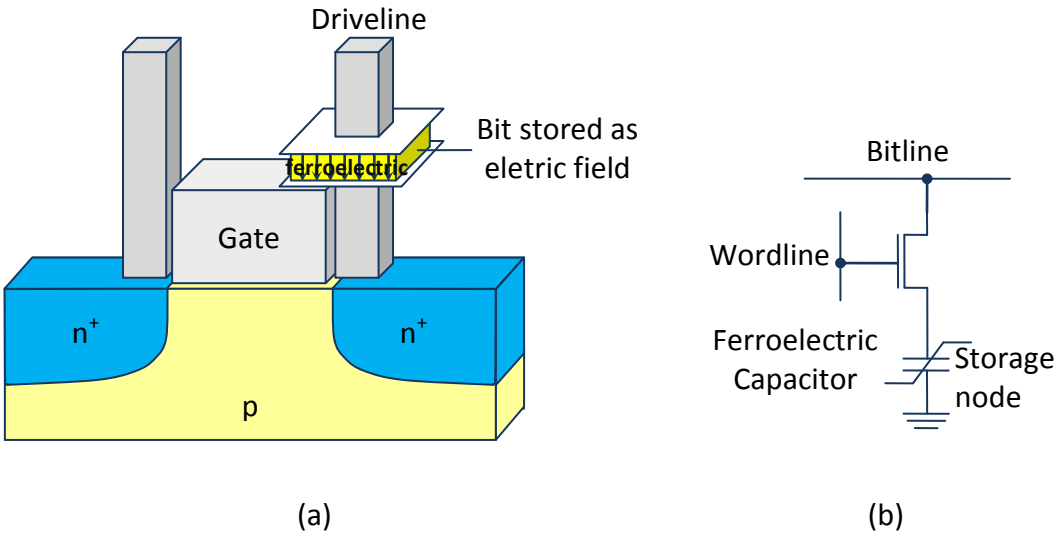

Fig. 2.17: (a) Schematic cross section of a ferroelectric nonvolatile DRAM. (b) Equivalent circuit.

Ferroelectric RAMs (FeRAMs) are structurally similar to dynamic-RAM discussed in 2.1.2, but the dielectric part of the capacitor is replaced by ferroelectric thin film Fig. 2.17. When the voltage applied to the capacitor exceeds a certain positive value V_{COERC}, the polarization becomes positive and increases up to a saturation value P_{SAT} (Fig. 2.18). The same applies for negative voltage lower than V_{NCOERC}, leading to a saturated polarization $-P_{SAT}$. When the electric field is removed, the ferroelectric film maintains its state of polarization, but the value of polarization is somewhat reduced to a relaxation value P_{REL} (or $-P_{REL}$ if a negative voltage has been applied). Two logic states are therefore possible, corresponding to the P_{REL} and $-P_{REL}$ polarizations. When, during the read operation, a positive voltage is applied to the ferroelectric capacitor the polarization changes from P_{REL}

[1] Pyroelectricity (from the Greek *pyr*, fire, and electricity) is the ability of certain materials to generate a temporary voltage when they are heated or cooled. The pyroelectric materials are crystalline substances capable of generating an electric charge in response to heat flow [***40***].

to P_{SAT}, thus requiring a low current (logic state "0") or from $-P_{REL}$ to P_{SAT}, which corresponds to a high current, or to the logic state 1. The difference in current between two memory states is sensed to generate the output. After reading the 1 state, the $-P_{REL}$ negative polarization must be regenerated by applying a negative voltage to the capacitor and vice versa.

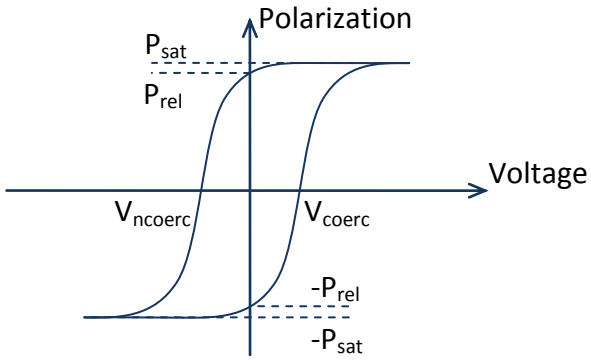

Fig. 2.18: Typical hysteresis curve of a ferroelectric capacitor, identifying polarization states.

2.2.2 Flash Memories

Flash memories inherit the EEPROM's structure, introducing a mechanism which allows simultaneously electrical erasure of a large number of cells (block, sector or page) inasmuch do not include the selected transistor. Such mechanism improves the performance in terms of speed, inasmuch the whole memory can be erased at the same time. Hence, flash memories combine the electrical in-system erase-ability of EEPROMs with the high density of the EPROMs and an access time comparable to DRAMs, and for this reason they are the most common technology used to store data permanently in different applications (such as microprocessors or microcontrollers mainly to allow software updates and reconfigure the system).

Programming is carried out selectively by means of the hot electron mechanism; erasing is based on tunneling, and is carried out in blocks of different sizes. The first cell based on this concept was presented in 1979 [***33***]; the first commercial product, a 256-K memory chip, was presented by Toshiba in 1984 [***34***].

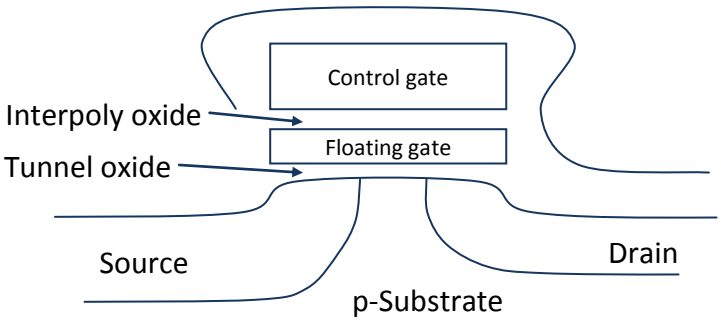

Fig. 2.19: Schematic cross-section of an industrial Flash cell.

Fig. 2.19 shows the cross-section of an industry-standard Flash cell. It is derived from an EPROM cell, but a few meaningful differences exist:

- The oxide between the substrate and the floating gate is very thin (~10 nm);
- If a high voltage is applied at the source when the control gate is grounded, a high electric field exists in the oxide, enabling electron tunneling from the floating gate to the source. This bias condition is dangerously close to the breakdown of the source-substrate junction.
- The source diffusion is realized differently from the drain diffusion, which does not undergo such bias conditions.

The cell is not symmetrical, but this is the only difference with respect to the standard EPROM process. This is a great advantage, since all the accumulated experience in process development can be used to produce these devices.

2.2.2.1 Flash Architectures

Different design approaches exist for flash memories. Each one exalts certain features that make each solution more suitable for a particular application. These approaches, give rise to different architectures of flash memories which, generally differ each other for [***35***]:

- Cell architecture;
- Cell functionality, i.e. voltages to be applied during program, erase and read operation;
- Target device performance;
- Array organization;

- Flash memory interface with the external world: communication protocol and I/O circuitry.

Moreover, flash memories are divided into two families, according to their main applications:

1. "EEPROM like" applications: telecommunications, automotive, hard disk's drivers, printers, etc. The most common features are:
   - Low density;
   - High capacity;
   - Easy integration with other device.

   The market requirements for this kind of memory are:

   - Speed;
   - Low power consumption;
   - Low supply voltages;
   - Density and maximum number of cycles.
2. Mass storage application, as multimedia card and palm-top. The market requirements for these applications are:
   - Profitability (cost/MB);
   - Density;
   - Number of cycles;
   - Low power consumption and speed.

Architectures of flash-arrays can be also divided according to data access and data write organization. In fact, arrays which have random access and random program (parallel) are used for embedded applications, whereas arrays which have page read and page program (serial) accesses are consistent with mass storage applications.

The device organization depends on the cell array architecture. Three implementations exist. They differ to each other for following features:

- Sector size;
- Read, program and erase performances;
- Power supply requirements;
- Cycling performance (Program / Erase);
- Complexity of process manufacturing and device size.

These implementations are:

- **NOR**: is the most commonly used in a wide range of applications that require both medium density and performances.
- **NAND**: is similar to the NOR, but the access to the matrix is different because the cells are arranged along the array in serial chains: the drain of a cell is connected with the source of the following one. The name comes from the way in which the read operation is performed. The access time is very slow and the high voltages needed to program/erase operation can cause some reliability problem.
- **EEPROMs**: derive from the EEPROM cells but the contents inside the memory can be altered on a block rather than on a byte basis. The functionality is the same of an EEPROM cell. The reliability is the main problem due to the high voltages and thick oxide.

Cells which use the CHE program and FN erase can be grouped into two main categories:

1. one-transistor cells and their array architectures (NOR common ground)
2. merged cells (split-gate triple poly, split-gate source injection)

Cells which use FN tunneling for program and erase operation can be grouped as:

1. NOR arrays (DINOR, asymmetrical contact less transistor, EEPROM like cells);
2. AND arrays (AND, HiCR cells);
3. NAND cells.

The motivation to use Fowler-Nordheim tunneling to the channel for both programming and erase comes from the need to change the programming mechanism to simplify the supply scaling and to reduce the cell sizes.

2.2.2.1.1 NOR Architecture

The NOR organization is shown in Fig. 2.20: cells belonging to the same word share the same word line, while many bit lines as the word length, are activated simultaneously.

For a read operation, the cell address has to be provided to the row and column decoders. The row decoder selects the corresponding word line by raising

its voltage while all the others are kept at ground. The addressed bit line, which is connected to the sense amplifier, will be flowed by a significant current if the addressed cell is erased (low threshold voltage). Otherwise, if the cell is programmed (high threshold voltage), the cell will not be "ON", so no current flows through the bit line. The sense amplifier deals to detect a significant current to reveal the value of the bit stored in the addressed cell. Note that this solution requires a sense amplifier per bit.

For both operations, programming and erasing, the word line is selected to activate the cell to be written. If the input data is "0" (a programming operation has to be performed), the bit line will be driven at high voltage, to allow the cell threshold voltage be incremented through channel hot electron injection (Fig. 2.21).

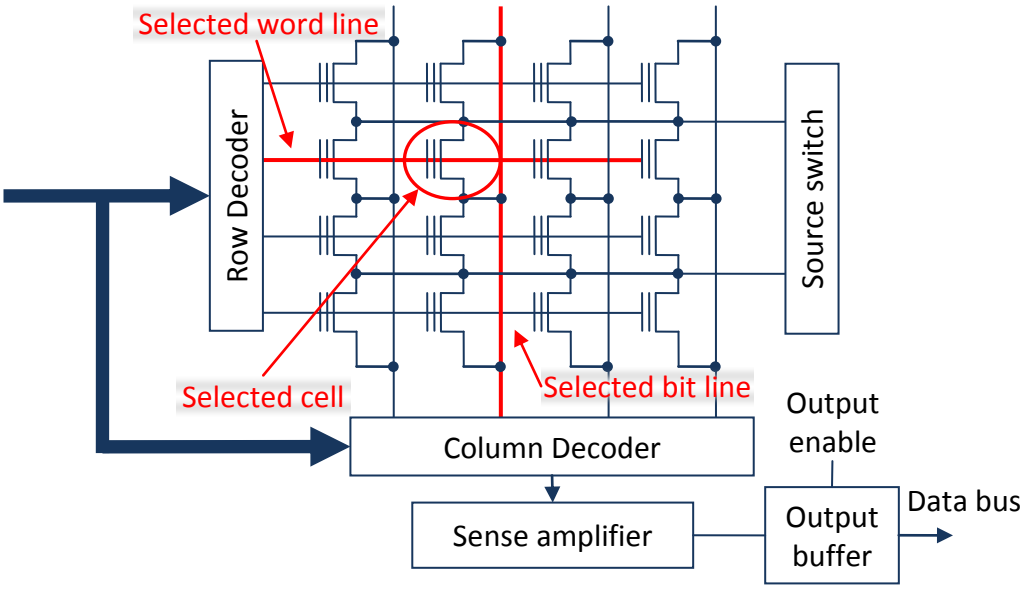

Fig. 2.20: Schematic structure of the read path in a NOR organization. Only one bit at a time is here considered as addressable.

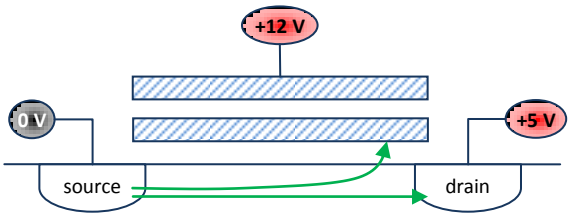

Fig. 2.21: Channel Hot Electron injection: voltage application for programming (writing value "0").

After the application of the programming voltages, it is necessary to verify whether the cell has been correctly programmed, i.e. if the threshold voltage is larger than a minimum acceptable value pointed as $V_{T_{pm}}$. This task is carried out by reading the cell selecting the word line with voltage higher than that applied during normal reading and by comparing the read data with that to be programmed (some digital blocks are also necessary e.g. register to keep the value which has to be programmed). If the two values are equal, it will mean that the threshold voltage of the cell has risen from the erased value to the programmed one and that, since this read operation has provided a correct result with the more critical condition of a higher reading voltage, the correct value is expected to be detected even in a conventional reading. If the verification fails, another programming pulse is applied to the cell (always by raising both gate and drain voltages), until the cell is correctly programmed or a maximum number of pulses has been reached, so that a fail signal will be produced.

For several reasons the erase procedure is even more complicated. First of all, it is performed on an entire sector, so that the verification process requires that all the cells of the sector are read in sequence. In addition, it is important to check the thresholds of some cells will not become too low and, in case, to raise their threshold to a higher value. A schematic behavior of the thresholds' distribution for cells belonging to the same sector is shown in Fig. 2.22.

Starting from a typical situation existing before erasing (Fig. 2.22(a)), once the erase procedure has been activated, all the cells of the sector are programmed with "0", so that their thresholds are increased (Fig. 2.22(b)). This normalization task reduces the possibility of over-erasing cells written with a "1" (that could become leaky when unaddressed), and it allows for a more uniform distribution of the erased thresholds, since all the initial thresholds belong to the same range.

To erase a single sector, a high electric field must be applied between the sources and the gates of the cells belonging to the sector, to allow Fowler-Nordheim current to discharge the floating gates. This task is accomplished in two different ways: by applying a high voltage (in the range of 12 V) to the source of the cells to be erased while grounding their gates (source erase), or by splitting the biasing voltage between source (at 5 ÷ 7 V) and gate (at -8 ÷ -10 V) (negative gate erase) (Fig. 2.23). Both solutions present a drawback: the highly negative bulk-source voltage drop can activate avalanche injection in the former case, while the generation of a negative voltage is required in the negative gat erasure.

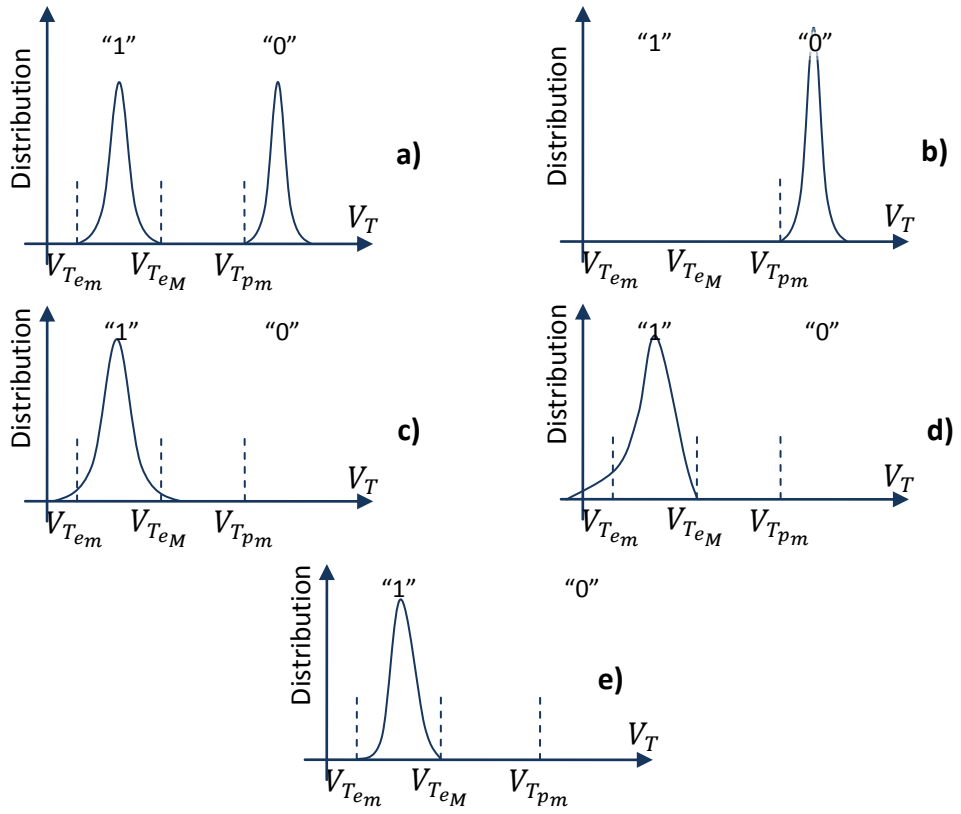

Fig. 2.22: Schematic distribution of the threshold voltages during an erase operation: a) before erase; b) after a "program all-0" operation; c) after a single erase pulse; d) after the erase verify procedure has been successfully performed; e) after soft-programming.

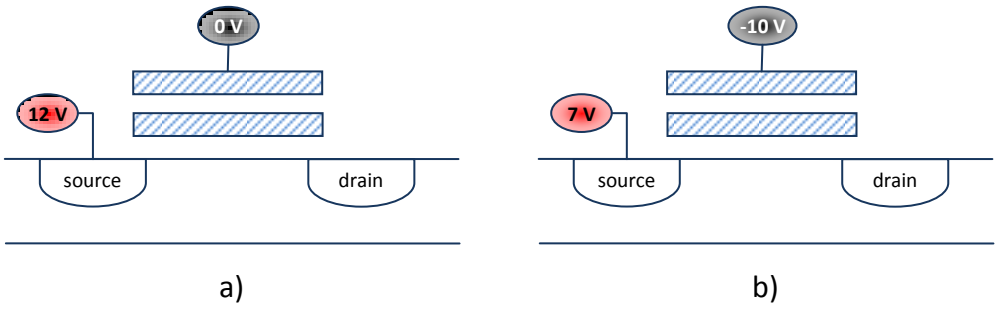

Fig. 2.23: Fowler-Nordheim voltage application for a) source erasing b) negative gate erasing (writing value "1").

Fowler-Nordheim tunneling current depends on many physical and technological parameters, so that even adjacent cells can discharge at different rates. After a single erase pulse has been applied, the threshold distribution may be

similar to that depicted in Fig. 2.22 c. In particular, many cells are not fully erased ($V_T > V_{T_{e_m}}$). Hence, as for the programming operation, it is mandatory to check for the correctness of the erase procedure, by reading the entire sector with a gate voltage lower than that usually applied during normal reading. If the data read is "1", it means the threshold voltage of the cell has been lowered from the programmed value to the erased one and that, since this read operation has provided a correct result with a low reading voltage, the correct value is expected to be detected even in a conventional reading. If the verification fails for at least one cell, another erasing pulse is applied to the sector until all the cells are correctly erased or a maximum number of pulses has been reached, so that a fail signal is produced. After the erase verify procedure (see Fig. 2.22 d), it is important to check whether some cells are over-erased ("depletion verify") and, in case, their thresholds must be driven to the correct range ("soft programming"). The former operation must check whether some cells feature low or even negative threshold (depleted cells), so that they would draw current even if not biased, thus preventing from a correct reading of cells belonging to the same bit line. With the latter operation, these cells are written with suitable gate and drain voltages that, lower than those used during the normal program procedure, allow for a slight increase of the threshold voltage. The final threshold distribution is then bounded between $V_{T_{e_m}}$ and $V_{T_{e_M}}$ (see Fig. 2.22 e).

2.2.2.1.2 NAND Architecture

The elementary unit of a NAND architecture flash memory is not a single cell but a serial chains of more floating gate transistors connected to the bit line and ground through two selection transistors (Fig. 2.24). This organization permits to eliminate all contacts between WLs, reducing in this way the occupied area. The reduction of the matrix area, thanks to the scaling of the word line pitch, is the main advantage of this solution. This feature is possible because of:

- Decrease of the number of contacts;
- Scaled source and drain junctions with respect to the standard ETOX cell, made possible by the physical mechanism used for cell writing.

Selection transistors are biased to connect the chain to the bit line and isolate it from the ground. If the memory is organized in a NAND array, both program and erase mechanisms are electron tunneling. Since tunneling is more efficient than CHE injection, currents are smaller and different supply voltages can be internally generated by charge pumping circuits implemented on the same die. NAND array are preferred for high-density Flash memories.

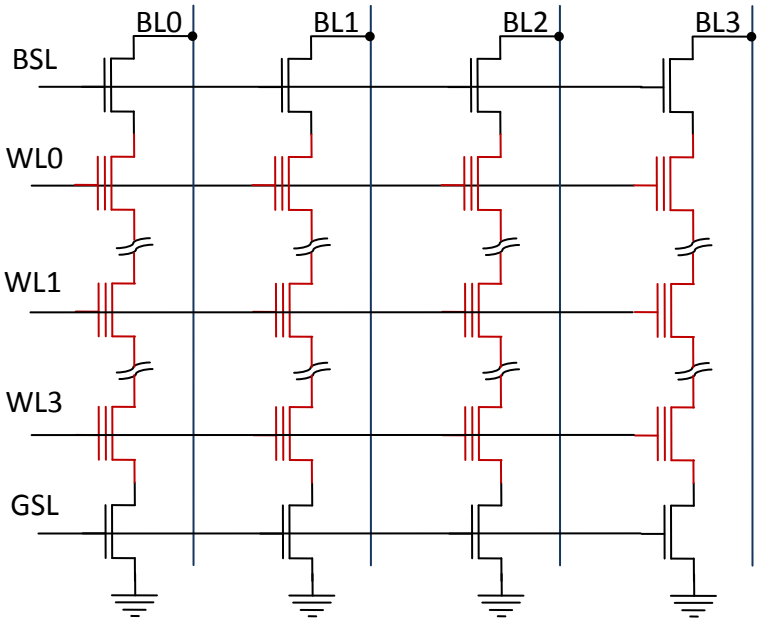

Fig. 2.24: NAND array architecture for flash memories.

During reading operation the selected cell has the control gate at 0V while the other cells in series are driven at high voltage, thus acting as ON pass gates independently of their actual thresholds. The current will flow along the series only if the selected transistor shows a negative threshold. Such current will be detected by the sense amplifiers present at the end of each bit line. So this operation becomes slow.

2.2.2.2 Reading Techniques

A reading operation in flash memories consists in sensing the current which flows among the bit line and comparing with a given references. If the current provided by the selected matrix cell is greater than the reference the content cell will be read as “1”, otherwise as “0”. The issue involves analog circuitry, in particular for comparing the currents.

The simplest sense circuit is show in figure Fig. 2.25 a. Transistors M_2 and M_3 comes from row and column decoders, $C_{BITLINE}$ is the parasitic capacitor of the bit line and it is due substantially to the drains of all floating gate devices of the bit line. At the beginning of the read procedure, the transistor M_1 charges the capacitor until about 1 V is reached, when the transistor is turned off by the negative feedback. The gate of the cell is biased to V_{DD}, so if the cell is erased (e.g. its

thresholds is 2 V) the capacitor will discharge through the floating gate device, lowering the voltage V_{OUT}. In the other hands, if the cell is programmed (threshold higher than V_{DD}) the capacitor will not discharge and the output node remains at V_{DD}. In a CMOS process the resistor denoted by "R" is replaced with a current generator. For a better comprehension of the circuit operation is useful keeping in mind the chart of the Fig. 2.7 which, as explained in 2.2.1, depicts the characteristic of the floating gate device when erased (read as "1") and programmed (read as "0").[2]

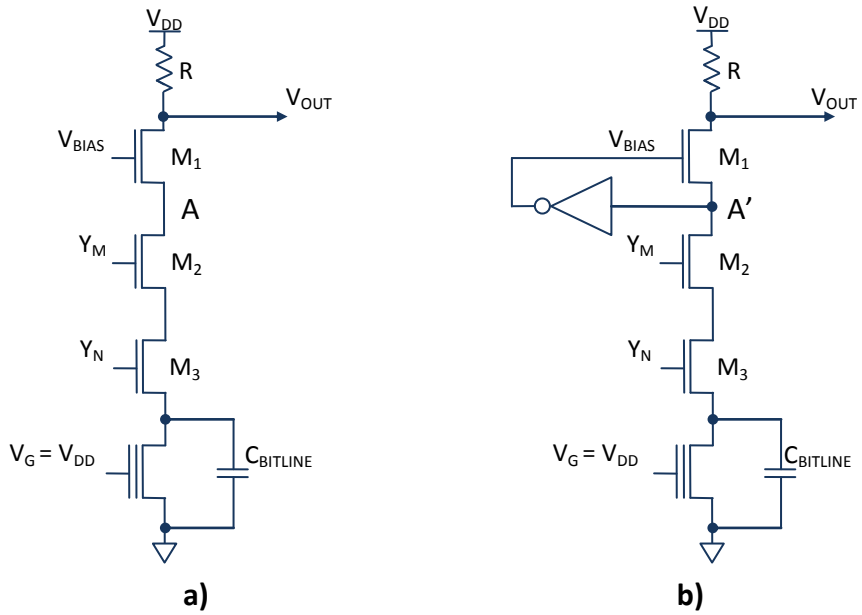

Fig. 2.25: Biasing configuration for read mode: a) arbitrary V bias; b) self-generated V bias.

Considering the current generator as ideal, two different situations can occur:

- The cell is erased: if it is over-erased (i.e. the threshold voltage is lower than a minimum value, V_{DD_m}) it will not be able to sink the current supplied by the ideal generator, so the output voltage will erroneously rise up at V_{DD}. If, instead, the threshold voltage is higher than the

[2] In this context the voltage at the output node will be low when the cell is erased and high when programmed. An output stage will invert the signal according to the previous definition of programmed and erased cell.

minimum value V_{DD_m}, the reference current will be completely sunk by the floating gate, keeping the output voltage low.

- The cell is programmed: if the threshold voltage of the floating gate device is greater than a maximum value V_{DD_M} the current supplied by the ideal generator will be completely sunk by the cell, so the output voltage will erroneously drop. If, instead the threshold voltage is less than the maximum value V_{DD_M}, the reference current will not be sunk by the cell and will charge the output capacitor increasing the output voltage.

Therefore, the threshold voltage for a correct sensing of the cell is $V_{DD_m} < V_{TH} < V_{DD_M}$. The right choice of the reference current should also satisfy dynamic constraints; it should be large enough to charge $C_{BITLINE}$ quickly, but not so high as to prevent a erased cell from pulling down the output node.

The way V_{BIAS} is generated has also an effect on the dynamic behavior of the circuit: if the gate voltage of M_1 is kept at a fixed value, when M_2 and M_3 are turned on, the V_{GS} of M_1 is equal to V_{BIAS}, since node "A" (Fig. 2.25(a)) is dynamically grounded, therefore the charging of the $C_{BITLINE}$ is performed through M_1 with a limited V_{GS}. To overcome this issue, the bias voltage network is replaced by an inverter as shown in Fig. 2.25 b. When the node A' is grounded, the bias voltage is at V_{DD} therefore the V_{GS} of M_1 is the maximum available, and the charging of $C_{BITLINE}$ is as fast as possible. Calibration of the threshold voltage of the inverter – by designing the nMOS aspect ratio greater than the pMOS one – is needed to guarantee the M_1 is turned off as soon as the A' node reaches the voltage V_{TH_n}, of about 1 V. The main drawback of this solution is the power consumption: feedback needs current to work properly.

The Fig. 2.26 depicts a differential current sensing. Its operation is based on comparing two voltages, V_R and V_M, which are proportional to the reference current and the current provided by the cell to be read respectively.

The floating gate denoted as M_R in the figure is a virgin cell, i.e. its threshold voltage is equal to an EPROM cell UV-erased, about 2 V; whereas the M_M is the floating gate device which makes up the cell to be read.

The circuit's structure is symmetrical, so M_4, M_5 and M_6 are identical to M_1, M_2 and M_3 respectively. They convert the current supplied by the floating gate devices M_R and M_M into the voltages V_R and V_M respectively.

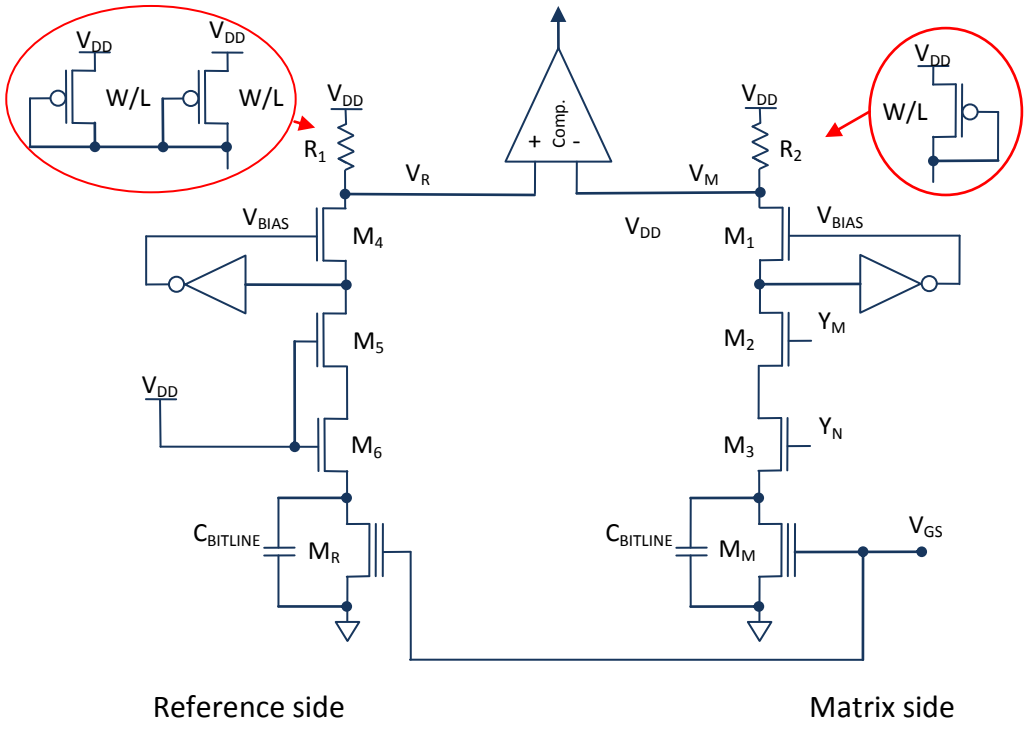

Fig. 2.26: Differential architecture.

If M_M is programmed, no current will be sunk in the matrix side and V_M is at V_{DD}, while M_R is active and V_R is pulled down: therefore a voltage difference exists, and the comparator switches its output. The problem with the structure arises when MM is virgin: of course, VR and VM have the same potential and the comparator is not able to decide what type of information it is reading. The parameters that can be changed to obtain a correct behavior in every conditions are the value of the loads, R_1 and R_2. V_R node should always be between V_{M_V} (the potential due to a virgin matrix cell) and V_{M_P} (the potential due to a programmed matrix cell), and it should always have the same value, chosen according to dynamic considerations, independently of the matrix side cell. The relation is therefore the following:

$$V_{M_V} < V_R < V_{M_P} \tag{2.20}$$

The upper limit is always satisfied, whereas two ways exist to obtain the lower one:

- Decreasing R_1, thus pulling up node V_R
- Increasing R_2, thus pulling down the node V_{M_V}.

In both cases, if the two branches sink the same current, node V_R is higher than V_{M_V} because voltage drop on R_1 is lower than on R_2 ($R_1 < R_2$). The optimal solution is the first one, since smaller load values mean higher current to charge $C_{BITLINE}$ quickly. As mentioned before, in a CMOS integrated process the resistor will be replaced by a transistor. Therefore R_1 and R_2 will be replaced by pMOS transistors in diode configuration as shown in Fig. 2.26, where R_1 is composed by two pMOS in parallel to form a dynamic resistance half than R_2.

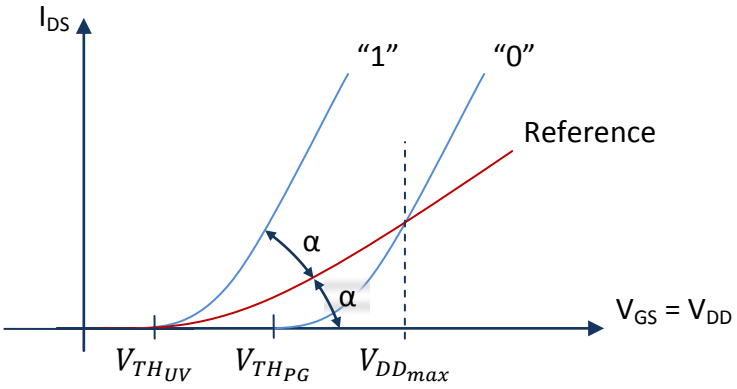

Fig. 2.27: Read with unbalanced loads.

Fig. 2.27 depicts both reference and matrix cell characteristic, assuming that MR and MM have the same size. The reference characteristic (red curve in Fig. 2.27) starts at the same point of the erased cell one, but with half angular coefficient. It is clear that if the cell characteristic is above to the reference one, the cell will be recognized as erased; vice versa the cell will be recognized as programmed. Supposing that the characteristics are straight lines, the following relation holds:

$$V_{DD_{max}} = V_{TH_{UV}} + \frac{n}{n-1}\Delta V_{TH} \tag{2.21}$$

$$V_{DD_{min}} = V_{TH_{UV}} \tag{2.22}$$

with n equals to the ratio between the size of the loads in the reference and in the matrix side. By choosing different values of n, it is possible to modify the current differences between the reference and the matrix sides thus giving more margin either to the virgin or to the programmed cell, as needed.

The main problem related to current-voltage converter with unbalanced loads is that $V_{DD_{max}}$ fixes a maximum supply voltage for the device operation; in fact, if V_{DD} becomes higher than $V_{DD_{max}}$, the read circuitry misinterprets a programmed

cell as erased. A typical solution to avoid this issue consists in realizing a reference whose characteristic is parallel to that of the matrix cells as shown in Fig. 2.28.

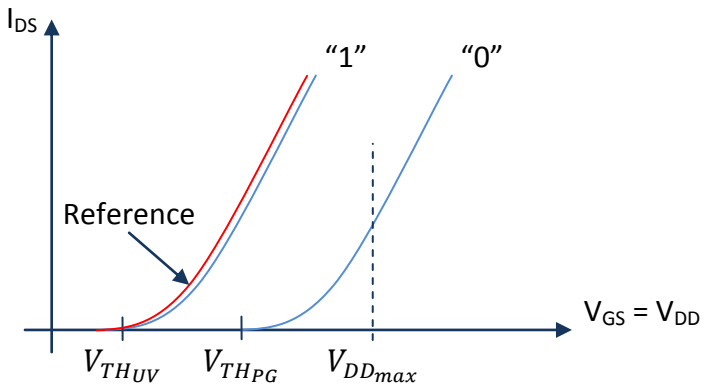

Fig. 2.28: Parallel characteristics approach.

By shifting rigidly the characteristics of both the programmed and erased cells upwards of a given offset current, as shown in Fig. 2.29, the reference characteristic will completely be on the right side of the erased one, whereas that of the programmed cell will be on the left side of the reference just up to a low voltage denoted as $V_{DD_{min}}$. Therefore for $V_{DD} > V_{DD_{min}}$ the reference characteristic is always separated from that of the matrix; solving the problem related to $V_{DD_{max}}$, but introducing a problem related to $V_{DD_{min}}$.

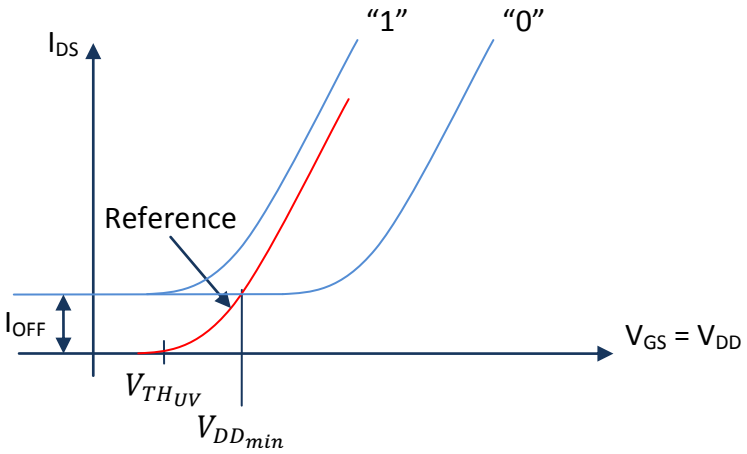

Fig. 2.29: Characteristic separation using offset current.

The offset current has to be chosen by keeping in mind the compromise between the $V_{DD_{min}}$ and the separation of the reference and erased cell characteristics. Fig. 2.30 shows the circuitry used to obtain these characteristics: the

voltage generator consists in a network based on a mirror current fed by a current generator.

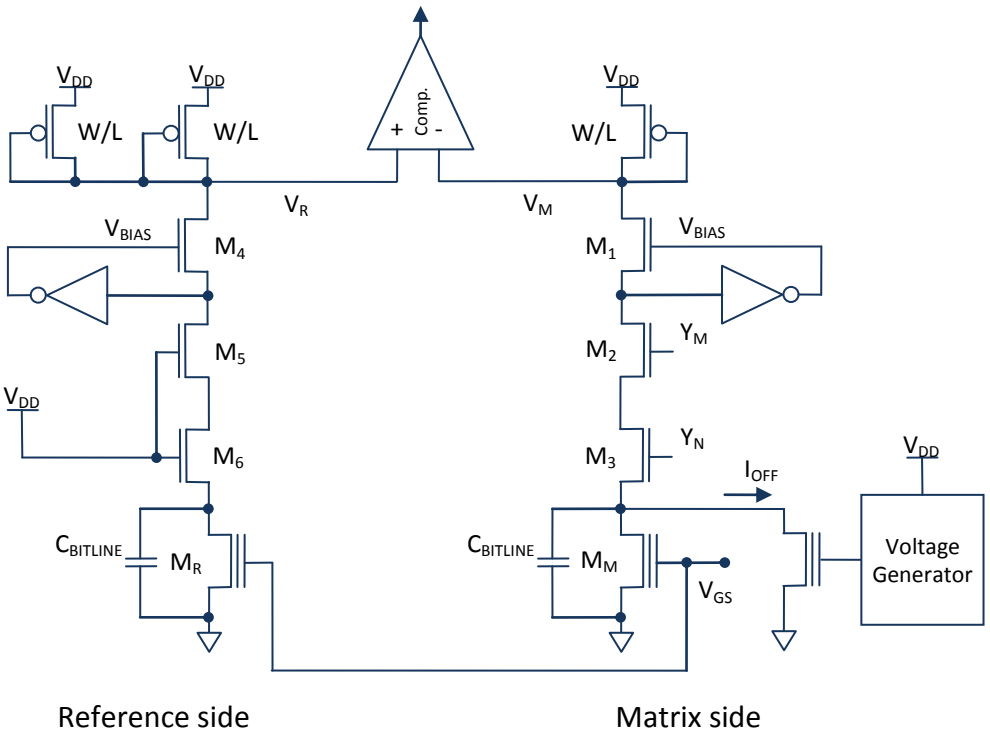

Fig. 2.30: Sense amplifier with current offset configuration.

This configuration is used in many devices obtaining good results even if paying particular attention to the shrink influence on the offset current value. Different structures exist in literature such as “Differential Semi-Parallel Sensing Technique” [***35***] and “Current Mode Analog Sense Amplifier” [***36***].

2.3 Conclusions

In this chapter the issues related to the flash memories have been introduced. As discussed, they involve both digital and analogue themes. Many of them have been neglected, such as the generation of the voltages needed to either program or erase the cells’ content. Flash memories result one of the most complex units embedded in a microcontroller for both the generation and handling of the analogue signals, and the treatment of the disturbances in either reading, programming or erasing. The complexity of the flash memory makes the unit which contains the flash memory the largest in terms of area occupied in a microcontroller. For this reason a great effort has to be spent by the manufacturers to guarantee the flash operation both in terms of reliability and durability.

Chapter 3 TriCore® TC27x microcontroller

One of the aims of this thesis is the development a simulator of the Program Memory Unit (PMU) included in the Infineon® "TriCore Aurix TC27x" microcontrollers. This simulator will be interfaced with the existing CPU simulator so that it is possible to have a complete simulator of the whole microcontroller. The goal of the user is to test the correct execution of the firmware application. In particular, for the Infineon's Padua team, it is to test the firmware applications, which perform operations on flash memory unit.

In this chapter, a brief presentation about the microcontroller will be given, focusing on the Program Memory Unit.

3.1 TriCore® Aurix Device

The Infineon TriCore® Aurix device is a 32 bit microcontroller DSP based on the Infineon TriCore architecture. It combines three powerful technologies within one silicon device, achieving new levels of power, speed and economy for embedded applications:

- Reduced Instruction Set Computing (RISC) processor architecture;
- Digital Signal Processing (DSP) operations and addressing modes;
- On-chip memories and peripherals.

DSP operations and addressing modes provide the computational power necessary to efficiently analyze complex real-world signals. The RISC load/store architecture provides high computational bandwidth with low system cost. On-chip memory and peripherals are designed to support even the most demanding high-bandwidth real-time embedded control system tasks.

The TC27x is a high-performance microcontroller with three TriCore CPUs, program and data memories, buses, bus arbitration, interrupts system, DMA controller and a powerful set of on-chip peripherals, such as serial controllers, timer units, and analog-to-digital converters. Within the TC27x, all these peripheral units

are connected to the TriCore CPUs / system via the System Peripheral Bus (SPB) and the Local Memory Bus (SRI). A number of I/O lines on the TC27x ports are reserved for these peripheral units to communicate with the external world. A block diagram is shown in Fig. 3.1.

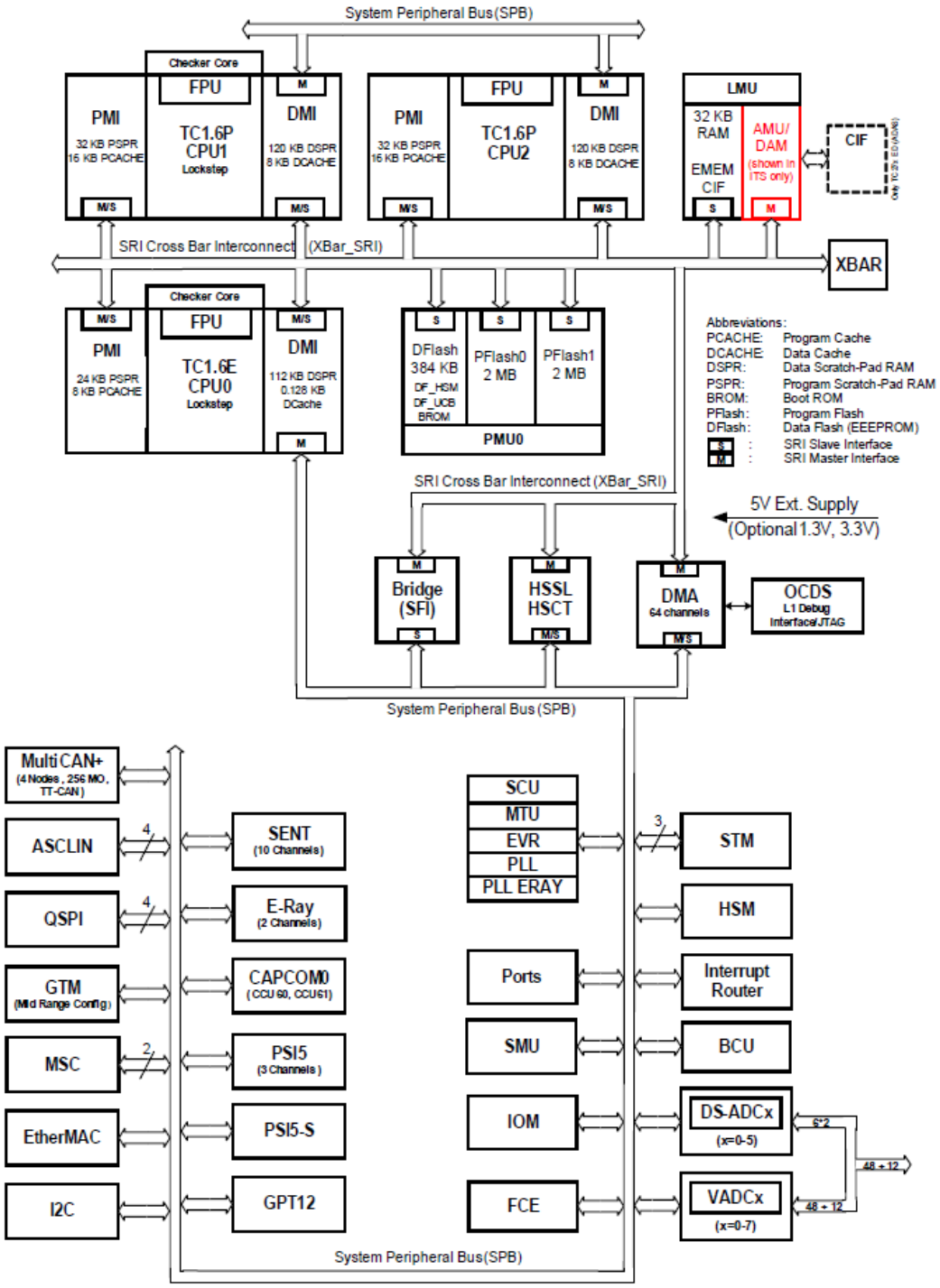

Fig. 3.1: TC27x block diagram.

The TC27x has two independent on-chip buses:

- Shared Resource Interconnect (SRI)
- System Peripheral Bus (SPB)

The SRI connects the TriCore CPUs, the high bandwidth peripherals and the DMA module to its local resources for instruction fetches and data accesses. Further details about this bus will be treated afterward inasmuch it connects the CPUs with the Program Memory Unit which is the subject of this thesis.

The System Peripheral Bus connects the TriCore CPUs, the high bandwidth peripherals and the DMA module to the medium and low bandwidth peripherals.

3.2 Program Memory Unit

The Program Memory Unit is the module which includes the flash memory. It is connected to other microcontroller's peripherals through the "Shared Resources Interconnected" (SRI) bus.

In Infineon® microcontrollers the silicon area is mainly taken up by the Program Memory Unit, therefore it is subjected to the highest probability of execution failure than any other peripheral. Thus great efforts are made to test, characterize and validate the PMU in order to reduce as much as possible the probability of an error.

The Program Memory Unit, sometimes also referred as Flash Block, can be thought as a finite state machine which interfaces the actual flash memory with all other microcontroller parts. The Fig. 3.1 depicts some functional device blocks, including the PMU. The most important components are the three CPUs denoted by CPU0, CPU1 and CPU2. These CPUs execute the instructions which are stored into the program flash, involving all the needed subsystems via SRI bus.

The Program Memory Unit contains some sub circuitries which handle the local memory, as shown in Fig. 3.2. The local memory presents the following features:

- 2 MBytes of program flash memory (PFlash)
- 128 kBytes of data flash memory (DFlash)
- 16 kBytes of boot ROM (8 kB boot code, 8 kB factory test routines)
- Emulation memory interface for access to ECC SRAM in emulation device.

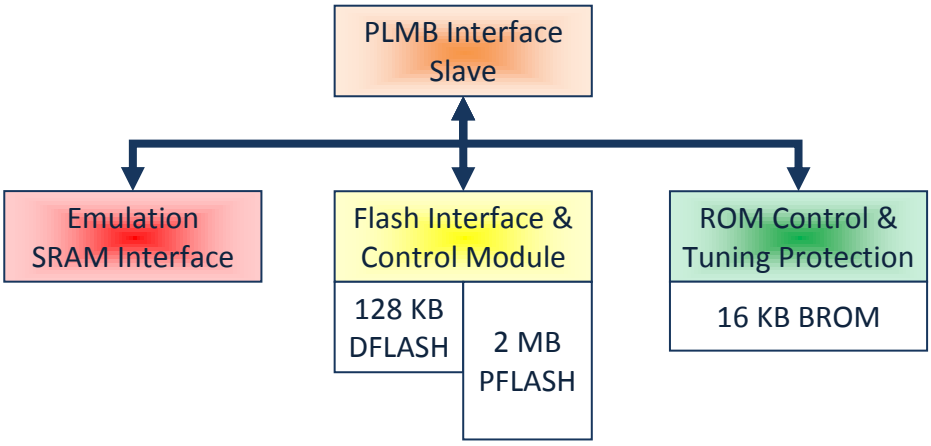

Fig. 3.2: Flash arrays inside the Program Memory Unit.

The whole flash block is composed by three modules as shown in Fig. 3.3:

- Flash Array Module (FAM): contains cell arrays with all analogue and digital blocks needed to handle it.
- Flash Interface and Control Module (FIM): provides an interface between the PMU and the remaining parts of microcontroller.
- Flash Standard Interface (FSI): manages the flash operations. It is the most important PMU's part as it manages all the operations to and from the Program Memory Unit.

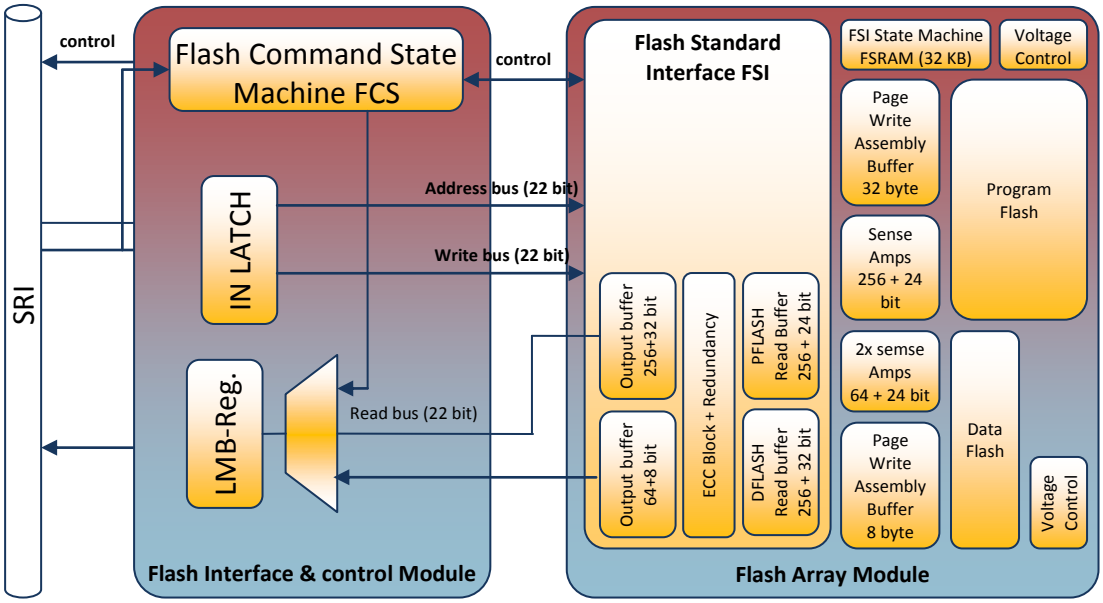

Fig. 3.3: Program Memory Unit's sub circuitries.

Fig. 3.3 depicts all the PMU's sub modules: data and program instructions are stored into Data Flash and Program Flash respectively. Some sense amplifiers and voltage controls handle analogue signals needed to write and erase flash arrays. A detailed description of the other parts will be given afterwards.

Since TC27x microcontroller is designed for automotive purpose, all data are stored in flash array with some ECC and redundancy bits for safety reasons, therefore "ECC and Redundancy" block exists to calculate these fields.

The simulator developed in this thesis focuses on the Data and Program Flash memory; whereas the boot ROM is not considered (it could be a next improvement). For the "Flash Interface and Control Module" and "Flash Standard Interface" just behavioral aspects are implemented.

3.2.1 Pump Voltage

Reduction of the device power supply voltage required the inclusion of an additional internal circuitry to obtain all the voltages necessary for the various operations such as erase and program. At the same time, the decrease of the device size emphasized the problems related to stress and, as a consequence, the sector organization had to be redesigned.

The first generation of a single supply System-on-Chip with Flash memory with a V_{DD} of 5V was created by Infineon in 2001. In the case of single supply, the voltages greater than V_{DD} are produced on-chip by charge pumps. The current-voltage characteristic of a charge pump realized with diodes and capacitors can be approximated to a line, the slope of which represents the pump output resistance R_{OUT}. The R_{OUT} value amounts to tens of KΩ, whereas the maximum current supplied amounts to mA.

3.2.2 Sense Amplifier

Another analog component of the Flash system is the sense amplifier connected to the bit lines for reading the corresponding bit.

The sense amplifier is a comparator. In order to identify the information in the cell, the current is converted into voltage and the voltage of the cell addressed is compared with the reference cell by a differential amplifier. In this section the sense amplifier embedded in the Infineon microcontrollers will not be treated. General information about this task was provided in 2.2.2.2.

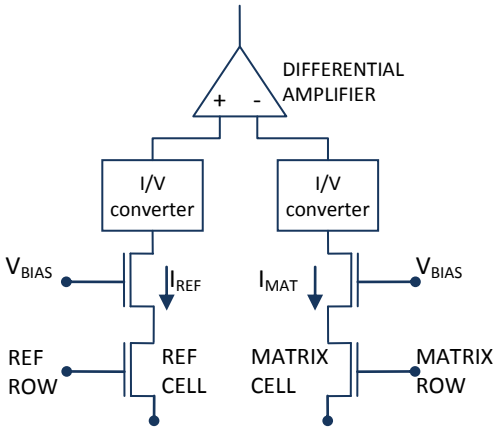

Fig. 3.4: Sense amplifier basic block diagram.

3.2.3 Flash Structure

The flash memory is divided into different banks, physical sub-sectors and logical sub-sectors respectively.

The flash module contains separate banks: one set for the program flash (PF0, PF1 and so on) and another one for data flash (DF0, DF1 and so on). This division exists to support concurrent operations, although some existing limitations due to the common logic.

A flash bank is further divided into physical sub-sectors, which are independent memory areas. The sub-sectors are separated into logical sectors, which can be thought as groups of word-lines which share the same bit line.

A logical sector is the smallest erasable memory unit, but after a bit of erasure the whole physical sub-sector has to be erased to avoid problems due to over-erased and / or soft erased cells.

3.2.3.1 Program Flash Structure

Microcontrollers belonging to the TC27x family can have different size of the Program Flash banks, all based on the same sectorization:

- 2 MBytes bank implements logical sectors S0 to S26;
- 1.5 MBytes bank implements logical sectors S0 to S24;
- 1 MBytes bank implements logical sectors S0 to S22.

The Fig. 3.5 shows the sectorization for a 2 MBytes program flash bank which is divided into four physical sub-sectors and more logical sectors.

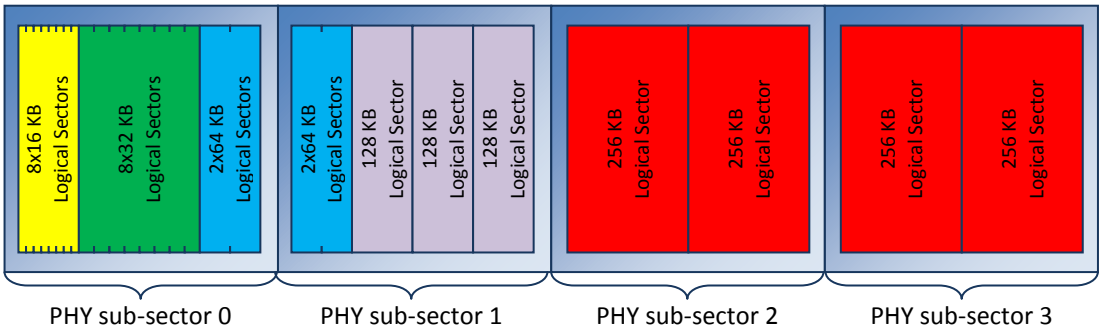

Fig. 3.5: Program Flash bank partition.

A TC27x microcontroller can have at most two program flash banks (PF0 and PF1).

In Program Flash, a word-line is an aligned group of 512 kBytes. It is divided into sixteen pages which are the smallest units that can be programmed. A page is an aligned group of 32 bytes. Fig. 3.6 gives a graphic word-line description where the pages are represented by rectangles.

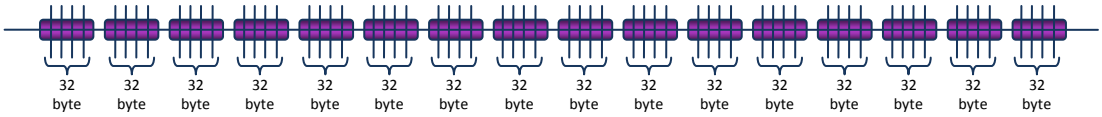

Fig. 3.6: Graphic representation of a word-line.

Logical sectors are groups of word-lines that share the same bit lines. Different physical sub-sectors have independent word-lines and bit-lines. The sub-sectorizations of the physical sub-sectors are represented in figures Fig. 3.7, Fig. 3.8 and Fig. 3.9.

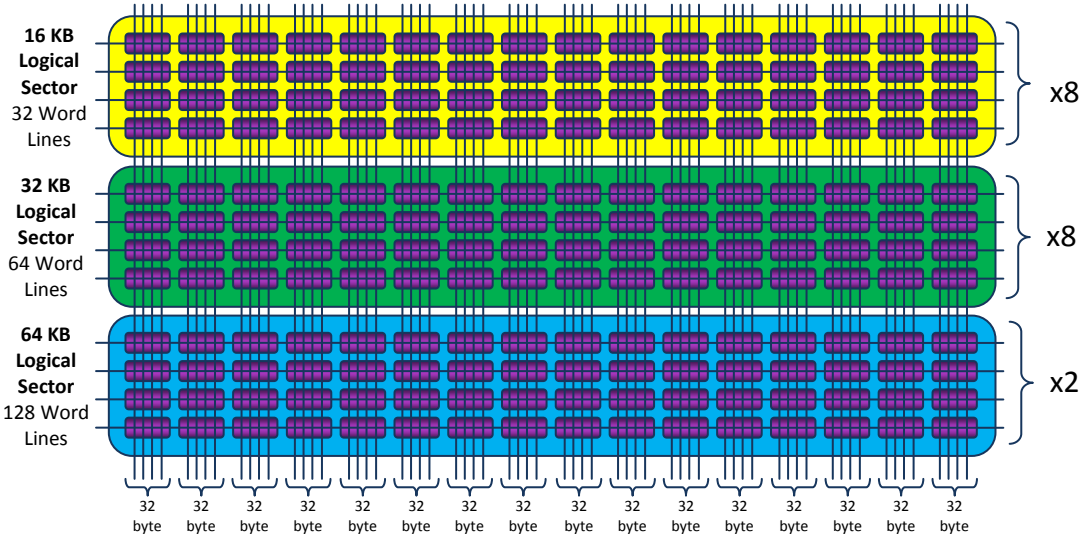

Fig. 3.7: Graphic representation of logical sectors belonging to the first program flash physical sub-sector.

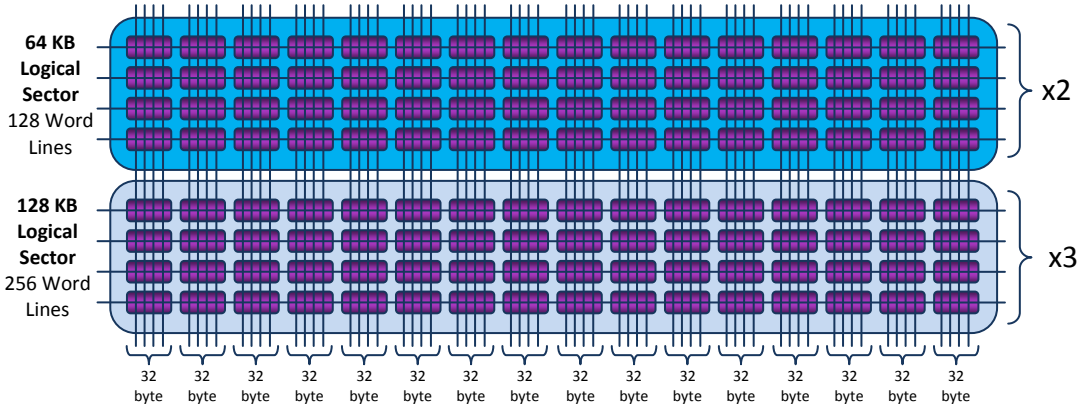

Fig. 3.8: Graphic representation of logical sectors belonging to the second program flash physical sub-sector.

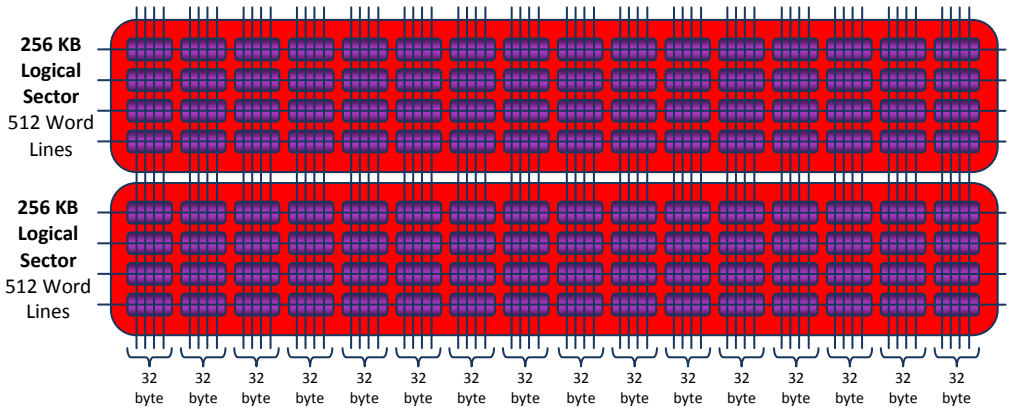

Fig. 3.9: Graphic representation of logical sectors belonging to the third and the fourth flash physical sub-sectors.

3.2.3.2 Data Flash Structure

The general structure of the 1 MByte Data Flash is chosen to support the emulation of an EEPROM, e.g. 384 kByte EEPROM. In this context the main difference between a flash memory and an EEPROM is the endurance. For EEPROM, an endurance of e.g. 120.000 write/erase cycles is required, that is not supported with the standard flash memory.

For EEPROM emulation and thus for increasing the endurance, the data flash is used like a circular stack-memory: the newest data updates are programmed always on top of the actual region. When the top of sector is reached, all actual data (representing the EEPROM data) are copied to the bottom area of the next sector and the last sector is then erased. This Round robin procedure, using multi-fold replication as the emulated EEPROM size, significantly increases the endurance. This method will allow emulating at least 120.000 write/erase cycles to a 16 kByte Flash-EEPROM by programming each virtual 16 kByte block only 1/8 of the total number of endurance (15.000 for Data Flash with retention of 5 years).

The simulator will ignore this method because it is developed with the intent to execute a firmware application written for the purpose of testing the flash array.

3.2.4 Flash Standard Interface

When the CPU has to perform a flash operation, it has to send some appropriate commands to the Program Memory Unit which will take charge of the execution. These commands are called “Command Sequences” and are computed by the most important sub-module into the PMU: the Flash Standard Interface (FSI).

The FSI is an 8 bit RISC processor, the firmware of which, called micro-code, is designed to define and execute these specific command sequences. The FSI has been implemented mainly to get the possibility of changing some parameters after the end of the silicon process. This achievement is very important because many differences exist between chips made in regions far apart from each other of the same wafer. These mismatches result in different performances and / or features, in particular for analogue sub circuits, which can be adjusted by the microcode. Some minor reasons exist such as the possibility of changing the command sequence definitions.

The FSI is interfaced with the remaining PMU’s parts by Special Function Registers (SFRs) which could be thought as a kind of Application Program Interfaces (APIs). So SFRs plus microcode allow parameterization of erase and / or the programming algorithms.

In the simulator object of this thesis, the FSI is replaced with an algorithm implementing a finite state machine, which means that only the behavior of the microcode is replicated.

3.2.5 Register Set

The Program Memory Unit contains a set of registers which control all functionalities of the PMU itself and the flash arrays.

Some microcontroller belonging to the TC27x family is realized with reduced functionalities, removing some data flash sub-sectors and / or its functionalities. Anyway, registers which control these functionalities are implemented, but the contained values have no effect on the operation of the Program Memory Unit.

This section will provide a brief introduction to register set, focusing on the most important ones to handle the Program Memory Unit. For a complete description of all PMU's registers, please refer to [***37***]. Note that not all registers have been implemented in the simulator object of this thesis. Many of them, could be introduced as next improvements.

The register access conditions use the following abbreviations:

- **"U"**: Access permitted in User Mode "0" or "1" (applicable to write and read).
- **"SV"**: Access permitted in Supervisor Mode (applicable to write and read).
- **"E"**: ENDINIT protected write. "E" means a write access is only allowed before ENDINIT or after disabling this protection with a password as described in the SCU chapter.
- **"SE"**: Safety ENDINIT protected write.
- **"P"**: The access is controlled by the master specific register access protection
- **"–"** or **"BE"**: Access not permitted.

3.2.5.1 Flash Status Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CFU	ORIER	VIS	SLM	SPND	EVER	PVER	REC	0	SRIADDERR	DFMBER	DFTBER	DFDBER	DFSBER	RES17	PFMBER
rh	rh	rh	rh	rwh	rwh	rwh	rwh	r	rwh	rwh	rwh	rwh	rwh	rwh	rwh
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFDBER	PFSBER	PROER	SQER	OPER	DFPAGE	PFPAGE	ERASE	PROG	RES6	RES5	P1BUSY	P0BUSY	D1BUSY	D0BUSY	FABUSY
rwh	rwh	rwh	rwh	rwh	rh	rh	rwh	rwh	rh	rh	rh	rh	rh	rh	rh

Fig. 3.10: Flash Status Register definition.

The most important Flash register is Flash Status Register, the definition of which is reported Fig. 3.10.

In this context just some field explanations are provided:

- D0BUSY / D1BUSY indicate the data flash 0 / 1 bank is busy: a flash operation such as erase or programming is running.
- P0BUSY / P1BUSY indicate the program flash 0 / 1 bank is busy: a flash operation such as erase or programming is running.
- PROG / ERASE: a program/erase operation is running or has been completed.
- PFPAGE / DFPAGE: Program / Data Flash bank is in page mode.
- OPER: an error has occurred during an operation.
- SQER: some errors regarding command sequence have occurred.

3.2.5.2 Flash Configuration and Control Register

The Flash Configuration Register FCON reflects and controls the general Flash configuration functions.

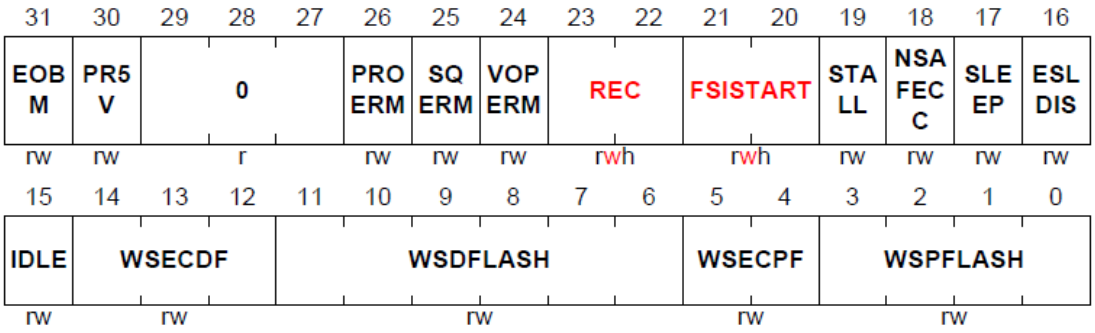

Fig. 3.11: Flash Configuration and Control Register definition.

- WSPFLASH / WSDFLASH: defines the number of wait states in number of FSI2 clock cycles, which are used for an initial read access to the Program / Data Flash memory area.
- WSECPF / WSECDF: defines the number of wait-states for the ECC correction of Program / Data Flash in terms of FSI2 clock cycles.
- SLEEP: defines if the Flash Module is sleeping or is active.
- STALL: This field selects if reading from busy Flash banks causes a bus error or wait-states until busy is cleared again.
    - **Stall**, Reading busy Flash banks suppresses the SRI bus ready effectively causing wait states until busy is cleared.
    - **Error**, Reading busy Flash banks causes a bus error.

3.2.6 PMU Operations

Since erasing and writing flash memory involve analogue circuits and the duration is longer than a simple RAM access, the dynamic of flash operation is different than a simple register access. Therefore, all flash operations, except reading, are performed through command sequences handled by the FSI. In other words, compliance with certain rules is needed to compute flash operations. The whole set of these rules makes it seem the flash module as a complex finite state machine, inside which other finite state machines exist to run single operations.

Fig. 3.12 depicts the possible states of the flash module and the possible modes of the flash bank.

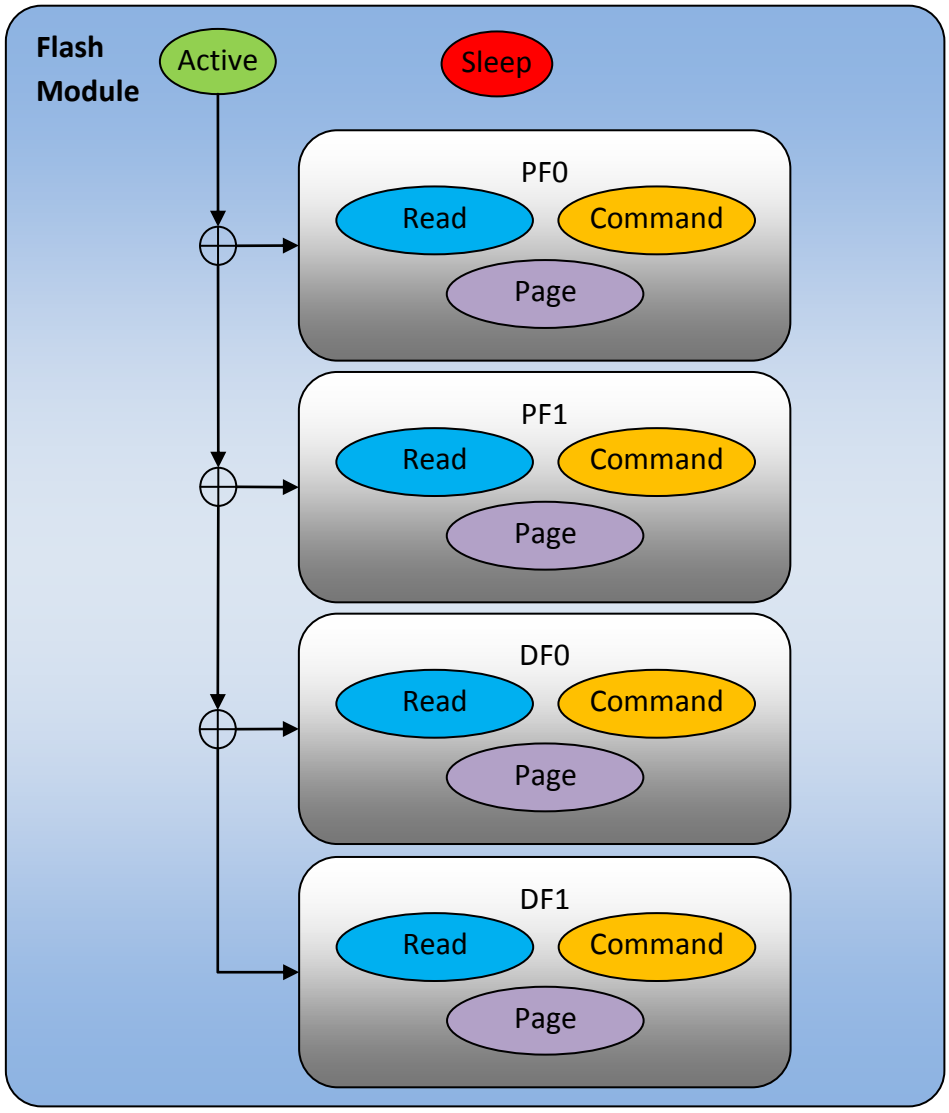

Fig. 3.12: Flash Module and Flash Bank states.

A Flash Module can be in one of the following state:

- Active (normal) state.
- Sleep.

In sleep mode, write and read accesses to all flash ranges of the Program Memory Unit are refused with bus error.

When the flash module is in normal mode a flash bank can be in one of the following modes:

- Read mode.
- Page mode.
- Command mode.

In order to either erase or write a flash location, the flash module which contains the location has to be in “Active State”, and the related flash bank has to be in “Command Mode”. To clarify, “state” and “mode” words are referred to Flash Module and Flash Bank respectively.

In read mode a Flash bank can be read and command sequences are interpreted. In read mode a Flash bank can additionally enter in page mode which enables it to receive data for programming.

In command mode an operation is performed involving all circuitry needed. During this time, the Flash bank reports BUSY on the Flash Status Register and it does not allow any access. In particular, a read access to a busy flash bank can be either refused with a bus error or put on hold, according to the “Flash Configuration and Control Register”. At the end of an operation the Flash bank returns to read mode, and BUSY is cleared. Only operations with a significant duration (i.e. all write and erase operations) set BUSY.

In command mode, further command sequences in this flash module are not allowed, but refused with a bus error.

3.2.6.1 Command Sequences

A command sequence consists in more write accesses to Data Flash 0 bank which are called “command cycles”. Every write access to DF0 memory range is interpreted as command cycle belonging to a command sequence. On the other hand, write accesses to flash location outside the DF0 memory range will be refused with a bus error.

Whenever a write access to DF0 memory range is executed, the Flash Standard Interface will run an algorithm in the microcode called “Command Interpreter” similar to a finite state machine which deals the identification of command sequence. The command interpreter checks whether a command cycle is correct in the current state of command interpretation, otherwise an error is reported by the SQER bit on Flash Status Register.

When the command sequence is accepted the last command cycle finishes read mode and the Flash bank transitions into either command mode or page mode, depending on the given command sequence.

As mentioned before, the command cycle is a write access addressed to the data flash bank 0, therefore two parameters are needed:

- The Address (32 bit)
- The Data (64 bit)

These values are provided by the CPU via the SRI bus. The signals width is disposed by the SRI bus though not all bits of Data signal are used. In fact just one command sequence, the “Load Page” (see 3.2.6.3), makes use of whole signal.

3.2.6.2 Reset to Read

In general, a command sequence is longer than one command cycle. When the command interpreter has detected some command cycles belonging to a command sequence, just three situations can occur:

- Command interpretation fails because incorrect command cycle has been given, in this case sequence error will be reported on SQER bit of Flash Status Register;
- Command sequence is accepted as soon as the last command cycle has been given, in this case the mode of the flash bank will change in read mode (or page mode if the command given is Enter in Page Mode);
- Command interpretation is reset because the “Reset to Read” command has been given, in this case the bank mode will change in read mode and the first command cycle of any command sequence is expected.

The “Reset to Read” command sequence resets all sequence errors clearing all error bits in the Flash Status Register; therefore the “Reset to Read” must be detectable in any state of the command interpreter.

“Reset to Read” definition is reported in the Tab. 3.1.

Address	Data
0xAF00_5554	0xXXF0

Tab. 3.1: "Reset To Read" definition.

3.2.6.3 Enter in Page mode and Load Page

To execute a programming operation, data have to be loaded in a dedicated SRAM called “Assembly Buffer” before they are stored in the flash memory. There are two assembly buffers: one for Program Flash and the other for Data Flash. The flash bank where the data is written has to set the assembly buffer in page mode to allow the loading of the data into the related assembly buffer.

To change assembly buffer in page mode from read mode the “Enter in Page Mode” command sequence has to be given. This is one-cycle command sequence the definition of which is reported in table Tab. 3.2.

Address	Data
0xAF00_5554	0xXX5”y”

Tab. 3.2: "Enter in Page Mode" definition.

The parameter “y” indicates which of the two assembly buffers, either program flash or data flash, has to enter in page mode:

- Y = 0x0: program flash assembly buffer enters in page mode;
- Y = 0xD: data flash assembly buffer enters in page mode.

When one of the two assembly buffers is in page mode the related PFPAGE / DFPAGE bit of Flash Status Register is set depending on program and data flash.

Just one assembly buffer can be in page mode. If the “Enter in Page Mode” is given when any buffer is still in page mode a sequence error will be reported on the SQER bit of Flash Status Register, and the existed page mode will be aborted.

After the assembly buffer enters in page mode, data can be loaded through the “Load Page” command sequence. This command sequence loads either 64 bit or 32 bit data on the assembly buffer in page mode. Its definition is reported in table Tab. 3.3.

Address	Data
0xAF00_55Fy	“WD”

Tab. 3.3: "Load Page" definition.

Where “WD” is the data to load on assembly buffer, and the parameter “y” indicates its width.

The width of the assembly buffers depends on the microcontroller. For the most popular TC27x microcontrollers, the program flash assembly buffer is in-depth 256 byte while the data flash 32 byte. To fill the whole buffer, more “Load Pages” have to be executed, all with the same data width. Otherwise, a sequence error will be reported.

If “Load Page” is called more often than allowed by the available buffer space, the overflow data is discarded, but the page mode is not left. This overflow will be reported by the following Write Page/Burst command with SQER.

3.2.6.4 Write Page, Write Page Once and Write Burst

When the assembly buffer is full, three command sequences are available to store the data in the flash memory:

- “Write Page”: transfers one page (32 byte for program flash, 4 byte for data flash) from the assembly buffer to the flash memory.
- “Write Page Once”: it is the same of “Write Page” with the only difference that it checks if the related page is erased. If not the command will fail with PVER and EVER bits set on the Flash Status Register.
- “Write Burst”: transfers a burst of page (8 pages: 256 byte for program flash, 4 pages: 32 byte for data flash) from the assembly buffer to the related flash memory.

These command sequences contain an argument which indicates the address location on the flash memory whereon the data are stored. In order to correctly execute the command sequence, the following situation must occur:

- The destination address has to point to the location within the flash bank range;
- The address has to be a page start address;
- The flash bank containing the address has to be in page mode.

If just one of these conditions is not true, the command will fail with a sequence error.

Since the programming operation takes a bit of time, the BUSY bit, related to the interested bank, is set on the Flash Status Register until the operation ends.

If just one of the following conditions occurs, the command sequence will be performed completely, although a sequence error will be reported:

- The related assembly buffer is not completely full;
- The related assembly buffer is overflowed.

The command sequence definitions are reported in tables Tab. 3.4, Tab. 3.5 and Tab. 3.6:

Address	Data
0xAF00_AA50	“PA”
0xAF00_AA58	0xXX00
0xAF00_AAA8	0xXXA0
0xAF00_AAA8	0xXXAA

Tab. 3.4: "Write Page" definition.

Address	Data
0xAF00_AA50	“PA”
0xAF00_AA58	0xXX00
0xAF00_AAA8	0xXXA0
0xAF00_AAA8	0xXX9A

Tab. 3.5: "Write Page Once" definition.

Address	Data
0xAF00_AA50	“PA”
0xAF00_AA58	0xXX00
0xAF00_AAA8	0xXXA0
0xAF00_AAA8	0xXX7A

Tab. 3.6: "Write Burst" definition

When an Erase operation begins, the PROG bit on Flash Status Register will be asserted. This bit will remain set after the end of the operation. There are two ways to clear it:

- Writing directly the Flash Status Register;
- Performing the “Clear Status” command sequence (see 3.2.6.7).

3.2.6.5 Erase Logical Sector Range and Erase Physical Sectors

As mentioned in “3.2.3 Flash Structure” the smallest erasable unit is the logical sector, but the whole physical sector can be erased as well. Microcode provides two dedicated command sequences to perform these operations:

- “Erase Logical Sector Range”: erases some consecutive logical sectors;
- “Erase Logical Sectors”: erases some consecutive physical sectors.

The command sequences provide two arguments to the FSI needed to perform the operations:

- The address of the first sector has to be erased;
- The number of sectors which have to be erased.

The command will fail with SQER set on the Flash Status Register if the address argument does not point to the base address of a correct sector.

The “Erase Logical Sector Range” command sequence definition is reported in table Tab. 3.7.

Address	Data
0xAF00_AA50	“SA”
0xAF00_AA58	0xXX”nn”
0xAF00_AAA8	0xXX80
0xAF00_AAA8	0xXX50

Tab. 3.7: "Erase Logical Sector Range" definition.

The “Erase Physical Sectors” command sequence definition is reported in table Tab. 3.8.

Address	Data
0xAF00_AA50	“SA”
0xAF00_AA58	0xXX”nn”
0xAF00_AAA8	0xXX80
0xAF00_AAA8	0xXX5A

Tab. 3.8: "Erase Physical Sectors" definition.

When an Erase operation begins, the ERASE bit on the Flash Status Register will be asserted. This bit will remain set after the operation ends. There are two ways to clear it:

- Writing directly the Flash Status Register;
- Performing the “Clear Status” (see 3.2.6.7) command sequence.

3.2.6.6 Verify Erased Logical Sector Range

After the “Erase Logical Sector Range” command sequence has been performed the “Verify Erased Logical Sector Range” command can be run to check if the erasing operation has been correctly executed. If not, the flag EVER on Flash Status Register will be asserted.

Even if “Erase Logical Sector” has not been executed before, this command sequence can be performed.

Address	Data
0xAF00_AA50	“SA”
0xAF00_AA58	0xXX”nn”
0xAF00_AAA8	0xXX80
0xAF00_AAA8	0xXX5F

Tab. 3.9: "Verify Erased Logical Sector Range" definition.

3.2.6.7 Clear Status

The flags FSR.PROG and FSR.ERASE and the error flags of the Flash Status Register (OPER, SQER, PROER, PFSBER, PFDBER, PFMBER, DFSBER, DFDBER, DFTBER, DFMBER, ORIER, PVER, EVER) are cleared. These flags can also be cleared in the status registers without any command sequences.

Address	Data
0xAF00_5554	0xXXFA

Tab. 3.10: "Clear Status" definition.

3.2.6.8 Operation Suspend and Resume

The following operations can be suspended.

- “Erase Logical Sector Range”
- “Erase Physical Sectors”
- “Verify Erased Logical Sector Range”
- “Write Page”
- “Write Burst”

These operations take a bit of time so they set the related BUSY bit on Flash Status Register and can be suspended by asserting SPND bit on MARD register.

After the MARD.SPND assertion, the FSI will check if one of these operations is still running and in this case it waits until that command sequence reaches a suspendable state; otherwise the request will be ignored, clearing the MARD.SPND bit.

When an operation has been suspended, the SPND bit of MARD register will be automatically cleared and the SPND bit of Flash Status Register will be asserted. Also the BUSY bit on the Flash Status Register will be cleared because de facto, flash bank is not busy anymore. The Program Memory Unit will be able to execute other command sequences on different flash banks. When these have been completed, the suspended operation will be resumable giving the “Resume Prog Erase” command, the definition of which is reported on table Tab. 3.11.

Address	Data
0xAF00_AA50	“PA/SA”
0xAF00_AA58	0xXX”nn”
0xAF00_AAA8	0xXX70
0xAF00_AAA8	0xXXCC

Tab. 3.11: "Resume Prog Erase" definition.

This command sequence accepts two arguments which have to match with the suspended command ones, otherwise the sequence error will be asserted and the command will remain suspended.

If the “Resume Prog Erase” command is accepted, the SPND bit of the Flash Status Register will be cleared and the related BUSY bit will be re-asserted.

In the suspended programming state:

- Reading Flash is allowed.
- New programming and erase commands are rejected with a sequence error.

In the suspended erase state:

- Reading Flash is allowed.
- New erase commands are rejected with SQER.
- Programming commands can be performed on any bank.
- However programming in the target range of the suspended erase fails with SQER.
- Suspending a programming command is not possible and fails by setting MARD.SPNDERR.

3.3 Shared Resource Interconnect bus

The “Shared Resource Interconnect” is the high speed system bus for TriCore1.6x CPU based devices. The central module of the interconnection is the XBar_SRI which connects all the components in one SRI system.

The Shared Resource Interconnection Bus (SRI) is a synchronous, pipelined bus with variable block size transfer support. All signals are related to the positive clock edge.

The protocol supports 8, 16, 32 and 64 bits single beat transactions and block beat transactions with a length of 2 or 4.

The central block of the SRI-Bus implementation is the crossbar (XBar_SRI) managing and monitoring the whole SRI-Bus action.

The elements which put together the SRI system are:

- The crossbar (XBar_SRI);
- The master interface (MIF_SRI);
- The slave interface (SIF_SRI).

The Program Memory Unit is interfaced with the SRI bus via three slave interfaces:

- SCI6: for both of Data Flash banks, register set and BootROM;
- SCI7: for Program Flash bank 0;

- SCI8: for Program Flash bank 1.

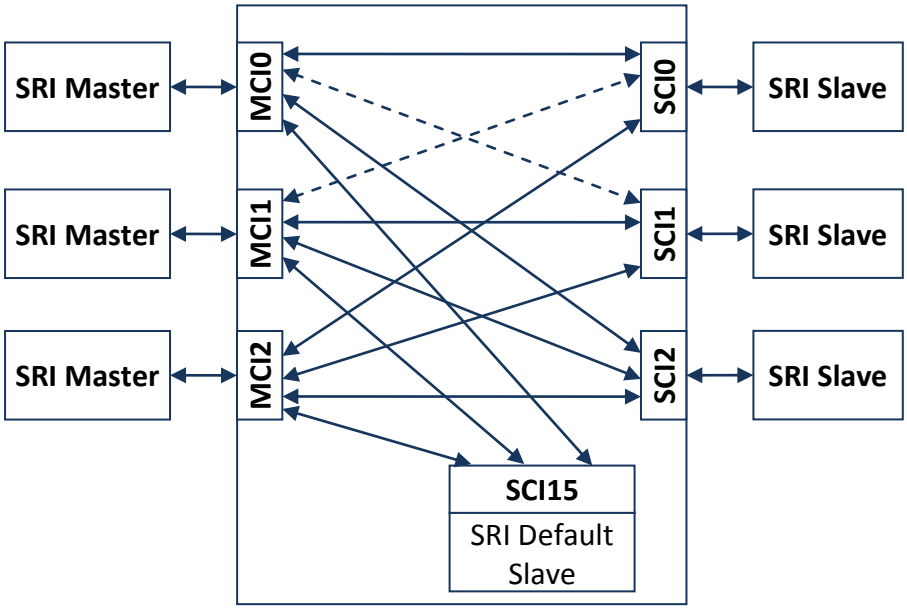

Fig. 3.13: XBar_SRI point to point connection scheme.

The SCI6 is the only one which is enabled to execute “write transactions”, because, as told in section 3.2.6, the command sequences consist in write accesses to Data Flash bank 0. Moreover write accesses to register set access are allowed.

The SCI7 and SCI8 interfaces are enabled just for reading transaction, to Program Flash bank 0 and Program Flash bank 1 locations respectively.

In this context, the slave view will be presented, because it is the Program Memory Unit point of view. According to the SRI bus protocol the following kind of transactions are distinguished:

- SRI Write Transaction: the master performs a write access, the slave reads;
- SRI Read Transaction: the master reads data sent by the slave.

All transactions are univocally identified by the “transaction id” parameter. It especially serves for transaction longer than one clock cycle, in order to allow either the master or slave peripherals to recognize the transaction the data of which is on bus. A transaction is composed by the following three phases:

- Arbitration phase;

- Addressing phase;
- Data phase.

3.3.1 SRI Slave Signals

The most important signals of an SRI Slave peripheral are the following:

- “sri_clk_i”: input clock reference;
- “sri_rd_n_i”: input which indicates if the transaction master is running is a read transaction;
- “sri_wr_n_i”: input which indicates if the transaction master is running is a write transaction;
- “sri_addr_i”: 32 bit input which indicates the location address the master would like either read or write;
- “sri_wdata_i”: 64 bit input which indicates the data master would like write. It is used just for write transaction;
- “sri_rdata_o”: 64 bit output which indicates the data slave sends to master. It is used just of read transaction;
- “sri_wrdvalid_n_i”: input which indicates whether data present on “sri_wdata_i” input is valid;
- “sri_ready_n_o”: output which informs the master that the slave has sent the valid data on “sri_rdata_o” signal during a read transaction;
- “sri_tr_id_i”: input which indicates the transaction identifier during the arbitration phase of transaction;
- “sri_wr_tr_id_i”: input which indicates the transaction identifier during the data phase of a write transaction;
- “sri_rd_tr_id_o”: output which indicates the transaction identifier during the data phase of a read transaction;
- “sri_id_err_n_o”: output which will be asserted when a transaction mismatch occurs during the data phase of transaction;
- “sri_err_n_o”: output which indicates a generic error has occurred during transaction; e.g. generic error can be due to address requested from master is out of the possible range of the slave peripheral. In case of the Program Memory Unit, a generic error can occur whether a transaction begins while the PMU is sleeping.
- “sri_opc_i”: op code, 5 bit input which indicates how long the data phase is, in terms of clock cycles. There could be four six cases:
    - SDTB (Single Data Transfer Byte): data phase takes one clock cycle and just one 8 bits data are transferred;

- SDTH (Single Data Transfer Half-Word): data phase takes one clock cycle and just 16 bits data are transferred;
- SDTW (Single Data Transfer Word): data phase takes one clock cycle and 32 bits data are transferred;
- SDTD (Single Data Transfer Double-Word): data phase takes one clock cycle and 64 bits data are transferred;
- BTR2 (Burst Transfer Request – 2 transfers): data phase takes two clock cycles where 64 bits data are transferred per everyone;
- BTR4 (Burst Transfer Request – 4 transfers): data phase takes four clock cycles where 64 bits data are transferred per everyone;

3.3.2 SRI Arbitration Phase

The arbitration is the transaction's phase during which the master requests to communicate with a slave. In this phase the master specifies the address, the op code and the transaction id parameters, while by the slave's point of view no signal changes.

The arbitration phase serves especially for the SRI crossbar to address the communication to the selected slave.

3.3.3 SRI Address Phase

During the address phase the slave peripheral receives the transaction request. In this phase, in fact, one of either "sri_rd_n_i" or "sri_wr_n_i" is asserted (set to 0) to indicate that a read or write transaction has started respectively. The transaction identifier and the op-code are also received by the slave during this phase.

The slave pushes the transaction into a "queue". So, during the next steps (data phase), it sends (or receives) the data associated to the corresponding transaction in the queue.

More than one transaction could be in the queue at the same time. If the data phase is longer than one clock cycle the slave peripheral can recognize the transaction the sent or received data belongs to.

3.3.4 SRI Data Phase

In a read transaction, the data phase starts with the "sri_ready_n" assertion by the slave. The time passed while the address phase is completed and the ready

signal has not been asserted yet is measured in wait states. For a write transaction: no one, one or more than one wait states can occur. Instead, for a write transaction the address phase ends immediately after the master sends the address and transaction identifier information.

During the data phase, the data is transferred between the master and the slave peripherals. The arbitration phase duration depends on the op code sent by the master during the arbitration phase.

3.3.5 SRI Transaction Examples

The following figures give some example of write and / or read transactions.

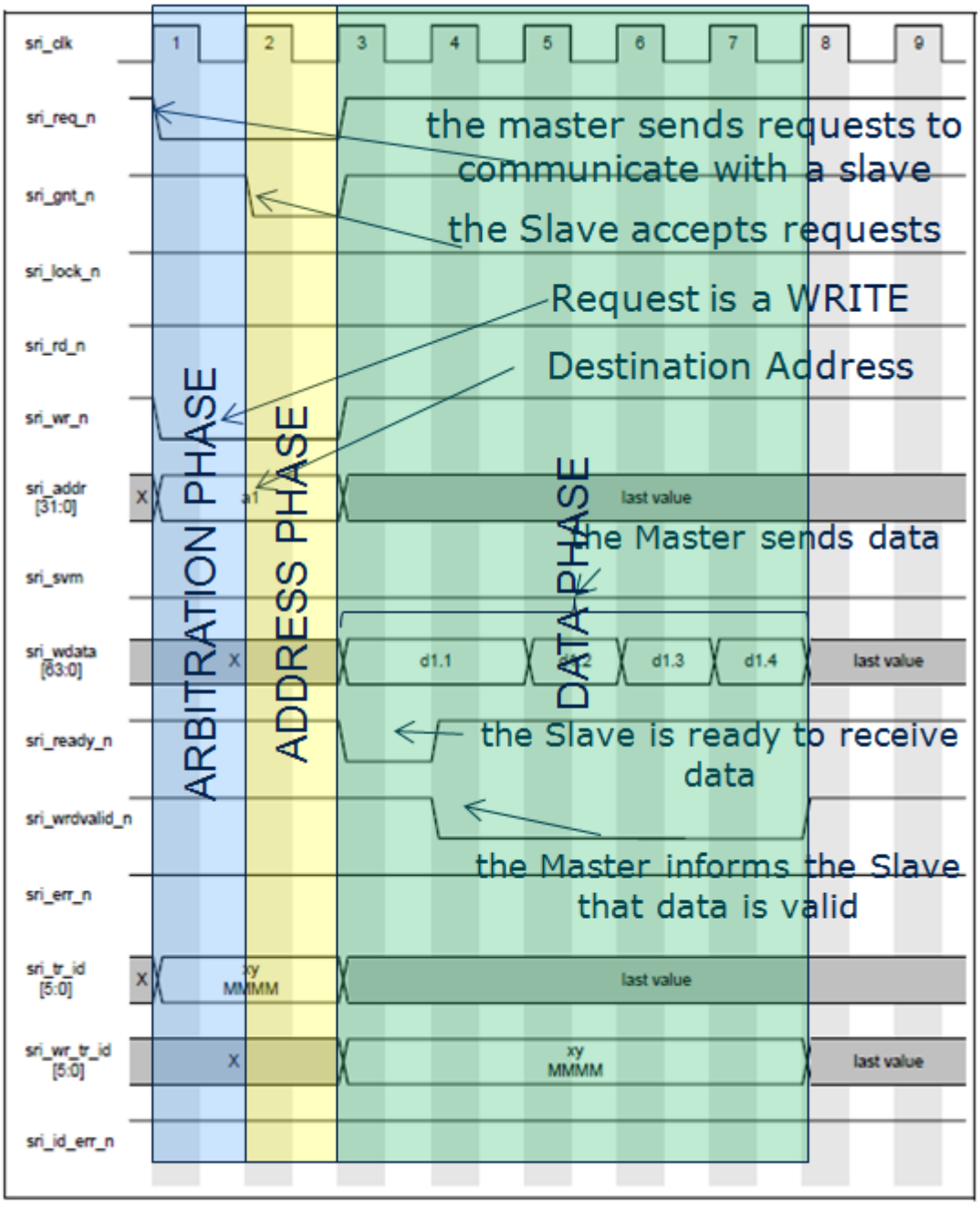

Fig. 3.14: Basic write block transfer 4 transaction, master view.

In Fig. 3.14 a write transaction is shown from the master peripheral point of view. The master during the arbitration phase asserts the “sri_req_n” signal to send a request to communicate with a slave. This request is forwarded to SRI crossbar which provides the interconnection with the selected slave. During this phase the master also sets the “sri_opc” signal which specifies the length of data phase (BTR4). During the arbitration and the address phases, the master asserts the “sri_wr_n” signal to indicate the transaction is a write. During the address phase the master receives the “sri_gnt_n” signal by the slave, thereafter data phase begins sending data on write bus. The master receives the “ready” signal by the slave at the end of the address phase. By the Fig. 3.14, it is clear that the first clock cycle of the data phase is lost. This phenomenon happens for all kind of transaction in the SRI bus. Fig. 3.15 shows the same transaction of Fig. 3.14 from the slave point of view.

In Fig. 3.16, a read transaction is shown from the master peripheral point of view. The master during the arbitration phase asserts the “sri_req_n” signal to send a request to communicate with a slave. This request is forwarded to SRI crossbar which provides the interconnection with the selected slave. During this phase the master sets also the “sri_opc” signal which specifies the length of data phase (SDTD). During the arbitration and the address phases the master asserts the “sri_rd_n” signal to indicate the transaction is a read. During the address phase the master receives the grant signal by the slave. The data phase begins when the slave asserts the “sri_ready_n” signal such that the master can read the data on the read bus. Fig. 3.17 shows the same transaction of the Fig. 3.16 from the slave point of view.

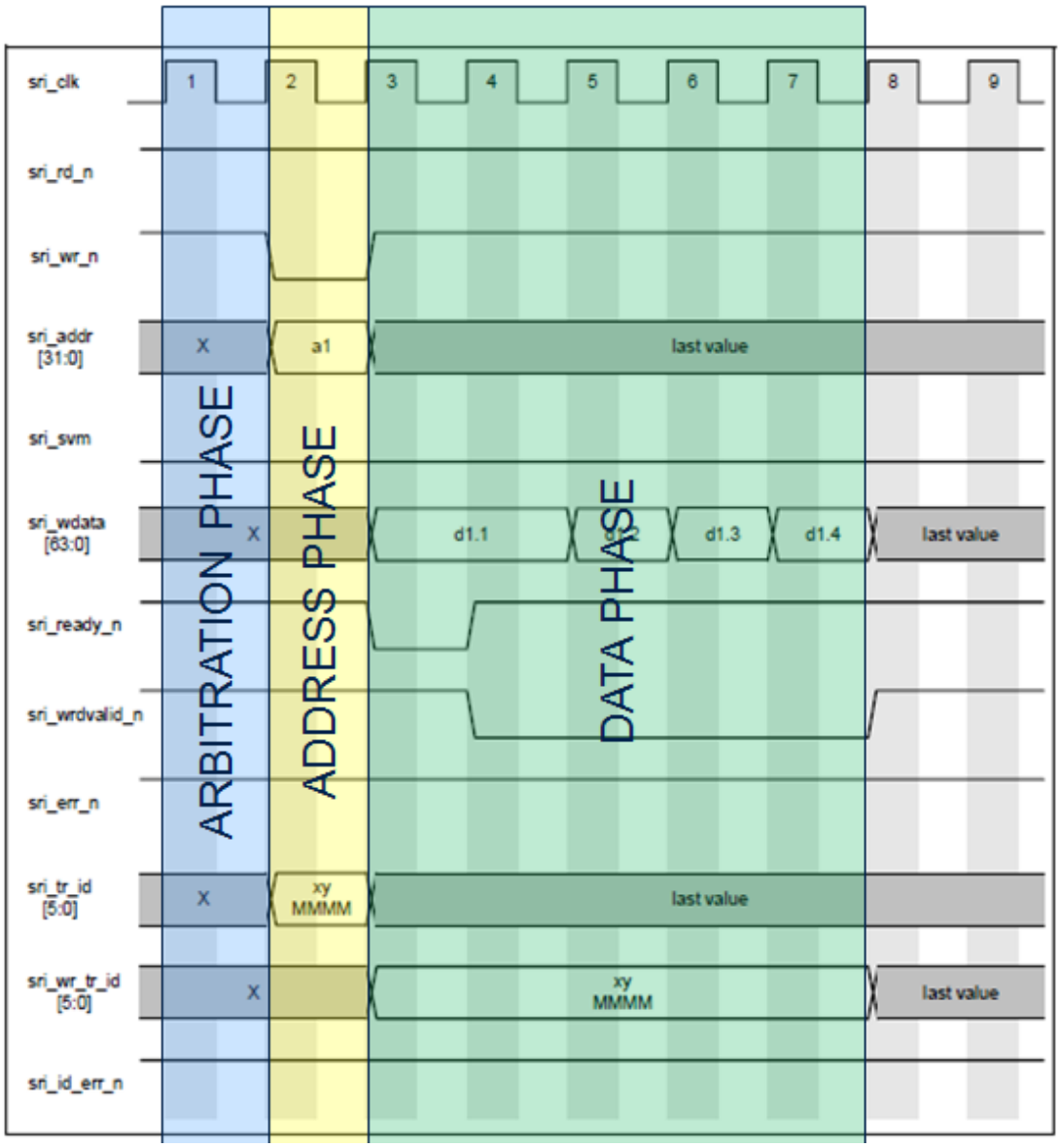

Fig. 3.15: Basic write block transfer 4 transaction, slave view.

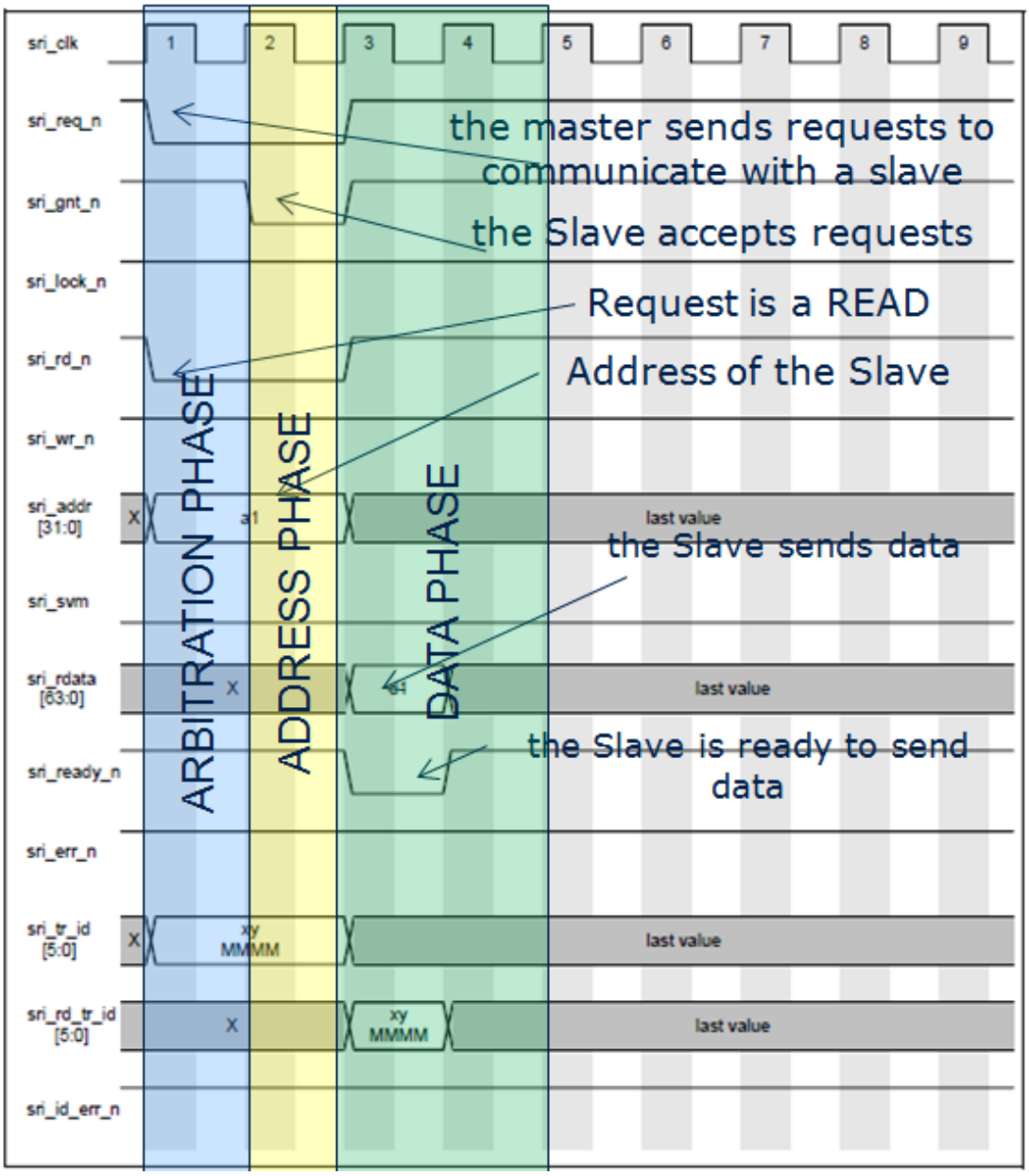

Fig. 3.16: Basic read block transfer, master view.

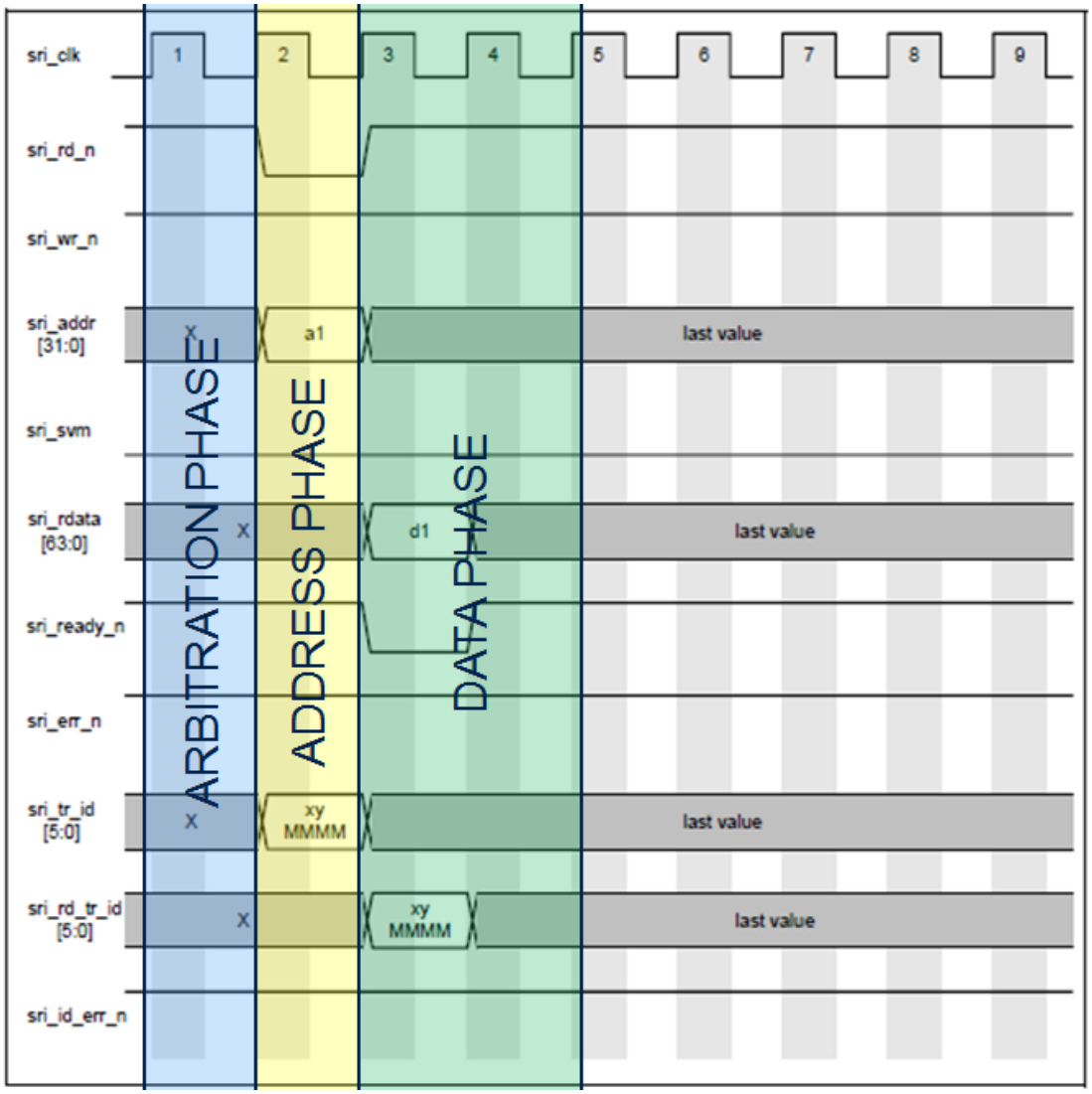

Fig. 3.17: Basic read block transfer, slave view.

3.3.6 SRI Transaction Id Error

During each cycle in the data phase of a write transaction, the slave receives the identifier of the transaction the data belongs to. It can happen that some received identifier does not correspond to any identifier in the "queue" of transactions. In this case, the slave peripheral will assert the "sri_id_err_n" bit for one clock cycle for each mismatch detected.

Transaction id errors during read transactions are not treated in this context because they do not regard the slave peripherals.

Fig. 3.18 shows an example of BTR2 transaction with two cycles ID error.

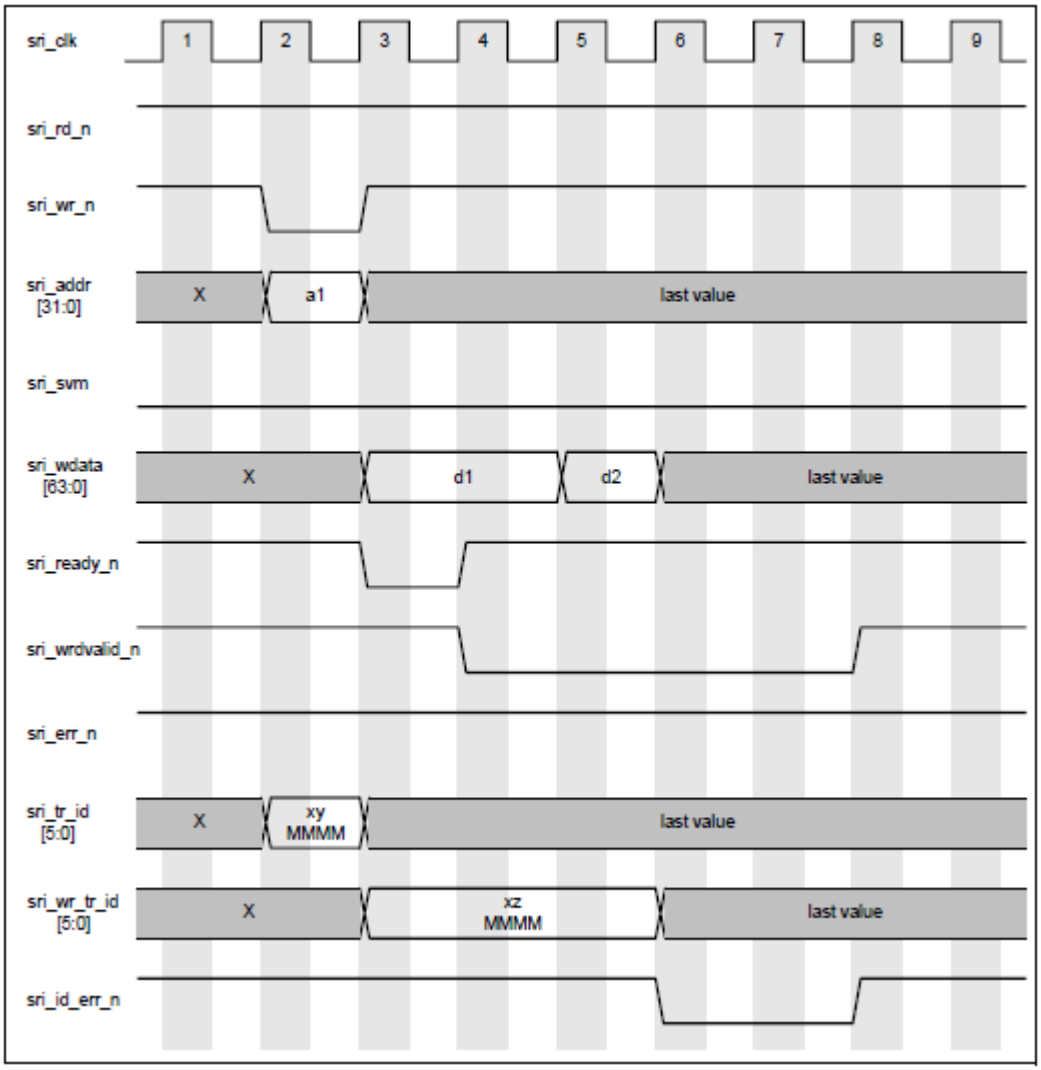

Fig. 3.18: Basic write block 2 transaction with two cycles ID error.

Let's suppose that the queue of transaction is empty, and that in the address phase a transaction with id equals to "xy" is pushed into the queue. During the data phase, two data cycles are received by the slave with transaction id equal to "xz" such that two ID mismatches are detected. After the data phase is completed, the slave asserts the "sri_id_err_n" signal in order to flag the master that an error occurred.

3.4 Conclusions

In this chapter, the general characteristics and behavior of the Program Memory Unit were explained. Some of the aforementioned theoretical notions are

of crucial importance to better understand the content of the later chapters of this work. However, the reader is referred to [***37***], [***38***] for a complete and thorough description of this specific unit.

Chapter 4 Simulation theory

Simulation is the process of modeling a proposed or real dynamic system and observing its behavior over time. A model is a representation of the real system that includes entities of the system, and the behavior and interactions of those entities [***39***]. A simulation environment must support some type of modeling framework to facilitate development and implementation of the simulator. The world view supported in a parallel simulation language should provide a framework for modeling the physical system.

Simulation is used in many contexts, such as simulation of technology for training, education or, closer this thesis, performance optimization, safety engineering and testing. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Simulation is also used when the real system cannot be engaged, because it may not be accessible, or it may be dangerous or unacceptable to engage, or it is being designed but not yet built, or it may simply not exist.

A computer simulation is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables in the simulation, predictions may be made about the behavior of the system. It is a tool to virtually investigate the behavior of the system under study.

Traditionally, the formal modeling of systems has been via a mathematical model, which attempts to find analytical solutions enabling the prediction of the behavior of the system from a set of parameters and initial conditions [***40***]. Computer simulation is often used as an adjunct to, or substitution for, modeling systems for which simple closed form analytic solutions are not possible. There are many different kinds of computer simulation; the common feature they all share is the attempt to generate a sample of representative scenarios for a model in which a complete enumeration of all possible states would be prohibitive or impossible.

Modern usage of the term "computer simulation" may encompass virtually any computer-based representation.

The simulator developed in this thesis has to be intended as behavioral, i.e. many non-ideal effects such as problems related to the physical layer of flash memories, explained in Chapter 2 are not considered. The purpose is to create a tool which allows validating the firmware written to test the flash module.

To reach the goal, two approaches have been analyzed:

- Verilog® simulation;
- C++ simulation.

The main difference is that Verilog® is a hardware description language (HDL), whereas the C++ is a sequential language. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time. HDLs form an integral part of Electronic design automation systems, especially for complex circuits, such as microprocessors.

An HDL is a specialized computer language used to describe the structure, design and operation of electronic circuits, and most commonly, digital logic circuits. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of an HDL program into a lower level specification of physical electronic components, such as the set of masks used to create an integrated circuit.

4.1 Verilog® simulation

The design of a very large scale integrated (VLSI) digital system commonly begins with a behavioral description of the system through a hardware description language, such as VHDL and Verilog®, and, subsequently to verify the functionality of the description. Modern digital system design strongly relies on simulation to ensure correctness of the design and to maximize system performances in terms of speed, power consumption, etc. [***41***].

The Verilog®[3] Hardware Description Language (Verilog HDL) became an IEEE standard in 1995 as IEEE standard 1364-1995 [***42***]. It was designed to be simple, intuitive, and effective at multiple levels of abstraction in a standard textual format for a variety of design tools, including verification simulation, timing analysis, test

[3] Verilog® is a registered trademark of Cadence Design Systems, Inc.

analysis, and synthesis. It is because of these rich features that Verilog has been accepted to be the language of choice by an overwhelming number of IC designers.

Verilog® contains a rich set of built-in primitives, including logic gates, user-definable primitives, switches, and wired logic. It also has device pin-to-pin delays and timing checks. The mixing of abstract levels is essentially provided by the semantics of two data types: nets and variables. Continuous assignments, in which expressions of both variables and nets can continuously drive values onto nets, provide the basic structural construct. Procedural assignments, in which the results of calculations involving variable and net values can be stored into variables, provide the basic behavioral construct. A design consists of a set of modules, each of which has an I/O interface, and a description of its function, which can be structural, behavioral, or a mix. These modules are formed into a hierarchy and are interconnected with nets.

Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation of time and signal dependencies (sensitivity). There are two types of assignment operators: a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Its control flow keywords (“if”/”else”, “for”, “while”, “case”, etc.) are equivalent, and its operator precedence is compatible. Syntactic differences include variable declaration (Verilog requires bit-widths on “net”/ “reg” types), demarcation of procedural blocks (“begin”/”end” instead of curly braces “{ }”), and many other minor differences.

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (“wire”, “reg”, “integer”,

etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a “begin”/”end” block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of “wire” consists of both signal values (4-state: "1, 0, floating, undefined") and strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements is synthesizable in Verilog language. Verilog modules that conform to a synthesizable coding style, known as RTL (register-transfer level), can be physically realized by synthesis software. Synthesis software algorithmically transforms the (abstract) Verilog source into a netlist, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flip-flops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC or a bit stream file for an FPGA.

4.2 Simulator based on C++

The simulation based on C++ programming language is absolutely more flexible than a simulation based on hardware description language. In fact, since the simulation will be computed, in any case, by an electronic calculator, surely an HDL could be replaced by a sequential programming language like C/C++. This flexibility is paid back in terms of higher possibility to make mistake, especially to control the passage of time.

4.3 Introduction to Verilator

Verilator is a synthesis tool which is able to join both of advantages of C++ and Verilog languages. Precisely, Verilator is a tool which converts synthesizable Verilog code into C++, therefore the output system model is described in C++ but is obtained starting from a Verilog code, avoiding the high possibility of making errors by writing the C++ code manually [***43***].

Verilator is therefore not a complete simulator, just a compiler. It represents the technology used to develop the simulator.

Code produced by Verilator contains a class which imitates the behavior of hardware described by the Verilog code. Around of C++ class made by Verilator we build simulation, writing other C++ classes linked to it. The main aim of these

classes is to connect signals and probes to the represented device. Moreover, they could be used to create a graphical user interfaces or any other feature to improve the whole simulator.

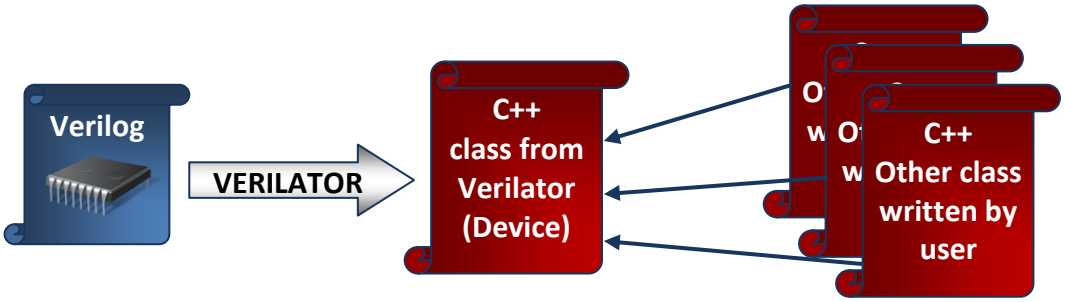

Fig. 4.1: Files which compose the simulator.

The existed CPU's simulator is written in Verilog code, so it is enabled to be compiled by Verilator tool.

4.3.1 How to interface C++ with Verilog

The simulator has a much different behaviour than the original device. Furthermore some flexibility is required such as:

- Dynamic memory range;
- Dynamic sectorization of the flash array;
- Dynamic command sequence definitions.

For this reason, most parts are developed directly in C++. Verilog code does not provide any possibility of resizing an array which represents the flash memory at run time.

Verilator provides some techniques to interface the Verilog code with C++. The Fig. 4.2 depicts the architecture of the whole simulator. Whenever some SCIx inputs change the PMU class generated by Verilator, it will invoke evaluation method on PMU class written directly in C++. The latter replies with the SCIx output values to forward to the CPU simulator.

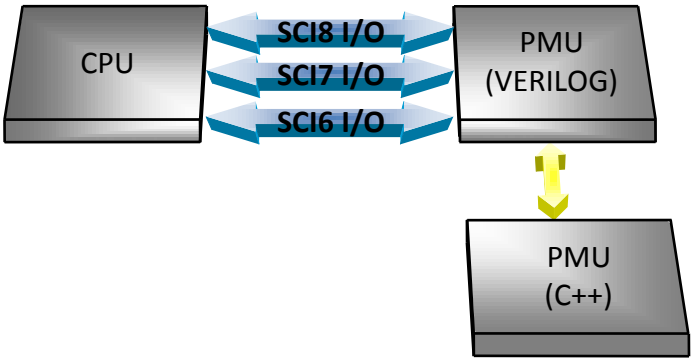

Fig. 4.2: System model architecture.

The PMU Simulator is interfaced to the CPU Simulator in a single Verilator run, in order to produce a single TOP module which represents the whole microcontroller (obviously only the implemented parts, but it could be improved in the future).

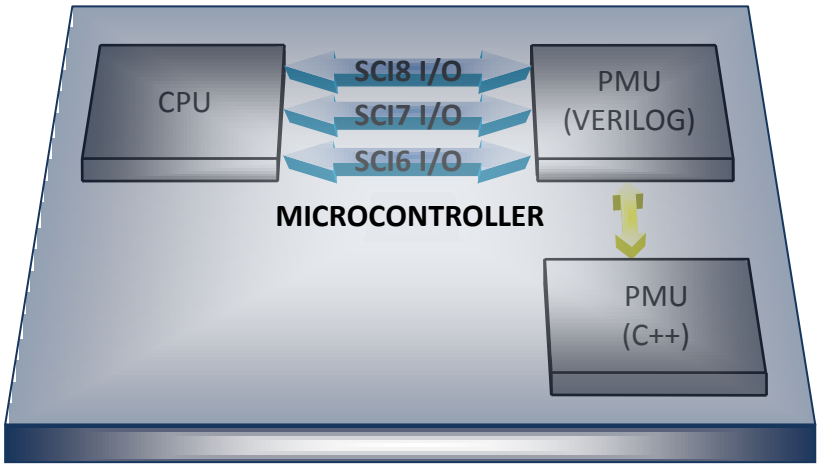

Fig. 4.3: Microcontroller model architecture.

4.3.2 Unit of time

The “Verilator” technology provides a function which gets back the simulation time at the moment it is invoked.

This function is called “sc time stamp” and it is written in the C++ code, generally in the same file where the “main” function is.

The distinctive trait of the “sc time stamp” is that it can be referred into the Verilog code by calling the directive “$time”, which in native Verilog language gets

back the current simulation time at the moment it is invoked. It is certainly not a synthesizable function, just usable in the test bench.

```
module PMU(
        input                   sri_clk_i,              // Clock input

        /* INPUTS/OUTPUTS FOR DFLASH, BootROM, RegisterSet */
        //  inputs
        Input                   sci6 _sri_lock_n_i,     // Slave Lock Request
        Input                   sci6 _sri_rd_n_i,       // Read Indication
        Input                   sci6 _sri_wr_n_i,       // Write Indication
        Input       `ADDR_RANGE sci6_sri_addr_i,        // Address Bus
        Input       [3:0]       sci6 _sri_opc_i,        // Operation Code
        Input                   sci6 _sri_svm_i,        // Supervisor Mode
        Input       `DATA_RANGE sci6_sri_wdata_i,       // Data Bus
        Input       [5:0]       sci6_sri_wr_tr_id_i,    // Write Transaction ID
        Input                   sci6 _sri_wrdvalid_n_i, // Write Data Valid
        Input       [5:0]       sci6 _sri_tr_id_i,      // Transaction Identifier
        //  outputs
        Output      `DATA_RANGE sci6_sri_rdata_o,       // Data Bus
        Output                  sci6 _sri_ready_n_o,    // Slave Ready Indicator
        Output                  sci6_sri_err_n_o,       // Error signal
        Output      [5:0]       sci6 _sri_id_err_n_o,   // Transaction ID Error
        Output      [5:0]       sci6_sri_rd_tr_id_o,    // Read Transaction
        …
);
```

The class produced by Verilator represents the hardware. It provides a member variable for each input/output of the top module. To use this class the input signals have to be set and the method “eval”, which evaluates the outputs and all the needed internal states of the represented device, has to be invoked.

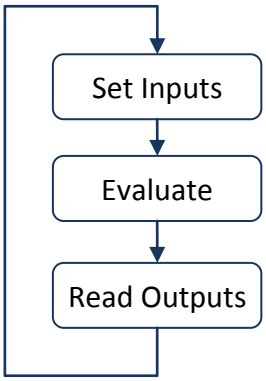

Fig. 4.4: Graphic representation of simulation cycle.

In general, the code which implements the simulator is a loop cycle wherein the inputs are set and the evaluation is performed as graphically represented in Fig. 4.4. At the beginning of each cycle, a variable which represents the simulation time is incremented. The value of that variable is the same as provided by the “sc time stamp” to Verilog code.

Fig. 4.5 gives an example of the temporal evolution of two signals: “clock” and “signal”. The first row represents the time axis, which is obviously discrete. The evaluation can be computed more than once per clock cycle, for example in Fig. 4.6 it is performed twice.

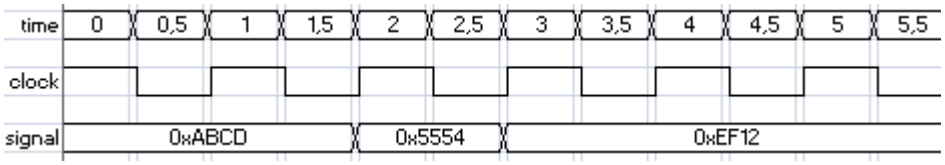

Fig. 4.5: Example of time variation of signals. The evaluation is performed once per clock cycle.

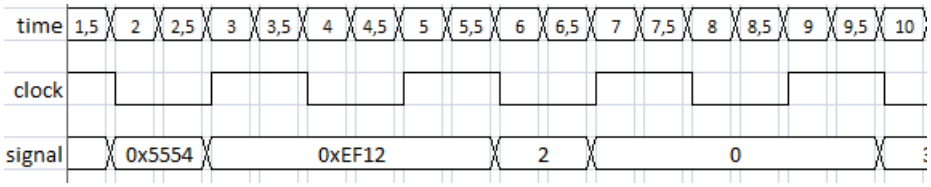

Fig. 4.6: Example of temporal variation of signals. The evaluation is performed twice per clock cycle.

The time is generally a floating point value, but sometimes it is used as an integer value. The user (simulator developer) has to take care the time value is correct (e.g. it is non-negative).

Chapter 5 Implementation

In Chapter 3, the microcontroller features have been described focusing on all the aspects regarding the Program Memory Unit and its interface with the other peripherals, in particular the CPUs.

These aspects will be taken into account again inasmuch the simulator has to imitate the PMU behavior, especially for the “Shared Resource Interconnect” bus, in order to guarantee the correct interface with the existing CPU simulator.

This chapter describes the technical issues regarding the implementation of the Program Memory Unit, focusing on the code development.

5.1 Program Memory Unit

From the Central Processing Unit’s point of view (or other microcontroller’s peripherals), the Program Memory Unit can be treated as a black box which executes all the operations on its internal flash array memory.

The management of the PMU passes through the Shared Resource Interconnect bus, which represents the communication media. The command sequences described in 3.2.6.1 are the high-level interfaces with the CPU.

The next paragraphs will thoroughly analyze this “black box” to discover all the internal implementations.

The Program Memory Unit, in this context, is represented by a C++ class wherein other classes are instantiated representing the internal modules.

Fig. 5.1 depicts the SRI slave modules within the Program Memory Unit. They are just the classes which manage the communication with other peripheral using the Shared Resource Interconnect bus. A complete representation of the module is given by Fig. 5.2.

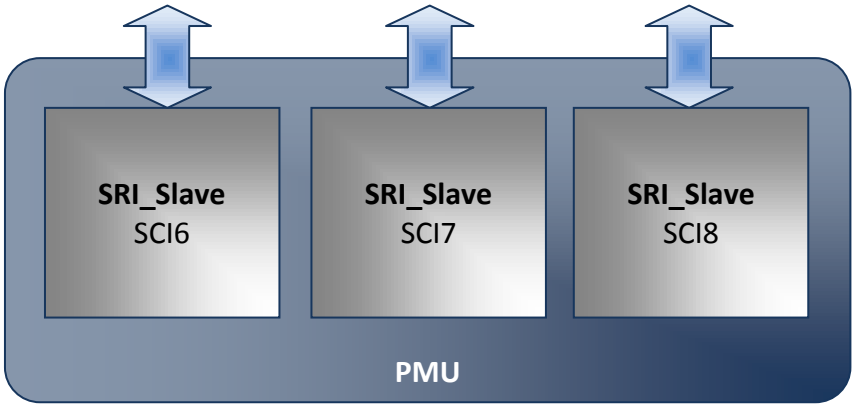

Fig. 5.1: Program Memory Unit as black box.

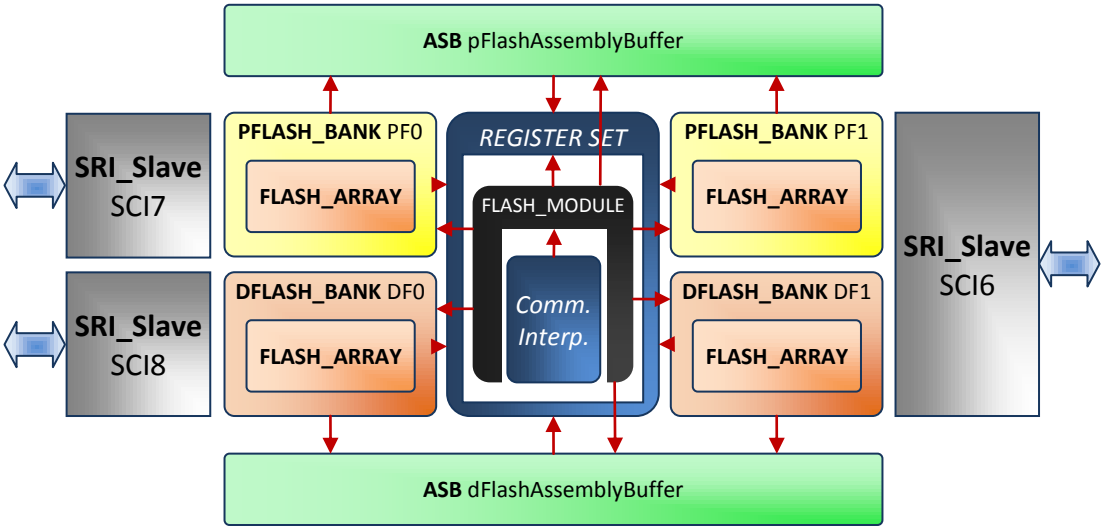

Fig. 5.2: Graphic representation of the whole Program Memory Unit.

In Fig. 5.2 every rectangle represents a class instance, except for the “Register Set” which denotes a set of static variables emulating the PMU and Flash registers. Rectangles contained within other rectangles symbolize the instantiation made inside the respective class. The arrows represent the references to other class instances. By looking at the figure, it is apparent that all classes are built within the top class called PMU, hereinafter also referred to as FLASH MODULE.

The top module is the class PMU which has to be interfaced with the Verilog code as explained in 4.3.1.

5.1.1 Error management

Each method invoked on the various instances of the sub modules gets back an instance of a structure which provides information about eventual errors occurred during its execution. This structure is called “ERROR INFO”.

For example, if the “Enter in Page Mode” command sequence is sent to the FLASH MODULE while an assembly buffer is already in PAGE MODE, the former will invoke the method “Enter in Page Mode” on the related assembly buffer. This has to produce a sequence error signaled by the SQER bit on the Flash Status Register, according to the command sequence definition explained, in 3.2.6.1.

As already stated above, the Flash Module is the top level of the PMU simulator and it manages all the operations in the PMU. In light of this, all the methods invoked by the instances representing the sub-modules, will always return an instance of “ERROR INFO” in order to prevent the different classes from simultaneously accessing the same bits of the different registers. The FLASH MODULE is responsible for enforcing designated bits on the Flash Status Register and eventually for printing further information to the user.

5.1.2 Shared Resource Interconnect slave port

The “SRI_Slave” objects in Fig. 5.2 represent the “Shared Resource Interconnect” slave peripherals which will be connected to the SRI crossbar. Their goal is both the generation and the interpretation of the digital signals which come from and / or go to the SRI crossbar.

The Program Memory Unit has three available ports, each enabled to read and / or write to different memory locations, as reported in Tab. 5.1.

	WRITE	READ
SCI6	DF0, DF1, Register Set, Boot ROM	DF0, DF1, Register Set, Boot ROM
SCI7	-	PF0
SCI8	-	PF1

Tab. 5.1: Memory range which slave ports are enabled for.

In this context, it is important to remember that every write access to the DF0 memory range is interpreted as a command cycle belonging to a command sequence, while a write accesses to Program Flash memory range will be refused

with a bus error. Therefore, the class representing the SRI slave port has to take into account this specific aspect embedding in its instance a set of writable and a set of readable ranges.

As explained in paragraph 3.3, a SRI slave peripheral has to perform a write transaction, when the data sent by the master are received by the slave itself; and a read transaction one, when the master requires getting the data located at a specific memory address. All the SRI communications are divided into three phases: arbitration, address and data, but just the last two are affected by the slave peripheral.

The code was developed to freely specify the range, so a specific structure has been created. This structure is called “AddressRange”, and it is visible in the fragment Code 5.1.

```
typedef struct
{
        IData    StartAddress;
        IData    StopAddress;
} AddressRange;
```

Code 5.1: Definition of the structure which indicates an address range.

Therefore two AddressRange arrays exist on the SRI slave to represent the writeable and readable address range.

When the SRI Slave instance recognizes a SRI transaction a notification has to be sent to the top class, the PMU. This is realized by two member function pointers: one for the read transactions and the other for the write transactions as depicted by the. Their definition and declaration are reported in the fragments Code 5.2 and Code 5.3 respectively, whereas the Fig. 5.3 graphically represents the member function pointers from SRI_Slave to the PMU class.

```
typedef void(PMU_Sim::PMU::*WRITE_DELEGATE)(IData, QData, unsigned);
typedef QData( PMU_Sim::PMU::*READ_DELEGATE)(IData);
```

Code 5.2: Declaration of function pointers which notify to the PMU that a SRI transaction occurred.

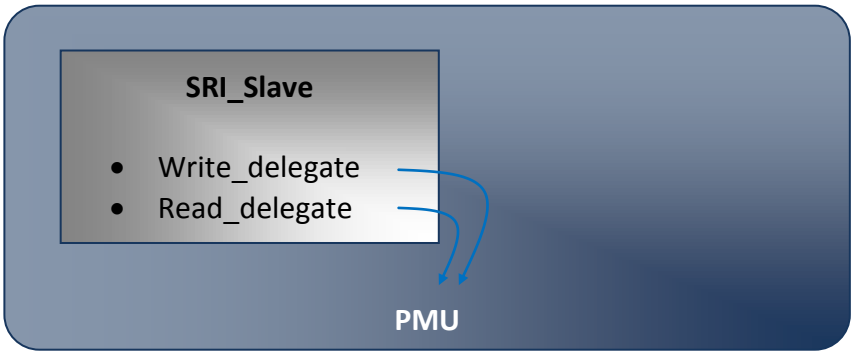

Fig. 5.3: The SRI_Slave class invokes a method on PMU when a SRI transaction has been recognized.

```
class SRI_Slave {
    WRITE_DELEGATE write_delegate;
    READ_DELEGATE read_delegate;
      …
};
```

Code 5.3: declaration of PMU's member function pointers which notify to the PMU that a SRI transaction occurred.

5.1.2.1 Read Algorithm

As explained in section 3.3, the Shared Resource Interconnect bus supports multiple transactions, in the sense that a transaction is able to execute the arbitration phase, while another existing transaction is being performed.

The read algorithm, which is graphically represented in Fig. 5.4, is based on a queue of “Read Transaction” to support multiple transactions and it is executed on every rise edge of the SRI clock system signal. In particular, a “Read Transaction” is a structure which contains the following parameters:

- Address;
- Transaction ID;
- Value sent to SRI bus;
- Data phase cycles computed;
- Data phase cycles to compute;
- A flag which tells if the transaction is in error.

When a new SRI read transaction is required from a master peripheral, a new transaction instance is pushed into the queue and it is signaled as “in arbitration phase”, because it is referred by a pointer called “arbitration_read”.

If the oldest transaction in the queue is not in arbitration phase, a data phase cycle will be computed sending data to SRI read bus. The number in the transaction, that counts the computed data phase cycles, will be incremented.

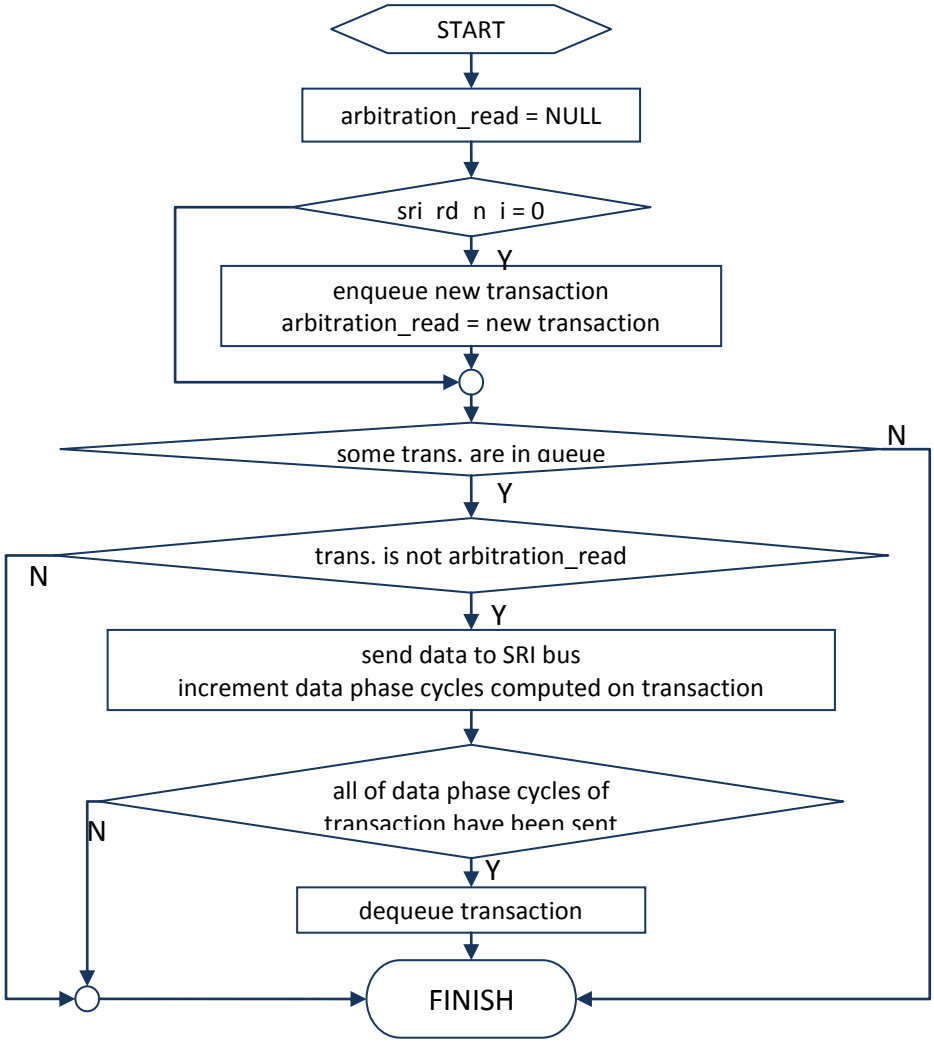

Fig. 5.4: Flow chart of the read algorithm.

If all of the data phase cycles of the oldest transaction in the queue have been completed, this transaction will be removed from the queue and the algorithm will be executed for the next oldest transaction in the queue.

In order to improve the readability of the code, this algorithm is divided into two sub algorithms:

1. “Evaluate Read Arbitration Phase” which performs the arbitration phase;
2. “Perform Read Data Phase” which computes the data phase of the transaction.

Fig. 5.4, Fig. 5.5 and Fig. 5.6 show the flow charts representing the algorithm. In particular, Fig. 5.4 represents the whole algorithm in a single flow chart, Fig. 5.5 and Fig. 5.6, instead, split the same algorithm in the two sub algorithms above mentioned.

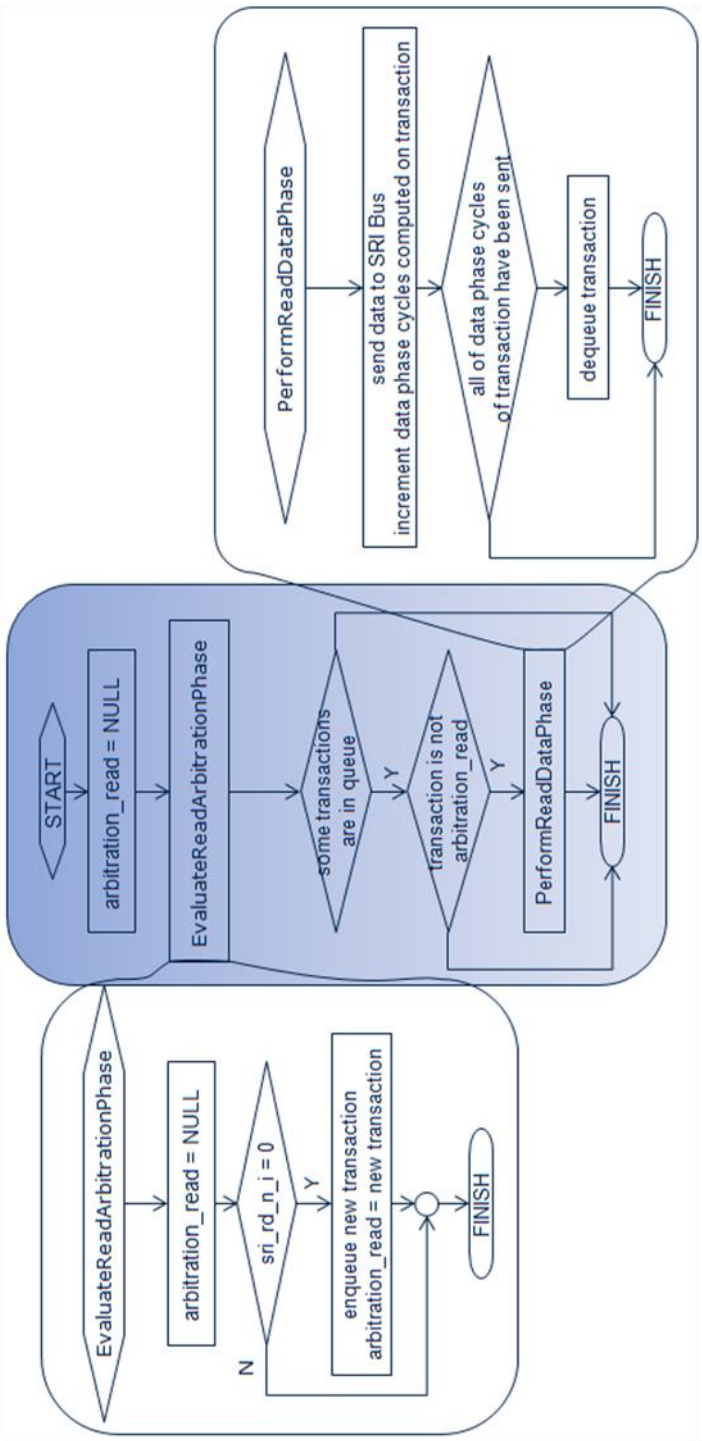

Fig. 5.5: Read algorithm divided into "Evaluate Read Arbitration Phase" and "Perform Read Data Phase" sub-algorithms.

It is worth remembering that the read algorithm has also to manage the bus errors. Typically, a bus error occurs when the master requires reading an address

out of the possible range of SRI slave port. In this case the slave asserts the error clearing “sri_err_n_o” bit.

The sub-algorithms have to change according to the flow charts shown in Fig. 5.6.

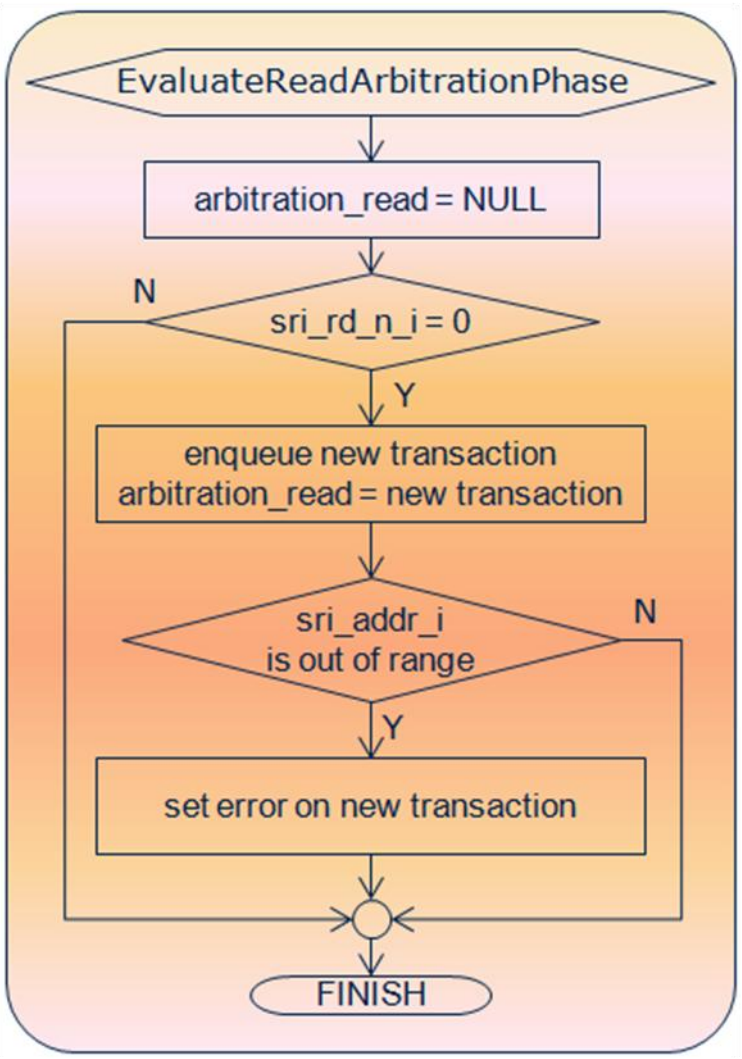

Fig. 5.6: Flow chart which represents the "Evaluate Read Arbitration Phase" algorithm which takes into account the bus error.

Observing the picture, it is clear that an if-statement is added to the previous version in order to check whether the address required by the master is out of range.

It is worth noting that this is not the only case that a SRI bus error can occur. In fact, when the master executes a read access while the Program Memory Unit is sleeping, it should get a bus error. Actually, this aspect is not yet implemented

because the Program Memory Unit's simulator is not provided with the sleep mode.

5.1.2.2 Write Algorithm

Like the read one, also the write algorithm has to support multiple SRI transactions, therefore it is based on a queue of "write transactions" which are specific data structures containing the following parameters:

- Address;
- Transaction ID;
- Number of cycles computed during data phase with a wrong transaction ID;
- The number of data phase remaining cycles;
- A flag which tells whether the ready signal has been sent to the SRI Port Master;
- A flag which tells whether the transaction is in error.

When a write transaction is started, a new instance of "write transaction" is pushed into the queue and is set as "in arbitration phase" by setting the specific "arbitration_write" variable to point to this new transaction.

If the oldest transaction in the queue is not in arbitration phase, and write data present on the bus is valid, a data phase cycle will be performed invoking the relative PMU member function through the related pointer and the number of data phase remaining cycles on transaction will be decremented.

If no data phase cycle remains on the oldest transaction, it will be removed from the queue.

Fig. 5.7 reports what has just been outlined in a simple flow chart. It is a general description of the actual algorithm, since bus errors and transaction ID error have to be managed. In order to make it easier to understand, the algorithm description is divided into some sub-algorithms.

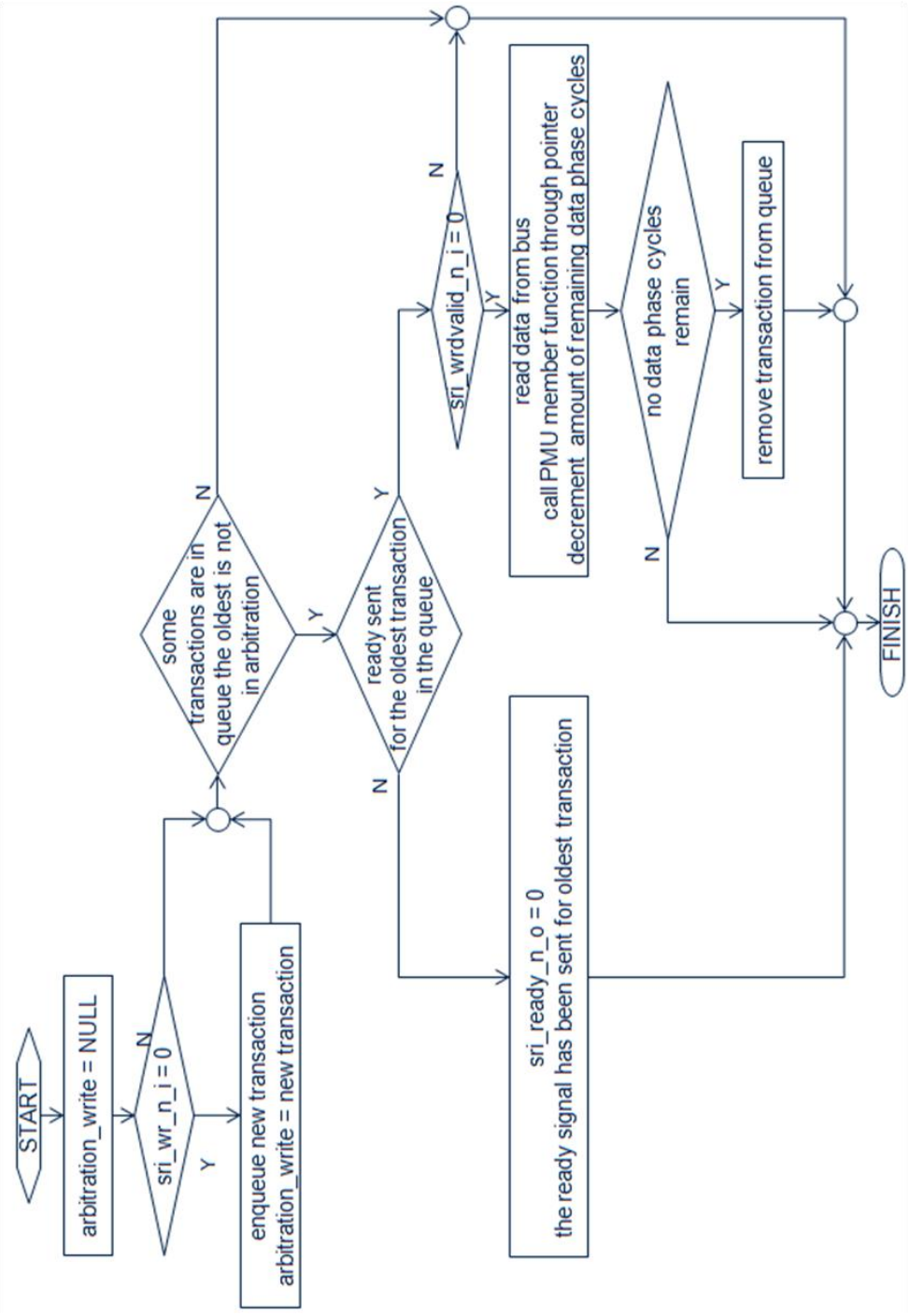

Fig. 5.7: Flow char which describes generally the write transaction algorithm.

The "Evaluate Write Arbitration Phase" sub algorithm, which is graphically represented in Fig. 5.8, detects a new SRI write transaction and puts the new

instance of “write transaction” in the write queue. Thereafter, it detects if the address required by the master is out of range. In this case, it will assert a flag on the new transaction instance.

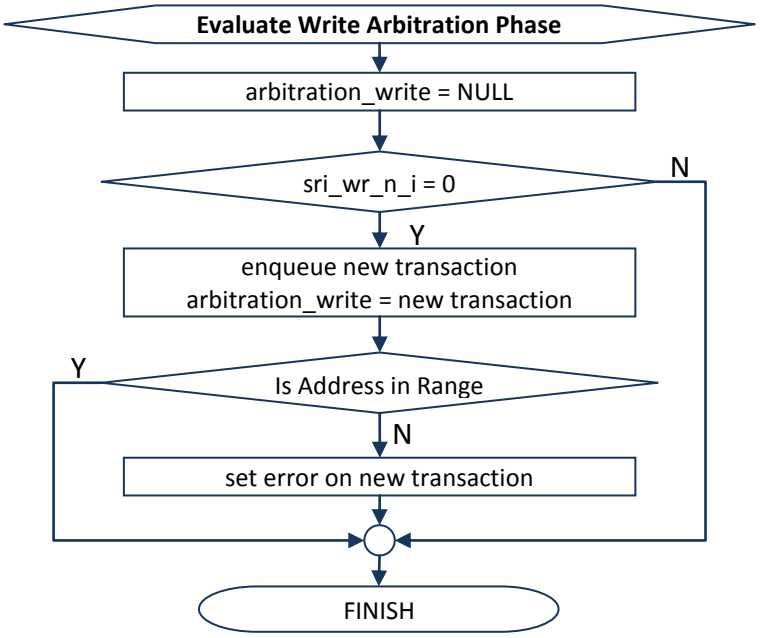

Fig. 5.8: Flow chart which represents the "Evaluate Arbitration Phase" sub algorithm.

The “Evaluate Write Data Phase” function works on the oldest transaction in the queue. It sends the ready signal on the bus if it has not been sent yet for that transaction; otherwise it will call the next method, “Perform Write Data Phase”, which will read the data from the bus. A graphical description of this specific sub-algorithm is reported in the flow chart in Fig. 5.9.

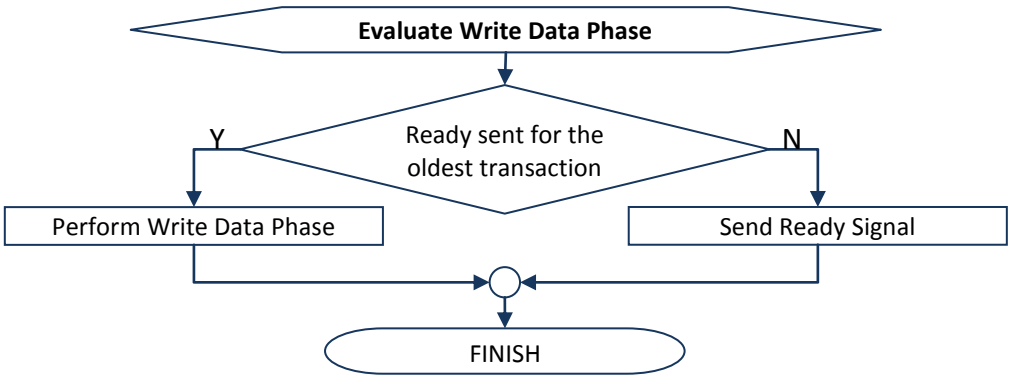

Fig. 5.9: Flow char which represent the "Evaluate Write Data Phase" sub algorithm.

The “Perform Write Data Phase” method, which is graphically described by Fig. 5.10, works on the oldest transaction in the queue. If it has a generic error, it will assert the related signal on the bus and resets the remaining data phase cycles (so in the next function the transaction will be removed from the queue); otherwise, if the data on the bus is valid and no transaction id error occurs, it will invoke the appropriate function on PMU.

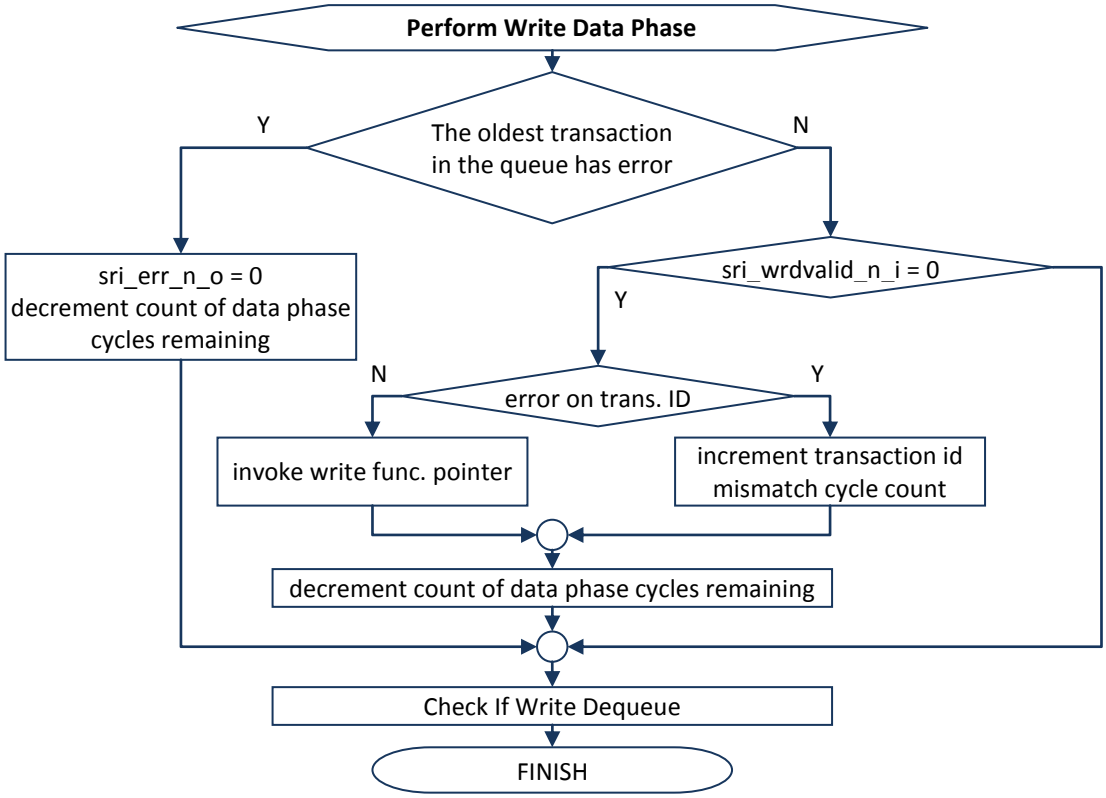

Fig. 5.10: Graphic description of the "Perform Write Data Phase" sub algorithm which takes into account transaction ID error.

If transaction ID occurs the related counter in the transaction instance will be incremented. After a data phase cycle is computed, the remaining cycles counter will be decremented.

“Check If Write Dequeue”, the flow chart of which is shown in Fig. 5.11, is the last sub method of the whole write algorithm. It controls whether all the data phase cycles of the transaction have been completed and, in this case, it verifies if the transaction ID errors occurred. If no Transaction ID error occurs, the instance will be correctly removed from the queue and the SRI transaction will end; otherwise it will assert the related output signal on the bus and decrement the counter which contains the number of mismatched cycles.

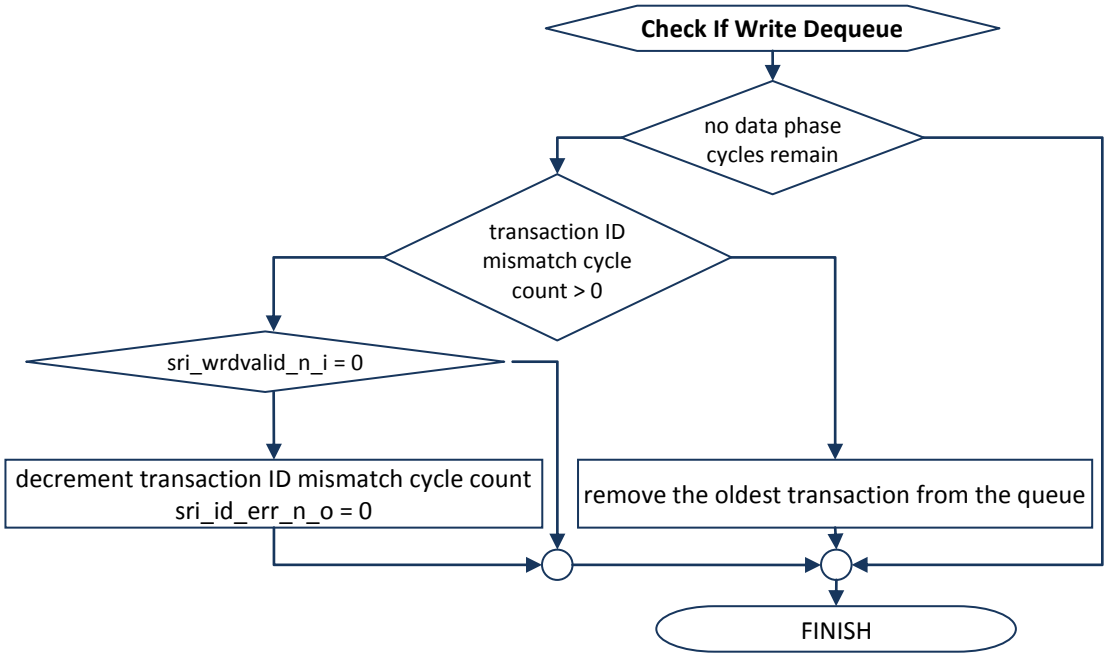

Fig. 5.11: Flow chart which describes the "Check If Write Dequeue" sub algorithm.

5.1.2.3 Test of SRI Port Slave

The algorithms that have been presented so far make up the SRI port slave. In this section, the results of some tests that were carried out are reported.

In Fig. 5.12, input and output signals of an SRI slave port are shown for a Single Data Transfer Double-Word read transaction. These can be compared with those reported in Fig. 3.17, to understand that the signals are compliant to the SRI bus protocol [***38***].

In Fig. 5.13, the signals are reported for a Single Data Transfer Double-Word write transaction. These can be compared with those of Fig. 3.15 to verify the compliance with the SRI bus protocol.

In Fig. 5.14, an example of a Write Burst Transaction 4 is reported, while in Fig. 5.15 the same transaction reports an error because the address required by the master peripheral is out of the possible range of the slave port.

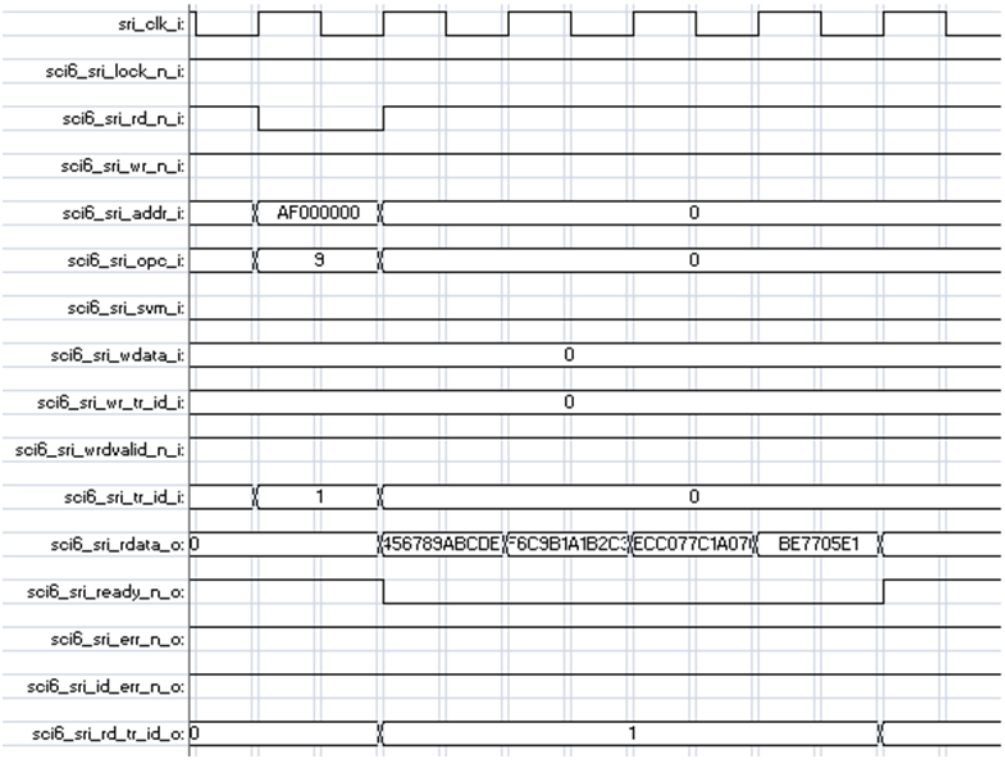

Fig. 5.12: Read transaction performed with SDTD op-code.

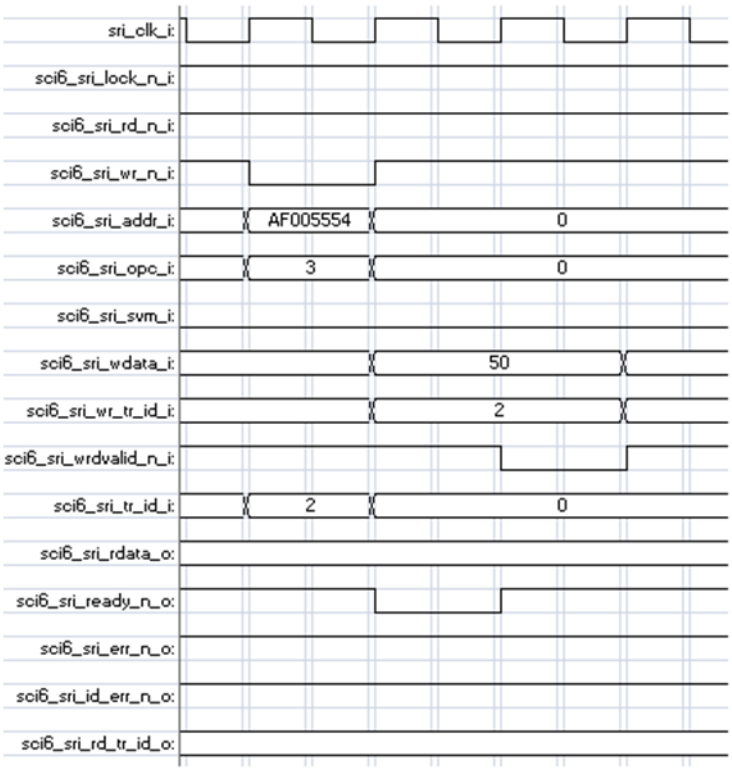

Fig. 5.13: Write transaction performed with SDTD op-code.

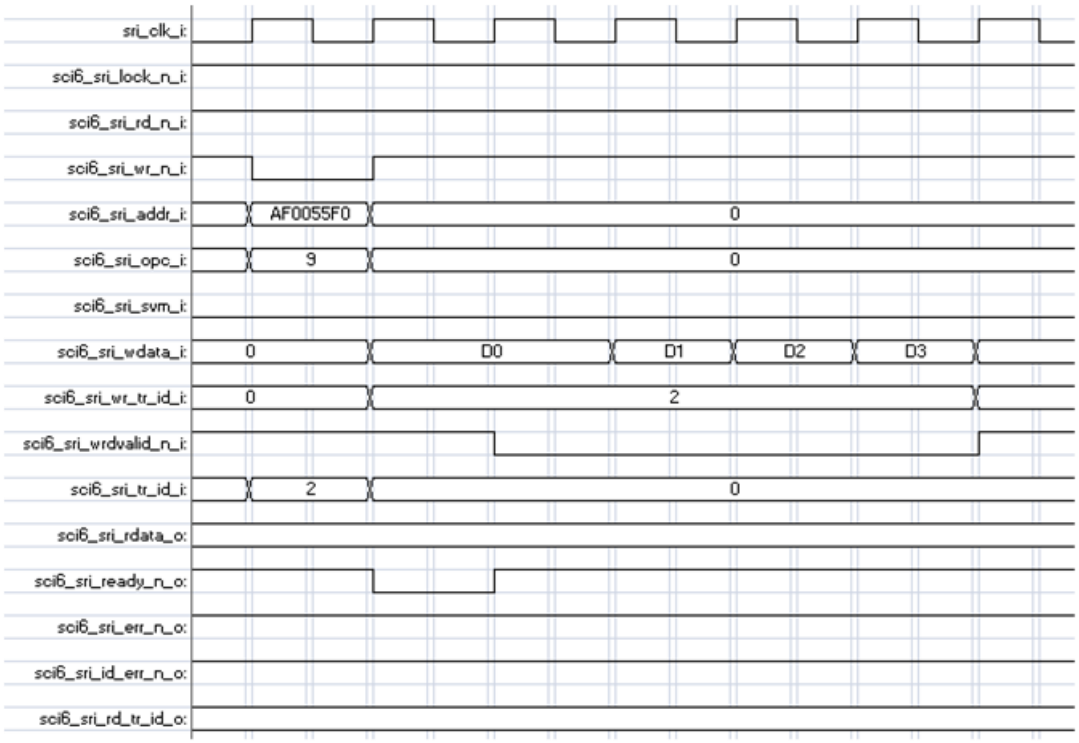

Fig. 5.14: Burst Write transfer 4 transaction.

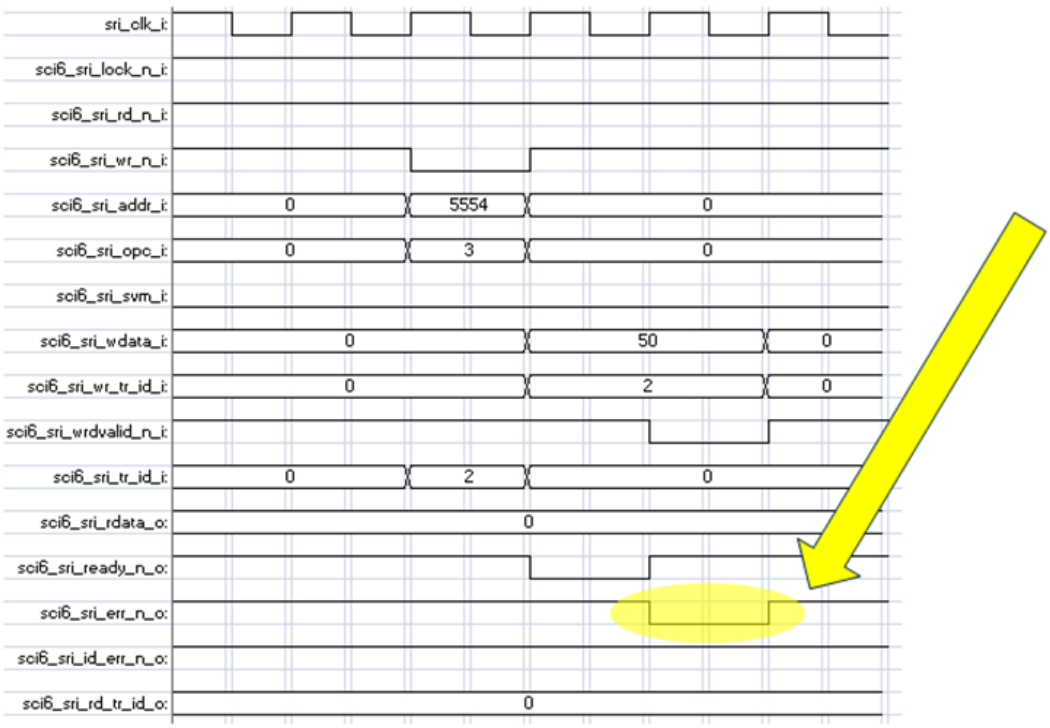

Fig. 5.15: SDTD transaction with bus error occurred because address is out of range.

5.1.3 Assembly Buffer

In Fig. 5.2, two classes both named “Assembly Buffer” are depicted. They emulate the modules of the same name in the Program Memory Unit hardware.

As explained in 3.2.6.3, to execute a programming operation the data, before being stored in the flash memory, has to be loaded in one of the two dedicated assembly buffers, depending on the destination: data or program flash. In the simulator, these buffers are represented by two instances of the ASB class.

Inside the PMU module, two assembly buffers exist:

- Data Flash Assembly Buffer: used during programming operations on Data Flash banks;
- Program Flash Assembly Buffer: used during programming operations on Program Flash banks.

Since a “Write Burst” operation programs eight pages of either data or program flash, the assembly buffers have to be designed large enough to contain all these pages. In this context, it is useful to recall that a program flash page is 32 byte

large, whereas data flash 8 byte, hence the size of the program flash assembly buffer is 256 bytes and the one of the data flash assembly buffer is 64 bytes.

However, an assembly buffer size is determined during its instantiation, reading the parameters from a configuration file, so that it is possible to simulate different assembly buffer sizes without re-compiling the simulator, just changing the configuration file.

The main goal of the assembly buffers is to keep the data while programming operation is running. An assembly buffer can be marked as:

- “Page Full”
- “Page Overflowed”
- “Full”
- “Overflowed”.

When an assembly buffer is marked as “Page Full” enough data has been loaded to cover a page size. If the assembly buffer is not “Page Full” and a write page is performed, the SQER bit of the FSR will be asserted.

When in the assembly buffer more data than a page size is loaded, it will be marked as both “Page Full” and “Page Overflowed”. In this case, if a “Write Page” is performed, the SQER bit of the FSR will be asserted.

When an assembly buffer is completely full, i.e. the size of the data loaded is equal to the burst size, it is marked as “Full”. In this case a “Write Burst” operation will be correctly executed.

If more than a burst size of data is loaded into the assembly buffer, the buffer will be marked as “Overflowed” and in this case both operations “Write Page” and “Write Burst” would be executed giving back a sequence error.

The “Assembly Buffer” class can be thought as an array which can be read and filled through some methods according to the TriCore TC27x assembly buffer’s specifications. Consequently, an assembly buffer can be in one of the following states:

- READ MODE: the assembly buffer can be read using either two dedicated methods: “Get 32 Bit Data” or “Get 64 Bit Data”. They are usually invoked more than once during a programming operation to get 32 bit data or 64 bit data respectively. Arguments passed to these methods are:

  - An index which indicates the page index into the assembly buffer;
  - An index which indicates the word (or double-word in case of 64 bit) inside the page indicated by the previous page's index;
- PAGE MODE: the command sequence "Enter in Page Mode" has been received by the PMU. The assembly buffer can receive either 32 bit or 64 bit of data to load into the array which emulates the assembly buffer's SRAM;
- PAGE MODE (WORD): with the assembly buffer in PAGE MODE, a "Load Page 32 bit" has been received by the PMU. The next "Load Page" command sequences have to be in the form of 32 bit data, otherwise a sequence error will be asserted and the data will not be loaded;
- PAGE MODE (DOUBLE-WORD): with the assembly buffer in PAGE MOE, a "Load Page 64 bit" has been received by the PMU. The next "Load Page" command sequences have to be in the form of 64 bit data, otherwise a sequence error will be asserted and the data will not be loaded;
- FROZEN FOR PROGRAMMING: the assembly buffer cannot receive data to fill the array because a programming operation is still running. If a "Load Page" is sent, a sequence error will be asserted.

Referring to Fig. 5.2, the FLASH MODULE class handles both assembly buffers. This concept emulates the same behavior of the actual Program Memory Unit. The difference is that in the hardware the assembly buffers are handled using digital signals, while in the simulator by invoking class methods.

The FLASH MODULE invokes the method "Enter in Page Mode" on the assembly buffer when receives the command sequence of the same name. There are two "Enter in Page Mode" commands: one for the program flash and other for the data flash (see 3.2.6.3). The method will change the assembly buffer's state from READ MODE in PAGE MODE. If the current state is not READ MODE, it will return a sequence error. After entering in page mode, the FLASH MODULE expects either "Load Page 32 bit" or "Load Page 64 bit" command sequences. When received, the method of the same name will be invoked on the assembly buffer which is in page mode. This method loads the data passed as an argument in the array that emulates the assembly buffer's SRAM In hardware.

When a programming operation begins, the involved assembly buffer has to change its state from PAGE MODE (WORD) or PAGE MODE (DOUBLE-WORD) to

FROZEN FORM PROGRAMMING. To accomplish this, the class provides the method “Freeze for Programming”, which is invoked by the “Flash Array” instance in the “Flash bank” involved in the programming operation. The “Flash Array” is able to invoke then either “Get 32 bit data” or “Get 64 bit data” method to obtain the data to store in flash memory.

The last command provided by the assembly buffer is “Reset to Read”. It is usually invoked by the flash array after the programming operation ends, but it can be invoked also by the FLASH MODULE once the “Reset to Read” command sequence has been sent.

Fig. 5.16 depicts the chronological sequence of how the assembly buffer is handled during a “Write Burst” operation. The “Enter in Page Mode” command produces the “Enter in Page Mode” invocation on the related assembly buffer. This method will change the PFPAGE / DFPAGE bit on the Flash Status Register (see 3.2.5.1). In figure the “Load Page” command is performed at 64 bit data. From Fig. 5.16 it is clear that the FLASH MODULE no longer needs to read the status directly by the assembly buffer, but it automatically recognizes which assembly buffer is in page mode only by using the PFPAGE, DFPAGE bit of the Flash Status Register.

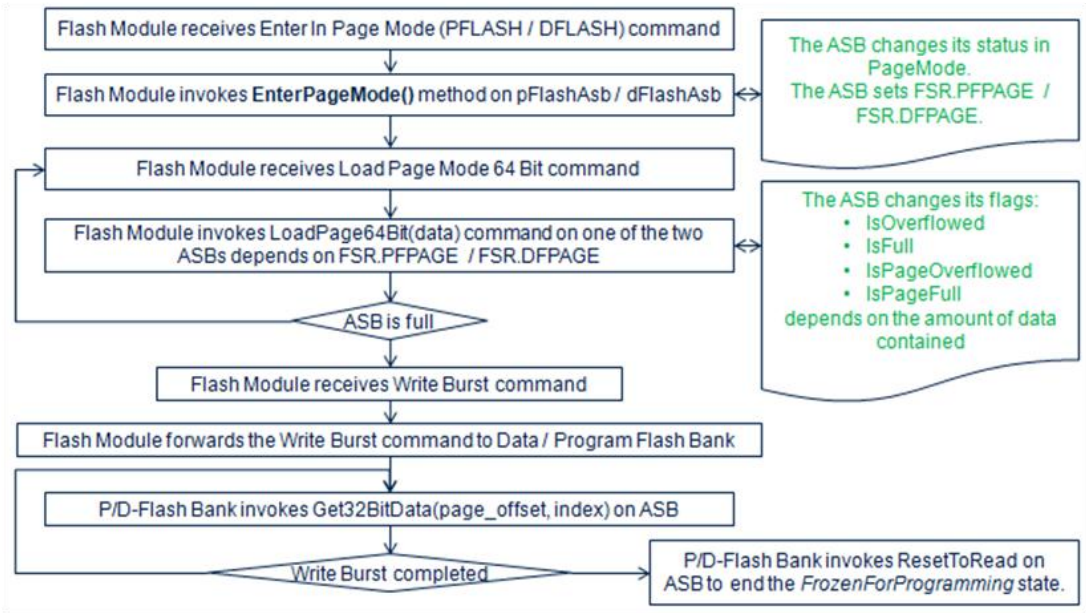

Fig. 5.16: Handling of the Assembly Buffer during write burst command.

5.1.4 Flash Bank

The flash bank class represents the “entity” containing the flash array memory. It is not a simple data container, but rather an actual digital module which provides all the methods necessary to perform the various operations. For instance, it has to simulate the passage of time during the programming or erasing and the possibility of suspending and resuming the operations.

Two classes were realized to represent the flash banks:

- Program Flash Bank
- Data Flash Bank

Since these two classes are very similar, the next pages will give a description valid for both of them. The main difference is that the “Write Page Once” command sequence is not implemented for the Data Flash Bank, inasmuch it is not provided by the hardware functionalities.

Two instances of each class are used to simulate both the PF0 and the PF1 program flash banks and both the DF0 and the DF1 data flash banks.

The flash bank behavior is similar to that of a finite state machine; in fact, depending on the running operation, it can be in one of the states reported in Tab. 5.2.

State	Description
IDLE	The flash bank can be read and it is ready for an eventual operation.
WRITING PAGE	The flash bank is busy for a “Write Page” command. No operation can be started unless the suspension is performed.
WRITING PAGE ONCE	The flash bank is busy for a “Write Page Once” command. No operation can be started unless the suspension is performed.
WRITING BURST	The flash bank is busy for a “Write Burst” command. No operation can be started unless the suspension is performed.

State	Description
ERASING LOGICAL SECTOR	The flash bank is busy for a “Erase Logical Sector Range” command. No operation can be started unless the suspension is performed.
ERASING PHYSICAL SECTOR	The flash bank is busy for a “Erase Physical Sectors” command. No operation can be started unless the suspension is performed.
WRITING PAGE SUSPENDED	A “Write Page” command has been suspended. The flash bank cannot be read, neither a new operation can be started on the same bank, just the “Resume Prog Erase” command can be performed.
WRITING PAGE ONCE SUSPENDED	A “Write Page Once” command has been suspended. The flash bank cannot be read, neither a new operation can be started on the same bank, just the “Resume Prog Erase” command can be performed.
WRITING BURST SUSPENDED	A “Write Burst” command has been suspended. The flash bank cannot be read, neither a new operation can be started on the same bank, just the “Resume Prog Erase” command can be performed.
ERASING LOGICAL SECTOR SUSPENDED	An “Erase Logical Sector Range” command has been suspended. The flash bank cannot be read, neither a new operation can be started on the same bank, just the “Resume Prog Erase” command can be performed.

State	Description
ERASING PHYSICAL SECTOR SUSPENDED	An “Erase Physical Sectors” command has been suspended. The flash bank cannot be read, neither a new operation can be started on the same bank, just the “Resume Prog Erase” command can be performed.
VERIFY ERASED LOGICAL SECTOR	A “Verify erased logical sector range” on the flash bank is running.

Tab. 5.2: Status set of the flash bank.

As already mentioned, the flash bank has to take into account the passage of time while carrying out an operation. In this context, the time is expressed through the function “sc time stamp” (see 4.3.2). To accomplish this target the constants reported in Tab. 5.3 are defined.

CONSTANT	DESCRIPTION
PAGE WRITE TIME	It indicates the time spent to program a page. The “Write Burst” command will spend eight times the “PAGE WRITE TIME”.
ERASE LOGICAL SECTOR TIME	It indicates the time spent to erase a whole logical sector. The “Erase Logical Sector Range” command will spend n times the “ERASE LOGICAL SECTOR TIME” where n is the number of logical sectors to erase.
ERASE PHYSICAL SECTOR TIME	It indicates the time spent to erase a whole physical sector. The “Erase Physical Sectors” command will spend n times the “ERASE PHYSICAL SECTOR TIME” where n is the number of physical sectors to erase.

CONSTANT	DESCRIPTION
VERIFY ERASE LOGICAL SECTOR TIME	It indicates the time spent to verify that one logical sector is erased. The “Verify Erased Logical Sector Range” command will spend n times the “VERIFY ERASE LOGICAL SECTOR RANGE TIME” where n is the number of logical sector to verify.

Tab. 5.3: Constant definitions to take into account the passage of time.

Furthermore, the flash bank contains the following variables:

- “Operation start time”: it is set at the moment an operation begins;
- “Operation suspended time”: it is initially set at the moment an operation begins, then if this is suspended it will be updated at the suspension time.

As explained in 4.3, for each clock cycle the evaluation method is invoked on the whole microcontroller’s simulator. This will call the evaluation method on the PMU which in turn will invoke the “Tick” method on the flash bank. The second method compares the difference between the current simulation time and the value kept by the “operation start time” with one constant among those reported in Tab. 5.3 depending on the flash bank state as explained in Tab. 5.2. So, the flash bank is able to detect whether enough time has elapsed to complete the operation.

If enough time has elapsed, the operation will be really executed on the flash array and the current state of the flash bank will change in “IDLE”.

The concept is graphically depicted by Fig. 5.17.

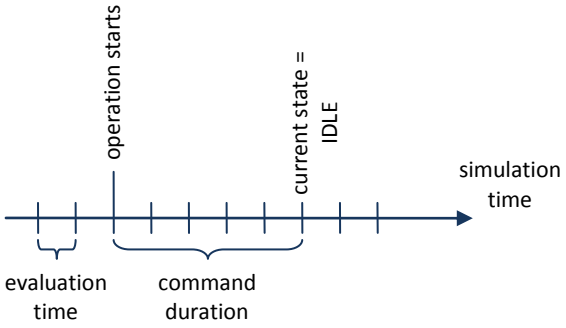

Fig. 5.17: Graphic representation of the duration of an operation.

For a “Write Burst” operation, the mechanism just expressed is a bit different: first of all another variable, called “page written during write burst”, is needed to keep in memory the number of pages written during the operation. For each evaluation of the PMU, the “Write Burst Tick” method is executed. It evaluates if the difference between the current simulation time and the “start operation time” is greater than the “PAGE WRITE TIME” constant, which means enough time has elapsed to program a single page. In this case, one page-size will be copied from the assembly buffer into the flash array; the “start operation time” will be updated to the current simulation time in order to allow controlling the passage of time for the next page; and the “page written during write burst” will be incremented.

If eight pages are written, the current state of the flash bank will change into “IDLE”, so that at the next PMU evaluation no tick method will be executed on the flash bank. Instead, if the pages written are less than eight, the current PMU’s state will remain on “WRITING BURST”, so that at the next evaluation the same process will be executed again. The flow chart in Fig. 5.18 gives a graphic representation of the method which performs this mechanism: “Write Burst Tick”.

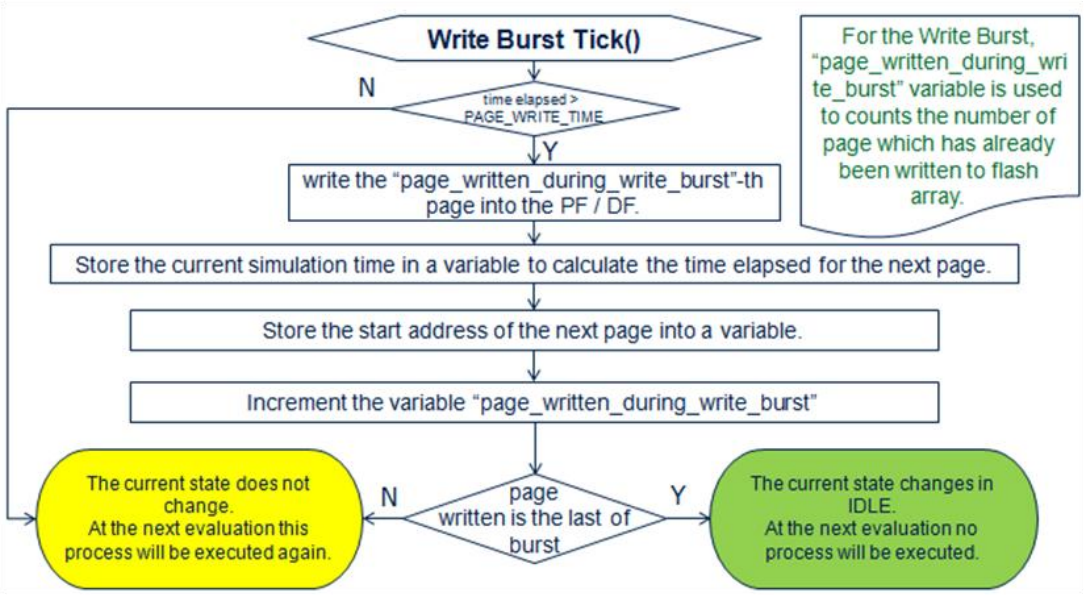

Fig. 5.18: Flow chart which describes the "Tick" method executed during a "Write Burst" command.

For the “Erase Logical Sector Range” or the “Erase Physical Sectors” or the “Verify Erased Logical Sector Range”, the “Tick” method is similar to “Write Burst Tick”. A graphic representation of the “Erasing Logical Sector Range Tick” is given in Fig. 5.19.

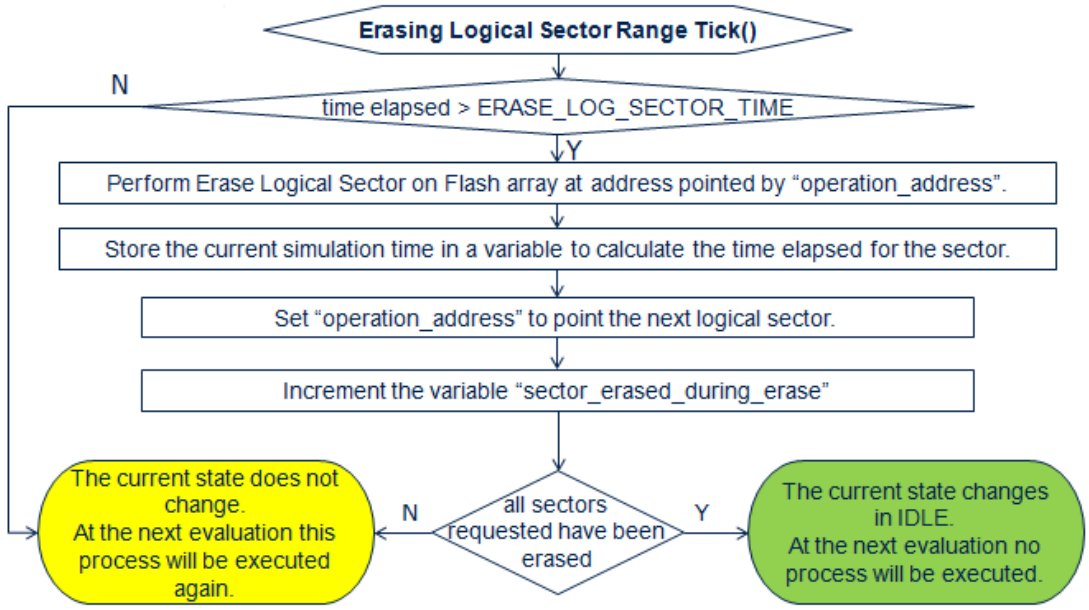

Fig. 5.19: Flow chart which describes the "Tick" method executed during an "Erase Logical Sector Range" command.

The main difference with the "Write Burst Tick" method is that to detect if the whole command sequence has been completed, the number of erased sector has to be compared with the command sequence argument which tells how much sectors have to be erased.

5.1.4.1 Flash array

The flash bank instance contains the flash array which can be thought as the class containing the actual memory and its partition. Even though the Flash array and the flash bank could be developed as the same class, they have been created as two different classes to improve the readability and to make the code documentation easier to understand. The main objects composing the flash array are:

- One array which emulates the actual memory and contains the stored data;
- One array of sectors to subdivide the whole memory array into logical sectors;
- One array of sectors to subdivide the whole memory array into physical sectors;
- One array of sectors to subdivide the whole memory array into pages.

In this context the word “sector” is used with reference to the structure reported in the fragment Code 5.4.

```
typedef struct Sector {
        /* The first address of the sector. */
        unsigned StartAddress;
        /* The last address of the sector. */
        unsigned StopAddress;
        /* The sector size (in Byte). */
        unsigned Size;
        /* The data contained in the sector. */
        CData*   Data;
} Sector;
```

Code 5.4: "Sector" structure definition.

The flash array provides methods to control the address and data arguments of a command sequence needed to perform the operation[4]. These methods are generally indicated as “begins methods”, inasmuch they are invoked before a command sequence starts.

For example: if the “Write Page” command sequence has been sent to the Flash Bank, the Flash Bank will invoke the “Begin Write Page” command on its Flash Array, passing the address argument received by the command sequence. The method controls no errors have occurred (e.g. the address is the first of a page) and in this case it will set the assembly buffer as “Frozen for programming” (see 5.1.3). At last it will get back an instance of “ERROR INFO” to the flash bank so the latter will be able to control whether it is possible to perform the command with the errors that eventually occurred[5]. In this case it will change its state in “WRITING PAGE”.

[4] The argument of the command sequence “Write Page”, for example, is the address of the page.

[5] If the “Write Page” command is given when the assembly buffer is not completely filled, or more data than one page size are loaded into the assembly buffer, a sequence error will be reported on the Flash Status Register, although the operation will be executed.

The same mechanism, which is graphically shown in Fig. 5.20 and Fig. 5.21, is repeated for the “Erase Logical Sector Range” or the “Erase Physical Sector Range” or the “Verify Erased Logical Sector Range”.

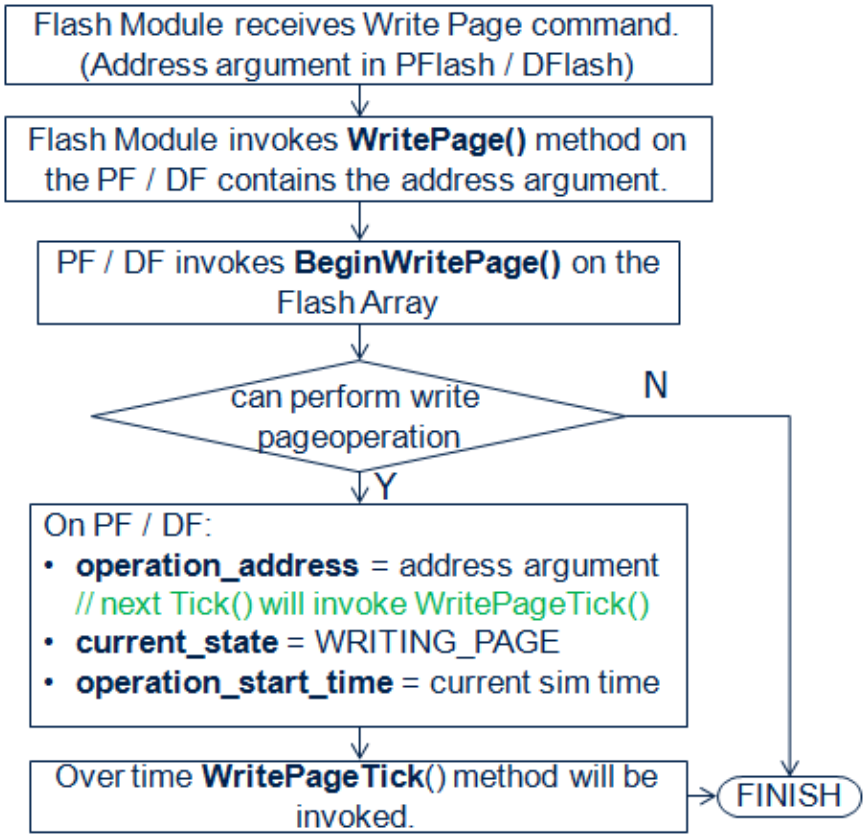

Fig. 5.20: Sequence performed during a "Write Page" command on the Flash Bank.

When a programming operation is performing, the data have to be copied from the assembly buffer into the flash array. A dedicated method called “Copy Assembly Buffer Page” is provided by the flash array. It takes care not to change the bits into the flash memory which are set to “1” because in a flash memory, as explained in Chapter 2, the erasing operation has to be performed separately by the program. Since the data into the assembly buffer cannot be directly copied into the flash memory, the logical “OR” operation has to be used between the existing data into the flash memory and the next value present into the assembly buffer to calculate the actual value to store into the flash array.

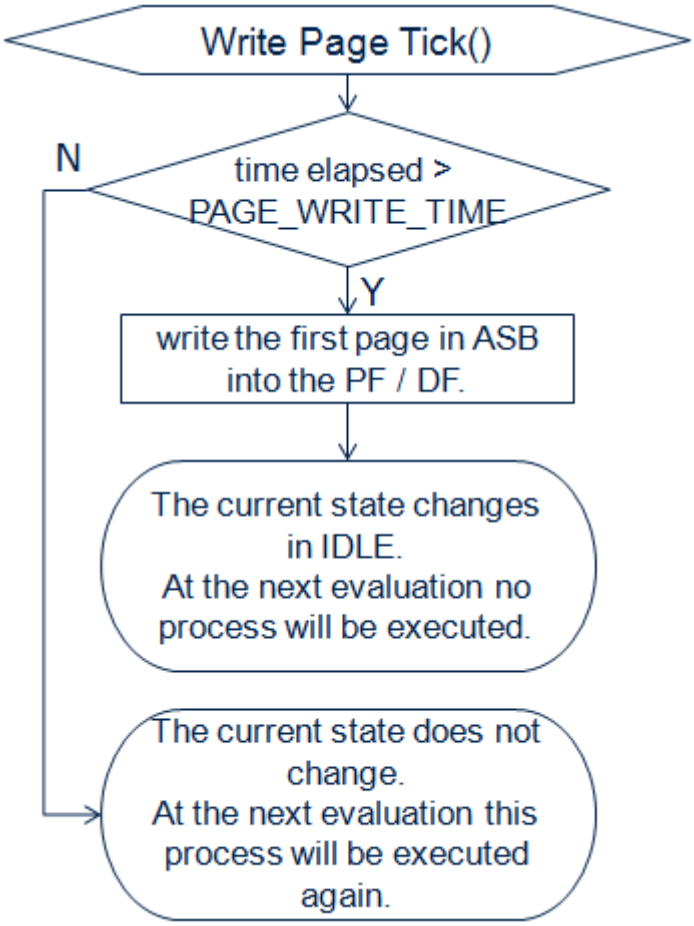

Fig. 5.21: Sequence performed during a "Write Page" command on the Flash Array.

This method is represented in Fig. 5.22. First of all, the page to program on the flash array is calculated by the address argument, thereafter for each word into the selected page the data from the assembly buffer is got, and after being filtered by the OR operation (refer to Tab. 5.4), it will be written into the flash array through the “Data” pointer of the “page” structure. Note that every “Data” element is one-byte so the word coming from the assembly buffer has to be divided into 4 bytes.

Previous Value	Next Value	OR
0	0	0
0	1	1
1	0	1
1	1	1

Tab. 5.4: Logic-OR operation can be used to calculate the new value in the flash array.

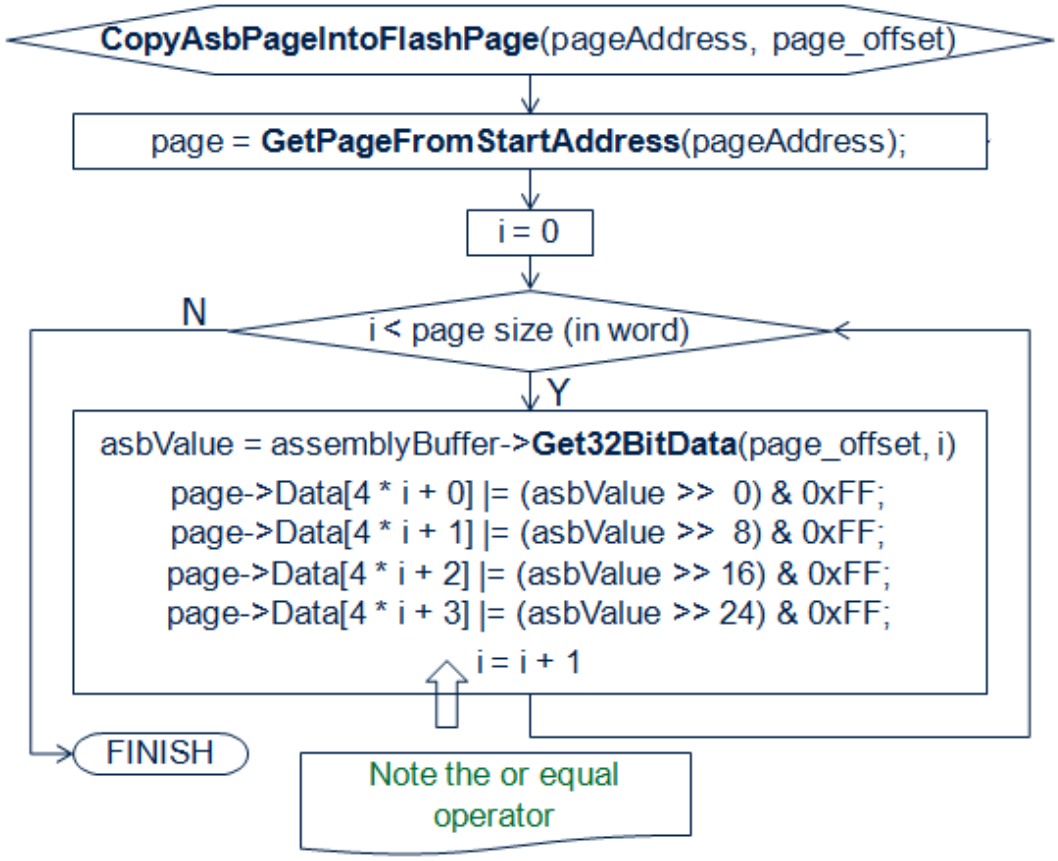

Fig. 5.22: Flow chart which describes the "Copy Assembly Buffer Page Into Flash Page" method.

A similar technique is adopted to erase the logical and the physical sectors. Obviously, the data have not to be read from assembly buffer (the sector of memory array has to be erased). The flow chart which describes the “Erase Logical (Physical) Sector” is reported in Fig. 5.23: each data into the selected sector, the reference of which is got by the sector address passed as argument, is set to “0”.

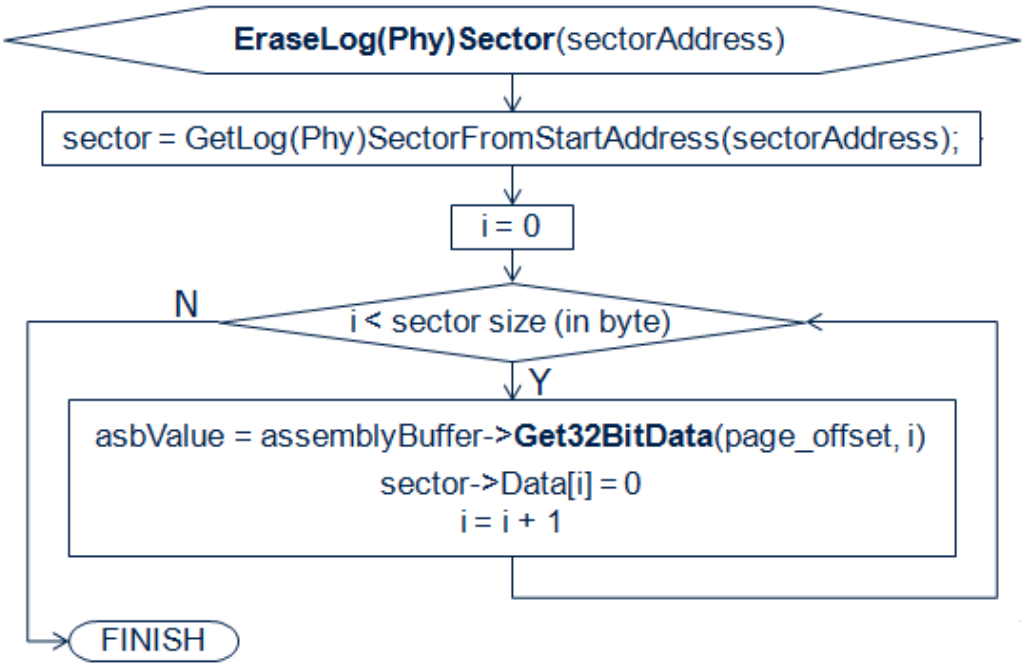

Fig. 5.23: Flow chart which describes the "Erase Logical (Physical) sector" method.

5.1.4.2 Save and read content on file

When a simulation starts, every flash bank inside the PMU has to read the content of its flash array from a file using a dedicated method.

This data is stored into the flash array before the simulation starts, and at the end it will be stored on the same file using another dedicated method. In this way manner the user is able to verify if any change on the memory content occurred.

The file structure, show in the text box below, is the same treated by the "Jazz" tool used in Infineon®.

```
<address>, <data>
<address>, <data>
...
```

5.1.5 Command Interpreter

The "Command Interpreter" is a single object instantiated inside the PMU, as show in Fig. 5.2. Its purpose is to detect the command sequence coming from the SRI bus through the "SCI6 - SRI Slave Port".

After the communication of the command interpreter, the flash module will execute the command sequence, involving assembly buffers, flash banks, flash arrays, register set and all other parts needed.

The flash module can recognize all the commands reported in the specs presented in 3.2.6.1, though with the following variants:

- Reset to Read
- Clear Status
- Enter in Page Mode P Flash
- Enter in Page Mode D Flash
- Load Page 32 bit
- Load Page 64 bit
- Write Page
- Write Page Once
- Write Burst
- Erase Logical Sector Range
- Erase Physical Sectors
- Verify Erased Physical Sector Range
- Resume Prog Erase

5.1.5.1 Command Sequence

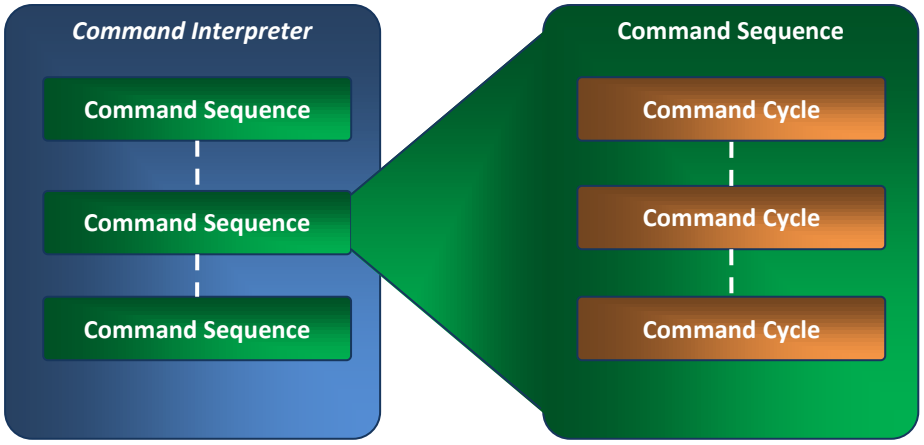

Fig. 5.24: "Command Sequences" contained into the "Command Interpreter".

The command interpreter makes use of a “Command Sequence” class instance for each command sequence it is able to detect. The “Command

Sequence" instance can be thought as a list of "Command Cycles" composing the definition of the Command Sequence. Fig. 5.24 depicts this structure.

A "Command Cycle" contains the following parameters:

- **Address**: specifies the value of the address signal coming from the SCI6 SRI port slave to match the command cycle;
- **Data**: specifies the value of data signal come from the SCI6 SRI port slave to match the command cycle;
- **Data Mask**: specifies the bits of the data signal that have to match with the Data variable above so that the command cycle will be recognized;
- **Data Type**: specifies the meaning of the Data value:
  - **CMD CYCLE**: the data signal has to match with a certain constant value which depends on the command sequence definition;
  - **CMD ADDR ARG**: the data signal is an address argument (e.g. for Write Page specifies the address to which store data);
  - **CMD DATA ARG**: the data signal is a data argument (e.g. for Load Page specifies the data to put in assembly buffer).

The set of all command cycles in a command sequence provides the command sequence's definition. These parameters, for all the command cycles of all the command sequences, are loaded from a dedicated INI file before the simulation starts.

5.1.5.2 Command interpreter's operation

The command interpreter is involved when SCI6 SRI Port receives a write transaction the address of which points in the data flash memory range. A dedicated method on PMU, called "SCI6 Sri Write", will be invoked by means of a member function pointer by the SRI port, as explained in 5.1.2.2. This method (on the PMU) will invoke the "Give Command Cycle" method on the "Command Interpreter" passing the address and the data values coming from SRI bus.

Therefore, the "Give Command Cycle" method is the first step computed to recognize a command sequence. It will invoke the method of the same name on each command sequence inside the command interpreter, which in turns deals to recognize whether the current command cycle belongs to that command sequence.

In order to explain the “Give Command Cycle” method on the “Command Sequence” object, it can be useful to keep in mind the variables that are defined within the cycle, which are:

- **Error**: indicates if the command sequence is in error, i.e. some command cycles previously given do not match with the related command cycle definition of the command sequence;
- **Cycle Index**: indicates the index of the last command cycle given;
- **Data Argument Caught**: the data argument which the SRI master has sent to the slave;
- **Address Argument Caught**: the address argument which the SRI master has sent to the slave.

A command sequence can be reset, so that it is not in error and the “cycle index” variable is set to “0”.

When a command cycle has been received, the “Give Command Cycle” method on the “Command Sequence” performs the algorithm depicted in Fig. 5.25 to detect if it belongs to the command sequence.

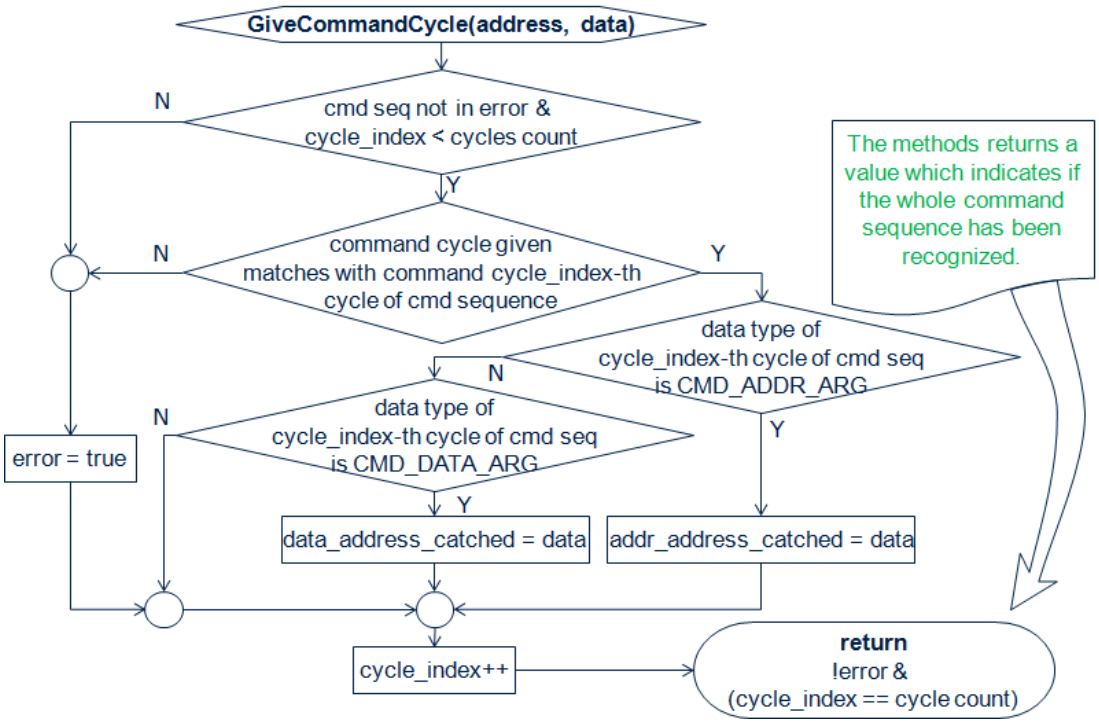

Fig. 5.25: Algorithm performed by the "Command Sequence" to recognize a command cycle.

If the command sequence is not in error (i.e. all the previous command cycles are recognized as belonging to that command sequence) and the number of cycles previously detected is less than the number of cycles belonging to the command sequence, and also if the current command cycle matches with the command cycle the command sequence is expecting, the data and / or address are eventually captured in the case they are arguments ("CMD CYCLE" in the cycle definition). Then the "cycle index" variable is incremented because a new command cycle has been recognized. Otherwise, if the current command cycle does not match with what the command sequence is expecting, the "error" flag will be asserted so that in the next steps the command sequence will not be able to be accepted.

When a command sequence has been accepted, the command interpreter resets all the "Command Sequence" instances, so that they will be able to begin a new interpretation.

The "Reset to Read" command sequence is the only one which is reset at the end of any given command cycle, so that it is always enabled to detect the "Reset to Read" command as provided by the specs (see 3.2.6.2).

Fig. 5.26 shows the code flow for the execution of a command sequence.

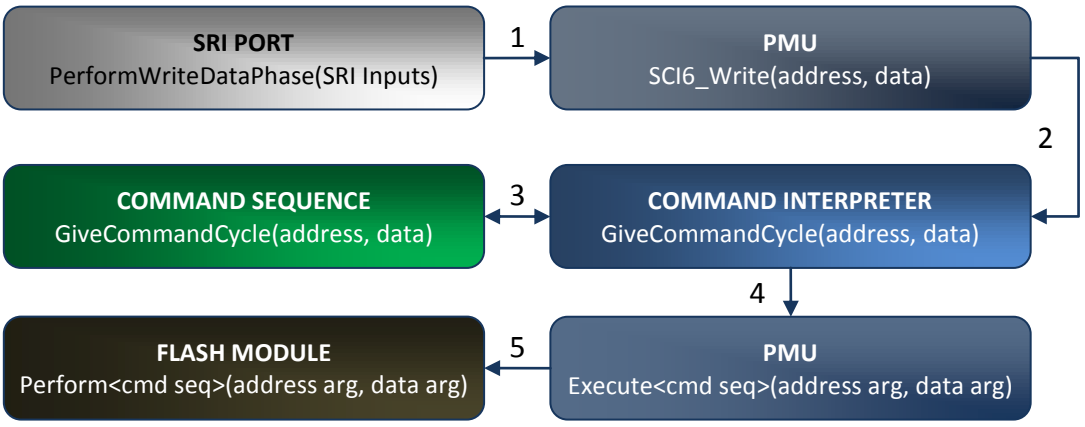

Fig. 5.26: Chronological execution of a command sequence.

5.1.6 Flash Module

The "Flash Module" is the manager of the whole PMU. Its purpose is to handle all the different classes described in this chapter in order to execute the command sequence.

Even though the functionalities of the flash module could be added inside the PMU class, it has been chosen to implement them as a separate object in order to improve the code readability and to make documentation clearer. Hence, the PMU

is just a simple container of all other instances, with the only feature to call the appropriate Flash Module methods when an operation has to be executed.

The flash module provides a method for each operation the PMU is able to perform, that is:

- Perform Reset to Read;
- Perform Clear Status;
- Perform Resume Prog Erase;
- Perform Erase Physical Sectors;
- Perform Erase Logical Sector Range;
- Perform Verify Erased Logical Sector Range;
- Perform Enter in Page Mode P Flash;
- Perform Enter in Page Mode D Flash;
- Perform Load Page 32 bit;
- Perform Load Page 64 bit;
- Perform Write Page;
- Perform Write Page Once;
- Perform Write Burst.

These methods accept the arguments they need (e.g. the “Perform Write Page” method accepts the page address argument; the “Erase Physical Sectors” accepts both the address and the number of sectors to erase).

These “performing methods” are invoked by the PMU through methods of the similar name: just the word “perform” is replaced with “execute”. They deal to recognize which instances have to be involved.

To give an example, the flow chart in Fig. 5.27 describes what the Flash Module computes in order to perform the “Write Page” command cycle.

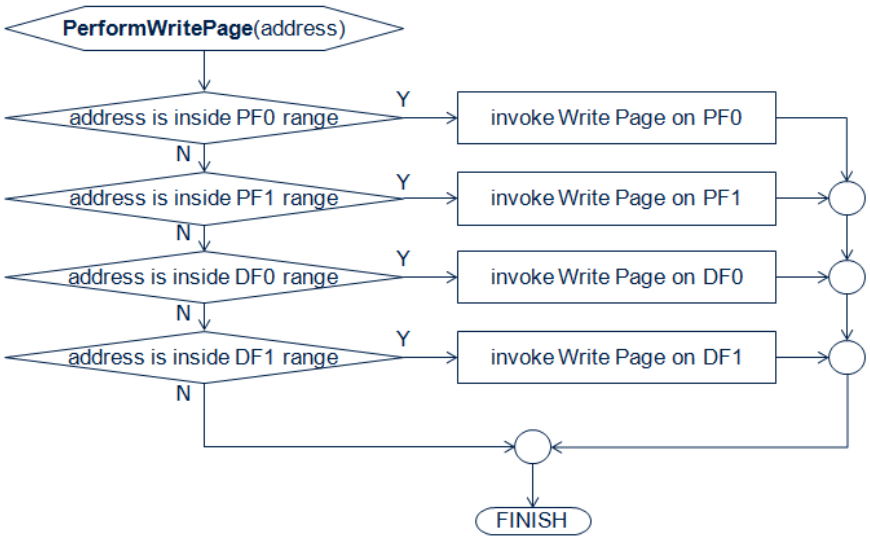

Fig. 5.27: Flow chart which describes the "Perform Write Page" method.

As mentioned in 5.1.4, since the flash bank has to take into account the passage of time during an operation, the PMU provides an evaluation method executed whenever at least one input signal changes.

```
if  (MARD.SPND)
{
      SUSPEND_REPLY  _pf0Reply=  pf0->Suspend();
      SUSPEND_REPLY  _pf1Reply=  pf1->Suspend();
      SUSPEND_REPLY  _df0Reply=  df0->Suspend();
      SUSPEND_REPLY  _df1Reply=  df1->Suspend();
      if  ( _pf0Reply == NO_OPERATION_PENDING &&
            _pf1Reply == NO_OPERATION_PENDING &&
            _df0Reply == NO_OPERATION_PENDING &&
            _df1Reply == NO_OPERATION_PENDING)
      {
            MARD.SPND = 0;
      }
}
```

Code 5.5: Suspend evaluation in flash module.

This method accomplishes also the evaluation for the flash module, which currently has the only purpose of detecting if the suspension of an operation has to be performed. This feature is reached by the dedicated “Evaluate Suspension” method on the flash module. Specifically, it controls the SPND’s status of the MARD register: if asserted the suspension begins for the current operation, calling the

"Suspend" method on the flash bank which is running the operation. The fragment Code 5.1 contains the instructions for the evaluation described above.

The "Suspend" methods on the flash banks return an instance of the enumeration "SUSPEND REPLY" which can assume one of the following values:

- NO OPERATION PENDING: suspension has not be performed because no operation is pending;
- CANNOT SUSPEND CURRENT OPERATION: suspension has not be performed because the operation is running cannot be suspended;
- OPERATION SUSPENDED: the operation was running has been suspended.

The second value is actually not used, but it has been defined for possible future improvements.

When an operation is suspended, the flash bank takes care to reset the SPND bit of the MARD register and to set the bit of the same name of the Flash Status Register. Moreover, it changes its own status depending on the current state:

- WRITING PAGE → WRITING PAGE SUSPENDED
- WRITING PAGE ONCE → WRITING PAGE ONCE SUSPENDED
- WRITING BURST → WRITING BURST SUSPENDED
- ERASING LOG SECTOR → ERASING LOG SECTOR SUSPENDED
- ERASING PHY SECTOR → ERASING PHY SECTOR SUSPENDED

If the flash bank is in a suspended state, its "Tick" method will perform no operation (see 5.1.4). Invoking "Resume" method on the flash bank the current state changes back and the suspended operation will start again.

5.2 Interface between CPU and PMU simulators

The interface between the two simulators consists in embedding the features of the Program Memory Unit into the project containing the CPU and the SRI bus. This can be accomplished by instantiating the PMU top module contained in the only Verilog file of the PMU project (see section 5.1) into the top module of the existing project.

As explained in section 3.3, the SRI bus is endowed of a crossbar which provides an interface between all the master and slave peripherals connected to the SRI system. The PMU can be connected to the crossbar through the "SCI Port", introduced in section 5.1.2.

Since the “SCI Port” module is developed comply with the SRI bus protocol, the connection with the crossbar is made by assigning all input and output signals of the “SCI Port” to the related crossbar ones. This goal is achieved in the Verilog top module of the entire microcontroller simulator, by assignment instructions.

The SRI bus protocol supports up to fifteen slaves and sixteen masters. At present only three masters are instantiated, the three CPUs of the TriCore®, and five slaves:

- The default slave, which complying with the SRI bus protocol, receives all the transactions having address out of the ranges of all the other slaves.
- An array memory used to test the CPU.
- The SCI6, SCI7 and SCI8 ports of the PMU object of this thesis.

Note that before this work was developed, the existed microcontroller was including a simple array memory which did not provide the PMU features. At present this memory has not been removed yet since it can be useful to test other features of the simulator.

Hence, with this work, just the addition of the Program Memory Unit has changed the configuration of the SRI system. This addition needed to update some settings. First of all the number of the SRI slaves is incremented at five. This setting, which is represented by a constant number, establishes the length of the array which contains the slave ports of the SRI crossbar. Moreover, in a special Verilog file, the address range of each slave is specified. In that context, only one address range can be specified per slave peripheral. For this reason the forbidden write transactions at the address range of the SCI7 and SCI8 slave are handled by the peripheral themselves by asserting the SRI bus, as explained in section 5.1.2.2 and as reported as example by Fig. 5.15.

Chapter 6 Test and Results

In this chapter some example are presented. In the first section, just the PMU simulator will be tested, whereas the examples of the PMU simulator interfaced with the CPU one will be explained in 5.2.

6.1 Example of use of the PMU simulator

The test bench provides to the device under test the same signals that the crossbar SRI would provide in a real scenario.

Fig. 6.1: Test bench schematic.

6.1.1 Example 1

In the first example a “Write Page” command is performed at the address 0xA000_0060 on the bank 0 of the Program Flash. According to section 5.1.4, 32 bytes must be loaded into the program flash assembly buffer. The data has been chosen randomly:

- 0xABCD_EF01
- 0x2345_6789
- 0xA0B1_C2D3
- 0xF9E8_D7E6
- 0xAB01_CD23
- 0xFE98_DC76

- 0xBA54_0101
- 0x34BE_56FC

Before running the simulator, the program flash is erased at all addresses within the range from 0xA0000_0060 to 0xA000_0080, i.e. all the addresses within the page will be programmed. In this context is useful to remember that for the program flash a page is 32 byte wide and addresses are aligned to byte.

The command sequences have to be executed are:

- “Enter in Page Mode P Flash”:
  - **Address**: 0xAF00_5554 **Data**: 0x50;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xABCD_EF01_2345_6789;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xA0B1_C2D3_F9E8_D7E6;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xAB01_CD23_FE98_DC76;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xBA54_0101_34BE_56FC;
- “Write Page”:
  - **Address**: 0xAF00_AA50 **Data**: 0xA000_0060;
  - **Address**: 0xAF00_AA58 **Data**: 0x00;
  - **Address**: 0xAF00_AAA8 **Data:** 0xA0;
  - **Address**: 0xAF00_AAA8 **Data**: 0xAA.

At the end of the simulation, the file containing data stored into the program flash bank 0 reports the values specified in Tab. 6.1. The addresses are aligned to 32 bit, so the first row reports the values for locations within the address range that goes from 0xA000_0060 to 0xA000_0063. The first two rows contain the values loaded into the assembly buffer by the first executed “Load Page 64 bit” command. This command is the same of the two “Load Page 32 bit” commands, in which the first has the least significant argument. For this reason, the values contained in the first two rows are swapped with respect to the command sequence’s data argument.

Address	Value
0xA000_0060	0x2345_6789
0xA000_0064	0xABCD_EF01

0xA000_0068	0xF9E8_D7E6
0xA000_006C	0xA0B1_C2D3
0xA000_0070	0xFE98_DC76
0xA000_0074	0xAB01_CD23
0xA000_0078	0x34BE_56FC
0xA000_007C	0xBA54_0101

Tab. 6.1: Data stored into the PF0 after the simulation was completed.

The simulator prints some information during the simulation, regarding the executed actions. These are reported in Info 6.1. By looking at the last two messages, it can be appreciated the time spent by the simulator to complete the Write Page operation: 20.0 units.

```
COMMAND SEQUENCE ENTER PAGE MODE P FLASH HAS BEEN RECEIVED: (Address: 0; Data: 0)
PFLASH ENTERS IN PAGE MODE:
      FSR: 0000_0200
(Sim. time: 4.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: ABCDEF0123456789)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 8.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: A0B1C2D3F9E8D7E6)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 9.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: AB01CD23FE98DC76)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 10.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: BA54010134BE56FC)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 11.500000)
COMMAND SEQUENCE WRITE PAGE HAS BEEN RECEIVED: (Address: A0000060; Data: 0)
WRITE PAGE(Address: A0000060)
      FSR: 0000_0088
(Sim. time: 27.500000)
WRITE PAGE COMPLETED ON PF0 AT ADDRESS 0xA0000060!
      FSR: 0000_0080
(Sim. time: 47.500000)
```

Info 6.1: Information provided during the simulation.

6.1.2 Example 2

The following example fills the page at address 0xA0000060 with the same data as

Example 1. During execution of Write Page command the MARD.SPND bit will be set to suspend the operation. Write Page will be resumed after a few time through “Resume Prog Erase” command sequence. The command sequences have to be executed are:

- “Enter in Page Mode P Flash”:
  - **Address**: 0xAF00_5554 **Data**: 0x50;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xABCD_EF01_2345_6789;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xA0B1_C2D3_F9E8_D7E6;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xAB01_CD23_FE98_DC76;
- “Load Page 64 bit”:
  - **Address**: 0xAF00_55F0 **Data**: 0xBA54_0101_34BE_56FC;
- “Write Page”:
  - **Address**: 0xAF00_AA50 **Data**: 0xA000_0060;
  - **Address**: 0xAF00_AA58 **Data**: 0x00;
  - **Address**: 0xAF00_AAA8 **Data:** 0xA0;
  - **Address**: 0xAF00_AAA8 **Data**: 0xAA.
- **After a bit of time the SPND bit of MARD register will be set**
- “Resume Prog Erase” (after a bit of time)
  - **Address**: 0xAF00_AA50 **Data**: 0xA000_0060;
  - **Address**: 0xAF00_AA58 **Data**: 0x00;
  - **Address**: 0xAF00_AAA8 **Data:** 0x70;
  - **Address**: 0xAF00_AAA8 **Data**: 0xCC.

The messages printed during the simulation are described in the following.

The “Enter in Page Mode P Flash” command sequence switches the mode of the program flash assembly buffer, so that the PFPAGE bit of the flash status register is asserted.

```
COMMAND SEQUENCE ENTER PAGE MODE P FLASH HAS BEEN RECEIVED: (Address: 0; Data: 0)
PFLASH ENTERS IN PAGE MODE:
      FSR: 0000_0200
(Sim. time: 4.500000)
```

Info 6.2: FSR.PFPAGE bit is set after "Enter in Page Mode".

The four “Load Page 64 bit” command sequences fill the program flash assembly buffer with data passed as argument. The Flash Status Register does not change its value.

```
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: ABCDEF0123456789)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 8.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: A0B1C2D3F9E8D7E6)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 9.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: AB01CD23FE98DC76)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 10.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: BA54010134BE56FC)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 11.500000)
```

Info 6.3: The assembly buffer is filled via "Load Page 64 bit" command sequences.

When the command sequence “Write Page” is given the page mode ends so the PFPAGE bit of the Flash status register is cleared. The programming operation begins, so the BUSY bit is set for the program flash bank 0 and the PROG bit is set to flags a programming operation is running.

```
COMMAND SEQUENCE WRITE PAGE HAS BEEN RECEIVED: (Address: A0000060; Data: 0)
WRITE PAGE(Address: A0000060)
      FSR: 0000_0088
(Sim. time: 27.500000)
```

Info 6.4: FSR.P0BUSY, FSR.PROG are set and FSR.PFPAGE is cleared then the "Write Page" begins.

In order to suspend the operation that is running, the SPND bit of the flash status register is set via the SCI6 port: a SRI transaction at MARD's address writes the value 0x8.

At the same time, the flash module evaluation method begins the suspension of the program flash bank 0. The MARD.SPND bit is automatically cleared and the SPND of the Flash Status Register is set by the flash bank. Furthermore, also the P0BUSY is cleared, because the flash bank is not busy again.

```
Write value on register MARD: 0000_0008
(Sim. time: 41.500000)

SUSPEND REQUESTED:
Write Page suspended for PF0
      FSR: 0800_0080
      MARD: 0000_0000
(Sim. time: 41.500000)
```

Info 6.5: FSR.SPND bit is set by writing data directly to SCI6 port.

In this example, no other operations are performed during the suspension of the "Write Page". Clearly, it is not a useful scenario, but it could happen. The P0BUSY bit is set again, because the flash bank is running the operation again.

```
COMMAND SEQUENCE RESUME PROG ERASE HAS BEEN RECEIVED: (Address: A0000060; Data: 0)
RESUME PROG ERASE:
      FSR: 0000_0088
(Sim. time: 67.500000)
```

Info 6.6: The "Resume Prog Erase" will be given to resume the suspended operation.

After a bit of time the write page ends. The actual time taken to perform the entire operation is 20.0 units as the previous example in which the suspension was not involved. In fact the operation has started at 27.5 units of time (uof) and it has been suspended at 41.5 uof. So in the first step the actual time taken is 14.0 uof. After suspension, the operation is resumed at 67.5 uof and definitively ends at 73.5 uof so that the second step takes 16.0 uof. Therefore, the sum gives exactly 20.0 uof.

```
WRITE PAGE COMPLETED ON PF0 AT ADDRESS 0xA0000060!
      FSR: 0000_0080
(Sim. time: 73.500000)
```

Info 6.7: The "Write Page" is completed after 20.0 actual units of time.

6.1.3 Example 3

This example enhances the previous adding an “Erase Physical Sectors” command sequence on data flash bank 0 during the suspension of the “Write Page” command on the program flash 0 bank.

The Command Sequence added during the time span when the MARD.SPND bit is set and the command “Resume Prog Erase” is given is:

- “Erase Physical Sectors”
  - **Address**: 0xAF00_AA50 **Data**: AF00_0000;
  - **Address**: 0xAF00_AA58 **Data**: 0x01;
  - **Address**: 0xAF00_AAA8 **Data:** 0x80;
  - **Address**: 0xAF00_AAA8 **Data**: 0x5A.

Indeed, it has to be expected an error since, as reported in 3.2.6.8, an erasing operation is not allowed while a programming operation has been suspended. In fact, as it can be appreciated in Info 6.8, the simulator replies with a sequence error on the Flash Status Register.

```
COMMAND SEQUENCE WRITE PAGE HAS BEEN RECEIVED: (Address: A0000060; Data: 0)
WRITE PAGE(Address: A0000060)
      FSR: 0000_0088
(Sim. time: 27.500000)
Write value on register MARD: 0000_0008
(Sim. time: 41.500000)
SUSPEND REQUESTED:
Write Page suspended for PF0
      FSR: 0800_0080
      MARD: 0000_0000
(Sim. time: 41.500000)
COMMAND SEQUENCE ERASE PHYSICAL SECTORS HAS BEEN RECEIVED: (Address: AF000000; Data: 1)
      !*** DFLASH: Erase operation is not allowed while a command sequence is suspended.
      FSR: 0800_1080
(Sim. time: 67.500000)
COMMAND SEQUENCE RESUME PROG ERASE HAS BEEN RECEIVED: (Address: A00000A0; Data: 0)
RESUME PROG ERASE:
      FSR: 0000_1088
(Sim. time: 163.500000)
WRITE PAGE COMPLETED ON PF0 AT ADDRESS 0xA0000060!
      FSR: 0000_1080
```

```
(Sim. time: 169.500000)
```

Info 6.8: A sequence error is given because the suspension is not allowed.

6.1.4 Example 4

In this example the “Write Page” and the “Erase Physical Sectors” commands in the previous Example 3 are swapped.

Command sequences are not reported in order to focus the attention on the simulator’s behavior during the suspension of the erase command.

```
COMMAND SEQUENCE ERASE PHYSICAL SECTORS HAS BEEN RECEIVED: (Address: AF000000; Data: 1)
ERASE PHYSICAL SECTORS:
    FSR: 0000_0302
    FSR: 0000_0302
(Sim. time: 27.500000)
Write value on register MARD: 0000_0008
(Sim. time: 71.500000)
SUSPEND REQUESTED:
Erase Physical Sector suspended for DF0
(Sim. time: 71.500000)
COMMAND SEQUENCE WRITE PAGE HAS BEEN RECEIVED: (Address: A0000060; Data: 0)
WRITE PAGE(Address: A0000060)
    FSR: 0800_0188
(Sim. time: 127.500000)
WRITE PAGE COMPLETED ON PF0 AT ADDRESS 0xA0000060!
    FSR: 0800_0180
(Sim. time: 147.500000)
COMMAND SEQUENCE RESUME PROG ERASE HAS BEEN RECEIVED: (Address: AF000000; Data: 1)
RESUME PROG ERASE:
Erase Physical Sector resumed for DF0
    FSR: 0000_0182
(Sim. time: 163.500000)
ERASE PHYSICAL SECTOR COMPLETED ON DF0. LAST SECTOR'S ADDRESS: 0xAF000000
    FSR: 0000_0180
(Sim. time: 199.500000)
```

Info 6.9: the programming operation is performed on the PF0 when an erasing operation has been suspended for DF0.

6.1.5 Example 5

In this new example a four Load Page command sequences are sent while both assembly buffers are in read mode.

```
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: ABCDEF0123456789)
LOAD PAGE (64 bit):
    !*** PFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    !*** DFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    FSR: 0000_1000
(Sim. time: 4.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: A0B1C2D3F9E8D7E6)
LOAD PAGE (64 bit):
    !*** PFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    !*** DFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    FSR: 0000_1000
(Sim. time: 5.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: AB01CD23FE98DC76)
LOAD PAGE (64 bit):
    !*** PFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    !*** DFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    FSR: 0000_1000
(Sim. time: 6.500000)
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: BA54010134BE56FC)
LOAD PAGE (64 bit):
    !*** PFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    !*** DFLASH: Load Page (64 bit) sent when assembly buffer not in page mode.
    FSR: 0000_1000
(Sim. time: 7.500000)
```

Info 6.10: The simulator sets the sequence error because a "Load Page" command is requested when neither of the assembly buffers is in page mode.

6.1.6 Example 6

In this example a “Write Burst” command is executed after the assembly buffer has been completely filled.

The “Load Page 64 bit” is executed 32 times to fill the whole Program Flash Assembly Buffer.

```
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: ABCDEF0123456790)
LOAD PAGE (64 bit):
    FSR: 0000_0200
(Sim. time: 57.500000)

COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: AB01CD23FE98E376)
```

```
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 59.500000)

…

COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: BA54010134C556FC)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 60.500000)
```

Info 6.11: "Load Page 64 bit" command sequences fill the program flash assembly buffer.

```
COMMAND SEQUENCE WRITE BURST HAS BEEN RECEIVED: (Address: A00000A0; Data: 0)
WRITE BURST:
      FSR: 0000_0088
(Sim. time: 76.500000)
```

Info 6.12: Page mode is left when the "Write Burst" command begins.

```
WRITE BURST COMPLETED ON PF0. LAST PAGE'S ADDRESS: 0xA0000180
      FSR: 0000_0080
(Sim. time: 236.500000)
```

Info 6.13: At the end of the programming operation the P0BUSY bit of the Flash Status Register is cleared.

6.1.7 Example 7

This example is similar to the previous one, but the assembly buffer is overloaded before the “Write Burst” is given. The programming operation will be executed, but the SQER bit on the Flash Status Register is set. The simulator prints a message to inform the user on the reason why the sequence error has occurred.

```
COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: ABCDEF0123456790)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 57.500000)

COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: AB01CD23FE98E376)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 59.500000)
```

```
…

COMMAND SEQUENCE 64 BIT LOAD PAGE HAS BEEN RECEIVED: (Address: 0; Data: ABCDEF0123456789)
LOAD PAGE (64 bit):
      FSR: 0000_0200
(Sim. time: 64.500000)
```

Info 6.14: The "Load Page 64 bit" command sequence is executed enough time to overload the assembly buffer.

```
COMMAND SEQUENCE WRITE BURST HAS BEEN RECEIVED: (Address: A00000A0; Data: 0)
WRITE BURST:
      !*** PFLASH: Write operation with assembly buffer overflowed.
      FSR: 0000_1088
(Sim. time: 80.500000)
```

Info 6.15: "Write Burst" is performed with data contained into assembly buffer but SQER is set.

6.2 Example of use of the entire simulator

The simulator, as mentioned in Chapter 1, is intended to verify the correctness of executions of the test applications developed by the testware team.

A further example, which involves the whole microcontroller simulator, will be explained now.

The simulation concern a source code in which just the command sequence “Reset To Read” is specified. The code can be compiled by choosing the memory segment of the microcontroller where the instructions have to be loaded at the boot before to be executed. The segment can be specified through special options on the before the compiler runs.

The results of the simulation are reported in Info 6.16.

```
SREC: store.sre
SREC: Start: c0000000
FlashWidth=8, HitLatency=0, Prefetch=0, ExtraDelay=0, AddrChangeLag=0, Image=store1.1.sre

void PMU_Sim::SRI::SRI_Slave::EvaluateWriteArbitrationPhase(PMU_Sim::SRI::SRI_INPUTS)
> SCI6 Arbitration Phase for Write Transaction at address: AF005554     OK
> SCI6 SRIWriteTransactionParams enqueued(Address: AF005554; TransactionID: 0;
DataPhaseLength: 1)

void PMU_Sim::SRI::SRI_Slave::PerformWriteDataPhase(PMU_Sim::SRI::SRI_INPUTS)
> SCI6 Data sent(Address: AF005554; Data: F0; Transaction Id: 0)
> SCI6 Write transaction at address into the DF0 range.
```

```
void PMU_Sim::Command_Interpreter::CommandInterpreter::ExecuteResetToRead()
COMMAND SEQUENCE RESET TO READ HAS BEEN SENT: (Address: 0; Data: 0)
RESET TO READ:
>       PFlash Assembly Buffer Reset To Read
>       FSR: 0000_0000
>       PFlash Assembly Buffer Reset To Read
>       FSR: 0000_0000
>       DFlash Assembly Buffer Reset To Read
>       FSR: 0000_0000
>       DFlash Assembly Buffer Reset To Read
>       FSR: 0000_0000
      FSR: 0000_0000
(Sim. time: 55.000000)

void PMU_Sim::SRI::SRI_Slave::CheckIfWriteDequeue(PMU_Sim::SRI::SRI_INPUTS)
> SCI6 SRI Write dequeue.
(Sim. time: 55.000000)
```

Info 6.16: Simulation result of the "Reset To Read" command sequence.

The section in Info 6.16 which is highlighted in green contains information provided by the simulator of the CPU. Precisely “Image = store1.1.sre” gives the name of the file which contains the assembly instruction (firmware or testware) which the simulation processes. The string “Start = 0xC000_0000” provides the start address of the program counter. This parameter can be set by an external file, to a value different of the default.

The section in Info 6.16 which is highlighted in blue reports to the user that the “Evaluate Write Arbitration Phase” method (see the 5.1.2.2) is executed, so the arbitration phase is performed by adding into the “write queue” a transaction with address “AF005554” and the associated transaction identifier is “0”.

The section in Info 6.16 which is highlighted in gray reports that the “Perform Write Data Phase” method (see the 5.1.2.2) is executed, so the “Data Phase” is performed for the transaction in the queue (which in this case is the only one) with the transaction identifier equals to “0”. The write transaction contains the data argument “0xF0”.

The “Command Interpreter” (see section 5.1.5) decodes the received transaction and recognizes the command sequence “Reset To Read”. Hence it invokes the “Flash Module” (see the section 5.1.6) which perform the command. This information is reported in the yellow section in Info 6.16.

In the brown section in Info 6.16 the simulator reports that the transaction is removed from the “write queue”.

6.3 Conclusion

The ability of running a firmware which involves the Program Memory Unit has been demonstrated with the example in section 6.2.

Nevertheless, the ability to analyze the entire testware is affected by a problem due to an instruction executed before the actual testware takes place. This instruction concerns an initialization of a microcontroller peripheral, which has not been implemented yet in the simulator.

As will be explained in Chapter 7, the execution of this initialization, at present, is avoided in the simulation. However, a future improvement could be implementing the related peripheral, reaching so the actual ability of testing the testware.

Chapter 7 Future improvements

The last step to achieve the goal of this thesis is the development of the interface between the PMU simulator with the CPU one. In this way, the ability of executing the firmware is achieved for the whole microcontroller simulator, in particular the Infineon's Padua team will be able to test the testware.

Nevertheless, a problem subsists to reach the actual ability to analyze the entire testware. In fact, some initializations of a part of the microcontroller, which has not been implemented yet, are computed by the testware interrupting its execution.

The guilty microcontroller's part is the System Control Unit (SCU), which contains several control registers associated with other system functions, such as:

- External Request and Cross-triggering Unit (ERU)
- CPU Lockstep Comparator Logic (LCL)
- Die Temperature Sensor (DTS)
- Watchdog Timers (WDTx)
- Emergency Stop (EMS)
- Logic Built-in-Self-Test (LBIST)
- Overlay Control (OVC)
- Miscellaneous System Control Registers
- SCU register overview table

In this context the features of these functions will not be treated. The execution of the simulation stops when the instructions involving the SCU have to be executed. At present, the technique adopted as a workaround consists in omitting these procedures, avoiding so the involvement of the System Control Unit while keeping the operations computed on the flash banks valid.

The first improvement of the whole microcontroller's simulator could be the implementation of the System Control Unit, which, in the TC27x sub-units scenario represent the most important among the reminder.

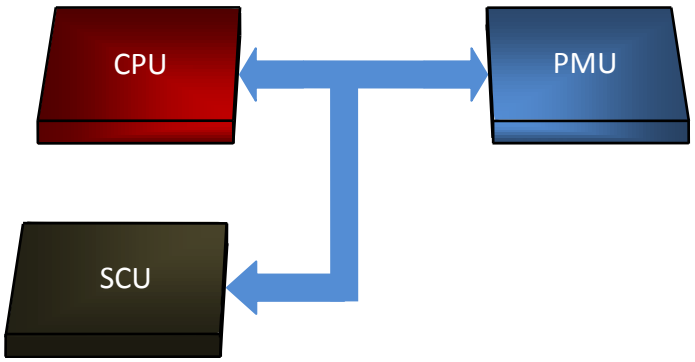

Fig. 7.1: Interface with the System Control Unit.

7.1 Watchdog timer

The implementation of the System Control Unit would enhance the microcontroller's simulator, such as enabling it to manage the Watchdog Timers.

A watchdog timer – which in this context is the object responsible of the interruption of the simulations – is a typical feature in a microcontroller. It provides a mechanism able to reset the CPU (by resetting the program counter) if a fault of the software execution occurs.

In principle the watchdog timer could be treated as a countdown timer which should not expire, otherwise a reset of the microcontroller will be performed. Therefore, an application has to avoid the timer expires by resetting its value periodically.

A typical application of the watchdog timer is in a firmware which has to execute a loop for a long time. In general, in a firmware the actual application is contained into an infinite loop, so that it is permanently executed. The watchdog timer is typically enabled at the beginning of the infinite cycle by setting the relative registers, and within the loop it is reset. If an error occurs and the execution is broken, the watchdog timer expires and the microcontroller will be restarted. The user of the watchdog feature has to take care to calculate the duration of one cycle in order to avoid erroneously expire of the timer.

Several microcontrollers enrich this basic mechanism by adding features, such as enabling the user to decide the timer frequency and change it at run time; or the

presence of more than one watchdog timer, which is very useful in all the applications executing parallel computing.

7.2 Improvements of Program Memory Unit

Presently, the most important features of the Program Memory Unit have been enabled. In fact, to achieve the goal of testing the "testware" and remembering that the goal of the testware is to discover possible malfunctioning of the Program Memory Unit, it can be asserted that the implementation of the PMU's simulator is accurate enough.

Nevertheless, some registers have not been implemented yet. A future improvement could be, certainly, the implementation of all the remaining features.

7.3 Graphic User Interface

The simulator provided is not equipped of a graphic user interface, so reading the state of the registers and giving commands to the simulator could be challenge, since the handles are represented by the command terminal. Therefore, a future improvement could be endowing the simulator of a graphic user interface.

Bibliography

[1] Infineon web site. [Online]. www.infineon.com

[2] A. P. Godse and D. A. Godse, *Microcontrollers*. Technical Publications Pune, 2008.

[3] H. Sulzer, "Microcontrollers," in *Semiconductors*. Infineon Technologies, 2004, ch. 7, pp. 266,270,308-314.

[4] P. Olivo and E. Zanoni, "Flash Memories: an overview," in *Flash Memories*. Kluwer Academic Publishers, 1999, vol. 1, pp. 7-26.

[5] E. Suzuki, H. Hirashi, K. Ishii, and Y. Hayashi, *A low-voltage alterable EEPROM with metal-oxide-nitride-oxide semiconductor structure.* IEEE Trans. Electron Devices, 1993, vol. ED-30.

[6] E. Suzuki, K. Miura, H. Hayashi, R. .-P. Tsay, and D. Schroder, *Hole and electron current transport in metal-oxide-nitride-oxide-silicon memory structures.* IEEE Trans. Electron Devices, 1989, vol. 36.

[7] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, "Flash Memory Cells - An Overview," Proceedings of the IEEE, VOL. 85, August, 1997.

[8] S. T. Wang, "On the I–V characteristics of floating-gate MOS transistors," in *IEEE transaction on electron devices*, 1979, vol. 26, p. 1292–1294,1979.

[9] M. Wada, S. Mimura, N. Nihira, and H. Iizuka, "Limiting factors for programming EPROM of reduced dimensions," 1980.

[10] K. Prall, W. I. Kinney, and J. Marco, "Characterization and suppression of drain coupling in submicrometer EPROM cells," vol. ED-34, no 12, 1987.

[11] M. Wong, D. K. .-Y. Liu, and S. S. .-W. Huang, "Analisys of the subthreshold slope and the linear transconductance techniques for the extraction of the capacitive coupling coefficients of floating-gate devices.," in

IEEE Electron Device Letters. 566-568, 1992, vol. 13, ch. 8.

[12] W. L. Choi and D. M. Kim, *A new technique for measuring coupling coefficients and 3-D capacitance characterization of floating gate devices*. IEEE Trans. Electron Devices, 1994, vol. 41.

[13] R. Bez, E. Camerlenghi, D. Cantarelli, L. Ravazzi, and G. Crisenza, "A novel method for the experimental determination of the coupling ratios in submicron EPROM and Flash EEPROM cells," 1999.

[14] B. Moison, C. Papadas, G. Ghibaudo, P. Mortini, and G. Pananakakis, *New method for the extraction of the coupling raratios in FLOTOX EPROM cells*. IEEE Trans. Electron Devices, 1993, vol. 40.

[15] M. Woods, "An E-PROM's integrity starts with its cell structure," in *Nonvolatile Semiconductor Memories: Technologies Design, and Application,*. IEEE Press, 1991, ch. 3, pp. 59-62.

[16] P. E. Cottrel, R. R. Troutman, and T. H. Ning, "Hot-electron emission in n-channel IGFET's," in *IEEE Transaction on Electron Devices*, 1979, vol. ED-26, ch. 4, p. 520–532.

[17] B. Eitan and D. Froham-Bentchkowsky, *Hot-electron injection into the oxide in n-channel MOS devices*. IEEE Trans. Electron Devices, 1981, vol. ED-28.

[18] G. A. Baraff, *Distribution functions and ionization rates for hot-electrons in semiconductors*. 1962, vol. 128.

[19] H. Chenming, "Lucky-electron model for channel hot-electron emission," University of California, 1979.

[20] S. Tam, P. K. Ko, C. Hu, and R. Muller, *Correlation between substrate and gate currents in MOSFET's*. IEEE Trans. Electron Devices, 1982, vol. ED-29.

[21] E. Takeda, H. Kune, T. Toyabe, and S. Asai, *Submicrometer MOSFET structure for minimizing hot-carrier generation*. IEEE Trans. Electron Devices, 1982, vol. ED-29.

[22] K. Hess and T. C. Sah, *Hot carriers in Silicon surface inversion layers*. IEEE Trans. Electron Devices, 1974, vol. 45.

[23] K. R. Hofmann, C. Wemer, W. Weber, and G. Dorda, *Hotelectrons and hole-emission effects in short n-channel MOSFET's*. IEEE Trans Electron Devices, 1985, vol. ED-32.

[24] L. Esaki, "Long journey into tunneling," in *PROCEEDINGS OF THE IEEE*.

Proc. IEEE, 1974, vol. 62, pp. 825-831.

[25] J. Moll, *Physics of Semiconductors*, McGraw-Hill, Ed. New York, 1964.

[26] M. Lezlinger and E. H. Snow, *Fowler-Nordheim tunneling into thermally grown SiO2*. J. Application Physics, 1969.

[27] D. Khang and S. M. Sze, *A floating gate and its application to memory devices*. Bell Syst. Technologies.

[28] F.-B. D., *A fully decoded 2048-bit electrically programmable MOS-ROM*. IEEE ISSCC, 1971, vol. 14.

[29] H. A. R. Wegener, et al., *The variable threshold transistor, a newly electrically alterable nondesctructive read-only storage device*. Electron Devices, IEEE Transaction on, 1967, vol. 15.

[30] E. Harari, L. Schmitz, B. Troutman, and S. Wang, "A 256-bit non-volatile static RAM," in *Solid-State Circuits Conference.* IEEE International, 1978, vol. 21, ch. 9, pp. 108-109.

[31] W. S. Johnson, G. Perlegos, A. Renninger, G. Kuhn, and T. Ranganath, "A 16 Knit electrically erasable nonvolatile memory," 1980.

[32] J. F. Scott, "Basic Properties of Ferroelectrics: Bulk Materials," in *Ferroelectric Memories*, S.-V. B. Heidelberg, Ed. Cambridge, 2000, pp. 2,18-22.

[33] D. C. Guterman, I. H. Rimawi, T. L. Chiu, R. D. Halvorson, and D. J. McElroy, "An electrically alterable non-volatile memory cell using a floating gate structure," in *Electron Devices*, ch. 26, p. 576.

[34] F. Masuoka, M. Asano, H. Iwahashi, T. Komuro, and S. Tanaka, "A new Flash EEPROM cell using triple polysilicon technology," 1984.

[35] M. Branchetti, et al., "Memory Architecture and Related Issues," in *Flash Memories*. ST Microelectronics, ch. 5.

[36] M. Pasotti, et al., "Analog Sense Amplifiers for high density NOR Flash Memories," STMicroelectronics - Innovative Systems Design Group - Central R&D 10.1109/CICC.1999.777284 , 1999.

[37] Infineon Technologies, "Aurix TC27x (with deltas) Internal Target Specification V3.5 2013-01," Data Sheet, 2013-01.

[38] Infineon Technologies, "Aurix Family - SRI bus protocol Specification V2.0D1," V2.0D1, 2010-02.

[39] T. Yong Meng, T. Seng, and K. Siew Teng, "Structured Parallel Simulation

Modeling and Programming," National University of Singapore, 1998.

[40] F. J. Vithayathil, "Hybrid Computer Simulation of WIND-DRIVE ocean currents," Oregon State University, 1974.

[41] L. Tun, G. Yang, and L. Si-Kun, "Design and Implementation of a Parallel Verilog Simulator: PVSim," National University of Defense Technology, 2004.

[42] IEEE, "Behavioral Languages - Verilog(R) hardware description.," in *INTERNATIONAL STANDARD*, 2004, ch. 4, pp. 23-29.

[43] W. Snyder, D. Galbi, and P. Wasson. Veripool / Verilator. [Online]. http://www.veripool.org

[44] http://en.wikipedia.org/wiki/Microcontroller.

[45] (2009) Integrated Circuit Engineering Corporation. [Online]. http://smithsonianchips.si.edu/

[46] J. Fraden, "Pyroelectric Thermometers," in *The measurement, instrumentation, and sensors handbook*, S. V. G. &. C. KG, Ed. CRC Press LLC, 1999, vol. 1, ch. 32, pp. 32-109.