Università di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Elettronica

Sviluppo di un sistema per il test di memorie NAND e PCM basato su FPGA

(Development of an FPGA-based testing system of NAND and PCM memories)

Laureando:
Davide Giovanelli

Relatore:
Prof. Alessandro Paccagnella

Anno Accademico 2010/2011

Sommario

Tutti i dispositivi a semiconduttore sono costantemente sottoposti ad irraggiamenti di particelle energetiche di varia natura; le fonti principali sono: onde elettromagnetiche irradiate dai circuiti presenti nelle vicinanze, radiazioni cosmiche e particelle emesse chip e dai package in cui i circuiti di silicio sono contenuti. In determinate condizioni questi irraggiamenti possono essere causa di malfunzionamenti e, soprattutto nei circuiti che sfruttano le tecnologie CMOS, possono generare problemi non trascurabili.

Il gruppo di ricerca RREACT (Reliability and Radiation Effects on Advanced CMOS Technologies) del Dipartimento di Ingegneria dell'Informazione si occupa dello studio degli effetti che questi irraggiamenti possono avere sui dispositivi CMOS. Attraverso test durante i quali i dispositivi vengono fatti lavorare in ambienti molto sfavorevoli (irraggiamenti di varie particelle ad alta energia che vengono effettuati in vari modi tra cui anche utilizzando gli acceleratori di particelle dei laboratori INFN), si studiano i problemi che queste energie hanno sulle varie tipologie di circuiti (principalmente FPGA e memorie). I risultati di queste ricerche sono importanti per capire quali circuiti sono più sensibili, quali meno agli irraggiamenti ai quali sono esposti ed eventualmente prendere le misure necessarie per aumentarne l'affidabilità.

Questa tesi descrive il lavoro di tirocinio svolto da me all'interno del gruppo di ricerca RREACT; il mio compito è stato quello di realizzare, utilizzando una board fornita di un circuito FPGA della Xilinx, un sistema per il test di due tipi di memorie, Flash NAND e PCM. Tale sistema deve permettere di scrivere una determinata parola (pattern) in tutte le celle comprese in un range di indirizzi della memoria ed essere poi in grado di verificare se il contenuto della cella cambia a causa degli irraggiamenti fatti, il tutto gestito attraverso personal computer. Per quanto riguarda la Flash esisteva già un sistema per testarle basato su una scheda con microcontrollore fornita dal produttore delle memorie; questa scheda è però costosa, poco versatile e non molto affidabile in ambienti ostili come possono essere quelli dove si realizzano gli irraggiamenti. Per questi motivi si è vista la necessità di realizzare un sistema adattabile a più tipi di memorie, più affidabile ed economico; i dispositivi FPGA sono perciò sembrati quelli con le caratteristiche più favorevoli a questa realizzazione. Per quanto riguarda invece la PCM non esisteva nessun sistema per il test, in quanto si è iniziato da poco lo studio degli effetti degli irraggiamenti su questa nuova tecnologia di memorie.

Nel primo capitolo verranno esposti i dispositivi presi in considerazione durante il lavoro, saranno quindi descritte in maniera abbastanza generale le tipologie di memorie non volatili sotto test (Flash NAND e PCM) e i dispositivi FPGA con i quali si è realizzato il sistema. Il secondo capitolo sarà invece dedicato alle cause e agli effetti che gli irraggiamenti di particelle ad alta energia possono avere sui circuiti CMOS, partendo dai primi problemi osservati sulle DRAM negli anni '70 fino a quelli che al giorno d'oggi sono maggior fonte di errori. Nel terzo capitolo viene descritto il sistema di test realizzato; usando VHDL, `C` e `C++` sono arrivato a scrivere parecchie migliaia di righe di codice e risulta ovviamente improponibile commentarlo interamente. Verranno perciò riportate e descritte solo le parti più interessanti di ogni blocco del sistema. Queste dovrebbero essere sufficienti per capire la metodologia con cui si è arrivati al risultato finale. Sono state realizzate due implementazioni del sistema, nella prima le comunicazioni col PC avvengono utilizzando la porta seriale e quindi in maniera abbastanza semplice; in una seconda, ho provato a spostare le comunicazioni sulla USB. Saranno quindi commentate le modifiche apportate al sistema.
Verranno infine tratte le conclusioni sul tirocinio, descritte le maggiori difficoltà riscontrate durante la realizzazione e i punti su cui è necessario lavorare ancora.

Indice

Sommario **iv**

1 Dispositivi **1**
1.1 FPGA . . . 1
1.1.1 Generalità . . . 1
1.1.2 Struttura interna . . . 1
1.2 Memorie non volatili . . . 4
1.2.1 Flash NAND . . . 4
1.2.2 PCM . . . 5
1.2.3 Tabella comparativa delle memorie . . . 8

2 Errori nei sistemi critici **9**
2.1 Sistemi critici . . . 9
2.2 Compatibilità elettromagnetica . . . 10
2.3 Soft Error . . . 11
2.3.1 Generalità . . . 11
2.3.2 Il problema dei Package . . . 12
2.3.3 Radiazioni cosmiche . . . 13
2.3.4 Prospettive future . . . 13

3 Sistema di test per memorie **17**
3.1 Introduzione . . . 17
3.2 Strumenti hardware e software utilizzati . . . 17
3.3 Sistema da realizzare . . . 18
3.3.1 Visione generale . . . 18
3.3.2 Operazioni da eseguire sulle memorie . . . 19
3.4 Codice sviluppato (prima implementazione) . . . 22
3.4.1 VHDL per la periferica (memoria NAND) . . . 22
3.4.2 VHDL per la periferica (memoria PCM) . . . 32
3.4.3 Codice C μblaze . . . 32
3.4.4 Risultati parziali . . . 36
3.5 Codice sviluppato (seconda implementazione) - Interfacciamento del sistema tramite USB . . . 37
3.5.1 Firmware per il cy7c67300 . . . 38
3.5.2 Modifiche apportate all'applicazione per il μblaze . . . 40
3.5.3 Programma per la gestione del sistema dal PC: . . . 42

4 Problemi riscontrati, risultati ottenuti, conclusioni **43**
4.1 Prima implementazione 43
4.2 Seconda implementazione 43
4.3 Foto varie 44
4.4 Conclusioni 45

Bibliografia **46**

Capitolo 1

Dispositivi

1.1 FPGA

1.1.1 Generalità

I dispositivi FPGA - Field Programmable Gate Array sono dei dispositivi programmabili utilizzati in elettronica digitale per la realizzazione di circuiti logici anche molto complessi ideati negli anni ’80. La loro introduzione sul mercato è avvenuta nel 1985 con il XC2064 della Xilinx. Al proprio interno essi contengono un gran numero di unità logiche programmabili e di segnali di instradamento, con i quali è possibile connettere le varie parti del chip e farlo comunicare con l’esterno. La particolarità di questi dispositivi, a differenza di quelli tipo PIC[1], è che quando sono programmati per eseguire una funzione logica non utilizzano una ALU per eseguirla ma viene realmente generato al loro interno un vero circuito logico, con porte and, nor, xor e anche con elementi base di memoria. Tale procedimento è equivalente a realizzare il circuito tramite componenti discreti su una basetta PCB[2]. Utilizzando un FPGA invece di saldare componenti discreti su una basetta si deve scrivere codice al computer, ottenendo circuiti molto più compatti, con minor consumo e con prestazioni decisamente superiori.

1.1.2 Struttura interna

Un dispositivo FPGA può essere visto come una grande matrice di blocchi logici connessi tra di loro con delle connessioni programmabili (matrici di commutazione) le quali consentono una grande quantità di combinazioni di connessione tra i blocchi (vedi Figura 1.1). Ogni blocco logico[3] contiene al suo interno uno o più sottoblocchi chiamati *slice*. Nel caso in cui ci siano più slice in un CLB esistono ulteriori segnali interni per poterle connettere senza ricorrere ai segnali globali e alle matrici di commutazione. Esistono vari tipi di slice: alcune possono essere configurate per essere utilizzate come elementi di memoria (RAM distribuita),

[1] i PIC sono una famiglia di circuiti integrati con funzioni di microcontrollore che si possono vedere come dei microprocessori per PC un poco meno performanti

[2] PCB - Printed Circuit Board ovvero circuito stampato

[3] I blocchi logici vengono anche chiamati CLB (Configurable Logic Block)

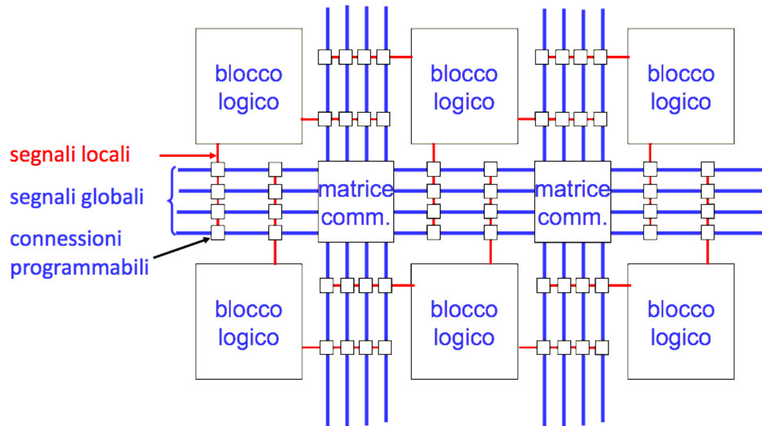

Figura 1.1: schema base di collegamento dei CLB tramite matrice di connessioni all'interno di dispositivi FPGA

o possono essere più adatte a realizzare sommatori grazie a circuiti dedicati per il riporto; il loro comportamento rimane comunque simile. Come si vede dalla Figura 1.2, che riporta lo schema interno di un CLB, ogni slice presenta al suo interno una cella chiamata LUT[4], anche in questo caso ci potranno essere più LUT in una *slice*. Le LUT, che sono la parte più interna di un FPGA, implementano un semplice circuito programmabile in grado di eseguire qualsiasi funzione logica ad x ingressi, dove x è il numero totale dei suoi ingressi[5].

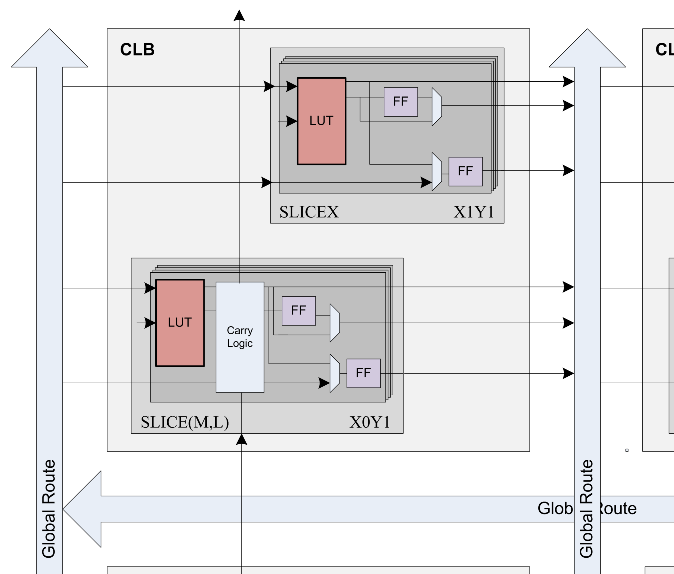

Figura 1.2: schema interno di un CLB all'interno di un'FPGA

Un'altra peculiarità di questo tipo di dispositivi è quella di poter essere riprogrammati un numero elevatissimo di volte in quanto la configurazione che memorizza il circuito logico che l'FPGA implementa (bitstream) è contenuta all'interno del chip in delle memorie (solitamente memorie volatili SRAM o simili).

[4] LUT - Look Up Table.

[5] Il numero degli ingressi delle LUT varia da modello a modello di FPGA ma è comunque nell'ordine di 4-6, quindi per creare funzioni con più ingressi verranno collegate due o più LUT anche di slice o CLB diversi.

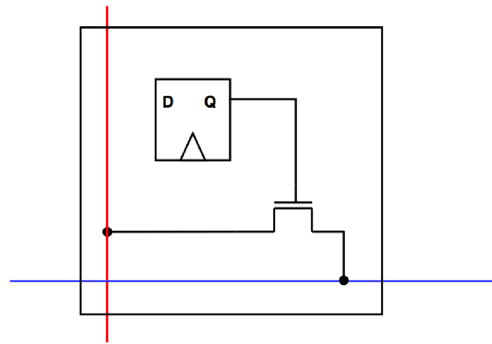

Figura 1.3: particolare di una matrice di commutazione, l'elemento di memoria (flip flop) memorizza la posizione della connessione e il transistor nMOS si occupa di effettuare o meno il cortocircuito tra i due segnali.

Il limite al numero di riprogrammazioni è determinato dalla vita della memoria. In alcuni chip sono disponibili anche memorie non volatili, col vantaggio di mantenere la programmazione anche in assenza di alimentazione esterna.
Per poter utilizzare un FPGA è quindi necessario caricare all'interno della sua memoria il circuito che si vuole riprodurre sotto forma di bitstream. Tale bitstream che non è altro che una successione di bit che imposta le funzionalità di ogni componente del FPGA; si impostano, ad esempio, le configurazioni delle connessioni all'interno delle matrici di commutazione (Figura 1.3), le funzioni booleane che tutte le LUT devono eseguire e i setteggi di tutti gli altri circuiti di cui è eventualmente fornito il dispositivo FPGA (sommatori, memorie ecc). Resta ora solo il problema di come creare tale bitstream. Durante i primi anni in cui si utilizzavano questi dispositivi veniva creato disegnando al computer il circuito che si voleva creare e questo era poi convertito in sequenza di bit con dei tool specifici. Attualmente questo metodo è stato quasi del tutto abbandonato e si preferisce usare il linguaggio VHDL (oppure il verilog) grazie al quale, utilizzando una sintassi specifica, è possibile descrivere il comportamento di un qualsiasi circuito logico; utilizzando poi dei software simili a dei compilatori la descrizione testuale viene trasformata in bitstream pronto ad essere caricato sul FPGA.

La complessità dei circuiti realizzabili con gli FPGA è molto spinta: si pensi che un XC5VLX110 della Xilinx contiene più di 17000 slices ed ha circa 800 pin di I/O! I possibili utilizzi sono ulteriormente allargati dal fatto che è possibile creare al loro interno dei veri e propri sistemi di calcolo composti da softprocessor (μblaze[6]), memoria, periferiche, sui quali è possibile far girare programmi scritti in `C` o addirittura interi sistemi operativi (principalmente versioni embedded di Linux), compilati appositamente per questo tipo di architettura.

[6] il μblaze è un softprocessor, cioè un microcontrollore creato all'interno dell'FPGA utilizzando i componenti programmabili del chip, i suoi componenti (registri, ALU, ecc) sono creati programmando in maniera adeguata i blocchi logici

1.2 Memorie non volatili

1.2.1 Flash NAND

Le memorie di tipo NAND si basano su una delle architetture più semplici con cui realizzare memorie digitali. All'interno della famiglia delle memorie NAND se ne possono distinguere due tipi principali: quelle di sola lettura, ROM, e quelle che possono essere lette e anche scritte più volte, dette FLASH. La struttura interna delle celle è pressoché identica e la differenza sta nel tipo di transitor di cui sono composte; nelle ROM vengono utilizzati normali MOS (solitamente del tipo ad arricchimento a canale n) mentre le FLASH sono composte da MOS a gate flottante.

Lo schema base di una memoria NAND è riportato in Figura 1.4. Nello specifico questa è una memoria di sola lettura (ROM). Per spiegarne il funzionamento si ipotizzi di dover leggere la parola $k = 2$; inizialmente, quando il segnale di clock è basso, i transistor pMOS precaricano al valore 1 tutti i nodi BL_x (questi sono le bit-line, cioè le linee di uscita da cui leggere il dato), dopodiché si pongono a 1 tutte le Word Line WL_i con $i \neq k = 2$ e a zero WL_2. Così facendo BL_0 e BL_2 si scaricano a massa, mentre BL_1 e BL_3 mantengono la tensione di precarica nelle capacità parassite, in quanto i transistor collegati alla WL_1 sono spenti e quindi interrompono il percorso a bassa impedenza verso massa. Ora è possibile estrarre il valore memorizzato $WL_2 = 0101$ leggendo le tensioni dei nodi BL_x con l'ausilio di appositi circuiti chiamati *sense amplifier*[7].

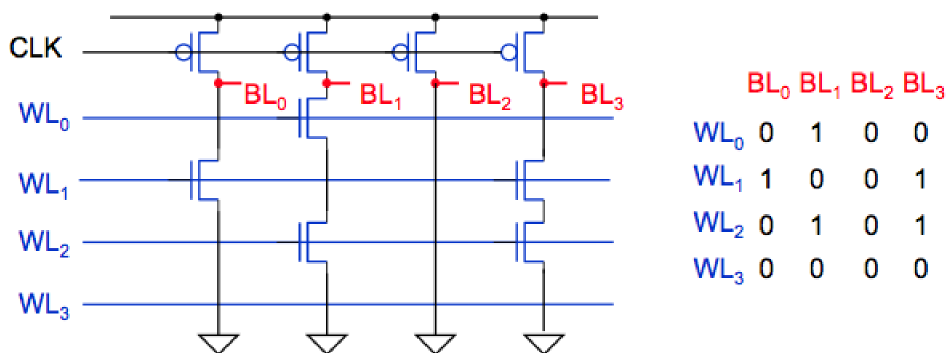

Figura 1.4: schema di una memoria di tipo NAND ROM in grado di tenere in memoria 4 parole da 4 bit ciascuna

Quindi se ne deduce che, per costruire una memoria di questo tipo, bisogna inserire un transistor nMOS, con il gate comandato dalla word-line corrispondente, sulle bit-line che devono memorizzare il valore 1 per quella determinata parola e non inserirlo invece dove il valore deve essere 0. Ovviamente, questo tipo di memoria, una volta costruito (e quindi creati i transistor), non può essere cambiata, quindi i valori memorizzati possono solo essere letti e non scritti.

Nelle memorie Flash i transitor nMOS vengono sostituiti da nMOS con *floating gate*. In questo caso non c'è bisogno di scegliere dove mettere oppure dove

[7]I *sense amplifier* servono per eliminare gli effetti delle perdite di carica nelle capacità di *storage* dovute ad elementi parassiti.

non mettere i transistor nMOS; saranno difatti messi su tutti gli incroci tra word-line e bit-line. Sarà poi la carica contenuta nel gate flottante a determinare se un transistor potrà scaricare a massa (quindi memorizzare uno 0) oppure tenere alta la tensione (quindi memorizzare un 1) una *BL*. Si può dire quindi che l'informazione in questo caso non è più associata alla presenza o meno del MOS, ma è associata alla quantità di carica contenuta nel gate flottante.

Mosfet a gate flottante

I *floating gate* mosfet (FG mosfet) sono un tipo di transistor derivati direttamente dai mosfet tradizionali. La loro particolarità sta nella presenza di un secondo gate (detto appunto *floating gate*) posto tra il canale e il gate vero e proprio (chiamato in questo caso control gate) Figura 1.5. Questo elettrodo è elettricamente isolato

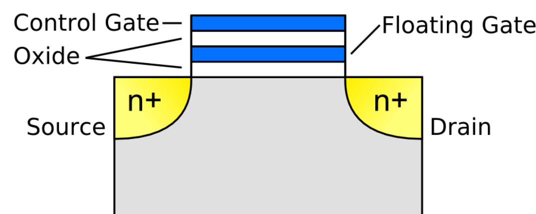

Figura 1.5: struttura di un nMos a gate flottante, si vede la presenza del gate intermedio isolato tramite due strati di ossido sia dal substrato che dal gate di controllo

da qualsiasi altra parte del chip. Questo significa che se viene posta una carica elettrica su di esso questa permane senza variare nel tempo (almeno in teoria). Dato che su un normale mosfet la conduzione è controllata dalla quantità di carica presente sull'elettrodo di gate, risulta immediato intuire che la presenza di una carica nella zona intermedia tra gate e canale farà cambiare le caratteristiche di conduzione del dispositivo. In particolare, la carica determina la tensione di soglia del transistor. Quindi, a parità di tensioni applicate tra i terminali (V_{gs} e V_{ds}) si avrà una corrente I_{ds} diversa a seconda di quanta carica è immagazzinata nel gate flottante. Questa differenza di comportamento nelle memorie viene tradotta in zero o uno permettendo la memorizzazione di un bit.
In anni più recenti sono state introdotte memorie Flash in grado di memorizzare 2 bit per ogni transistor, raddoppiando di fatto la capacità del chip a parità di area.

1.2.2 PCM

Le memorie a cambiamento di fase[8] sono una tecnologia relativamente nuova per la costruzione di memorie non volatili. Anche se gli studi sui materiali che le compongono sono iniziati già negli anni sessanta, si è riusciti solo recentemente (metà anni novanta) ad ottenere prestazioni tali da poter competere con i dispositivi Flash.
Le PCM si basano sulla capacità di cambiamento di stato dei materiali calcogenuri[9]. In particolare viene utilizzata una lega composta da Germanio (Ge),

[8] In inglese vengono chiamate Phase Change Memory da cui PCM o anche PCRAM.
[9] I calcogenuri sono una categoria di vetri.

Antimonio (Sb) e Tellurio (Te) chiamata GST. Questa lega è in grado, sotto determinate condizioni di temperatura, di cambiare il proprio stato da cristallino ad amorfo, e viceversa, che dal punto di vista elettronico viene tradotto in una variazione della resistività, bassa nello stato cristallino e alta in quello amorfo[10]. Per ottenere le condizioni di temperature necessarie al cambiamento di stato si sfrutta l'effetto joule generato dal passaggio di corrente. Nello specifico, se si porta la regione di programmazione alla temperatura di fusione per un breve periodo di tempo, seguita da un rapido raffreddamento, si mette il materiale in stato amorfo (reset della cella PCM). Per impostare invece lo stato cristallino (set della cella) bisogna invece portare il materiale ad una temperatura intermedia tra quella di fusione e quella di cristallizzazione per un tempo necessario alla cristallizzazione. Come mostrato dalla Figura 1.6 (b) il tempo di set è più lungo rispetto a quello di reset, quindi sarà questo a determinare la velocità della memoria nel caso peggiore.

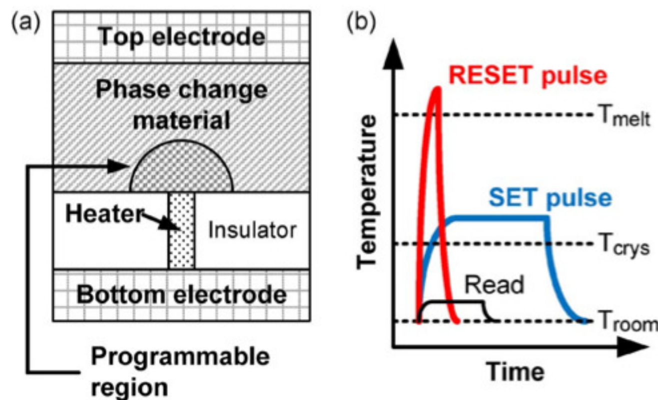

Figura 1.6: (a) Sezione della struttura di una cella PCM, questa tipologia viene chiamata cella mashroom (cella fungo).
(b) Impulsi di temperatura necessari per le operazioni di RESET, SET e Read della cella.

Nella Figura 1.6 (a) è schematizzata la struttura di una cella PCM; i due elettrodi (Top electrode e Bottom electrode) sono raggiungibili dall'esterno e sono i terminali tra i quali si dovrà misurare l'impedenza, per determinare se lo stato è amorfo o cristallino. Il funzionamento della cella è semplice: si possono vedere le tre regioni disegnate (Phase change material, Programmable region e Heater) come tre resistenze in serie. Quelle delle regioni Heater e di Phase change material (in fase di costruzione quest'ultima viene posta nello stato cristallino) sono relativamente basse e costanti nel tempo, mentre la resistenza equivalente della regione programmabile dipenderà dallo stato di cristallizzazione in cui si trova. La resistenza totale tra Top e Bottom sarà quindi determinata dallo stato in cui si trova la Programmable region.

[10]Questa proprietà di cambiamento di stato da parte dei calcogenuri viene sfruttata anche sui supporti CD-RW (CD riscrivibili) o nei recenti Blu-ray; in queste applicazione il cambiamento di fase del materiale si traduce in una variazione del coefficiente di riflessione che può quindi essere distinta come uno 0 o un 1 dal laser del lettore.

Per quanto riguarda la lettura basterà porre ai capi della cella una tensione e determinarne la corrente. Tramite l'utilizzo di sense amplifier (o simili) i circuiti di controllo della memoria potranno valutare se la resistenza è tale che lo stato sia amorfo, che solitamente rappresenta uno 0, oppure cristallino, che viceversa rappresenta un 1. Durante il processo di lettura l'effetto joule prodotto dalla corrente circolante tra Top e Bottom potrebbe far variare il contenuto della cella. In fase di progetto bisognerà perciò porre attenzione a questo e far in modo che la tensione di lettura sia adeguata, quindi indicativamente che non superi la tensione di soglia V_{th} (vedi Figura 1.7).

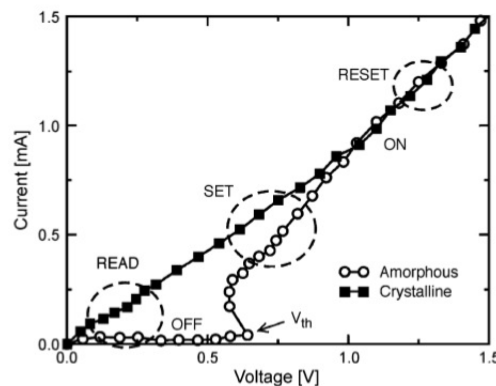

Figura 1.7: Caratteristica V-I di una cella PCM nei due stati, si nota che per tensioni basse (quelle utilizzate durante le operazioni di lettura) la differenza di impedenza è abbastanza marcata tra lo stato amorfo e quello cristallino

In Figura 1.7 è mostrata la caratteristica tensione-corrente di una cella PCM. Per basse tensioni (zona di READ) si nota la diversità di conduzione tra gli stati cristallino e amorfo. Se la tensione supera la soglia V_{th} si passa per uno stato a resistenza differenziale negativa e si arriva alla zona di SET. Se questa è raggiunta per tempi molto brevi (inferiori al tempo di cristallizzazione) lo stato non cambia; se invece la tensione permane per più tempo, lo stato della cella diventa cristallino, qualsiasi fosse quello precedente. Aumentando ulteriormente la tensione si arriva alla zona di RESET dove cioè il GST diventa amorfo (solo però se avviene un rapido raffreddamento).

Illustrato il procedimento utilizzato per la memorizzazione di un bit, estendere il ragionamento ad un array non risulta difficile; una struttura base per poter costruire una memoria PCM è molto simile a quella vista per le NAND. Serviranno perciò delle word-line che identificano quale parola si vuole leggere (cioè l'indirizzo) e delle bit-line che daranno in uscita il valore memorizzato di tale parola. Si farà quindi una matrice come quella riportata in Figura 1.8; in una direzione viaggeranno le WL e trasversalmente ci saranno le BL. Le une saranno collegate alle altre in ogni incrocio attraverso gli elementi di memoria (cioè le singole celle PCM) messi in serie ad un dispositivo di selezione. Con una struttura siffatta per leggere la parola k-esima basterà porre la relativa word-line a uno logico ($WL_k = 1$) e abilitare tutte le celle connesse ad essa. Questa caricherà ad uno le bit-line i cui elementi di memoria sono nello stato cristallino (bassa resistività). Le altre BL saranno tenute a zero da una resistenza di pull-down (non disegnata in Figura 1.8).

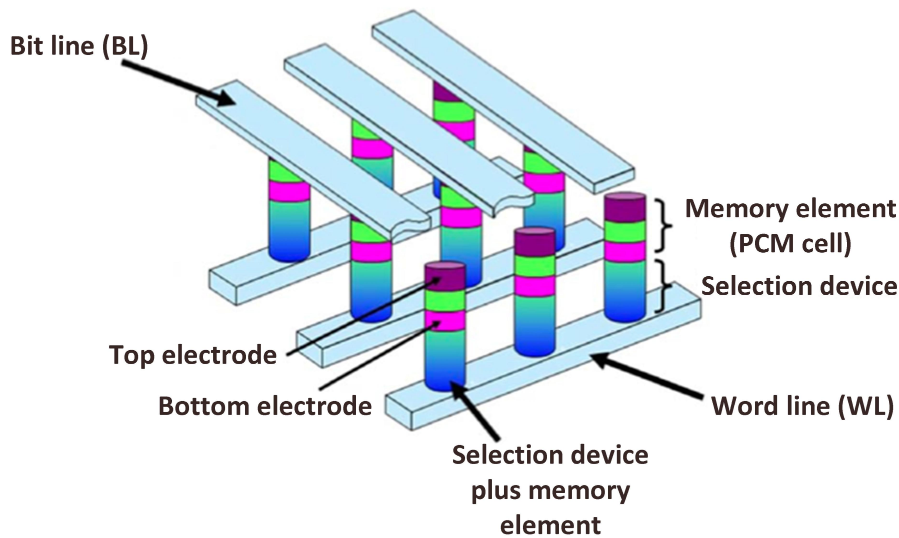

Figura 1.8: Struttura semplificata di una memoria basata su celle PCM.

1.2.3 Tabella comparativa delle memorie

Viene qui proposta una tabella comparativa tra vari tipi di supporti per la memorizzazione presenti in commercio. I dati sono stati presi da un paper della Numonix quindi potrebbero riportare valori troppo ottimistici in alcuni casi e pessimistici in altri; è utile però per capire gli ordini di grandezza in gioco.

COMPARISON OF HIGH-DENSITY MEMORY TECHNOLOGIES

Attributes	DRAM	PCM	NAND	MLC NAND	HDD
Non-Volatile	No	Yes	Yes	Yes	Yes
Erase Required	Bit	Bit	Block	Block	Sector
Software	Simple	Simple	Complex	Very Complex	Simple
Power	~W/GB	100→500mW/die	~100mW/die	~100mW/die	~10W
Write Bandwidth	~GB/s	1→100+ MB/s/die	10→100 MB/s/die	~10 MB/s/die	200→400MB/s
Write Latency	~20-50ns	~1µs	~100µs	~800µs	~10ms
Write Energy	~0.1nJ/b	<1nJ/b	0.1-1nJ/b	<1nJ/b	>10nJ/b
Read Latency	50ns	50 - 100 ns	10-25 µs	25-50 µs	~10ms
Read Energy	~0.1nJ/b	<<1nJ/b	<<1nJ/b	<<1nJ/b	>10nJ/b
Idle Power	~W/GB	<<0.1W	<<0.1W	<<0.1W	<10W
Endurance	∞	10^8	$10^5 \to 10^4$	$10^4 \to$?	∞

Figura 1.9: Tabella comparativa di alcuni tipi di memorie.

Capitolo 2

Errori nei sistemi critici

2.1 Sistemi critici

Al giorno d'oggi sono ormai pochi gli oggetti/accessori che non includono dei circuiti elettronici. Si pensi solamente a quello che normalmente una persona indossa o porta con se durante una giornata: telefonino, orologio digitale, carte di credito (incorporano tutte un chip), agende elettroniche e molto altro. Ci sono inoltre molti altri dispositivi, di cui molti ignorano addirittura l'esistenza, che operano in condizioni particolarmente impegnative e che magari, per la natura del compito che svolgono, non possono essere soggette a guasti o errori improvvisi; ad esempio il circuito di controllo dei freni di una automobile moderna, oppure i componenti elettrici di un'apparecchiatura biomedica o anche un computer che controlla il volo, il decollo e l'atterraggio di un aereo di linea. Se uno di questi circuiti cessasse, anche per in breve periodo, di funzionare correttamente le conseguenze potrebbero essere molto gravi con danni a cose e persone. Si parla quindi di sistemi critici quando un'apparecchiatura si trova in un contesto nel quale un malfunzionamento non è tollerabile.
Di che natura sono però gli errori e i malfunzionamenti di cui si sta parlando? Le cause possono essere innumerevoli (escludendo naturalmente errori di progettazione) e di seguito verranno discusse quelle principali.

Negli ultimi decenni si sta andando verso una sempre maggiore miniaturizzazione dei dispositivi CMOS con un'aumento esponenziale della densità spaziale di circuiti. Questo trend è giustificato dal fatto che dispositivi più piccoli permettono maggiore velocità, minori perdite, riduzione dei costi e aumento delle prestazioni. Si generano però non pochi problemi legati alla compatibilità elettromagnetica (EMC - Electro Magnetic Compatibility) tra elementi posti molto vicini tra di loro.
Un'altra causa di errori (più pericolosa della precedente) sono le particelle ad alta energia, in gran parte provenienti dallo spazio, che possono andare a colpire il die di silicio; se queste collisioni avvengono nel punto e nel momento sbagliato riescono a creare correnti impulsive tali da poter cambiare il livello logico di uno o più bit, creando non pochi problemi; tutto ciò vale sia che si stia parlando di circuiti combinatori che di circuiti di memoria.

In entrambi i casi (errori causati da onde elettromagnetiche o da particelle) i tipi di problemi cui si va incontro all'interno di circuiti digitali possono venire

divisi in due categorie e sono chiamati hard errors, non trattati qui, e soft errors di cui si parlerà più approfonditamente nelle sezioni successive.

2.2 Compatibilità elettromagnetica

Come è ben noto, ogni circuito nel quale circoli una corrente elettrica genera attorno a se dei campi elettrici e/o magnetici ed è a sua volta sensibile agli stessi campi che lo circondano. Si parla di suscettibilità ed emissione irradiata quando l'energia elettromagnetica è trasmessa o ricevuta attraverso l'aria e suscettibilità ed emissioni condotte quando la trasmissione avviene attraverso un mezzo come un cavo. Nell'immagine Figura 2.1 viene schematizzato il meccanismo con cui una corrente circolante in un dispositivo può influire sul funzionamento di un altro attraverso accoppiamenti capacitivi-induttivi e attraverso l'irraggiamento (questo fenomeno è a volte chiamato *crosstalk*). Tutto ciò può valere sia a livello di grossi circuiti sia a livello di piccoli chip (anche se comunque il fenomeno è meno pericoloso di quello generato dalle particelle ad alta energia di cui si parlerà nei prossimi paragrafi).

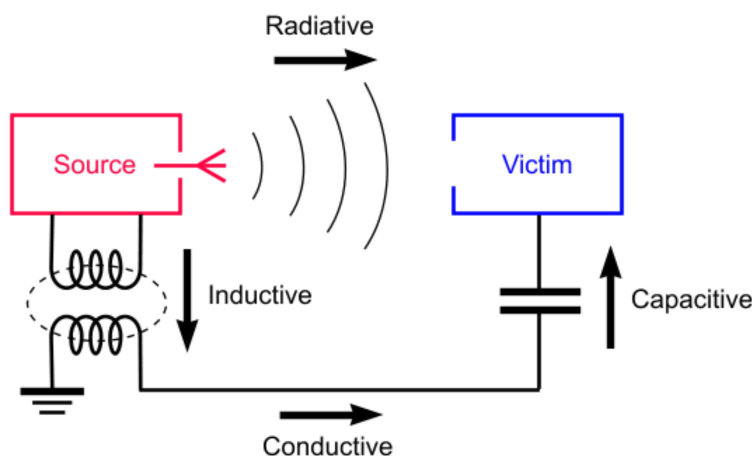

Figura 2.1: Vari modi di accoppiamento tra circuiti limitrofi

Fin dall'inizio del '900, con le prime trasmissioni radio, si iniziò a discutere di questo tipo di problemi legati alle energie elettromagnetiche e andando avanti negli anni, con l'inarrestabile corsa alle innovazioni tecnologiche, le questioni aumentarono. Furono quindi istituiti degli organi preposti a regolamentare questo campo, sia dal punto di emissione (quanti disturbi un'apparecchio può emanare) che da quello di immunità (quanti disturbi un'apparecchio deve saper sopportare). All'inizio le normative erano valide solo a livello nazionale, ma con l'aumento dell'interscambio commerciale si è sentita la necessità di una regolamentazione a livello internazionale. A tal scopo, con la nascita dell'Unione Europea, le normative sono state unificate, obbligando (negli scambi all'interno dell'EU) all'applicazione del simbolo CE, che indica che quel prodotto è conforme alle direttive europee (tra qui quelle riguardanti compatibilità elettromagnetica).

Gli organismi principali che si occupano di tale regolamentazioni sono:

- In l'italia:

  - CEI Comitato Elettrotecnico Italiano
  - IMQ Istituto per il marchio di qualità
  - SEV Association for Electrical Engineering, Power and Information Technologies
  - Ispesl Istituto Superiore Prevenzione e Sicurezza sul Lavoro

- A livello internazionale

  - CISPR (Comité International Spécial des Perturbations Radioéelectriques)
  - ETSI (European Telecommunications Standards Institute)
  - CENELEC (Comité Européen de Normalisation Electrotechnique)
  - BSI (British Standards Institute) per l'Inghilterra
  - FCC (Federal Communications Commission) per gli Stati Uniti

Per quanto riguarda le emissioni irradiate (che interessano di più quando si parla di soft errors) si riporta come esempio in Figura 2.2 un estratto della normativa CISPR 14-1[1] che si occupa dei disturbi a radiofrequenza irradiati da dispositivi casalinghi. Esistono poi normative simili, come le EN61000-3 per quanto

	Household and similar appliances		Tools					
1	2	3	4	5	6	7	8	9
Frequency range			Rated motor power not exceeding 700 W		Rated motor power above 700 W and not exceeding 1 000 W		Rated motor power above 1 000 W	
(MHz)	dB (pW) Quasi-peak	dB (pW) Average[a]	dB (pW) Quasi-peak	dB (pW) Average [a]	dB (pW) Quasi-peak	dB (pW) Average [a]	dB (pW) Quasi-peak	dB (pW) Average [a]
30 to 300	Increasing linearly with the frequency from:							
	45 to 55	35 to 45	45 to 55	35 to 45	49 to 59	39 to 49	55 to 65	45 to 55

a If the limit for the measurement with the average detector is met when using a receiver with a quasi-peak detector, the equipment under test shall be deemed to meet both limits and the measurement using the receiver with an average detector need not be carried out.

Figura 2.2: Estratto della normativa CISPR 14-1

riguarda i disturbi condotti, in particolare queste si occupano di limitare l'impatto armonico delle correnti assorbite dagli apparecchi collegati alla rete.

2.3 Soft Error

2.3.1 Generalità

Si definisce soft error (in alcuni casi viene chiamato anche SEU - single event upset) un errore temporaneo all'interno di un circuito digitale, cioè un evento

[1]CISPR - Comité International Spécial des Perturbations Radioélectriques, è un organismo internazionale i cui membri sono parte dei comitati nazionali dell'IEC e di altre organizzazioni internazionali che si occupano di questo tipo di disturbi e che detta le norme per quanto riguarda le interferenze a radio frequenza.

che porta ad avere dei valori non corretti in uscita dal circuito stesso. Si tratta quindi di una variazione dell'informazione contenuta in una memoria o un circuito digitale. Questo deve però essere un errore non ripetitivo: se lo fosse, quindi se ripetendo l'operazione l'errore si ripresentasse, è probabile che ci si trovi di fronte ad una rottura del circuito. Si parla in questo caso di hard errors, che però esulano da questa trattazione. Quindi, se una memoria si trova in un contesto tale da generare dei soft errors al suo interno, è sufficiente resettarla e poi riscriverla per cancellare tali errori. Se invece il malfunzionamento fosse dovuto ad uno o più hard errors è probabile che la memoria (o meglio le celle interessate, o i circuiti di lettura/scritturra) sia danneggiata e quindi non più utilizzabile.

2.3.2 Il problema dei Package

Questi tipo di errori sono stati studiato fin dagli anni settanta da molte aziende di sistemi digitali (ad esempio la IBM ha compiuto importanti studi in questo settore). Con l'introduzione delle prime RAM dinamiche i soft errors sono divenuti più frequenti e importanti. Si scoprì difatti che le particelle alfa emesse dalla piccolissima quantità di materiale radioattivo contenuta nei package delle memorie erano in grado, collidendo col chip su cui era costruito il circuito, di produrre una quantità di carica abbastanza grande (*carica critica*) da poter far variare l'informazione (1 logico trasformato in 0 o viceversa). Naturalmente la quantità di materiale radioattivo contenuto era irrisoria e non diversa da quella presente in natura in ogni oggetto. Le piccole dimensioni di tali circuiti integrati li rendevano però molto delicati e sensibili a questo tipo di radiazione. Effetti simili sono sempre stati presenti anche nei circuiti logici combinatori (i quali presentano package simili se non identici), ma erano molto più difficili da identificare in quanto una particella alfa può creare degli impulsi di carica (e quindi delle correnti) molto brevi e quindi di difficile osservazione. Cosa diversa nelle memorie, dove un errore può rimanere memorizzato nella cella: per questo il loro studio è cominciato con la costruzione delle prime DRAM[2].
I problemi di emissione di particelle alfa da parte del package ad oggi sono in gran parte eliminati, non senza difficoltà dato che non risulta facile ridurre la componente radioattiva dei materiali a valori desiderati (per ottenere una buona affidabilità bisogna arrivare ad un'emissione di particelle alfa nell'ordine di 0.001 $[cph/cm^2]$ [3], come termine di paragone si pensi che le emissioni di particelle di una suola di scarpa sono comprese tra 0.1 e 10 cph/cm^2 [4]).
I problemi relativi all'irraggiamento non sono però terminati, si scoprì difatti che alcune particelle ad alta energia provenienti dallo spazio (radiazione cosmica) riescono ad influenzare, anche pesantemente, il comportamento dei circuiti elettronici presenti sulla terra.

[2] Anche se storicamente le RAM dinamiche sono stati i primi chip sui cui si sono visti problemi relativi ai soft errors al giorno d'oggi sono uno dei dispositivi più resistenti agli irraggiamenti di particelle ad alta energia.

[3] Counts per hour per $[cm^2]$.

[4] Dati wikipedia.

2.3.3 Radiazioni cosmiche

L'esistenza di particelle ad alta energia provenienti dallo spazio sulla superficie della terra è nota da molti decenni. Già negli anni '60 J.T. Wallmark e S.M. Marcus teorizzarono la loro influenza sui circuiti elettronici. Risalgono invece alla fine degli anni '70 le ricerche della IBM, che dimostravano come le radiazioni cosmiche siano causa di soft errors (al giorno d'oggi è la causa principale), non solo per apprecchiature operanti nello spazio, ma anche per quelle sulla superficie terrestre. Tali radiazioni sono formate da particelle, principalmente neutroni, protoni e pioni[5] ad alta energia. L'atmosfera e il campo magnetico terrestre offrono un buono scudo contro molte di queste particelle. Tuttavia le collisioni fra un raggio cosmico e gli strati superiori dell'atmosfera provocano una cascata di particelle. Alcune di esse (soprattutto neutroni) riesce a raggiungere il suolo con una energia cinetica significativa (molte volte è sufficiente anche solo l'energia termica di un neutrone) a causare errori nei circuiti digitali.
Dato che l'atmosfera terrestre è un fluido non omogeneo la quantità di raggi cosmici che raggiunge la superficie può variare a seconda della località e soprattutto dell'altitudine. Un dispositivo a 2000 metri sarà sottoposto ad un irraggiamento maggiore rispetto ad uno posto al livello del mare e radiazioni ancora maggiori si avranno sui circuiti degli aeroplani o sulle installazioni spaziali. Le misurazioni mostrano che, negli strati più bassi dell'atmosfera, ogni mille metri il flusso di particelle aumenta circa di un fattore 2,2 così che un aereo in volo può essere sottoposto ad un irraggiamento anche 300 volte superiore rispetto a quello che ha quando è a terra.
Un altro fattore che può variare la quantità di radiazioni cosmiche (variazioni nell'ordine di $\pm 7\%$) è l'attività del sole; le radiazioni cosmiche sulla terra si riducono quando c'è una forte attività delle macchie solari, in quanto queste riescono a variare il campo magnetico terrestre, che contribuisce allo schermo dai raggi cosmici. All'indirizzo web `http://www.seutest.com/cgi-bin/FluxCalculator.cgi` è presente un tool in grado di calcolare il flusso di neutroni in ogni località (bisogna inserire latitudine, longitudine, altitudine e altre informazioni necessarie al calcolo). Il risultato viene proposto come valore relativo al flusso sul livello del mare a New York, che è considerato il valore di riferimento.

2.3.4 Prospettive future

Fin dall'inizio dell'elettronica digitale integrata il trend tecnologico tende a ridurre le dimensioni e le tensioni di lavoro dei transistor CMOS, sia per ridurre l'ingombro ma soprattutto perché circuiti più piccoli generano minore perdita per effetto joule, minori capacità parassite e quindi maggiori velocità. Nel grafico di Figura 2.3 è mostrato l'andamento della lunghezza di gate nei CMOS dagli anni settanta fino al 2020 (previsioni). A tal proposito si ricorda la prima legge di Moore: "*le prestazioni dei processori, e il numero di transistor per chip, raddoppiano ogni 18 mesi*". Non è una legge assoluta e dimostrabile, ma è stata verificata dall'andamento della tecnologia e assunta a paradigma.
Detto questo, è facilmente intuibile che la *carica critica*, ovvero la quantità di

[5] il pione è una particella subatomica della famiglia dei mesoni

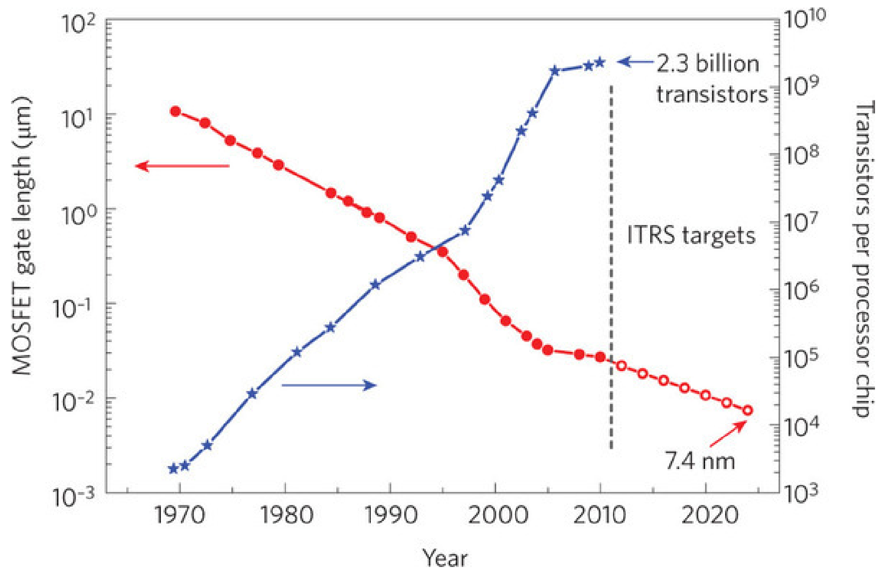

Figura 2.3: lunghezza da gate nei processi produttivi di circuiti CMOS dal 1970 al 2020 (previsioni)

carica il cui spostamento all'interno di un chip può causare il cambiamento di uno o più bit, diminuisce con la dimensione del transistor, questo trend è visibile in Figura 2.4. Si tenderebbe perciò a pensare che il SER[6] di un singolo bit all'interno di una memoria sia destinato a crescere con l'avanzare dei nodi tecnologici. Gli studi hanno dimostrato però che questo è stato verificato solo nel periodo iniziale. Facendo riferimento alle SRAM (in Figura 2.5 si guardi *SRAM bit SER*) il SER cresce solo fino a lunghezze maggiori di circa $200 \div 500nm$; al di sotto di queste c'è una saturazione per cui rimane pressoché costante (si tralasci la pesante discontinuità a $250nm$ che è dovuta all'eliminazione del Boro isotopo 10 nel vetro fosfosilicato di passivazione). Questo effetto è dovuto principalmente alla saturazione dello scaling della tensione di lavoro, che non è più ulteriormente abbassabile; altra causa è l'aumento del charge sharing coi nodi limitrofi dovuto agli effetti della miniaturizzazione.
Tutto ciò non minimizza però gli effetti degli irraggiamenti di particelle ad alta energia su circuiti CMOS di nuova generazione. Difatti la diminuzione della lunghezza di canale è legata ad un esponenziale aumento del numero di transistor per unità di area (Figura 2.3), aumentano quindi i potenziali nodi su cui si possono generare errori, quindi la saturazione del *SRAM bit SER* non implica una reale diminuzione degli errori nell'intero sistema (*SRAM system SER*).

Sarà quindi necessario prestare sempre maggiore attenzione al problema dei soft errors causati dalle radiazioni cosmiche, in quanto il mercato richiede dispositivi sempre più piccoli, performanti e affidabili; a tal proposito vengono eseguite ricerche (come quelle del gruppo RREACT) ed esperimenti su vari tipi di dispositivi per determinarne la robustezza in ambienti molto ostili così da poter capire se un determinato chip è abbastanza affidabile per lavorare in certe situazioni.

[6]SER - Soft Error Rate; è una grandezza che misura la frequenza con cui si presentano dei soft errors all'interno di un circuito, viene solitamente misurata in FIT (failures in time) fallimenti per unità di tempo dove l'unità di tempo è 10^9 ore, oppure in MTBF (mean time between failures) cioè tempo medio tra due errori.

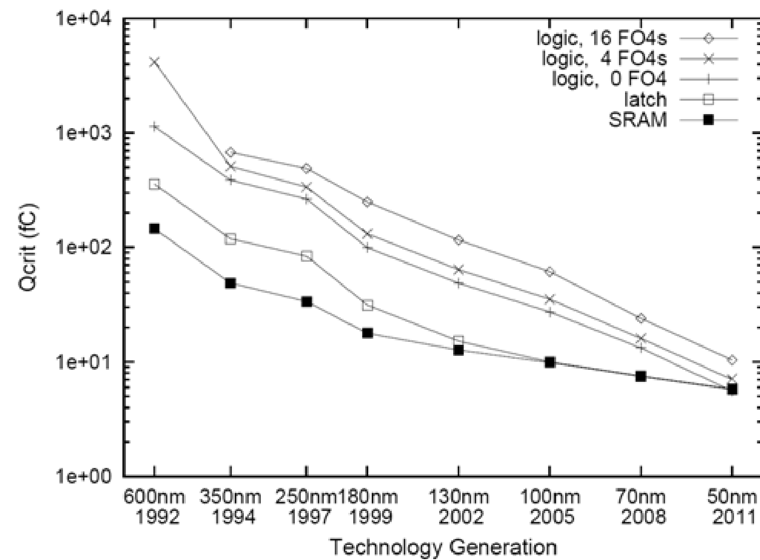

Figura 2.4: andamento della carica critica Q_{crit} rispetto al nodo teconologico di vari tipi di circuiti CMOS.

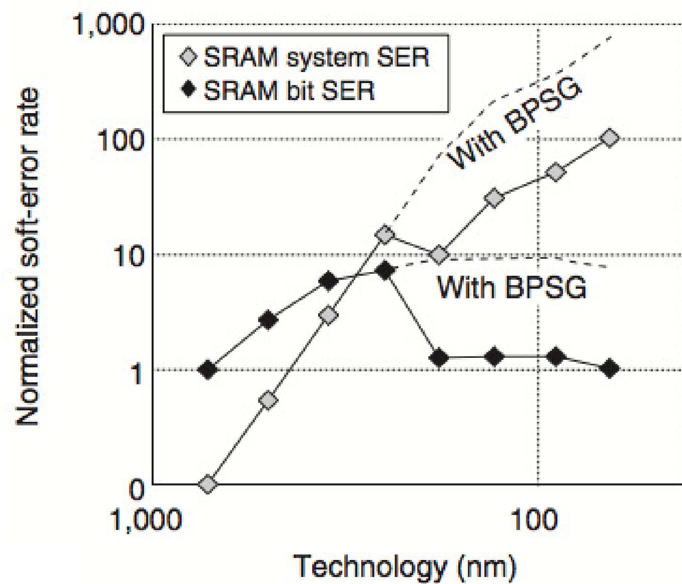

Figura 2.5: andamento del SER (soft error rate) in funzione del nodo tecnologico. La discontinuità che si ha sul nodo a $250nm$ è dovuta all'eliminazione del vetro boro fosfosilicato (BPSG) che veniva usato per creare strati isolanti nei wafer di silicio, si scoprì indurre soft errors per colpa dei prodotti di fissione generati quando B^{10} interagisce con neutroni a bassa energia.

Capitolo 3

Sistema di test per memorie

3.1 Introduzione

In questo capitolo verrà discusso il lavoro di tirocinio svolto da me presso il laboratorio RREACT (Reliability and Radiation Effects on Advanced CMOS Technologies) del Dipartimento di Ingegneria dell'Informazione. Come si capisce dal nome questo laboratorio si occupa di effettuare dei test su vari tipi di dispositivi CMOS per verificarne l'affidabilità. Il mio obbiettivo è stato quello di realizzare un sistema per il test di due memorie una NAND e una PCM (Sezione 1.2) attraverso l'utilizzo di una scheda di sviluppo dotata di un dispositivo FPGA (Sezione 1.1). Questo sistema dovrà a tal proposito essere in grado di eseguire operazioni di base sulle memorie, quindi letture e scritture. Non sarà richiesto di memorizzare dati significativi come file o testi ma semplicemente scrivere la stessa parola (pattern) all'interno di un range di indirizzi; una volta programmata, la memoria verrà irraggiata con determinate particelle (particelle alfa, neutroni o altro), dopodiché il sistema dovrà essere in grado di leggere la memoria e di verificare in quante e quali celle il contenuto è cambiato e non è più presente il pattern di partenza, in quali celle cioè si sono verificati soft errors (Sezione 2.3) o eventualmente hard errors. Poi una volta eseguite le operazioni richieste, i risultati (quindi il numero degli errori, il loro indirizzo, ecc) verranno spediti ad un PC che li potrà salvare o elaborare secondo le necessità.

3.2 Strumenti hardware e software utilizzati

Per la realizzazione di questo sistema mi è stata messa a disposizione una scheda di sviluppo (*ML402*) della Xilinx® con montato un FPGA della famiglia Virtex4®, per gestire più facilmente le comunicazioni col PC è stato implementato un μblaze all'interno del FPGA stesso. I collegamenti fisici dei pin della memoria con quelli del FPGA sono stati realizzati tramite una *dougther board*, cioè una scheda PCB sulla quale sono stati montati gli zoccoli per il collegamento ai bus di espansione della Virtex da una parte e alla memoria dall'altra (queste schede erano già pronte, il mio lavoro si è limitato al montaggio degli zoccoli e alla saldatura di qualche condensatore necessario al funzionamento della memoria).
Per quanto riguarda i software ho utilizzando quelli forniti da Xilinx, quindi sia l'ISE che l'XPS (Xilinx Platform Studio); il primo per sviluppare e testare le

periferiche scritte in VHDL e il secondo per integrare tali periferiche all'interno del sistema e per interfacciarle col μblaze.

3.3 Sistema da realizzare

3.3.1 Visione generale

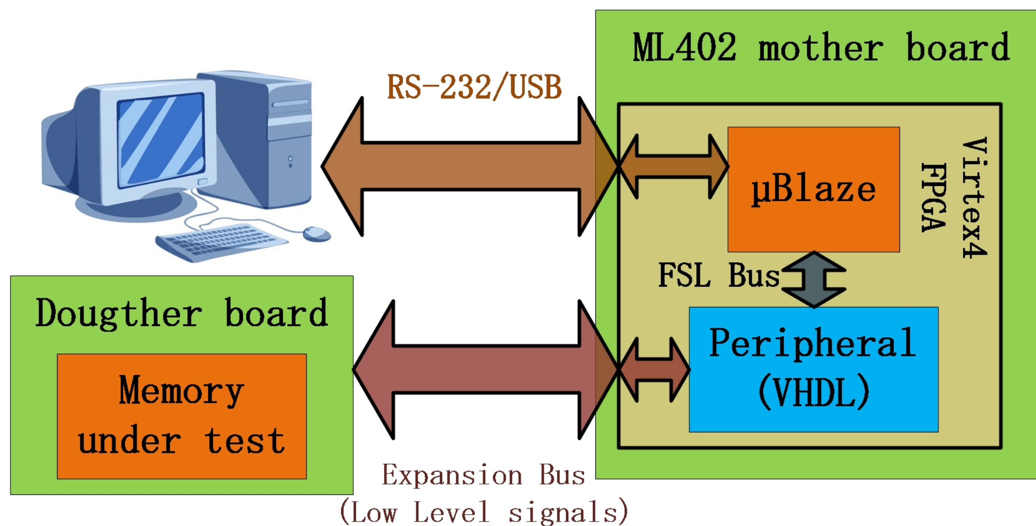

Figura 3.1: Schema del sistema di test da realizzare.

Come già detto nell'introduzione del capitolo il sistema dovrà interagire da una parte con la memoria sotto test (DUT - device under test) e dall'altra con un PC. Lo schema generale è riportato in Figura 3.1 e il funzionamento di base è il seguente: il PC invia (inizialmente tramite seriale e in un secondo momento via USB) uno o più pacchetti contenenti il codice dell'operazione richiesta e altre informazioni (le operazioni supportate sono descritte nella Sezione 3.3.2), come ad esempio indirizzi, pattern o codici di controllo, un programma scritto in `C` che viene eseguito sul microblaze decodifica tali operazioni, le traduce in un altro formato più compatto e le invia attraverso il *bus FSL* alla periferica[1]; quest'ultima sarà la parte centrale e più complicata dello sviluppo, in quanto dovrà tradurre le operazioni in segnali logici (0 o 1) comprensibili dalla memoria sotto test. Una volta eseguita l'operazione i risultati o i feedback percorrono il percorso in direzione opposta fino ad arrivare al PC.

Scendendo maggiormente in dettaglio le cose realizzate sono state:

Sul PC: In un primo momento le operazioni venivano inviate tramite seriale quindi non è stato necessario creare nulla in quanto in ambiente Windows la soluzione più semplice è utilizzare Hyperterminal con il quale basta aprire una connessione con la porta RS-232 dopodiché per inviare un carattere basta scriverlo nella finestra all'interno della quale vengono anche stampati

[1]Si noti anche dallo schema che sia la periferica che il microprocessore sono contenuti all'interno del FPGA, non sarà quindi possibile distinguerli fisicamente ma solo all'interno del programma XPS e a livello di codice

i valori inviati dalla periferica.
Successivamente si è provato ad utilizzare la porta USB per comunicare, questo ha comportato la scrittura di un software (in `C++`) che utilizzando delle DLL di Windows potesse inviare e ricevere abbastanza facilmente dati tramite USB.

Sul μblaze: Per quanto riguarda la creazione del processore si è utilizzato un wizard di XPS che lo crea automaticamente una volta inserite le caratteristiche volute.
Per far interagire il μblaze con il computer e la memoria è stata creata l'applicazione (in linguaggio `C`) da fargli eseguire. Date le poche risorse usate non si è vista la necessità di inserirvi un sistema operativo, quindi l'applicazione scritta girerà in modalità stand-alone.

Sulla periferica: Le periferiche (se ne sono sviluppate due, una per la memoria NAND e una per la PCM ma comunque con caratteristiche molto simili) sono la parte più consistente di codice scritta durante questo mio lavoro; mi è stato fornito uno scheletro del codice VHDL contenente la struttura dalla quale partire. Leggendo i datasheet delle memorie ho dovuto fare in modo di pilotare i segnali in maniera corretta per eseguire letture, scritture, reset, lettura dello stato e altre operazioni.

3.3.2 Operazioni da eseguire sulle memorie

Di seguito sono riportate le operazioni che il computer potrà richiedere di eseguire sulla memora alla mother board (la scheda di sviluppo), alcune di queste non necessitano nessun dato aggiuntivo (ad esempio **Read ID**), altre invece hanno bisogno di informazioni aggiuntive (ad esempio indirizzi), questi parametri verranno passati tramite il *bus FLS* con un pacchetto per ogni valore, ci si riferirà a questi parametri chiamandoli anche argomenti facendo una analogia col linguaggio `C`. Ci saranno poi alcune operazioni che restituiranno dei valori, come lo *status register* o il numero identificativo della memoria, a quelle invece che non hanno nessun dato significativo da trasmettere indietro al μblaze è stato comunque fatto inviare un valore, anche senza nessun significato, questo serve come pacchetto di ACK per informare il processore che l'operazione è terminata e si può quindi richiederne un'altra.
Verranno prima elencate le operazioni relative alla memoria NAND, per quanto riguarda la PCM ne verranno descritte solamente le differenze (dove ce ne sono) in quanto a questo livello di astrazione le differenze non sono molte.

Memoria NAND

Come la maggior parte delle NAND flash anche questa memoria della Hynix (modello: HY27UF da 256MByte x 8 bit = 2Gbit) è divisa in blocchi e pagine. In particolare questa contiene 2048 blocchi composti da 64 pagine ciascuno e ogni pagina a sua volta contiene 2112 byte; come tutte le memorie flash molte operazioni vanno effettuate su insiemi di parole, se si vuole ad esempio cancellare un byte non si può eliminarlo singolarmente ma bisogna agire a livello dell'intero blocco. Per questo motivo le operazioni supportate saranno divise in: **Operazioni generali**, cioè quelle che non vanno ad agire sulle celle di memoria ma

solamente sui circuiti di controllo, **Operazioni di pagina**, quelle che leggeranno o scriveranno a livello di una o più pagine e similarmente **Operazioni di blocco**, quelle che agiranno su uno o più blocchi.

Operazioni generali: Non necessitano di dati aggiuntivi ad esclusione di **Select** a cui bisogna passare il numero della memoria da selezionare.

Select: Dato che il sistema dovrà essere flessibile e magari utilizzabile da più memorie viene richiesto un comando di **Select**, questo dovrà abilitare una memoria piuttosto che un'altra; può anche essere visto come un modo per metterla in standby in quanto a livello fisico questo comando andrà ad agire sul segnale di `Chip Enable` della memoria.

Read status register: Ogni memoria possiede al proprio interno un registro chiamato *Status Register*, questo fornisce informazioni riguardanti lo stato in cui si trova la memoria, ad esempio dopo un'operazione di scrittura si può verificare se tutto è andato a buon fine. Usualmente ogni bit di questo registro ha un suo significato a seconda di quale operazione si sta eseguendo o si è appena eseguita. Lo *Status Register* (abbreviato con *SR*) è utile in quanto molte volte leggendolo si capisce se la memoria è ancora funzionante, sarà quindi importante poterlo verificare in situazioni di test, dove gli irraggiamenti potrebbero danneggiare il circuiti interni (e lasciare magari intatte le celle), portando a malfunzionamenti di difficile localizzazione.

Read ID: L'*ID* è un numero identificativo della memoria, contiene informazioni riguardanti il tipo le dimensioni della stessa oppure anche un identificativo della partita in cui è stata prodotta. Nel caso di questa NAND l'identificativo è rappresentato da quattro parole da 8 bit ciascuna. La conoscenza del *ID* serve per verificare con che memoria si sta lavorando ed anche questo un testare il reale funzionamento. Se il numero è diverso da quello che ci si aspetta: o si sta sbagliando memoria oppure è danneggiata.

Reset: Questa operazione, diversamente da quello che ci si può aspettare, non resetta il contenuto delle celle di memoria, bensì la macchina a stati e i circuiti interni di controllo della memoria. Dopo alcune operazioni la memoria potrebbe richiedere di essere resettata per poterla utilizzare ancora, anche dopo l'accensione è buona regola dare un comando di reset per essere sicuri di trovarsi sempre a lavorare in condizioni ottimali.

Operazioni di pagina: Tutte le operazioni sulle pagine necessitano di queste informazioni: **indirizzo di partenza** (formato da indirizzo di blocco, indirizzo di pagina e indirizzo di colonna), **numero di byte** da elaborare e **pattern**.

Program page: Come è facilmente intuibile questa operazione dovrà eseguire la programmazione della pagina, questa memoria supporta la scrittura anche parziale di una pagina, quindi la programmazione inizierà all'**indirizzo di partenza** e si programmeranno gli x byte che seguono, dove $x =$ **numero di byte**, col valore passato come **pattern**.

Page errors: Questa operazione dovrà leggere le parole contenute nel range di indirizzi [*(indirizzo di partenza) : (indirizzo di partenza + numero di byte)*] e verificare che queste coincidano con **pattern**, ogni volta che il valore è diverso si incrementa un contatore e alla fine verrà restituito il valore totale di errori.

Check Page: Simile a **Page errors** con la differenza di eseguire un'analisi più approfondita; per ogni errore restituisce l'indirizzo della cella e il valore in essa contenuto.

Operazioni di blocco: Il formato è simile alle operazioni sulle pagine, in particolare bisogna fornire: **blocco di inizio**, **blocco di fine** e **pattern** (il pattern non serve nel caso di **Erase block**).

Program block, Check block, Block errors: Come **Program page**, **Check page** e **Block page** con la differenza che le in questo caso si lavora con i blocchi e non con le pagine.

Erase Block: Come si intuisce facilmente questo comando cancella i dati contenuti nei blocchi specificati, quindi da **blocco di inizio** a **blocco di fine**.

Memoria PCM

La PCM utilizzata è una Numonix (modello: Omneo P8P PCM da 128Mbit), a differenza della FLASH gli indirizzamenti sono più semplici in quanto non c'è la divisione in indirizzi di pagina e blocco (anche in questa però l'operazione di **erase** viene effettuata in blocchi). La particolarità di questo modello è che i bus indirizzi e dati sono separati, quindi tutte le operazioni in cui si devono passare degli indirizzi saranno molto agevolate, questo a discapito di una maggiore complessità a livello di circuito stampato (in quanto ci sono molti più segnali). Altra sostanziale differenza rispetto alla memoria della Hynix è la lunghezza delle parole, su questa difatti le parole sono da 16 bit invece che 8 (questo comunque non introduce nessuna complicazione).

Operazioni (differenze rispetto alla NAND): Rispetto alla NAND le differenze a livello di operazioni elementari sono poche, ci saranno sempre **read ID** e **read status register**, il **reset** e il **select**; le operazioni **check** e **controllo degli errori** saranno uguali ad eccezione del fatto che non ci sarà più la separazione tra i controlli di pagina e di blocco. Ci saranno però due operazioni di programmazione differenti, **Bit Alterable Word Program** e **Bit Alterable Buffered Word Program**; la differenza sta nel modo nel quale i dati vengono scritti all'interno della memoria, nella prima la scrittura avviene non appena il valore viene inviato al chip, nella seconda invece i valori vengono temporaneamente salvati dentro un buffer (con dimensione di 32 word) e inviati all'array di memoria in un secondo momento. Con la scrittura *bufferizzata* si hanno tempi complessivi di scrittura più rapidi, soprattutto nel caso di scritture di tante celle consecutive. Verrà anche aggiunta l'operazione di **unlock block** in quanto questa memoria permette di bloccare dalla scrittura le celle all'interno di uno stesso blocco.

3.4 Codice sviluppato (prima implementazione)

3.4.1 VHDL per la periferica (memoria NAND)

In questa sezione verrà analizzato in codice scritto per creare la periferica VHDL che interfaccia la memoria della Hynix col μblaze. Dato che per la sola periferica dedicata alla NAND si sono scritte più di 1300 righe di codice non risulta possibile descrivere in maniera dettagliata tutte le sue parti, verranno perciò approfondite soltanto quelle più interessanti e in particolare saranno trattate le operazioni di **Read ID** e **Program page**.

Situazione di partenza: Come si vede dallo schema del sistema mostrato nella prima parte del capitolo (Figura 3.1) la periferica che si vuole realizzare sarà collegata da una parte alla memoria e dall'altra al microblaze. Dal lato memoria i segnali usciranno dai pin del FPGA e andranno alla NAND attraverso i bus di espansione della scheda *ML402*, mentre dal lato microprocessore i segnali rimarranno interni all'FPGA, la comunicazione avverrà tramite un *bus FSL*[2]; questo bus viene fornito da Xilinx come IP[3], quindi non sarà creato in VHDL ma verrà semplicemente aggiunto al progetto utilizzando un tool specifico. Il file contenente la descrizione della periferica sarà chiamato *rreact_nandflash_ft.vhd* che sarà anche il nome della entity che descrive il comportamento della periferca. Lo spezzone di codice che segue rappresenta la descrizione delle porte della entity *rreact_nandflash_ft*.

Listing 3.1: Prima parte del file *rreact_nandflash_ft.vhd* che ne descrive le porte, si noti alle righe 104÷106 la definizione dei segnali per la creazione della porta threestate collegata al *bus IO* della memoria

```
1  entity rreact_nandflash_ft is
2    port (
3    -- FLS bus signals --
4    FSL_Clk:     in std_logic;
5    FSL_Rst:     in std_logic;
6    FSL_S_Clk:     in std_logic;
7    FSL_S_Read:    out std_logic;
8    FSL_S_Data:      in std_logic_vector(0 to 31);
9    FSL_S_Control:   in std_logic;
10   FSL_S_Exists:    in std_logic;
11   FSL_M_Clk:     in std_logic;
12   FSL_M_Write:   out std_logic;
13   FSL_M_Data:      out std_logic_vector(0 to 31);
14   FSL_M_Control:   out std_logic;
15   FSL_M_Full:      in std_logic;
16
17   -- memory signals --
18   debug : out STD_LOGIC_VECTOR (7 downto 0);
19   -- three state IO port
20  -- -- -- -- -- -- --
21   IO_I : in  STD_LOGIC_VECTOR (7 downto 0); -- MSB a sinistra
22   IO_O : out  STD_LOGIC_VECTOR (7 downto 0);
23   IO_T : out STD_LOGIC; -- IO_T = '0' => out
```

[2]FSL - Fast Simplex Link
[3]IP - Intellectual Proprety

```
24  -- -- -- -- -- -- --
25   ALE : out STD_LOGIC;     -- address latch enable
26   CLE : out STD_LOGIC;     -- command latch enable
27   nCE : out STD_LOGIC;     -- chip enable
28   nRE : out STD_LOGIC;     -- read enable
29   nWE : out STD_LOGIC;     -- write enable
30   nWP : out STD_LOGIC;     -- write protect
31   RB : in STD_LOGIC        -- ready/busy
32   );
```

Tra le linee 4 e 15 sono elencate le porte relative ai due *bus FSL* (si faccia riferimento al file *fsl_v20.pdf*[4] per i dettagli sul significato preciso dei segnali), mentre nella seconda parte sono presenti le porte che saranno collegate alla memoria attraverso il bus di espansione. La porta threestate (Linee 21÷23) rappresenta il bus di comunicazione a 8 bit attraverso il quale la memoria riceve e invia comandi, indirizzi e altri dati; verrà chiamata nella seguente trattazione *bus IO*.

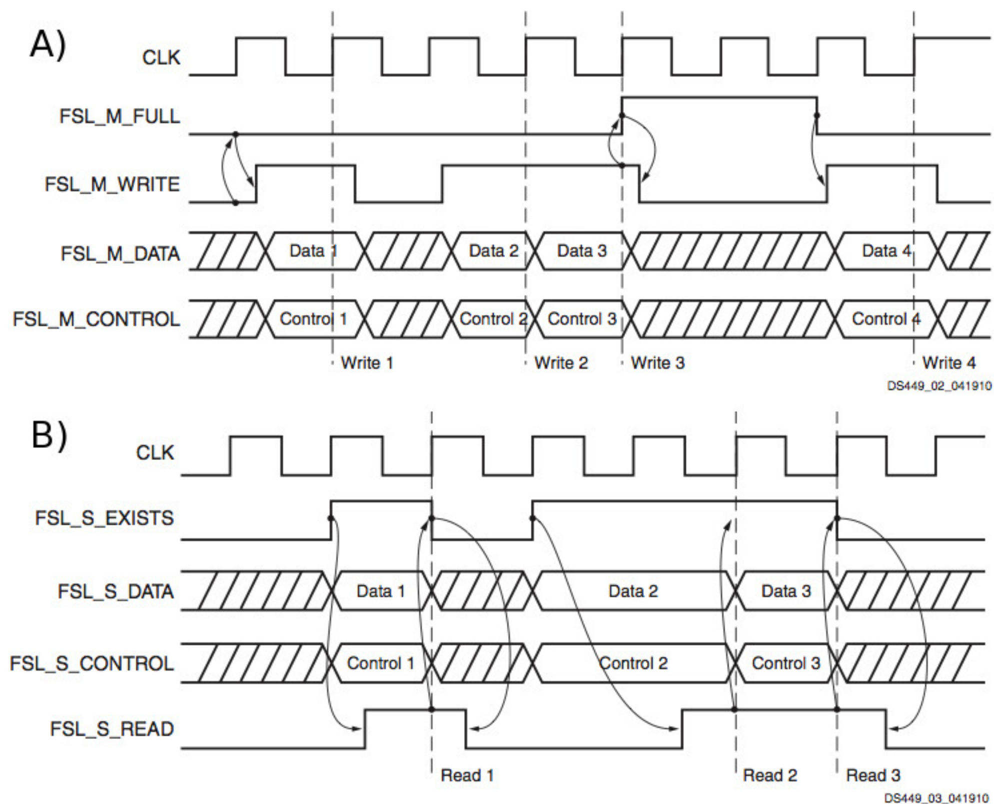

Figura 3.2: Estratto del file *fsl_v20.pdf* che mostra gli andamenti temporali dei segnali del *bus FSL* nel caso di invio A) e ricezione B).

Bus FLS: Questo bus di comunicazione implementa una coda unidirezionale di tipo FIFO, sia la dimensione delle parole sia la profondità della coda sono variabili e si possono cambiare facilmente all'interno di XPS, in questo progetto se ne utilizzeranno due (uno per i dati inviati dal processore e ricevuti dalla periferica e uno per i quelli che viaggiano in direzione opposta) e saranno formati da parole di 32 bit. É stato scelto questo tipo di bus per la sua facilità d'uso, risulta

[4]File reperibile sul sito della Xilinx

infatti molto semplice inviare e ricevere dati, sia dal lato del μblaze sia dal lato periferica; bisogna difatti far notare fin da subito che per inviare o ricevere dati dalla periferica si controlleranno i segnali fisici (basso/alto o 0/1), mentre dal lato del microprocessore non si avrà controllo dei singoli segnali ma si utilizzeranno delle funzioni a livello di codice `C`.
Grazie al file header *fls.h* fornito da Xilinx si hanno a disposizione le funzioni necessarie per inviare e ricevere dati sul *bus FSL*, quindi quando si vuole inviare un comando alla periferica basterà utilizzare la funzione `putfslx(...)` che è appunto definita all'interno del file `fsl.h`.
Per eseguire la stessa operazione dalla periferica verso il processore la procedura è completamente diversa, in questo caso il *bus FSL* viene visto come una serie di segnali a basso livello in ingresso e uscita (Linee 4÷15 del Listato 3.1); quando si ha la necessità di inviare un dato al microblaze si dovrà porre il valore a 32 bit sulla porta `FSL_M_Data` dopodiché basterà alzare il segnale `FSL_M_Write` e il dato verrà memorizzato dentro la FIFO e sarà possibile al microblaze leggerlo utilizzando la funzione `getfslx(...)`; in Figura 3.2 sono mostrati gli andamenti temporali tipici dei segnali per la scrittura e la lettura dal bus anche nelle condizioni critiche (coda piena).
Tutti i dettagli riguardanti questo bus sono contenuti nel file *fsl_ v20.pdf*.

Note sul file VHDL: Come si vedrà nei prossimi paragrafi, nel datasheet vengono forniti i diagrammi temporali dei vari segnali che comandano la memoria, ad esempio per un'operazione di scrittura di pagina si dovranno pilotare i pin nello stesso modo descritto dalla Figura 3.5. Per fare questo si è creata una grande macchina a stati, ad ogni operazione corrisponde una catena di stati percorsi in maniera sequenziale e ad ogni stato corrisponde la variazione di uno o più segnali; quando non si stanno eseguendo operazioni la FSM[5] sarà nello stato di `idle`, non appena viene ricevuto un pacchetto dal *bus FSL* questo viene letto e ne viene estratta la parte che contenente l'operazione richiesta, quindi la macchina entra nello stato corrispondente (nel caso di una scrittura `programPage0`) e percorre in maniera sequenziale tutta la catena di `programPage` comandando in maniera appropriata i segnali fino a tornare in `idle` una volta finita l'operazione.

Per ottenere un codice più efficace si sono usati tutti segnali sincroni, quindi ad esempio se si vuole assegnare un valore al segnale `s_nWE` che pilota la porta `nWE` (questo segnale verrà ad esempio usato nel Listato 3.2) non bisognerà imporlo direttamente su `s_nWE` ma assegnarlo al segnale `new_nWE`, ci sarà poi un processo che si occupa, ad ogni fronte di salita del clock, di copiare il valore di `new_nWE` su `s_nWE`. In questo modo tutti i segnali varieranno in sincronismo col segnale di clock.

Read *ID* (NAND): La lettura del numero identificativo della memoria è stata la prima operazione ad essere stata fatta, questo perché è sicuramente l'operazione più semplice in cui ci sono sia dati in ingresso che in uscita, in quanto bisogna inviare i comandi corretti e ricevere l'*ID*, queste comunicazioni avverranno utilizzando il *bus IO*[6] e i segnali di controllo della memoria (`CLE`, `nCE`, `nWE`, `ALE`,

[5]FSM - Finite State Machine, ovvero macchina a stati finiti

[6]Il *bus IO* è il bus dati a 8 bit che connette la periferica alla memoria, come si vede dal Listato 3.1 la porta è threestate, questo significa che i dati potranno viaggiare in entrambe le

nRE). Il protocollo da utilizzare per la lettura dell'*ID* è riportato sul datasheet della memoria HY27UF e la Figura 3.3 riassume gli andamenti tipici dei segnali.

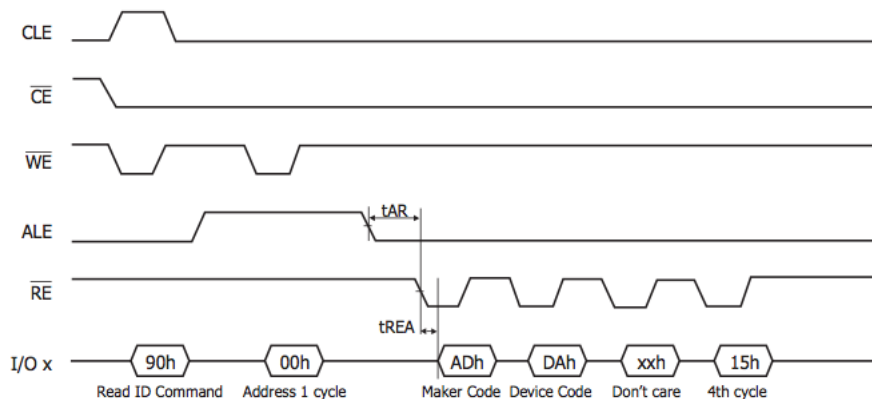

Figura 3.3: Estratto del datasheet della memoria della Hynix, viene mostrato il processo per leggere l'*ID*.

Si vorrà quindi che la periferica faccia seguire gli stessi andamenti ai segnali reali.

Listing 3.2: Parte del file *rreact_nandflash_ft.vhd* che rappresenta la descrizione della macchina a stati che si occupa della lettura dell'ID della memoria

```
1  if state = readId0 then
2      new_IODIR <= '0';    -- threestate port direction = OUT
3      new_IO <= X"90";     -- read ID command
4      new_CLE <= '1';
5      new_nWE <= '0';
6      new_state <= readId1;
7  end if;
8  if state = readId1 then
9      new_nWE <= '1';  -- command is latched on rising edge of nWE
10     new_state <= readId2;
11 end if;
12 if state = readId2 then
13     new_CLE <= '0';
14     new_ALE <= '1';
15     new_nWE <= '0';
16     new_IO <= (others => '0');  -- address X"00" is needed for
           reading ID
17     new_state <= readId3;
18 end if;
19 if state = readId3 then
20     new_nWE <= '1'; -- address is latched on rising edge of nWE
21     new_state <= readId4;
22 end if;
23 if state = readId4 then
24     new_ALE <= '0';
25     new_IODIR <= '1';
26     new_state <= readId5;
```

direzioni a seconda del valore dei segnali di controllo che nel Listato 3.1 si trovano alle righe 25÷31

```
27 end if;
28 -- -- Get maker code -- --
29 if state = readId5 then
30     new_nRE <= '0'; -- output is valid after falling edge of nRE
31     new_state <= readId6;
32 end if;
33 if state = readId6 then
34     new_FSL_M_Data(0 to 7) <= IO_I(7 downto 0);
35     new_state <= readId7;
36 end if;
37 if state = readId7 then
38     new_nRE <= '1';
39     new_State <= readId8;
40 end if;
41 -- -- -- -- -- -- -- -- -- --
42 -- -- Get device code -- --
43 if state = readId8 then
44     new_nRE <= '0'; -- output is valid after falling edge of nRE
45     new_state <= readId9;
46 end if;
47 if state = readId9 then
48     new_FSL_M_Data(8 to 15) <= IO_I(7 downto 0);
49     new_state <= readId10;
50 end if;
51 if state = readId10 then
52     new_nRE <= '1';
53     new_State <= readId11;
54 end if;
55 -- -- -- -- -- -- -- --
56 -- -- 3rd cycle -- --
57 if state = readId11 then
58     new_nRE <= '0';  -- output is valid after falling edge of nRE
59     new_state <= readId12;
60 end if;
61 if state = readId12 then
62     new_FSL_M_Data(16 to 23) <= IO_I(7 downto 0);
63     new_state <= readId13;
64 end if;
65 if state = readId13 then
66     new_nRE <= '1';
67     new_State <= readId14;
68 end if;
69 -- -- -- -- -- -- -- --
70 -- -- 4th cycle -- --
71 if state = readId14 then
72     new_nRE <= '0';  -- output is valid after falling edge of nRE
73     new_state <= readId15;
74 end if;
75 if state = readId15 then
76     new_FSL_M_Data(24 to 31) <= IO_I(7 downto 0);
77     new_state <= readId16;
78 end if;
79 if state = readId16 then
80     new_nRE <= '1';
81     new_State <= readId17;
82 end if;
83 -- -- -- -- -- -- -- -- -- -- --
84 -- Send ID through fsl BUS -- --
```

```
85 if state = readId17 then
86     if FSL_M_Full = '0' then
87       new_FSL_M_Write <= '1';
88       new_state <= idle;
89     else  -- If fsl FIFO is full remain in this state
90       new_state <= readId17;
91     end if;
92 end if;
```

Il segnale `state` rappresenta lo stato corrente della macchina a stati finiti mentre `new_state` è lo stato futuro, detto questo esaminando il Listato 3.2 si vede che la FSM percorre linearmente tutti gli stati da `readId0` a `readId17` non ci sono quindi particolari complicazioni e quindi passiamo ad analizzare le operazioni eseguite.

readId0÷readId1 La prima cosa che bisogna fare è passare alla memoria il comando di lettura dell'*ID* (come si vede dalla Figura 3.3 il suo codice è `0x90`), per fare questo la porta di uscita deve essere impostata come porta di OUT (`IODIR = 0`) in quanto il comando parte dalla periferica per andare alla memoria, il valore viene campionato dalla memoria sul fronte di salita di `nWE` (Linea 9).

readId2÷readId3 Per leggere l'*ID* è a questo punto necessario passare l'indirizzo `0x00`, per fare ciò prima si abilita la memoria a ricevere un indirizzo ponendo a uno `ALE` dopodiché si scrive `0x00` sul *bus IO*, anche in questo caso l'indirizzo verrà campionato sul fronte di salita di `nWE` (Linea 20).

readId4÷readId7 A questo punto la memoria è pronta per restituire il proprio numero identificativo, quindi si imposta la porta threestate come porta di IN (`IODIR = 1`), in quanto la direzione della comunicazione sul *bus IO* è ora dalla memoria alla periferica. Ora la NAND restituirà le quattro parole dell'identificativo scrivendole sul bus in maniera sequenziale, il valore è valido dopo il fronte di discesa di `nRE`. Perciò si porrà `nRE = 0` (Linea 30) dopodiché la prima parola (marker code) sarà presente sul *bus IO* e potrà essere salvata.

readId8÷readId16 Per leggere le altre tre parole non resta altro che alzare e abbassare `nRE`, dopo ogni fronte di discesa sarà presente la nuova parola e sarà possibile memorizzarla.

readId17 Una volta completata la procedura le parole dell'identificativo saranno memorizzate in un segnale all'interno della periferica, serve quindi inviarle al μblaze attraverso il *bus FSL*. Si noti che le quattro parole dell'*ID* (32 bit in tutto) sono già state raggruppate all'interno di `FSL_M_Data` mentre venivano lette, quindi per inviarle sarà necessario solamente dare un impulso su `FSL_M_Write`, solo però dopo aver effettuato un controllo che la FIFO del bus non sia piena (Linea 86), in tal caso la FSM rimane in `readId17` finché non si libera un posto nella coda.

Eseguendo le simulazioni con iSim[7] si possono verificare le forme d'onda, confrontandole con quelle del datasheet (Figura 3.3) si può verificare che sono esattamente gli andamenti voluti.

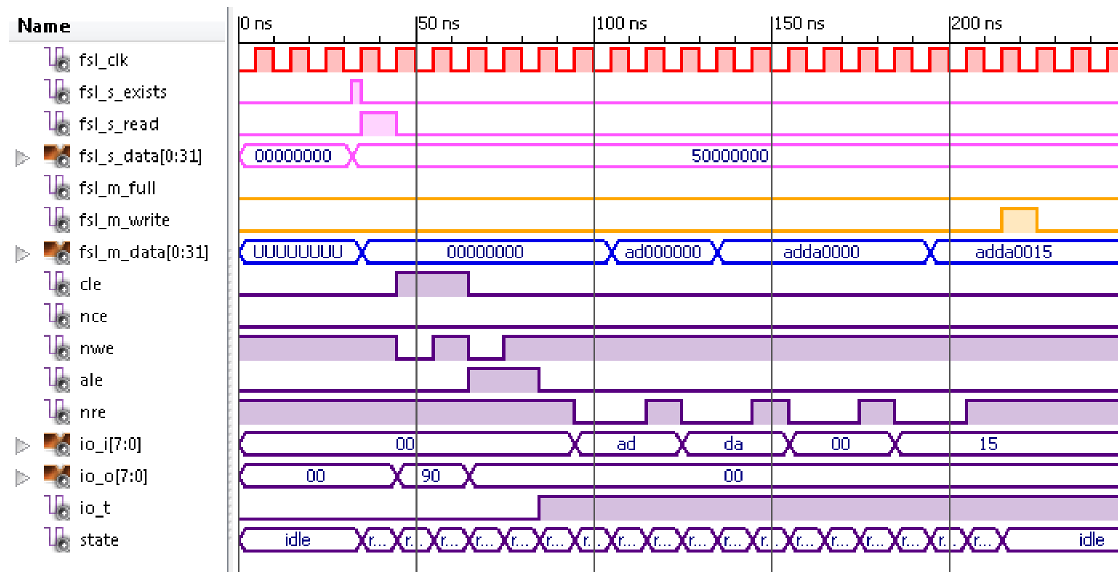

Figura 3.4: Risultato della simulazione di lettura dell'*ID* fatta con iSim. Se si confronta questa con Figura 3.3 si noterà che i segnali relativi alla memoria hanno gli stessi andamenti, si vede anche la parte relativa al *bus FSL*, sia in lettura (parte iniziale della simulazione) che in scrittura (parte finale). Per motivi grafici non si riescono a leggere i nomi degli stati correnti della FSM, si vedono comunque le variazioni che avvengono ogni ciclo di clock.

Program Page (NAND): La seconda parte del codice VHDL che verrà analizzata è quella relativa alla programmazione di una pagina. Anche in questo caso viene proposto l'estratto del datasheet delle memoria (Figura 3.5) che mostra l'andamento dei segnali per questa operazione e successivamente verrà analizzato il relativo codice VHDL.

Listing 3.3: Parte del file *rreact_nandflash_ft.vhd* che rappresenta la descrizione della macchina a stati che si occupa della programmazione di una pagina

```
1 if state = programPage0 then
2   new_IODIR <= '0';
3   new_IO <= X"80";     -- programPage command
4   new_CLE <= '1';
5   new_nWE <= '0';
6   if FSL_S_Exists = '1' then
7     new_FSL_S_Read <= '1';
8     new_ColumnAddress <= FSL_S_Data; -- Get column address
9     new_state <= programPage2;
10  else
11    new_state <= programPage;
```

[7]iSim è un tool di ISE che permette vari tipi di simulazioni su codici VHDL, è molto utile in fase di debug dei progetti perché si riescono a simulare gli andamenti di tutti i segnali in varie condizioni operative e con vari livelli di astrazione.

```
12    end if;
13  end if;
14  if state = programPage2 then
15    if FSL_S_Exists = '1' then
16      new_FSL_S_Read <= '1';
17      new_PageAddress <= FSL_S_Data; -- Get page address
18      new_state <= programPage3;
19    else
20      new_state <= programPage2;
21    end if;
22  end if;
23
24  [...]
25
26  if state = programPage6 then
27    new_nWE <= '1';    -- command is latched on rising edge of nWE
28    new_state <= programPage7;
29  end if;
30  -- 1st address cycle
31  if state = programPage7 then
32    new_CLE <= '0';
33    new_ALE <= '1';
34    new_nWE <= '0';
35    new_IO <= s_ColumnAddress (7 downto 0);
36    new_state <= programPage8;
37  end if;
38  if state = programPage8 then
39    new_nWE <= '1';  -- address is latched on rising edge of nWE
40    new_state <= programPage9;
41  end if;
42  -- 2nd address cycle
43  if state = programPage9 then
44    new_nWE <= '0';
45    new_IO <= s_ColumnAddress (15 downto 8);
46    new_state <= programPage10;
47  end if;
48  if state = programPage10 then
49    new_nWE <= '1';  -- address is latched on rising edge of nWE
50    new_state <= programPage11;
51  end if;
52
53  [...]
54
55  if state = programPage17 then
56    new_ALE <= '0';
57    new_IO <= s_Pattern(7 downto 0);
58    new_state <= programPage18;
59  end if;
60  if state = programPage18 then
61    if s_Bytes = 0 then -- all bytes have processed
62      new_state <= programPage20;
63      new_nWE <= '1';
64    else  -- more bytes need to be processed
65      new_nWE <= '0';
66      new_state <= programPage19;
67    end if;
68  end if;
69  if state = programPage19 then
```

```
70     new_nWE <= '1';   -- datum is latched on rising edge of nWE
71     new_Bytes <= s_Bytes - '1';
72     new_state <= programPage18;
73   end if;
74   if state = programPage20 then
75     new_IO <= X"10";   -- Program confirm command
76     new_CLE <= '1';
77     new_nWE <= '0';
78     new_state <= programPage21;
79   end if;
80   if state = programPage21 then
81     new_nWE <= '1';   -- command is latched on rising edge of nWE
82     new_state <= programPage22;
83     new_RB_counter <= (others => '0');
84   end if;
85   -- wait RB
86   if state = programPage22 then
87     new_CLE <= '0';
88     if RB = '0' OR s_RB_counter < X"0000000A" then
89       new_state <= programPage22;
90       new_RB_counter <= s_RB_counter + '1';
91     else -- when RB return hight the program process is completed
92       new_state <= programPage23;
93       new_FSL_M_Data <= s_RB_counter;   -- return RB_counter via FSL
94     end if;
95   end if;
96   if state = programPage23 then
97     if FSL_M_Full = '0' then
98       new_FSL_M_Write <= '1';
99       new_state <= idle;
100    else
101      new_state <= programPage23;
102    end if;
103  end if;
```

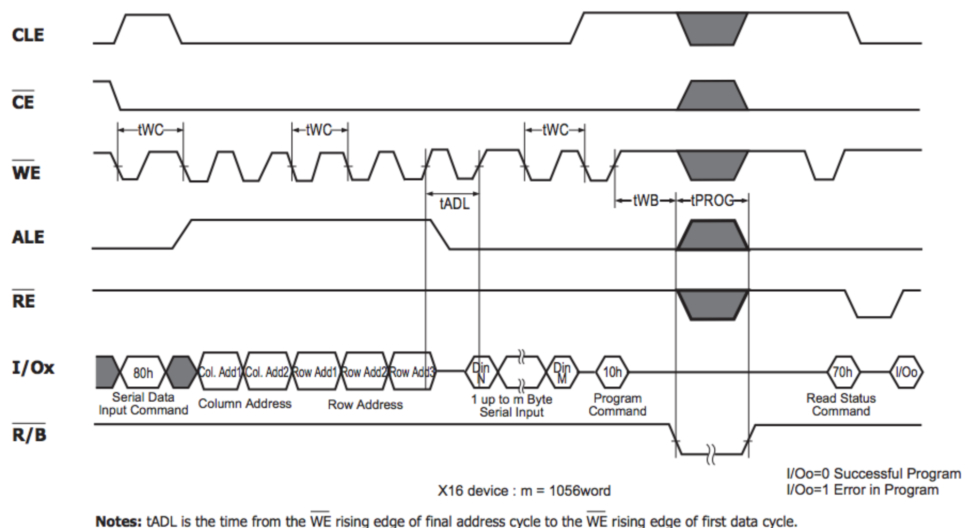

Figura 3.5: Estratto del datasheet della memoria della Hynix che si riferisce alla programmazione di una pagina.

programPage0÷programPage6 Come già detto nel paragrafo riguardante le operazioni di pagina (3.3.2) per eseguire questo comando sono necessarie più informazioni, queste vengono spedite ognuna con un diverso pacchetto tramite il *bus FSL*. I primi 8 stati di questa FSM si occupano di recuperare questi parametri, controllano la presenza del dato in arrivo (cioè controllano che `FSL_S_Exists` sia alto) e poi estraggono il valore presente sul bus, ovviamente l'ordine di invio deve essere corretto dato che non c'è modo di sapere cosa realmente si sta ricevendo, in particolare la periferia si aspetterà in ordine: indirizzo di colonna, indirizzo di pagina, indirizzo di blocco, numero di byte e in fine il pattern. Per alleggerire il listato sono stati rimossi gli stati `programPage3÷programPage5` dove venivano eseguiti gli stessi passi fatti in `programPage2` per gli altri argomenti in ingresso. Un'altra cosa fatta in questi stati è inviare il comando di programmazione (`0x80`) di pagina alla memoria, il meccanismo è lo stesso utilizzato per inviare il comando di lettura dell'*ID*, quindi si scrive `0x80` sul *bus IO* (Linea 3) e poi lo si fa campionare alla memoria alzando `nWE` (Linea 27).

programPage7÷programPage16 Ora che si hanno i valori di tutti gli argomenti bisogna passare l'indirizzo di partenza alla memoria, come si vede anche dalla Figura 3.5, questo viene inviato sul *bus IO* in cinque cicli. Dato che l'indirizzo ha in totale 29 bit e che con 5 cicli si inviano 40 bit risulta ovvio che alcuni bit saranno rindondanti, in particolare la Figura 3.6 mostra come devono essere impacchettati i bit all'interno dei cinque cicli di indirizzamento.
Anche in questo caso si sono eliminati dal listato alcuni stati in quanto i cinque cicli di indirizzamento sono pressoché identici.

	IO0	IO1	IO2	IO3	IO4	IO5	IO6	IO7
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7
2nd Cycle	A8	A9	A10	A11	L(1)	L(1)	L(1)	L(1)
3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27
5th Cycle	A28	L(1)	L(1)	L(1)	L(1)	L(1)	L(1)	L(1)

NOTE:
1. L must be set to Low.

Figura 3.6: Estratto del datasheet della memoria della Hynix; mappa degli indirizzi all'interno dei cinque cicli di indirizzamento durante le operazioni con le pagine.

programPage17÷programPage19 Dopo aver impostato l'indirizzo di partenza la memoria è in grado di accettare i dati veri e propri che si vogliono memorizzare, nel nostro caso si salverà lo stesso pattern di 8 bit su un numero di celle specificate (**numero di byte**) partendo dall'indirizzo di partenza, si imposta perciò il pattern sul *bus IO* (Linea 57) e poi è sufficiente pilotare `nWE` con un treno di impulsi (la FSM continua a passare da `programPage18÷programPage19` e viceversa e in questo modo generano gli impulsi con la durata di un periodo di clock), ogni impulso corrisponde alla scrittura di una cella, quindi si genereranno tanti impulsi quanti sono i byte da scrivere (**numero di byte**).

programPage20÷programPage21 La memoria naturalmente non ha modo di sapere quante celle vogliamo scrivere, quindi una volta raggiunto il numero desiderato bisogna passargli un comando di conferma (`0x10`) con il quale la memoria inizia la scrittura vera e propria (i dati in ingresso vengono inizialmente memorizzati in un buffer).

programPage22÷programPage23 A questo punto non resta che aspettare che la memoria trasferisca tutto il contenuto del buffer all'interno delle celle, per verificare quando il processo di scrittura è terminato è sufficiente controllare lo stato di `RB`[8] (Linea 88), questo viene abbassato, con un leggero ritardo, nell'istante in cui si dà il comando di conferma e riportato al valore 1 quando la scrittura è completata. Alla Linea 88 oltre al controllo sul `RB` ne viene effettuato uno anche su un contatore (`RB_counter`) che conta il numero di cicli di clock impiegati dalla memoria a ad effettuare la scrittura, questo secondo controllo serve per assicurarsi di rispettare il tempo minimo tWB(vedi Figura 3.5).
Per verificare se il sistema ha rilevato errori nella scrittura è possibile ora fare un controllo dello *status register* come indicato nel datasheet, qui non è stato implementato in automatico, se lo si volesse fare basta lanciare il comando di **Read Status Register**.
L'ultima operazione che è stata fatta è l'invio al μblaze del valore di `RB_counter` (Linea 98), questo serve per verificare che le durate dei periodi di scrittura abbiano valori ammissibili, in caso contrario ci può essere stato un errore.

Altre operazioni: Le altre funzioni elencate nella Sezione 3.3.2 non verranno mostrate, per la loro implementazione si sono seguiti gli stessi passaggi riportati per **read ID** e **program page**, costruendo cioè delle catene di stati consecutivi all'interno della macchina a stati e seguendo le indicazioni trovate sul datasheet della memoria per l'andamento dei segnali.

3.4.2 VHDL per la periferica (memoria PCM)

La periferica dedicata al controllo della memoria PCM è stata fatta nello stesso modo, ovviamente cambiano i nomi e le dimensioni dei segnali, le operazioni sono comunque eseguite sempre con una FSM che fa seguire ai pin della memoria gli andamenti voluti.

3.4.3 Codice `C` μblaze

Anche per quanto riguarda il programma da far eseguire al microblaze il mio lavoro non è partito da zero, mi è stata difatti consegnata una prima versione (relativa alla periferica per la memoria NAND), il mio lavoro si è limitato a completare le parti che mancavano e, partendo da questa, crearne una nuova da utilizzare con la periferica per la PCM.

Inizialmente la comunicazione col PC è stata fatta attraverso la porta seriale[9], i file *.h* (header) compresi nel XPS forniscono le definizioni delle

[8]Segnale di *ready-busy*

[9]La porta seriale verrà chiamata anche RS-232 o UART.

funzioni per gestire la RS-232 dal microblaze. Nel prossimo paragrafo sono elencati i comandi che l'applicazione dovrà saper gestire (in questo caso si prenderà in considerazione quella relativa alla PCM), alcune sono funzioni elementari che la periferica descritta nella sezione precedente è in grado di eseguire e quindi verranno semplicemente tradotte in un formato comprensibile e poi inviate sul *bus FSL*, altre saranno composte da più di una operazione elementare, ci saranno quindi ad esempio dei cicli che faranno eseguire l'operazione di **buffered word program** più volte su blocchi di 32 parole[10] consecutive.

L'elenco che segue rappresenta le operazioni che si potranno richiedere dal PC attraverso la seriale alla motherboard, il loro significato è già stato spiegato nella Sezione 3.3.2, qui se ne definisce il formato attraverso la porta UART. Il codice dell'operazione sarà formato da uno o più caratteri, mentre i parametri saranno passati come valori esadecimali da 8 a 32 bit, indicati con `0xXXXXXXXX` (ogni `X` rappresenta una cifra esadecimale quindi 4 bit).

Operazioni generali

select; [`s-0xXX`]: il valore esadecimale rappresenta la memoria con cui si vuole lavorare, per ora sarà considerato in caso di averne massimo due, quindi `0xXX` potrà assumere i valori 0 o 1.

write enable; [`e`]: questa operazione non prevede dati in ingresso o uscita.

read status register; [`t-0xXXXXXXX`]: nella PCM la lettura del SR richiede il passaggio di un indirizzo perché con questa operazione vengono passate anche informazioni riguardanti lo stato del blocco a cui l'indirizzo punta. Si avrà il valore dello status register di ritorno.

reset; [`r`]: nessun dato in ingresso o uscita.

read ID; [`i`]: nessun dato in ingresso, si avrà il valore del *ID* come valore di ritorno.

Operazioni di programmazione e di verifica

write buffered word; [`pb-0xXXXXXXXX-0xXXXXXXXX-0xXXXX`]: gli argomenti sono nell'ordine: indirizzo di inizio, indirizzo di fine, pattern, viene restituito il valore dello *status register*.

write word; [`pw-0xXXXXXXXX-0xXXXXXXXX-0xXXXX`] (stesso significato di `write buffered word`), viene restituito il valore dello *status register*.

errors; [`pe-0xXXXXXXXX-0xXXXXXXXX-0xXXXX`] (stesso significato di `write buffered word`), viene restituito il numero degli errori trovati.

check; [`pc-0xXXXXXXXX-0xXXXXXXXX-0xXXXX`] (stesso significato di `write buffered word`), per ogni errore trovato vengono restituiti l'indirizzo e il valore letto, per segnalare il completamento dell'operazione viene spedito sul *bus FSL* un pacchetto di 32 bit tutti al valore 1.

[10]Si ricorda che 32 parole è la dimensione del buffer utilizzato durante questo tipo di programmazione.

unlock blocks; [pu-0xXXXXXXXX-0xXXXXXXXX]: si passano gli indirizzi di inizio e fine, verranno sbloccati tutti i blocchi che comprendono quegli indirizzi, viene restituito 0 se l'operazione è andata a buon fine, 1 se ci sono stati degli errori.

Struttura programma: L'applicazione non è molto complicata, in quanto non sono richieste operazioni particolarmente complicate. Si analizzerà in questo caso l'operazione di scrittura bufferizzata, le altre sono eseguite in modo analogo.

Listing 3.4: Estratto del codice C dell'applicazione per il microblaze, questa parte si riferisce alla scrittura bufferizzata

```
1  while (1){
2    // Receive the operation
3    n = XUartLite_Recv(&UartLite, &operation, 1);
4
5    [ ... ]
6
7    switch (operation){
8
9    [ ... ]
10
11   // Programming and checking operations
12   case 'p':
13   // Receive which programming operation is required
14   ReadnBytes(&UartLite, &operation2, 1, 1);
15   // Get parameters
16   switch(operation2){
17     case 'b': // Write buffered word(s)
18     case 'w': // Write word(s)
19     case 'c': // Check
20     case 'e': // Errors
21       add = get_u32(&UartLite);
22       xil_printf("-");
23       stop_add = get_u32(&UartLite);
24       xil_printf("-");
25       pattern = get_u16(&UartLite);
26       break;
27     case 'u': // Unlock blocks
28       add = get_u32(&UartLite);
29       xil_printf("-");
30       stop_add = get_u32(&UartLite);
31       break;
32     default:
33       xil_printf("\r\nUnknown operation\r\n");
34     break;
35     }
36   switch(operation2){
37     case 'b': // write buffered word(s)
38     wordNumber = stop_add - add + 1;
39
40     for(i = 0; i < wordNumber; i = i + 32){
41       if(wordNumber - i >= 32){
42         data = BUFFEREDPROGRAM_CMD;
43         putfslx(data,0,FSL_DEFAULT);
44         putfslx(add + i,0,FSL_DEFAULT);
45         putfslx(add + i + 31,0,FSL_DEFAULT);
```

```
46          putfslx(pattern,0,FSL_DEFAULT);
47          getfslx(data,0,FSL_DEFAULT);
48        if( data != 0x00000080 ){
49          xil_printf("Some errors during programm.SR = 0x%X",data);
50          break;
51          }
52        }
53        else{
54          data = BUFFEREDPROGRAM_CMD;
55          putfslx(data,0,FSL_DEFAULT);
56          putfslx(stop_add - wordNumber + i + 1,0,FSL_DEFAULT);
57          putfslx(stop_add,0,FSL_DEFAULT);
58          putfslx(pattern,0,FSL_DEFAULT);
59          getfslx(data,0,FSL_DEFAULT);
60        }
61      }
62      if( data != 0x00000080 )
63        xil_printf("Some errors during programm. SR = 0x%X", data);
64      else
65        xil_printf("Word(s) programmed. SR = 0x%X", data);
66      break;
67
68      [ ... ]
69
70      }
```

Write buffered words: Escluse le varie inizializzazioni (che nel listato non sono mostrate), la prima operazione che si fa è acquisire il carattere dell'operazione utilizzando la funzione `XUartLite_Recv(...)` (definita nel file header *uartlite_header.h* fornito anche in questo caso da Xilinx), poi grazie al costrutto `switch-case` l'esecuzione salta alla Linea 12. A questo punto viene acquisito il secondo carattere che specifica quale `checking and programming operation` si vuole eseguire e a seconda della scelta verranno recuperati i parametri necessari (con **write buffered words** si eseguiranno le Linee 21÷25, ma nel caso di **unlock blocks** si salterà alle Linee 28÷30 in quanto non c'è nessun pattern da acquisire). A questo punto si devono inviare i comandi alla periferica; dato che la memoria, nel caso di scrittura bufferizzata accetta al massimo 32 parole ad ogni scrittura dovrò (nel caso in cui l'indirizzo di fine e l'indirizzo di inizio siano separati da più di 32 parole) dividere l'operazione e far eseguire alla periferica più volte **write buffered words**, questa scomposizione viene fatta all'interno del ciclo `for` alle Linee 40÷60.
Si noti che per inviare i vari pacchetti alla periferica si utilizza la funzione `putfslx(data,0,FSL_DEFAULT)`, dove `data` è il valore che si vuole inviare; l'operazione di scrittura bufferizzata prevede, come valore di ritorno, lo *status register*, questo verrà prelevato utilizzando `getfslx(data,0,FSL_DEFAULT)` (Linee 47 e 59), se il valore prelevato è diverso dal valore di default (`0x80`) significa che qualche errore è stato rilevato dalla memoria durante la scrittura.
Per recuperare gli argomenti dalla seriale vengono richiamate `get_u32(&UartLite)`, e `get_u16(&UartLite)`, queste funzioni sono definite nella prima parte del file (non riportata nel Listato 3.4), utilizzando `XUartLite_Recv(...)` si occupano di ricevere n bit, dove n può essere 32,

16 oppure 8 (il numero riportato nel nome della funzione rappresenta il numero di bit che essa acquisisce).
I risultati e i valori restituiti dalla periferica devono essere visualizzati sullo schermo del PC, si è detto che si utilizzerà l'Hyperterminal di Windows per interagire con il sistema, la soluzione più comoda risulta quindi utilizzare la funzione `xil_printf(...)` che stampa sullo *standard output* (che in questo caso è appunto la UART e quindi la schermata di Hyperterminal), questa funziona esattamente come la classica `printf(...)` ma è realizzata in modo da essere più compatta e poter essere integrata nella memoria del μblaze.

3.4.4 Risultati parziali

Si ha a disposizione ora un sistema in grado di interagire con le due memorie, quindi di scrivere un determinato pattern al loro interno, verificare il numero di celle che cambiano contenuto dopo un irraggiamento e far visualizzare indirizzi e valori delle celle contenenti errori. Tutte le interazioni avvengono con Hyperter-

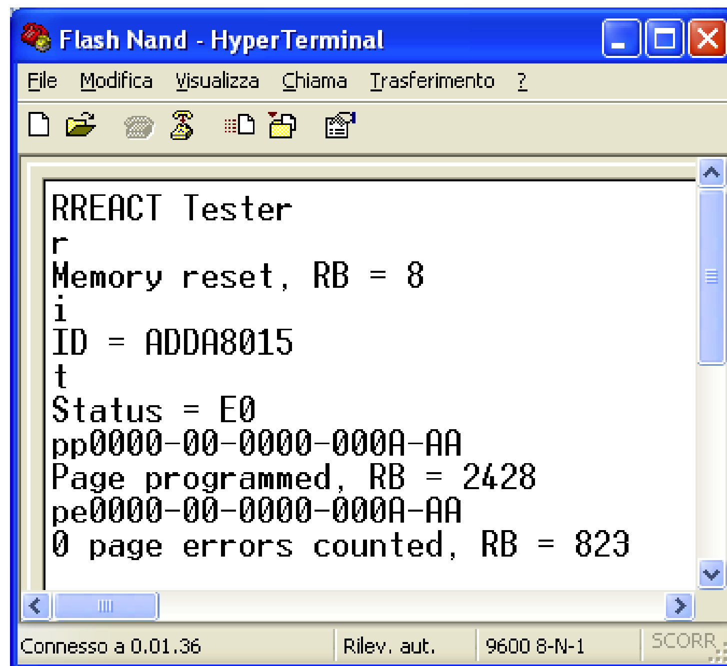

Figura 3.7: Schermata di Hyperterminal in cui si sono eseguite fatte eseguire al sistema in successione le seguenti operazioni: **Reset**, **Read ID**, **Read Status Register**, **Page Program** e infine **Page Errors**.

minal tramite la seriale (in Figura 4.2 si vede una schermata tipica con un paio di operazioni eseguite), questo non è il metodo migliore per svariati motivi:

- la RS-232 non è più ormai montata sui computer (soprattutto sui portatili) e questo obbliga all'utilizzo di convertitori USB-Seriale che non sempre funzionano a dovere
- la velocità della seriale non è molto elevata, quindi nel caso si lanciasse un'operazione che genera molto traffico sulla RS-232 (ad esempio un **check** su un gran numero di blocchi con molti errori) questa potrebbe essere un cono di bottiglia e rallentare tutto il processo

- i protocolli elettrici utilizzati per questa connessione sono abbastanza sensibili rispetto a quelli dell'interfaccia USB

Per questo si è cercato di spostare la comunicazione attraverso la porta USB, questo comporta le seguenti modifiche:

- la modifica dell'eseguibile per il microblaze, la struttura rimarrà la stessa, verranno però cambiate le funzioni relative alla comunicazione, sia in entrata che in uscita

- a livello di VHDL non verrà cambiato nulla in quanto le periferiche non sono influenzate, nel loro funzionamento, dal tipo di comunicazione utilizzato tra PC e motherboard

- sul PC non si potrà più utilizzare Hyperterminal in quanto questo non supporta connessioni su USB, per questo bisognerà sviluppare un'applicazione in grado di inviare e ricevere pacchetti da USB.

3.5 Codice sviluppato (seconda implementazione) - Interfacciamento del sistema tramite USB

Situazione di partenza: la scheda *ML402* è dotata di due porte USB, queste si possono utilizzare per collegare una periferica alla scheda (ad esempio un memoria USB) oppure per far diventare la scheda stessa una periferica da collegare al PC (la seconda è quella che si utilizzerà in questo caso). Data la complessità del protocollo USB le schede della Xilinx incorporano un chip della Cypress® (cy7c67300) in grado di gestire le due porte. Come si vede dallo schema a blocchi (vedi Figura 3.8) questo chip possiede al proprio interno un processore CY16 (architettura di tipo RISC a 16 bit) e svariati tipi di interfacce IO; nel sistema verrà utilizzata la porta HPI[11] tramite la quale il chip comunicherà col μblaze. Si otterrà quindi un sistema di comunicazione tale che il PC comunica col cy7c67300 tramite USB e questo comunica al microblaze tramite HPI. Sarà richiesto quindi **scrivere un firmware** da far eseguire al processore CY16 che gestisca le comunicazioni sia dal lato USB (ovvero lato PC) che da quello HPI (ovvero lato microblaze).

Come guida alla programmazione la Cypress fornisce dei programmi di esempio, questi sono pensati e scritti per delle schede di sviluppo costruite dalla stessa Cypress, ma possono essere utilizzati (con qualche limitazione) anche per i chip montati sulle schede della Xilinx. Tra questi, quello che più può essere d'aiuto in questo caso è un semplice contatore rappresentato da un software scritto in **Visual Basic** che virtualizza un display a 7 segmenti e dei bottoni sullo schermo del computer, il contatore incrementa il proprio valore (e lo visualizza sul display) sia se viene premuto un bottone reale sulla scheda di sviluppo sia se ne viene premuto uno virtuale sull'applicazione. Questo semplice contatore è realizzato in maniera tale che ci siano dati che viaggiano in entrambe le direzioni sulla USB,

[11]HPI - Host Port Interface.

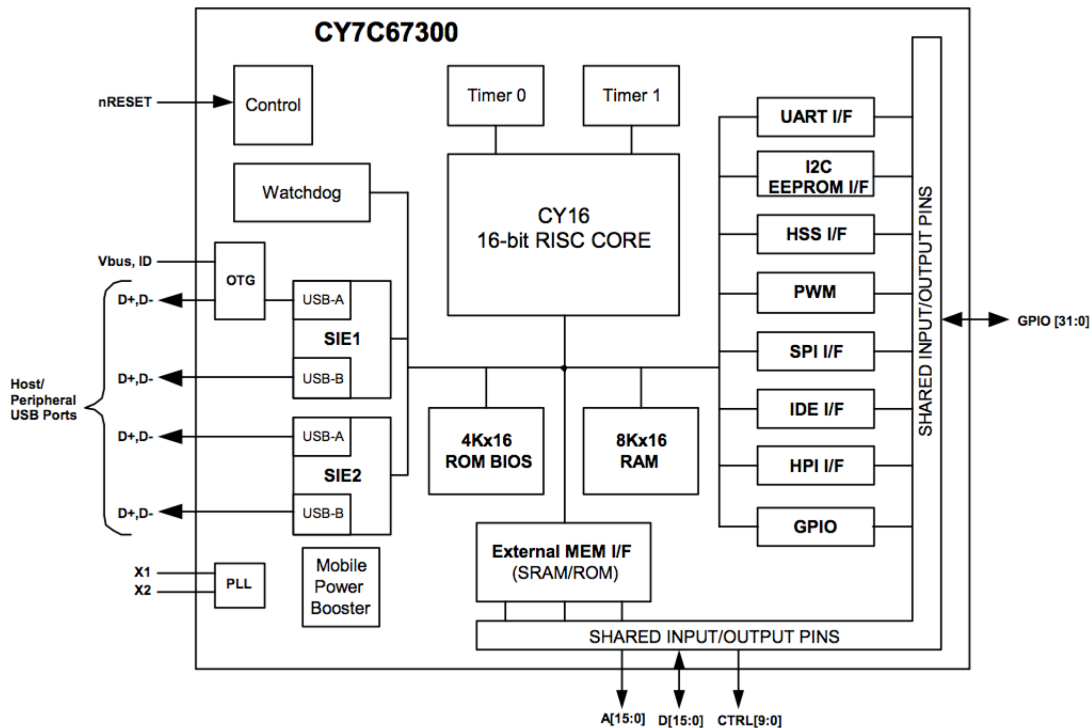

Figura 3.8: Estratto del datasheet del Cypress cy7c67300: diagramma a blocchi del chip, si notino sulla parte sinistra i blocchi SIE1 e SIE2 responsabili dell'interfacciamento USB e al centro il processore CY16.

proprio per questo si adatta bene al sistema di test per le memorie che si sta cercando di realizzare; in particolare la comunicazione utilizzerà il protocollo HID[12], in questo modo non sarà necessario scrivere driver per il PC in quanto Windows integra questi driver generici e le funzioni per gestirli; per questo motivo si partirà da questo progetto di esempio per lo sviluppo delle nuove parti di codice per far funzionare il sistema di test per le memorie attraverso la porta USB.

Integrazione del cy7c67300 all'interno del sistema: Per inserire il chip (che ovviamente è già presente sulla *ML402*) è necessario impostare correttemente l'FPGA, questa operazione viene effettuata all'interno del XPS, aggiungendo un bus di comunicazione mappato in memoria e collegato ai pin del chip, così facendo per comunicare tramite HPI si potranno usare le funzioni già presenti negli header del microblaze `XIo_Out16(...)` e `XIo_In16(...)` che consentono la scrittura e la lettura di locazioni di memoria.

3.5.1 Firmware per il cy7c67300

Per questa parte si è partiti dai sorgenti del contatore descritto nei paragrafi precedenti. Il chip è programmato utilizzato un BIOS[13] che si occupa della gestione di tutte le funzionalità del chip; utilizzando dei **#define** e degli **#undef** all'interno del file header *fwxcfg.h* si imposta il BIOS per abilitare e settare tutte

[12]HID - Human Interface Device. Protocollo USB utilizzato in periferiche come tastiere, mouse, joystick.
[13]BIOS - Basic Input-Output System.

le funzionalità di cui dispone il chip, verranno ad esempio disabilitati i pulsanti e il display della scheda e verranno abilitate la porta HPI e la USB.

Ricezione dei dati inviati dal computer: Sarà necessario a questo punto modificare il codice originale in modo che una volta ricevuto un pacchetto dalla USB questo venga inviato al microblaze; lo spezzone di codice seguente si occupa proprio di questa parte.

Listing 3.5: Spezzone di codice dell'applicazione per il cy7c67300; questa parte si occupa di passare al microblaze i pacchetti ricevuti dalla USB.

```
1 void setup_inData_report_callback( void ) {
2   inData_report_info.buffer = &inData_report;
3   inData_report_info.length = sizeof(inData_report);
4   inData_report_info.done_func = (PFNINTHANDLER) &
        inData_report_received;
5   susb_receive ( SIE1, 2, &inData_report_info);
6
7   if(waitingAck == TRUE && inData_report == 0xFFFFFFFF){
8     waitingAck == FALSE;
9     // If an ack is received reset the flag
10    WRITE_REGISTER( DATA_FLAG , 0x0);
11    }
12  else{
13    // Send via HPI using two registers
14    WRITE_REGISTER(OUT_DATA1, (uint16)((inData_report >> 16) & 0
        xFFFF));
15    WRITE_REGISTER(OUT_DATA0, (uint16)(inData_report & 0xFFFF));
16    }
17 }
```

La funzione `setup_inData_report_callback()` viene richiamata automaticamente dal BIOS quando viene ricevuto un *Report*[14], alla Linea 5 viene richiamata `susb_receive(...)`, questa è una funzione del BIOS che salva il report ricevuto, assieme alle sue informazioni, all'interno di `inData_report_info` il cui indirizzo gli viene passato come argomento. Quest'ultima è una struttura che raggruppa alcune informazioni sul pacchetto, la parte più importante è il campo `buffer` che contiene l'indirizzo dove salvare il dato ricevuto (tale indirizzo viene impostato nella Linea 2).
In fase di invio dei dati, se questi non vengono ricevuti subito c'è il rischio che vadano persi (non dovrebbe succedere ma se ne discuterà più avanti), per evitare questo ho inserito un controllo, ogni volta che un pacchetto viene inviato al PC esso risponde con un pacchetto di acknowledgement detto ACK segnalando che il dato è stato ricevuto ed è possibile inviarne un altro. L'ACK è un normale pacchetto il cui contenuto è `0xFFFFFFFF`.
Il costrutto `if-else` (Linee 7÷16) serve proprio per la gestione dell'ACK; se sto aspettando un acknowledgement da parte del PC (`waitingAck` sarà `TRUE`), quando ricevo un pacchetto devo verificare se questo è un ACK, se lo è dovrò solamente informare il microblaze che ora è possibile inviare altri pacchetti e per farlo imposto a zero un registro chiamato `DATA_FLAG` accessibile anche dal processore tramite HPI. Se invece non si sta' attendendo nessun acknowledgement

[14] Un pacchetto di dati sulla porta USB viene anche chiamato report.

significa che il pacchetto ricevuto via USB deve essere trasmesso al microblaze, per fare ciò si scrive il dato in altri due registri[15] (OUT_DATA1 e OUT_DATA1) anch'essi raggiungibili attraverso la HPI e quindi recuperabili dal microblaze.

Invio dati al computer: L'invio dei dati è effettuato con una procedura molto simile a quella appena descritta per la ricezione. Il microblaze memorizza il valore da inviare in due registri del cy7c67300 (IN_DATA0 e IN_DATA1), dopodiché mette DATA_FLAG = 1 questo comunica al chip che gli è richiesto di inviare sulla USB un report che trasmetta i dati contenuti nei due registri. Il chip li invia utilizzando la funzione susb_send(...) e pone waitingAck = TRUE così da mettersi nella condizione di attesa l'*ACK*.

3.5.2 Modifiche apportate all'applicazione per il μblaze

La struttura del programma che esegue il microblaze non viene variata, i codici delle operazioni rimangono gli stessi e anche il formato dei parametri non cambia vengono invece variate le funzioni che comunicano col PC. Nella prima versione si utilizzava XUartLite_Recv(...) per la ricezione e xil_printf(...) per la trasmissione dei risultati; si sono quindi create le funzioni get_u32 e put_u32 che in maniera simile ricevono e inviano utilizzando la USB invece che la seriale.

get_u32: Nel caso della ricezione di dati dal PC si fa il polling dello stato della porta HPI, quando il cy7c67300 ha ricevuto un dato dalla USB (e quindi il firmware ha già copiato il valore in ingresso nei due registri OUT_DATA0/1) il quinto bit del registro di stato della porta HPI diventa uno, una volta verificata la presenza di un valore in arrivo basta leggerlo dai due registri (dopo la lettura il BIOS azzera automaticamente il bit del registro di stato della porta HPI).
La funzione get_u32(...) (Listato 3.6) è stata realizzata scompattando il lavoro in due parti eseguite da funzioni separate richiamate in successione: wait_next_data() si occupa di fare il polling del registro di stato della porta HPI mentre USB_Recv() preleva il dato dai registri del cy7c67300.

Listing 3.6: Estratto del firmware riguardante la funzione get_u32. Per la sua realizzazione sono state utilizzate due funzioni di supporto che si occupano rispettivamente di aspettare la presenza di un nuovo dato dalla USB e di leggerlo

```
1  void wait_next_data(void){
2    u16 data16 = 0;
3    // Wait untill the 5th bit of HPI status become 1
4    do{
5      data16 = XIo_In16(USB_STATUS);
6      data16 = data16 & 0x0010;
7      data16 = data16 >> 4;
8    }while(data16 != 1);
9  }
10 //=====================================================
11 u32 USB_Recv(void){
12
```

[15]Si ricorda che la dimensione dei registri del processore interno al chip della cypress è 16 bit mentre il buffer dei report è di 32 bit.

```
13   u32 data;
14   // Receive the first part of data
15   XIo_Out16( USB_ADDRESS, OUT_DATA1 );
16   data = XIo_In16( USB_DATA );
17   data = data << 16;
18   // Receive the second part of data
19   XIo_Out16( USB_ADDRESS, OUT_DATA0 );
20   data = data | XIo_In16( USB_DATA );
21   return data;
22 }
23 //=====================================================
24 u32 get_u32(void){
25   wait_next_data();
26 
27   return USB_Recv();
28 }
```

put_u32 In maniera analoga a quanto fatto per la ricezione anche per la trasmissione si realizza una funzione che si occupa della trasmissione di dati. Questa deve prima di tutto verificare che il chip al momento sia in grado di inviare dati (quindi controllare il valore del registro `DATA_FLAG`) e in caso contrario bisogna aspettare. Una volta che il chip è pronto ad eseguire una trasmissione il dato viene memorizzato nei due registri `IN_DATA0/1` dopodiché si alza `DATA_FLAG` a uno, questo comunica al firmware che i dati presenti nei due registri sono pronti per essere inviati. Anche in questo caso la funzione è composta da una parte che si occupa di verificare VERIFICAREEEE

Listing 3.7: Estratto del firmware riguardante la funzione `put_u32`.

```
1 void put_u32(u32 Out_data){
2   u16 temp;
3   temp = (u16)(Out_data & 0xFFFF);
4   wait_ready_to_send_next_data();
5 
6   // Put the first part of data inside the cy7c67300
7   XIo_Out16(USB_ADDRESS, IN_DATA0);
8   XIo_Out16(USB_DATA, temp);
9   temp = (u16)((Out_data >> 16) & 0xFFFF);
10 
11   // Put the second part of data inside the cy7c67300
12   XIo_Out16(USB_ADDRESS, IN_DATA1);
13   XIo_Out16(USB_DATA, temp);
14   // Set flag to 1
15   XIo_Out16(USB_ADDRESS, DATA_FLAG);
16   XIo_Out16(USB_DATA, 1);
17   return;
18 }
19 //=====================================================
20 u32 wait_ready_to_send_next_data( void ){
21   u32 data = 0;
22   // Wait untill the cy7c67300 is ready to send new data
23   do{
24     XIo_Out16(USB_ADDRESS, DATA_FLAG);
25     data++;
26   }while((u8)XIo_In16(USB_DATA) == 1);
```

```
27   return data;
28 }
```

3.5.3 Programma per la gestione del sistema dal PC:

Il sistema è ora pronto serve però avere a disposizione un'interfaccia sul PC che ci consenta di inviare pacchetti tramite USB così da verificarne il funzionamento e le prestazioni. Anche in questo caso si è preso spunto dalle funzioni base per l'invio e la ricezione trovate nei file sorgenti del contatore descritto precedentemente, queste sono realizzate utilizzando le librerie a collegamento dinamico di Windows (DLL), vengono in particolare importate *hid.dll* e *setupapi.dll* e grazie a queste si hanno a disposizione le funzioni per la gestione della USB.
Il programma è stato realizzato con Visual C++, la semplice interfaccia

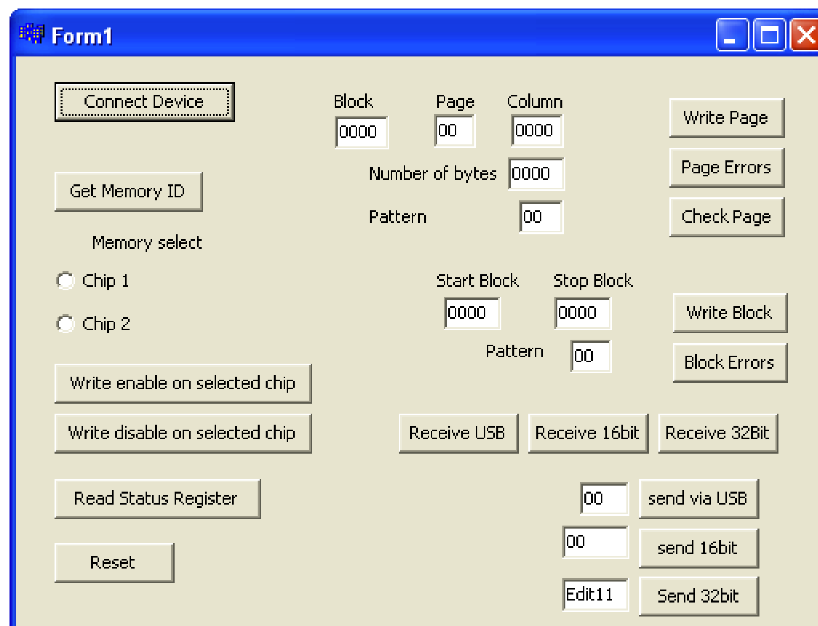

Figura 3.9: Interfaccia grafica del programma scritto per interagire con la memoria via USB, questa è una versione fatta solo per testare il funzionamento del sistema, non è adatta ad eseguire i test di irraggiamento.

realizzata è mostrata in Figura 3.9, si vede che si riescono a inviare le funzioni base come programmazioni e conteggio errori. Durante la sua scrittura ho mantenuto in funzione anche Hyperterminal, così da poter inviare i comandi da USB e riceverli tramite seriale, in questo modo il debug è risultato più semplice. L'interfaccia non è molto elaborata perché questa deve servire solo come test, per verificare che il collegamento USB funzioni, per i test veri e propri verranno sviluppati altri programmi con la possibilità di inserire script.

Capitolo 4

Problemi riscontrati, risultati ottenuti, conclusioni

4.1 Prima implementazione

Inizialmente si sono avuti parecchi problemi sulla periferica VHDL ma erano legati principalmente alla non completa conoscenza degli strumenti utilizzati sia software che hardware. Per risolvere tali problemi le prime prove si sono fatte senza utilizzare il microblaze, creando un semplice progetto con ISE[1] sia per semplificare il sistema ma soprattutto per risparmiare tempo dato che al compilazione di un progetto con XPS può impiegare anche più di 10 minuti. Ovviamente eliminando il processore non si ha più a disposizione la comunicazione attraverso la seriale, quindi ad esempio per leggere l'*ID* si è dovuto utilizzare un oscilloscopio collegato ai pin della memoria, così facendo sullo schermo si potevano leggere i valori binari dei vari bit. L'oscilloscopio è anche stato utilizzato per settare i parametri `SLEW` e `DRIVE`[2] dei pin dell'FPGA in modo da avere dei segnali senza sovraelongazioni.
Un altro grosso problema era legato al segnale di *Ready-busy*, si è scoperto (grazie all'aiuto di Simone Gerardin) che quando `RB` variava in corrispondenza di un fronte di salita del clock si creavano errori per cui il *bus FSL* si bloccava. Per risolvere questi problemi è bastato rendere sincrone (attraverso un flip flop) anche le variazioni del segnale `RB`.

4.2 Seconda implementazione

Una volta verificato il corretto funzionamento della prima implementazione si è iniziato a scrivere le modifiche per il funzionamento tramite USB. Questa parte è stata molto più densa di problemi, collegare il cy7c67300 al microblaze è inizialmente sembrato molto complicato, le difficoltà si sono superate una volta trovato sul web una guida per l'integrazione di tale chip in un progetto di XPS.
Anche la scrittura del firmware per il chip Cypress non è stata facile, in questo

[1] Utilizzando ISE si possono creare circuiti logici che non utilizzano processori ed effettuare le simulazioni in maniera abbastanza semplice.

[2] Questi parametri definiscono la velocità e la forza che devono avere i pin del FPGA durante le variazioni.

caso le difficoltà sono imputabili alla difficoltà nella comprensione del protocollo USB, anche con l'aiuto della molta documentazione presente sul web, la struttura dei file di esempio da cui si è partiti non era molto chiara.
Un grosso problema che non sono stato in grado di risolvere riguarda la ricezione dei pacchetti USB da parte del PC, questi difatti vengono sovrascritti come se non fosse presente un buffer che li memorizza, questo è stato risolto realizzando un semplice protocollo che fa utilizzo di pacchetti di ACK già descritti nella Sezione 3.5.1, inoltre se le letture vengono effettuate velocemente capita che in due letture successive venga letto lo stesso pacchetto. Ho provato a modificare molte cose sia sul software del PC sia sul firmware ma questi problemi persistono.
Altra cosa su cui è necessario lavorare ancora è la velocità, la comunicazione via USB risulta difatti molto più lenta della seriale, questo probabilmente è legato a un non correttissimo utilizzo del protocollo USB e probabilmente anche al problema citato nel paragrafo precedente.

4.3 Foto varie

Prima di trarre le conclusioni sul tirocinio vengono proposte alcune foto della scheda utilizzata per realizzare il sistema; la prima è una foto della motherboard con montata la dougtherboard sulla quale è presente la memoria Flash. Nella seconda si vede la motherboard subito prima di fare un test di irraggiamento ai laboratori di fisica di Legnaro.

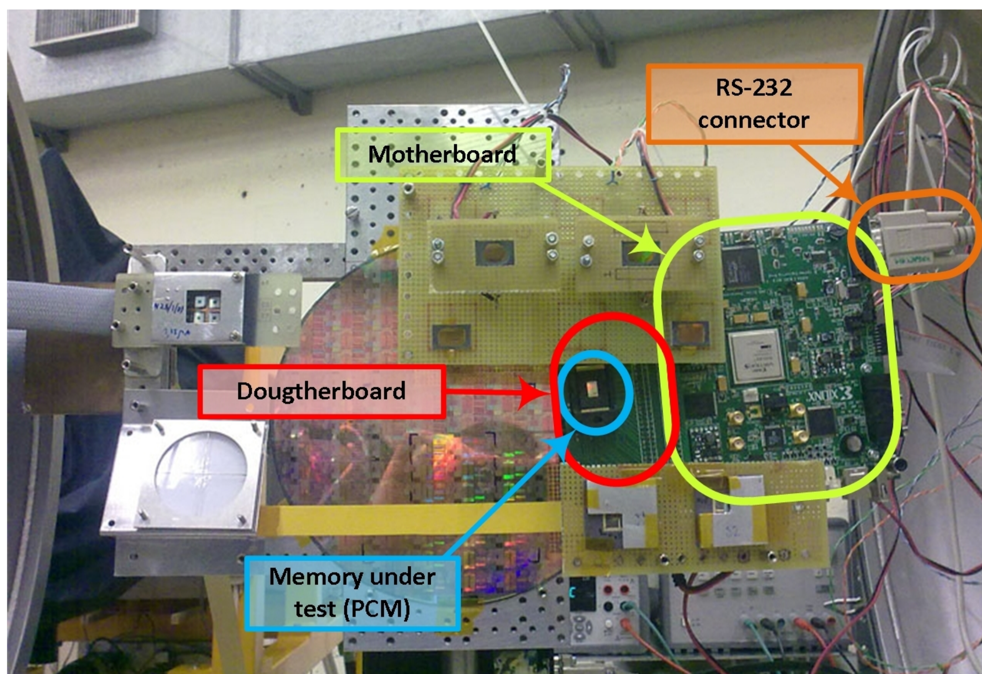

Figura 4.1: Foto scattata all'acceleratore di particelle di Legnaro prima di un irraggiamento; si vede la motherboard e gli altri elementi del sistema di test, sono montati assieme ad altri dispositivi che devono essere irraggiati (si vede anche un disco di silicio su cui sono realizzati circuiti ancora da inserire all'interno dei package) su un supporto che verrà poi inserito in una camera dove arriverà il fascio di particelle direttamente dall'acceleratore.

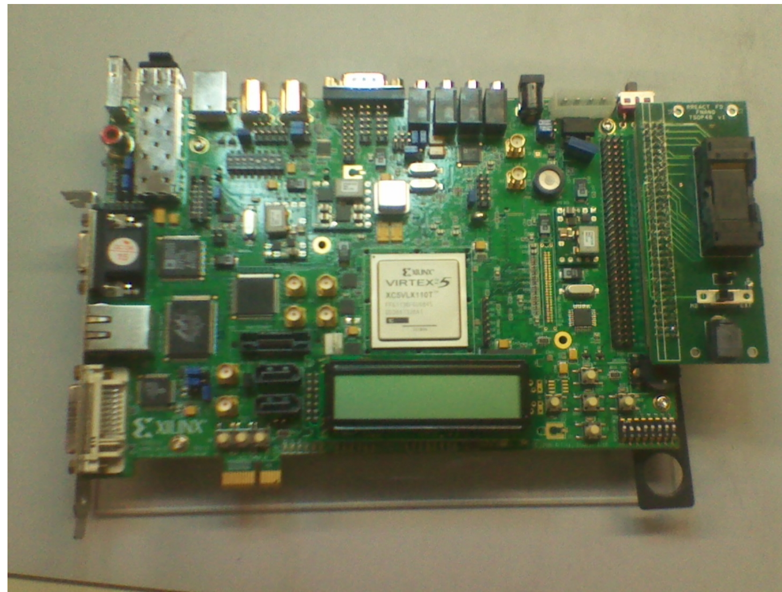

Figura 4.2: Foto della scheda della Xilinx con montata la memoria Flash (sulla destra).

4.4 Conclusioni

L'attività del tirocinio dal mio punto di vista è stata un'esperienza molto stimolante; dover realizzare un sistema con così tanti elementi da far interagire sembrava all'inizio qualcosa di veramente molto difficile, dopo invece analizzando i blocchi separatamente sono riuscito ad avere una visione più chiara del sistema e sono riuscito a portare a termine (non senza difficoltà che però sono state superate) il lavoro che mi era stato assegnato. É stato molto soddisfacente vedere che l'insieme di periferica e microblaze riuscivano interagire con la Flash e la PCM esattamente nel modo che si voleva. Riguardo ai componenti mi è capitato parecchie volte di non trovare le informazioni che cercavo sui relativi datasheet, questo a volte era dovuto a delle carenze proprie dei datasheet, molte altre volte invece era solo colpa della mia scarsa esperienza nel campo, esperienza che però durante il tirocinio sono riuscito, almeno in parte, ad acquisire.

Bibliografia

[1] New Approach - Directives & Standards http://www.newapproach.org/Directives/DirectiveList.asp

[2] DIRETTIVA 2004/108/CE DEL PARLAMENTO EUROPEO E DEL CONSIGLIO

[3] James F. Ziegler, William A. Lanford - IBM Research Center, University of New York at Albany - The Effect of Sea Level Cosmic Rays on Electronic Devices - 1980

[4] Marta Bagatin - Effects of Ionizing Radiation in Flash Memories - 2010

[5] wikipedia - http://it.wikipedia.org/wiki/Compatibilit%C3%A0_elettromagnetica

[6] Adam David Fogle, Don Darling, Richard C. Blish, II, Senior Member, IEEE, and Eugene Daszko - Flash Memory under Cosmic & Alpha Irradiation - 2004

[7] Giorgio Cellere, Member, IEEE, Alessandro Paccagnella, Member, IEEE, Angelo Visconti, Mauro Bonanomi, S. Beltrami, Jim R. Schwank, Fellow, IEEE, Marty R. Shaneyfelt, Fellow, IEEE, and Philippe Paillet, Senior Member, IEEE -Total Ionizing Dose Effects in NOR and NAND Flash Memories - 2007

[8] Robert Baumann - Texas Instruments - Design & Test of Computers, IEEE p.258-266 (Soft Errors in Advanced Computer Systems) - ISSN: 0740-7475 - 2005

[9] Andrea Cester - Slide del corso di circuiti integrati digitali 2009

[10] Daniele Vogrig - Slide del corso di programmazione di sistemi digitali 2009

[11] H.-S. Philip Wong, Fellow IEEE, Simone Raoux, Senior Member IEEE, Sang-Bum Kim, Jiale Liang, Student Member IEEE, John P. Reifenberg, Bipin Rajendran, Member IEEE, Mehdi Asheghi, and Kenneth E. Goodson - Phase Change Memory, a comprehensive and thorough review of PCM technologies - 2010

[12] Matthew J. Breitwisch - IBM/Macronix PCRAM Joint Project IBM T J Watson Research Center Yorktown Heights, New York, USA - Phase Change Memory - 2008

[13] Sean Eilert, Numonyx Director of Architecture, Principal Engineer; Mark Leinwander, Numonyx Systems Manager, Principal Engineer; and Giuseppe Crisenza, Numonyx Vice President of Strategic Alliances - Phase Change Memory (PCM): A new memory technology to enable new memory usage models - 2009

[14] Hynix® - 2Gbit (256Mx8bit / 128Mx16bit) NAND Flash Memory datasheet(rev.0.5)

[15] Numonix® - Numonyx® OmneoTM P8P PCM 128-Mbit Parallel Phase Change Memory datasheet (rev.06)

[16] Xilinx® - LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c) datasheet (*fsl_v20-2.pdf*)

[17] John Hyde - USB Design By Example

Ringraziamenti

Ringrazio prima di tutto i miei genitori per avermi dato la possibilità di arrivare fino a questo punto, gli amici e colleghi con cui mi sono confrontato e divertito in questi ultimi anni. Un ringraziamento a Alessandro Paccagnella per avermi dato l'opportunità di fare il tirocinio nel laboratorio RREACT e in particolare a Simone Gerardin per la sua disponibilità durante tutta la durata del tirocinio.