Università Politecnica delle Marche

Facoltà di Ingegneria

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

Tesi di Laurea

CODICI CORRETTORI D'ERRORE IN MEMORIE NON VOLATILI PER APPLICAZIONI BIOMEDICHE

ERROR CORRECTION CODES IN NON-VOLATILE MEMORIES FOR BIOMEDICAL APPLICATIONS

Laureanda:

Jennifer Guaitini

Relatore:

Franco Chiaraluce

Anno Accademico 2019 – 2020

INDICE

INTRODUZIONE

Nell'ambito biomedico è spesso necessario analizzare una enorme mole di dati per poter trovare delle regolarità, dei pattern, che si ripetono in concomitanza con certi eventi. Questa correlazione tra cause e eventi può essere sfruttata nel campo medico per determinare gli agenti che hanno un ruolo primario e secondario nell'avvenimento di una determinata patologia.

Da questo tipo di analisi può scaturire la capacità di prevenire, curare o monitorare lo stato di salute di un gruppo di individui sulla base di dati statistici.

Per lavorare in maniera efficace su una mole estremamente elevata di dati (i quali possono essere oggigiorno disponibili in rete in maniera condivisa), ogni aspetto dovrebbe essere analizzato. Non si tratta semplicemente di valori numerici (ad esempio pressione sanguigna, glicemia…), ma anche di immagini (come radiografie, ECG, EEG ...) che prese nel loro complesso potrebbero rivelare delle ripetizioni associabili a determinate patologie. L'occhio esperto di medici e ricercatori è sicuramente stato in grado di effettuare queste associazioni, tuttavia poter analizzare contemporaneamente molti più casi e individuare pattern non ovvii richiederebbe molto tempo e personale, con possibilità di errore dovuto a fattori umani.

In tempi recenti si è ricorsi sempre più all'utilizzo di tecniche di intelligenza artificiale nei campi più disparati dell'attività umana, in particolare l'utilizzo dell'intelligenza artificiale nella classificazione dei dati medici a fini diagnostici e terapeutici ha conosciuto un grande sviluppo [1].

Alla base delle tecniche di intelligenza artificiale si trova il concetto di rete neurale. Le reti neurali sono delle strutture topologiche che possono essere descritte matematicamente e che sono in grado di "imitare" il comportamento dei neuroni del sistema nervoso (da cui il nome neurale) le quali hanno la capacità di processare velocemente le grandezze di ingresso dopo aver "imparato" la giusta associazione tra ingresso e uscita attraverso la modifica di parametri interni detti *pesi*.

La differenza tra una rete neurale e un algoritmo scritto in un linguaggio di programmazione è che quest'ultimo è sviluppato conoscendo a priori il problema che dovrà risolvere e tutti i passi necessari per risolverlo, mentre una rete neurale ha una struttura generica che dovrà "apprendere" la giusta associazione tra stimoli di ingresso e uscita corretta, attraverso una serie di esempi ad essa presentati, esattamente come avviene in una mente umana che impara dall'esperienza. La rete neurale addestrata (tramite algoritmi come il *Back Propagation*) è quindi in grado di associare a nuovi stimoli mai visti determinate uscite.

Le reti neurali sono costituite da un gran numero di piccoli e semplici nuclei di elaborazione (neuroni) interconnessi tra loro in una struttura a strati, vedi figura 1 (fonte Wikipedia). La teoria matematica che le descrive è nota da diversi decenni, e sta vivendo un momento di sviluppo pratico grazie alla grande potenza di calcolo oggi disponibile. Tramite le reti neurali possiamo simulare relazioni complesse tra ingressi e uscite. Nel caso clinico questo significa migliorare la diagnostica di malattie: presentando durante la fase di apprendimento un gran numero di dati relativi a soggetti malati e soggetti sani. Si crea cioè un'associazione tra un insieme di dati in ingresso e categorie di malattia.

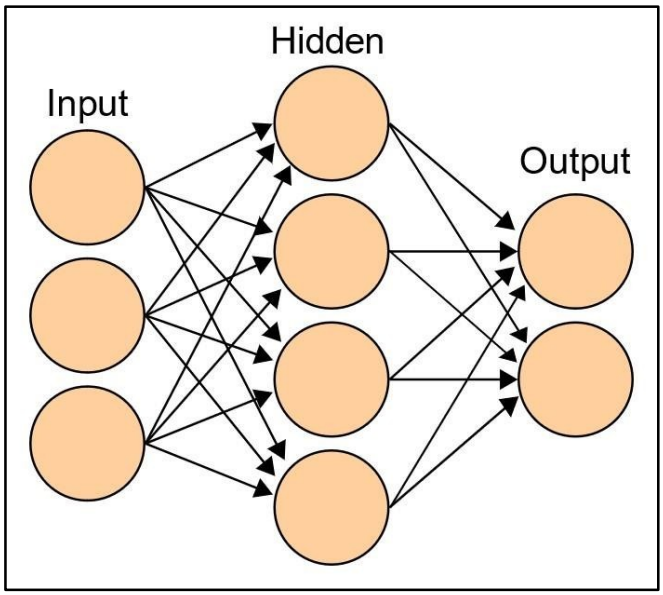

Figura 1

Le architetture di reti neurali possono essere di vario tipo, così come il meccanismo di apprendimento, ma tutte concentrano il risultato dell'apprendimento nel valore dei pesi: è chiaro, cioè, che la caratterizzazione della rete neurale al termine dell'apprendimento è tutta contenuta nelle matrici dei pesi e nei vettori di *bias*, che altro non sono che numeri. Potremmo dire che l'equivalente della memoria del cervello, cioè quello che noi ricordiamo e ci permette di giudicare eventi futuri, nella rete neurale è rappresentata dai pesi.

Ovviamente questi pesi debbono essere mantenuti a tempo indefinito ed eventualmente modificati sulla base di nuovi apprendimenti. Una rete neurale molto evoluta è caratterizzata da un gran numero di pesi.

Nelle implementazioni pratiche tutto questo si traduce nel memorizzare dei valori in una memoria "non volatile", cioè in grado di mantenere il proprio contenuto anche in mancanza di alimentazione elettrica. Questa memoria deve essere:

1) Densa, cioè capace di contenere molte informazioni in piccoli spazi;
2) Riscrivibile, cioè capace di essere cancellata e riletta;
3) Veloce, cioè capace di essere modificata e letta in tempi comparabili con gli algoritmi di apprendimento della rete neurale;
4) Affidabile, cioè capace di essere programmata e letta molte volte e in grado di contenere l'informazione senza perdita di dati per il maggior tempo possibile.

Lo scopo del presente lavoro è illustrare alcune tecniche utilizzate per la correzione di errori che possono verificarsi nelle memorie flash, in particolare si parlerà dei codici *Low-Density Parity-Check* o LDPC.

Negli ultimi decenni la tecnologia delle memorie flash ha reso possibile l'ottenimento delle caratteristiche sopra elencate.

La necessità di introdurre codici di correzione di errore nelle memorie flash è dettata dal fatto che la prestazione di tali memorie si è fatta via via più spinta, di pari passo con le esigenze sempre più stringenti delle nuove applicazioni. Se da un lato è stato possibile miniaturizzare lo spazio occupato su silicio da tali dispositivi o, a parità di spazio, contenere sempre più dati, dall'altro i vincoli delle tecnologie a semiconduttore hanno

reso più delicati i singoli dispositivi (celle flash) cosicché le non idealità di funzionamento risultano sempre più importanti.

L'architettura NAND ad esempio, che è quella attualmente più diffusa, soffre di problematiche quali la perdita di dati al passare del tempo, l'usura causata da frequenti cicli di scrittura e cancellazione, i disturbi causati dalla lettura, scrittura, cancellazione di celle adiacenti e fenomeni correlati. Questi effetti sono più gravi oggi rispetto a qualche anno fa a causa della diminuzione degli spessori degli ossidi nel silicio e dell'impiego della memorizzazione multilivello.

Se da un lato c'è un lavoro in corso per migliorare la qualità costruttiva di tali memorie, al tempo stesso molta ricerca viene svolta nell'ambito della "correzione" degli errori quando capitano. In sostanza si tratta di attuare tutte quelle tecniche che, per mezzo di una ridondanza, minimizzano da un lato la possibilità di insorgenza di errore, dall'altro offrono la possibilità di correggere gli errori che si verificano.

Potremmo fare un parallelismo tra le problematiche di trasmissione dati e quelle di memorizzazione. Nel primo caso un flusso di dati che parte dalla sorgente è separato da uno *spazio* prima di arrivare ad un ricevitore. Durante il tragitto le non idealità del canale tendono a corrompere il flusso dei dati che arriva alterato al ricevitore. Nel secondo caso un insieme di dati è scritto in una memoria ed è separato da un *tempo* arbitrario rispetto al momento in cui sarà riletto. Le non idealità della memoria tenderanno a corrompere il dato in lettura.

Da quanto detto si può osservare che le due problematiche hanno molti punti in comune, e quindi anche gli approcci per risolverle sono molto simili.

In particolare, così come le tecniche di codifica e correzione di errore sono utilizzate con successo nell'ambito delle telecomunicazioni, le stesse tecniche vengono impiegate nella codifica dei dati memorizzati nelle celle flash.

Nel seguito (capitolo I) verranno innanzitutto descritte le caratteristiche generali e il principio di funzionamento delle memorie non volatili flash, in particolare le due architetture più popolari, le NOR e le NAND.

A seguire (capitolo II) verranno indicate le problematiche che portano alla corruzione dei dati in memoria, infine si descriverà il codice di correzione di errore LDPC che ha trovato largo impiego nelle memorie flash ordinarie e multilivello.

Desidero ringraziare il Prof. Chiaraluce, relatore di questa tesi, per l'interessante lavoro proposto e per la sua disponibilità.

Capitolo I

Architetture di memorie non volatili Flash

Introduzione

Le memorie a semiconduttori si dividono in due categorie principali: RAM, Random Access Memories, e ROM, Read Only Memories. La RAM perde i suoi contenuti non appena la corrente viene interrotta, mentre la ROM la mantiene, nominalmente, "per sempre". La categoria di memoria non volatile include tutti i tipi di memorie i cui contenuti possono essere cambiati elettricamente e che, una volta che non siano più forniti di energia, sono anche in grado di mantenerli.

La storia delle memorie non volatili inizia negli anni '70 con l'introduzione della EPROM memory, Erasable Programmable Read Only Memory [2]. Da quel momento le memorie non volatili divennero la più importante famiglia nell'ambito delle memorie a semiconduttore e, fino agli anni '90, il loro interesse era più legato al ruolo svolto come ausilio nello sviluppo di nuove tecnologie piuttosto che alla convenienza economica in prodotti a larga diffusione. Dall'inizio degli anni '90, con l'introduzione delle memorie Flash non volatili [3] in dispositivi portatili il mercato di queste memorie è aumentato esponenzialmente.

Con il passare del tempo, l'evoluzione della tecnologia per le memorie non volatili, chiamate floating gate memories, ha permesso la riduzione dell'area di una singola cella di memoria, che è passata da 4 a 0,4 micrometri quadrati. Anche la capacità di storage è passata da 16 Mb, nel 1993, a 16 Gb, nel 2008, con una variazione equivalente a 1000 volte, in perfetto accordo con la legge di Moore.

In una Flash memory le informazioni vengono registrate in un transistor MOSFET floating gate, un tipo di transistor in grado di mantenere la carica elettrica per lungo tempo. Ogni transistor costituisce una cella di memoria che conserva il valore di uno o più bit.

Le memorie flash sono di due tipi: NOR [4] e NAND [5], che differiscono nell'architettura e nel procedimento di programmazione, cancellazione, lettura.

I.1 MOSFET a gate flottante

Le memorie flash sono basate su MOSFET a gate flottante. Sono transistor ad effetto di campo la cui struttura è simile a un MOSFET con però l'aggiunta di un ulteriore terminale gate, detto *floating gate* (gate flottante), situato tra il substrato e il *control gate* (terminale di gate accessibile dall'esterno).

Il condensatore MOS (Metallo Ossido Semiconduttore) è costituito da tre materiali: un metallo, che può essere o alluminio o silicio policristallino, un ossido, biossido di silicio che fa da isolante e non permette il passaggio di corrente, e un semiconduttore, che può essere o un silicio di tipo p o di tipo n.

Nel caso di un MOSFET (a canale n) la sua struttura è costituita da un substrato (Bulk) di silicio drogato p, due isole con drogaggio n (Source e Drain) contattate da metallo, un isolante nella zona centrale e sopra una zona di silicio policristallino, detto anche polisilicio, (Gate).

La struttura ha quindi 4 terminali: Source, Drain, Gate e Bulk.

Come noto l'equazione che descrive la corrente di drain in funzione della tensione sui tre terminali, supponendo per comodità V_b=V_s è:

$$I_d = \begin{cases} 0 & V_{gs} < V_{th} \\ \beta\left[(V_{gs} - V_{th})V_{ds} - \frac{V_{ds}^2}{2}\right] & V_{gs} \geq V_{th} \quad V_{ds} < V_{gs} - V_{th} \\ \frac{\beta}{2}(V_{gs} - V_{th})^2 & V_{gs} \geq V_{th} \quad V_{ds} \geq V_{gs} - V_{th} \end{cases} \tag{1.1}$$

In una cella flash è presente un gate flottante (non connesso ad alcun terminale) come mostrato in Figura 1

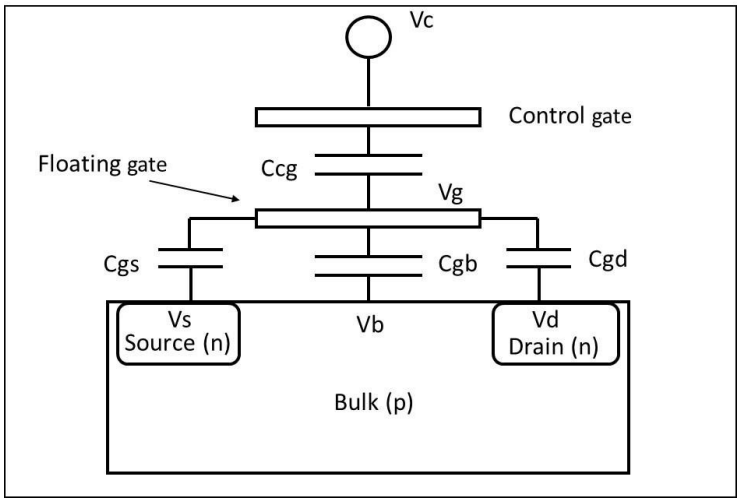

Figura 1

Dalla figura si può vedere che sono presenti dei *condensatori parassiti*. Andando a risolvere il circuito elettrico equivalente e supponendo che una quantità di carica *Q* sia rimasta intrappolata nel floating gate possiamo legare la tensione nel floating gate alle tensioni sugli altri terminali secondo la formula: (supponendo che $V_b = V_s = 0$)

$$V_g = \frac{Q}{C_{TOT}} + \frac{C_{gd}}{C_{TOT}} V_d + \frac{C_{cg}}{C_{TOT}} V_c \qquad (1.2)$$

Dove $C_{TOT} = C_{cg} + C_{gb} + C_{gs} + C_{gd}$

Il mosfet equivalente si potrà accendere solo quando $V_g > V_{th}$ quindi

$$V_c > (V_{th} - \frac{Q}{C_{TOT}} - \frac{C_{gd}}{C_{TOT}} V_d) \frac{C_{TOT}}{C_{cg}} \qquad (1.3)$$

Perciò la tensione sul control gate deve essere maggiore di una tensione di soglia equivalente.

Questa tensione di soglia dipende dalla carica *Q*. Se *Q* è negativa (elettroni) la tensione aumenta.

Nella pratica si riesce ad iniettare o estrarre elettroni nel floating gate tramite due meccanismi dovuti ad effetti quantistici (effetto tunneling): elettroni caldi (*channel hot electron*) e *Fowler Nordheim*.

Nel primo caso si fa in modo di accelerare gli elettroni nel canale (V_{ds} elevata) aumentando al tempo stesso il campo elettrico tra gate e bulk (V_{gs} elevata) in modo che una piccola parte degli elettroni veloci o "caldi" possa saltare dal canale al floating gate.

Nel secondo caso si fornisce un campo elettrico sufficientemente grande tra, ad esempio, gate e bulk, e gli elettroni sottoposti a tale campo hanno una certa probabilità di saltare nel floating gate.

Con questi meccanismi la tensione di soglia della cella è stata variata in modo permanente. Il valore di soglia costituisce l'informazione che può essere memorizzata.

Per rileggere l'informazione memorizzata bisogna fornire al control gate una tensione di lettura di valore intermedio tra il valore di soglia di una cella vergine e quello della cella scritta. La cella vergine lascerà passare una corrente, mentre una scritta avrà corrente nulla.

Questa corrente è rilevata da un circuito chiamato *sense amplifier*, la cui uscita è alta o bassa (bit 1 o bit 0) a seconda del valore memorizzato nella cella.

Al fine di aumentare la densità, il meccanismo di memorizzazione e lettura di una cella flash può in realtà essere raffinato così da poter posizionare con precisione le soglie su più livelli (ad esempio 4, 8, 16 livelli distinti) e rileggerle in modo da realizzare memorie contenenti più di un bit in ciascuna cella.

Per riassumere: una cella a floating gate è in grado di essere scritta, letta e cancellata.

Come già anticipato, in base all'architettura con cui vengono combinate moltissime celle a floating gate si distinguono due principali categorie di Flash memory: NOR e NAND.

In ragione del modo in cui le celle sono collegate, i meccanismi di scrittura, lettura e cancellazione differiscono.

I.2 Architettura delle Flash Memory.

L'architettura di una flash memory determina il tipo di memoria flash. I due principali tipi, NOR e NAND, con i propri vantaggi e svantaggi, hanno metodi diversi di programmazione, lettura e cancellazione.

I.2.1 NOR Flash

Negli array di memorie NOR (Figura 2) ogni cella ha il terminale di source connesso con quello delle altre celle (il source, durante la lettura e la programmazione è connesso a massa, ma viene connesso a una tensione positiva durante la cancellazione), e il terminale di drain alla *bit line* (BL in figura), che è comune a una stessa colonna dell'array. Il gate invece è in comune con i gate delle celle in una stessa riga, detta *word line* (WL in figura).

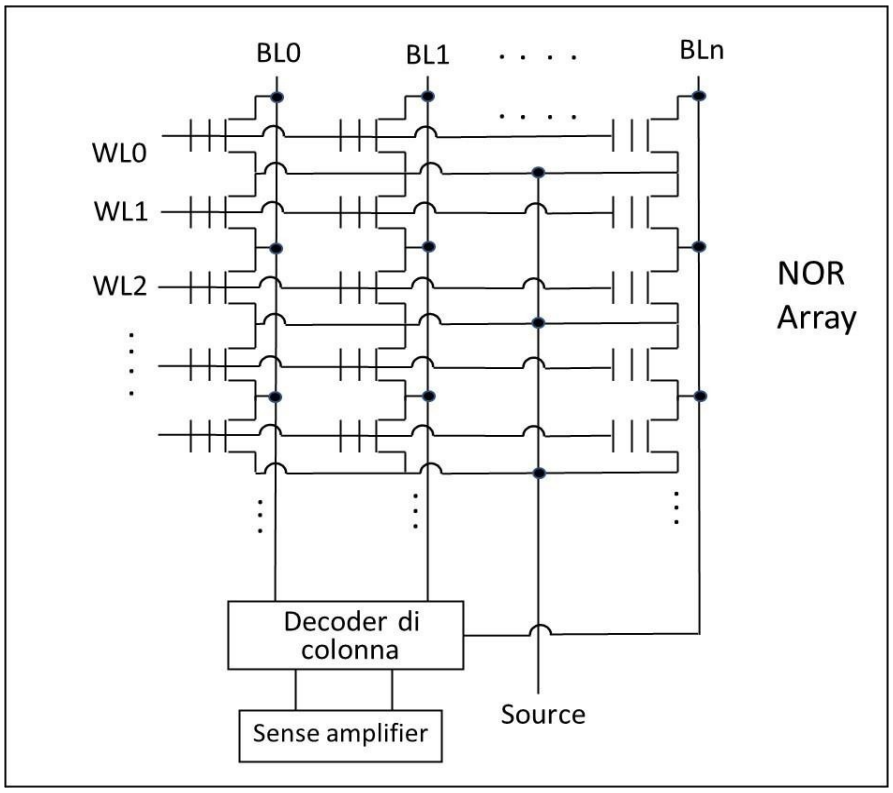

Figura 2

Quando una word line è posta a una tensione ben precisa di lettura, il corrispondente transistor si accende o rimane spento a seconda della carica immagazzinata; la conseguenza è che scorre o meno della corrente nella bit line. Questa corrente viene inviata tramite il decoder al sense amplifier che la confronterà con un valore di riferimento per fornire il bit di uscita. A prescindere dal tipo di sense amplifier utilizzato, nelle memorie NOR il tempo di accesso per una lettura random è generalmente minore rispetto ad altre architetture.

La lettura della cella viene effettuata selezionando la riga e la colonna di interesse, come mostrato in figura 3. In questo esempio vogliamo leggere la cella appartenente alla seconda riga e prima colonna.

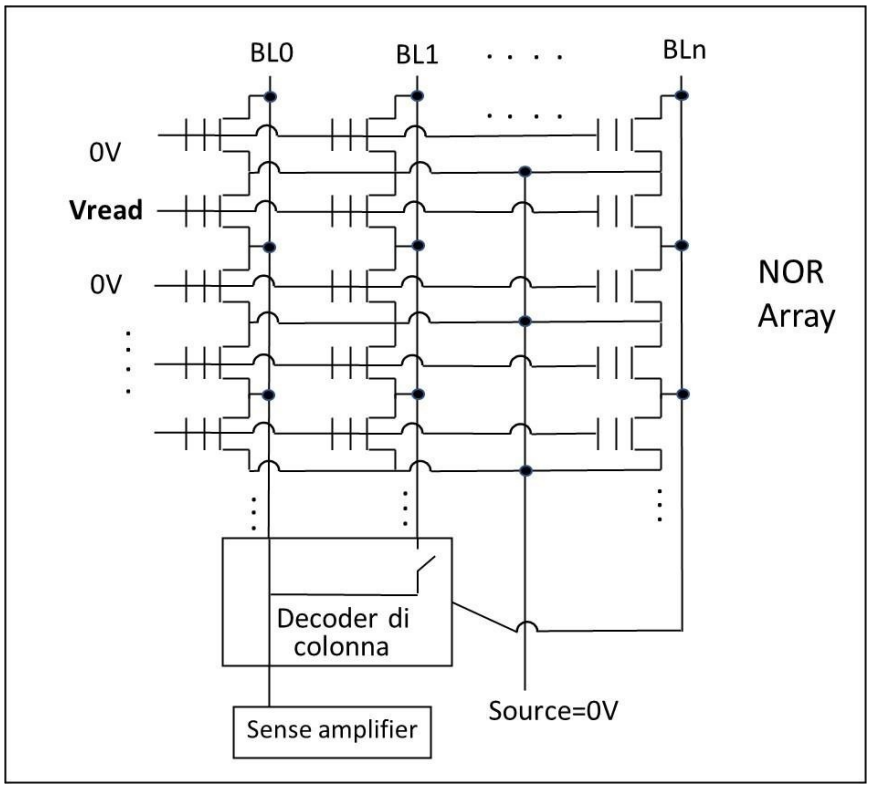

Figura 3

Il source, comune a tutta la matrice, viene posto a 0V. La colonna selezionata viene collegata al sense amplifier mentre le colonne non selezionate vengono tenute sconnesse (in figura ciò è schematizzato con degli interruttori nel decoder di colonna).

Alle righe non selezionate viene applicata una tensione di 0V, per assicurarsi che le celle

corrispondenti siano spente. Alla riga selezionata viene applicata una tensione di lettura Vread. In questo modo, solo alla cella selezionata è permesso di far scorrere una corrente nel sense amplifier. Questa corrente viene confrontata con una corrente di riferimento, per decidere il valore del bit memorizzato. In realtà la lettura viene solitamente effettuata in maniera differenziale, comparando la corrente della cella con quella di una seconda cella (di riferimento), fisicamente identica a tutte le altre celle e avente le stesse tensioni V_{gs} e V_{ds}.

Poiché la cella scritta condurrà una corrente più piccola rispetto ad una cella cancellata, se applichiamo una certa tensione comune alle celle da leggere e a quella di riferimento, saranno riconosciute come cancellate quelle la cui corrente risulterà più alta di quella di riferimento, altrimenti risulteranno scritte. Il vantaggio di avere una cella di riferimento è legato al fatto che essa altera le proprie caratteristiche nel tempo in maniera molto simile alle celle da leggere, e questo crea un utile effetto di compensazione.

È importante, durante il periodo di lettura, evitare che il drain delle celle abbia un potenziale troppo elevato in modo da evitare di programmare celle accidentalmente. Solitamente il sense amplifier è progettato in maniera da mantenere tale tensione sufficientemente bassa.

Per programmare le celle, la flash NOR utilizza il meccanismo degli hot electron (elettroni caldi) che è stato descritto precedentemente.

In figura 4 è mostrata la configurazione per programmare la stessa cella dell'esempio precedente (seconda riga, prima colonna).

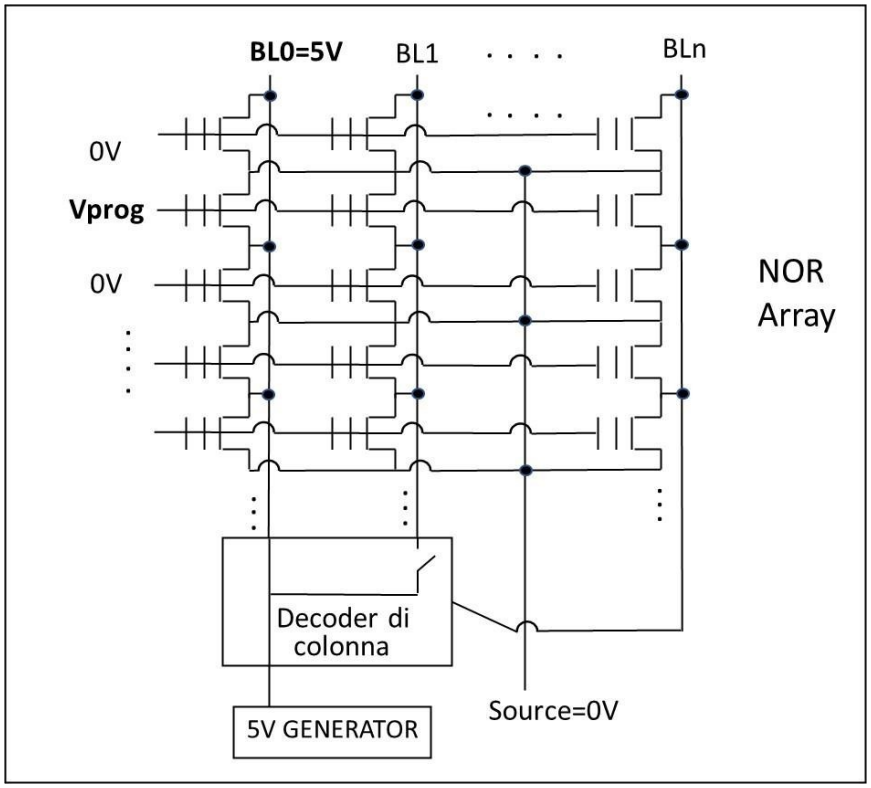

Figura 4

Anche in questo caso, il source viene posto a 0V. Avvalendosi del decoder di riga, le righe non selezionate sono a 0V e quella selezionata è collegata alla tensione di programmazione (che può arrivare a valori dell'ordine di 10-15V).

Con il decoder di colonna facciamo in modo che le bit line non selezionate siano flottanti, mentre alla bit line selezionata (e quindi al drain della cella di interesse) arriva una tensione tra i 5 e i 6 V. Il campo elettrico longitudinale prodotto dalla V_{ds} elevata fornisce un'energia notevole agli elettroni che, sottoposti al campo trasversale causato dalla V_{gs}, possono saltare nel floating gate grazie all'effetto tunneling.

Questo processo è veloce, ma occorre fornire una corrente molto elevata. Ovviamente più celle si vogliono programmare contemporaneamente, maggiore sarà l'energia da fornire.

Per cancellare una cella, figura 5, si utilizza il metodo di Fowler Nordheim.

In questo caso bisogna applicare una elevata tensione positiva sul source e una tensione negativa sul control gate (il drain viene lasciato flottante).

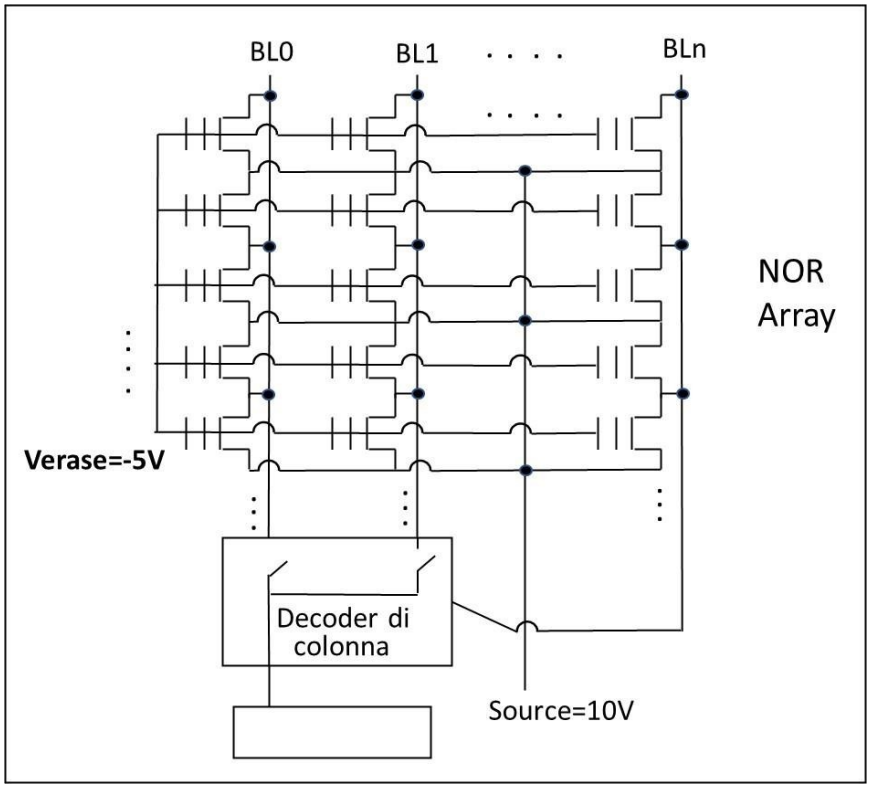

Figura 5

Data la struttura della NOR flash, in cui il source è comune a tutte le celle del blocco, la cancellazione le interesserà tutte. Necessariamente dovremo pilotare negativi tutti i gate: tutte le righe, tramite il decoder di riga, vengono poste a una tensione negativa. Il decoder di colonna, semplicemente, lascerà tutte le bit line flottanti.

Pertanto, in ciascuna cella gli elettroni vengono estratti dal floating gate, il che è equivalente a dire che si rende più negativa la tensione di soglia sulla cella. Per quanto appena detto, tutte le celle di quel blocco verranno cancellate. Ovviamente se una cella era programmata l'effetto è quello desiderato, ma se fosse già cancellata la soglia diventerebbe troppo negativa (cella *depleta*), creando problemi successivi di lettura. Infatti, una cella con soglia negativa risulterebbe accesa anche quando sul gate viene applicata una tensione di 0V. Confrontando questo scenario con la figura 3 è facile capire che una tale cella, anche quando non selezionata, farebbe passare una corrente che si aggiunge a quella della cella da leggere, alterando il risultato della lettura stessa. A questo inconveniente si rimedia alternando a impulsi di cancellazione piccole fasi intermedie di programmazione dette "soft program".

I.2.2 NAND Flash

Negli array di memorie NAND (Figura 6) le celle di una colonna sono connesse in serie (il source di una cella è connesso al drain della cella sottostante). Le celle sono connesse in serie, in gruppi costituiti da 16, 32 o 64 celle. In serie con le celle ci sono due altri transistor che connettono la serie delle celle o alla *Source Line* (SL in figura) o alle bit line pari oppure dispari. Nella figura, per semplicità, non sono mostrati il decoder di colonna e il sense amplifier, che sono comunque analoghi a quelli delle memorie di tipo NOR.

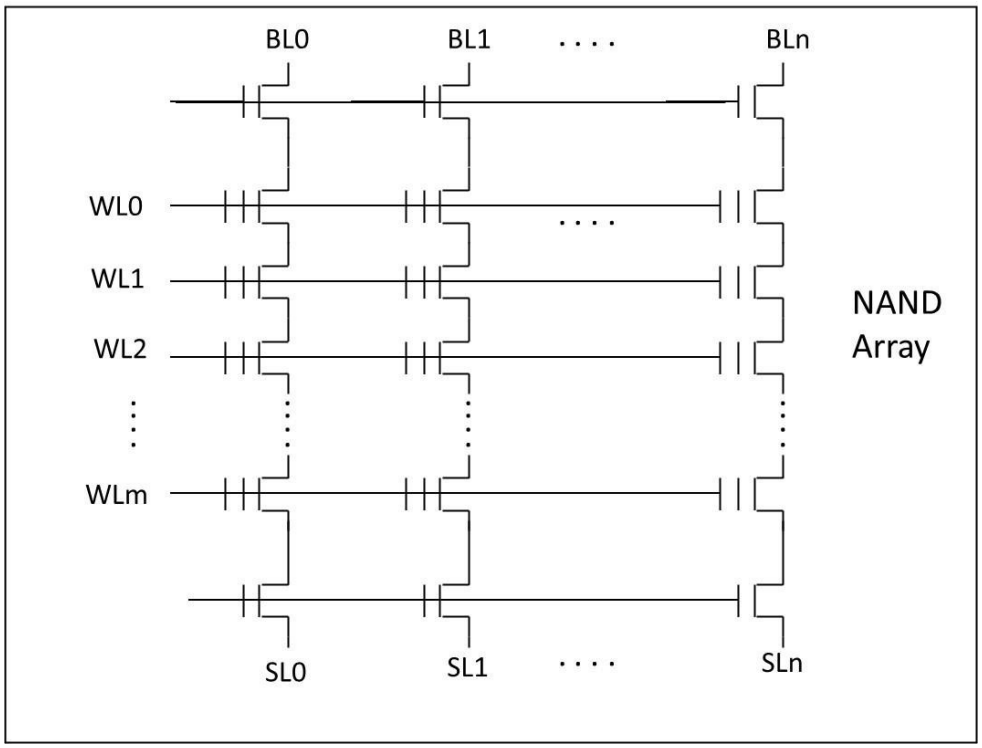

Figura 6

Durante la lettura (figura 7: supponiamo di voler leggere, come nell'esempio NOR, la cella sulla seconda riga e prima colonna) il decoder di riga fa in modo che le tensioni su tutte le word line (righe) tranne quella da leggere siano a un valore VPASS maggiore della più alta soglia programmata mentre la word line della cella da leggere è al valore di lettura, cioè compreso tra la tensione di soglia di un bit cancellato e di uno

programmato. Nelle memorie NAND la tensione di lettura è posta a 0V, in quanto è possibile lavorare anche con tensioni di soglia negative. Entrambi i mosfet della colonna selezionata vengono accesi. Inoltre, solo la bit line (colonna) selezionata viene collegata al sense amplifier (non mostrato). Visto che tutte le celle flash di una colonna sono in serie, per leggere una cella specifica di quella colonna è necessario accendere tutte le altre celle della stessa colonna come fossero degli interruttori ideali. È per questo che si applica la tensione VPASS alle righe non selezionate.

Una volta stabilito il collegamento del drain della cella selezionata con il sense amplifier e del source con il potenziale 0V, la cella può essere letta dal sense amplifier in maniera analoga a quanto avviene in una NOR. Anche in questo caso, solo se la cella selezionata non è programmata, sulla bit line scorrerà una corrente maggiore del livello di riferimento (bit di uscita = 1).

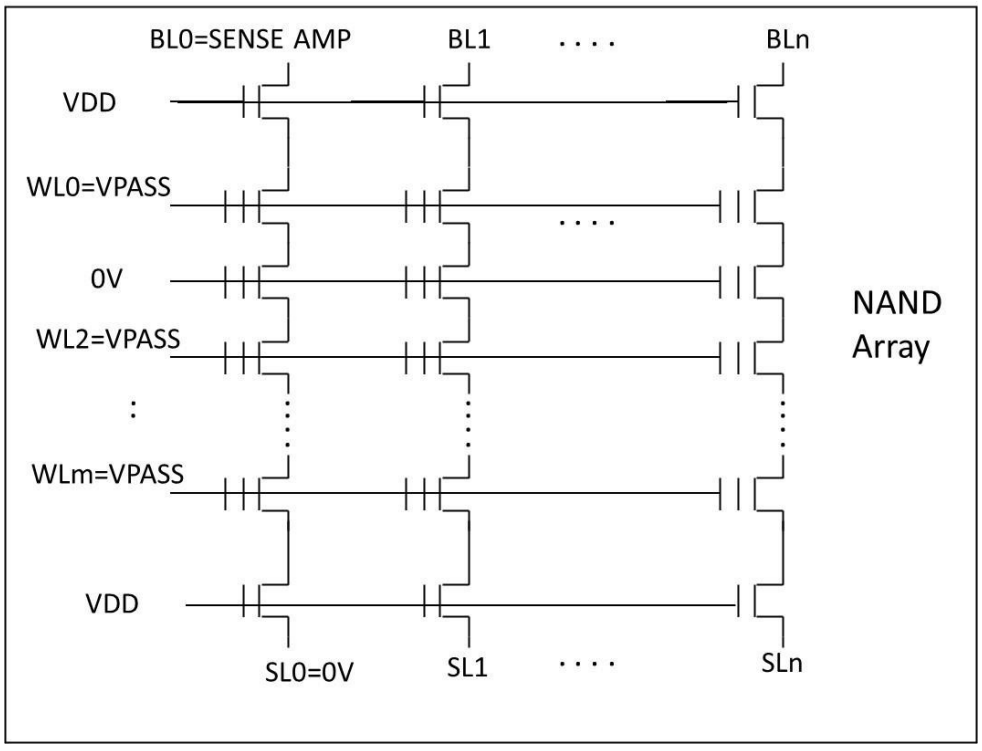

Figura 7

La configurazione di programmazione è mostrata in figura 8. Cerchiamo, come negli esempi precedenti, di programmare la solita cella della seconda riga e prima colonna.

In questo caso, tramite i decoder di riga, le word line non selezionate vengono poste a VPASS (stesso valore che in lettura), mentre la riga selezionata viene posta alla tensione positiva di programmazione (tra 8V e 10V). Tramite il decoder di colonna, la bit line selezionata viene posta a 0V, mentre quelle non selezionate vengono poste a VDD. Il transistor della parte superiore viene acceso, mentre quello della parte inferiore viene spento.

Con questa configurazione, la cella selezionata si ritrova con una tensione positiva alta sul gate, 0V sul drain, mentre il source è lasciato flottante. Siamo in sostanza nelle condizioni che attivano il meccanismo di Fowler-Nordheim, in base al quale gli elettroni salteranno nel floating gate aumentando la soglia equivalente della cella. In questo, poiché la tensione è applicata tra il gate e il terminale di drain, si parla di *junction tunneling*. Nella programmazione delle celle, che utilizza il meccanismo di channel tunneling, più intenso è il campo elettrico applicato, cioè più alta è la differenza di potenziale, maggiore è la probabilità di iniettare elettroni nel floating gate.

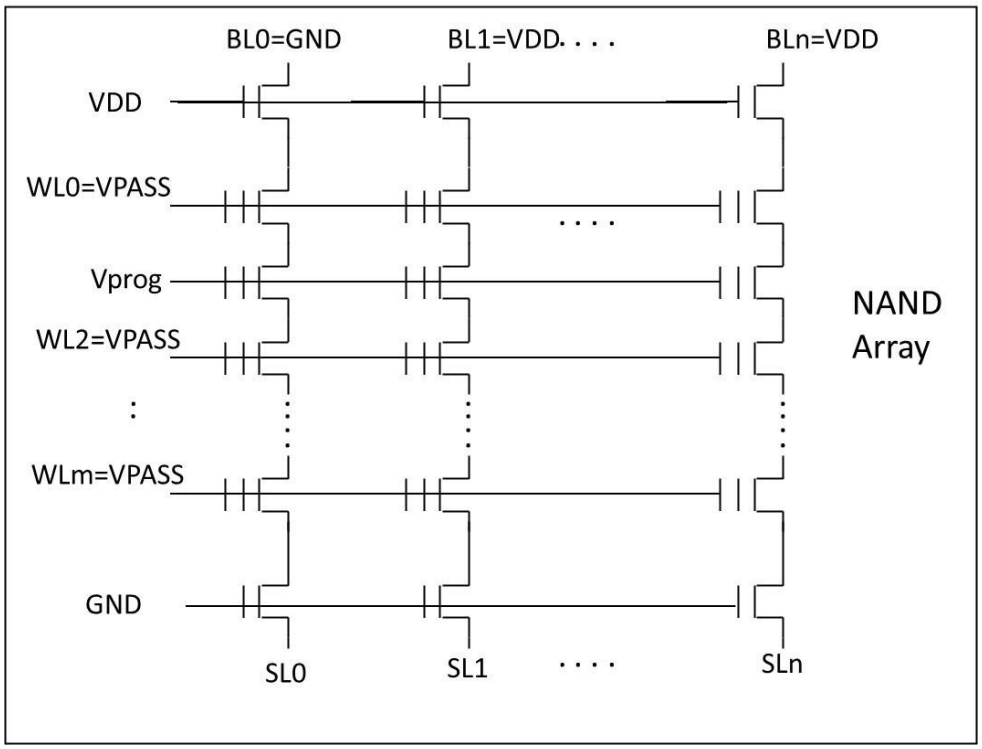

Figura 8

Per quanto riguarda le celle non selezionate, quelle appartenenti alla stessa colonna della cella da scrivere non si programmano poiché hanno il gate a VPASS < Vprog e sono soggette a un campo elettrico nettamente inferiore rispetto a quest'ultima (sebbene non così basso da evitare effetti indesiderati). Anche le celle che si trovano sulla stessa riga rispetto alla cella da programmare, avendo il drain alla tensione positiva VDD anziché a 0V, sono soggette a un campo elettrico inferiore e non si programmano (effetti indesiderati a parte).
Le celle appartenenti a righe e colonne entrambe non selezionate, infine, avendo una differenza di potenziale VPASS-VDD che è la minima possibile, a maggior ragione non si programmano e sono sottoposte al livello minimo di stress.

Per quanto riguarda la cancellazione, essa viene effettuata ponendo tutte le word lines a 0V, spegnendo i mosfet superiori e inferiori, e alzando la tensione di bulk (substrato comune a tutte le celle) a un valore positivo elevato. Come flash NOR, questa configurazione è applicata a tutte le celle contemporaneamente.
In questa situazione, ci troviamo di nuovo nelle condizioni di Fowler-Nordheim ma il campo elettrico è invertito, causando il passaggio di elettroni dal floating gate al canale. In questo caso, poiché la tensione è applicata tra il canale e il gate, si parla di *channel tunneling*. Nel channel tunneling il campo elettrico è applicato tra il gate e il substrato, mentre i terminali di drain e di source sono flottanti: in questo caso l'esodo degli elettroni dal floating gate è dovuto al fatto che gli elettroni transitano dal gate al substrato.

Un notevole vantaggio della struttura NAND è che le soglie negative sono consentite, poiché le celle sono tutte in serie e non causano inconvenienti durante la lettura.
In effetti, utilizzando soglie negative, si può impiegare una tensione di lettura a 0V come descritto precedentemente e si può utilizzare un valore di VPASS sufficientemente basso, tale da minimizzare lo stress sulle celle non selezionate.

Un altro vantaggio delle memorie NAND è il seguente: durante la cancellazione si potrebbero tenere tutte le word line tranne una alla stessa tensione del bulk, permettendo la cancellazione selettiva di una sola riga alla volta.
Inoltre, dal punto di vista costruttivo, la NAND risulta sicuramente più compatta della NOR, poiché, al contrario di quest'ultima, si possono evitare i contatti per collegare insieme i drain e i source delle celle di una stessa colonna.
Dato che per scrivere e cancellare una memoria NAND si utilizza esclusivamente il meccanismo di Fowler-Nordheim, il consumo energetico risulta nettamente inferiore a quello di una NOR.
Tutte queste caratteristiche hanno fatto sì che l'architettura NAND diventasse dominante nel mercato delle memorie non volatili.

Tuttavia, per garantire il funzionamento di una memoria NAND sono necessarie delle tensioni di pilotaggio piuttosto elevate, fatto che comporta diversi svantaggi tecnologici e costruttivi.
Per poter abbassare le tensioni di pilotaggio si può considerare una riduzione nello spessore dello strato dielettrico. In questo modo l'efficienza aumenta mentre l'energia richiesta diminuisce. Ma se si assottiglia troppo il dielettrico si sviluppano effetti degeneranti che creano correnti di perdita che ridurranno nel tempo la carica intrappolata.
Un altro problema è nel tempo che si impiega per programmare ogni cella, maggiore rispetto al metodo degli elettroni caldi usato nelle NOR. A quest'ultimo aspetto si rimedia con un elevato "parallelismo", cioè scrivendo tante celle contemporaneamente: si dovrà aspettare un periodo relativamente lungo perché la scrittura sia completata (latenza), ma alla fine di questo tempo molte celle saranno state programmate, diminuendo così il tempo medio di programmazione.

I.3 Cenni sulla memorizzazione multilivello.

Volendo ottenere una maggiore densità di informazione memorizzata in un dato spazio di silicio, è possibile utilizzare un approccio *multilivello* in modo da memorizzare più di un bit per cella.

La prima memoria flash NOR commerciale capace di memorizzare 2 bit per cella è stata la StrataFlash [6] sviluppata da Intel negli anni 90. Negli anni successivi molti sforzi sono stati dedicati al miglioramento di questa tecnologia in termini di densità, affidabilità e prestazioni.

Andremo qui a illustrare i risultati descritti in [7] che si riferiscono ad una architettura NOR, anche se gli stessi principi possono applicarsi all'architettura NAND.

Come già accennato in precedenza, è possibile affinare il meccanismo di programmazione in modo da iniettare nel floating gate di una cella flash una quantità controllata di carica. In questo modo la tensione di soglia risultante può essere posizionata in modo piuttosto preciso entro un range di valori possibili.

La figura 9 illustra il risultato sperimentale in cui le soglie di una matrice di celle sono state programmate su 32 livelli, corrispondenti a 5 bit per cella.

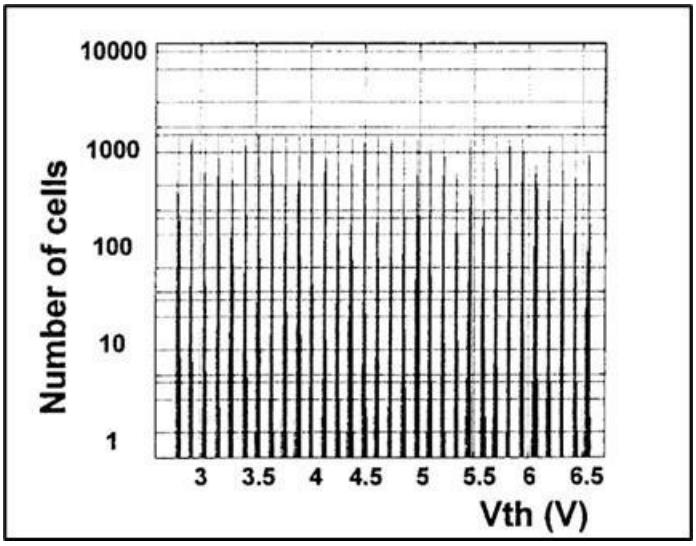

Figura 9

Per ottenere una tale distribuzione è ovviamente necessario utilizzare circuiti di programmazione più sofisticati nonché più lenti che nel caso binario.

La figura 10 mostra l'andamento della tensione applicata al gate della cella durante la fase di programmazione. Essenzialmente, anziché applicare immediatamente una tensione elevata al gate, si applica una tensione crescente nel tempo, tale da far innalzare la soglia della cella a poco a poco. Il processo viene interrotto nelle fasi B e D al fine di effettuare una verifica sui valori intermedi raggiunti, prima di completare il processo e arrivare alla soglia desiderata. La fase F comporta l'ultima verifica sul livello raggiunto.

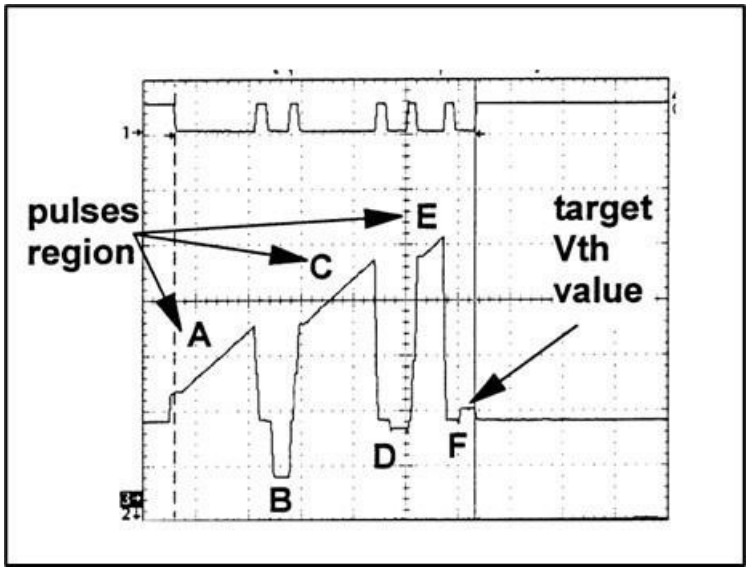

Figura 10

Ovviamente la memorizzazione multilivello comporta una maggior probabilità di errore di lettura, data la minor distanza tra i livelli delle tensioni di soglia.
Inoltre, al passare del tempo la distribuzione delle tensioni di soglia su ciascun livello tende ad allargarsi a causa delle non idealità della memoria, come mostrato in figura 11 (il passare del tempo viene simulato mettendo la memoria in un forno a 250°C per 200 ore). Appare evidente che al passare del tempo aumenta ulteriormente la probabilità di errore.
Come già notato precedentemente, un modo per mitigare questi effetti è quello di utilizzare delle celle di riferimento opportunamente programmate sui possibili livelli di

soglia: essendo soggette a degrado simile a quello delle celle da rileggere, l'effetto di invecchiamento viene in qualche modo compensato.

Un secondo approccio è quello di applicare un *refresh* periodico ai valori memorizzati nelle celle, vale a dire rileggerle periodicamente (prima che il degrado sia eccessivo), e riscrivere i dati nelle stesse celle (oppure in altre celle non ancora utilizzate, per minimizzare lo stress).

Nonostante questi accorgimenti, è ovvio che la probabilità di leggere erroneamente il contenuto delle celle è più alta che nel caso binario, il che giustifica la necessità di adottare codici di correzione di errore.

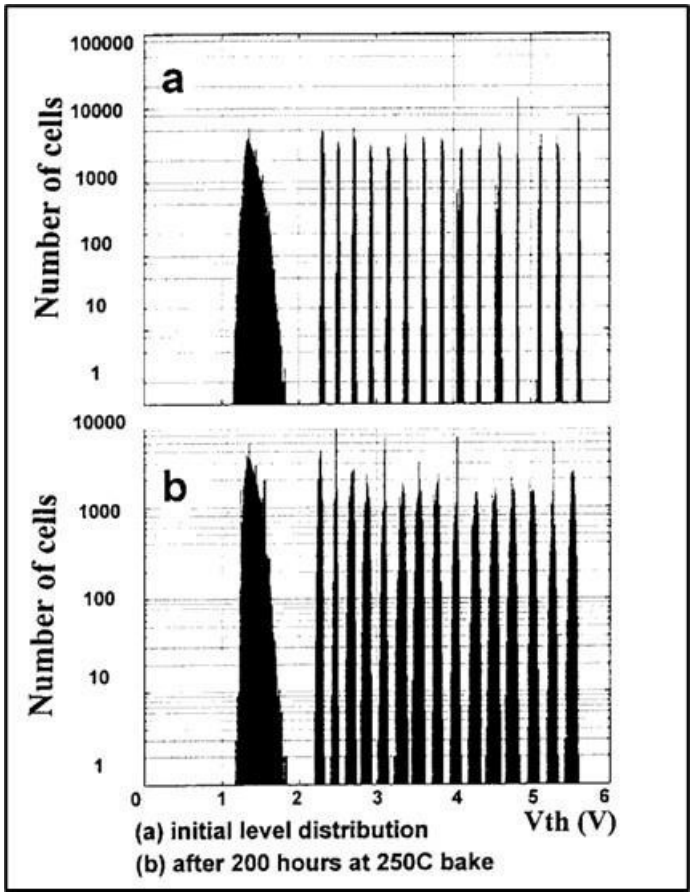

Figura 11

I.4 Problematiche delle memorie Flash

Come già accennato nell'introduzione, le memorie NAND offrono una maggiore densità e più bassi consumi, ma sono più soggette a non idealità che comportano da un lato la perdita di capacità di memorizzazione con il prolungato utilizzo (problemi di *endurance*), dall'altro perdita del dato memorizzato in una certa cella al passare del tempo (problemi di *retention*). Nel seguito verranno quindi affrontate le problematiche di questa famiglia di memorie Flash.

Possiamo classificare le non idealità in tre categorie [8]:

1) Disturbi dovuti alla scrittura e cancellazione;
2) Disturbi dovuti alla lettura;
3) Disturbi dovuti ai dati già scritti in precedenza nell'array.

Per quanto riguarda il primo tipo di disturbi dobbiamo notare che per programmare una certa cella di memoria di un array è necessario, come già mostrato nella figura 8, applicare una tensione elevata alla riga cui la cella appartiene e porre a 0 V il drain della colonna relativa alla cella. Al tempo stesso le righe non selezionate debbono essere pilotate con una tensione sufficientemente alta per far funzionare come un corto circuito le celle in serie a quella di interesse. Le colonne da non programmare debbono essere poste a una tensione positiva.

Questa configurazione è tale da causare accoppiamenti capacitivi tra le linee delle celle da programmare e le celle ad esse adiacenti. In sostanza il disturbo causato all'atto di scrivere una certa cella si traduce in un impulso di programmazione non desiderato sulle celle circostanti il cui effetto dipende dalla posizione relativa di ciascuna cella e dalla durata dell'impulso di programmazione. La figura 12 [8] riassume in modo schematico gli effetti dovuti al disturbo di scrittura.

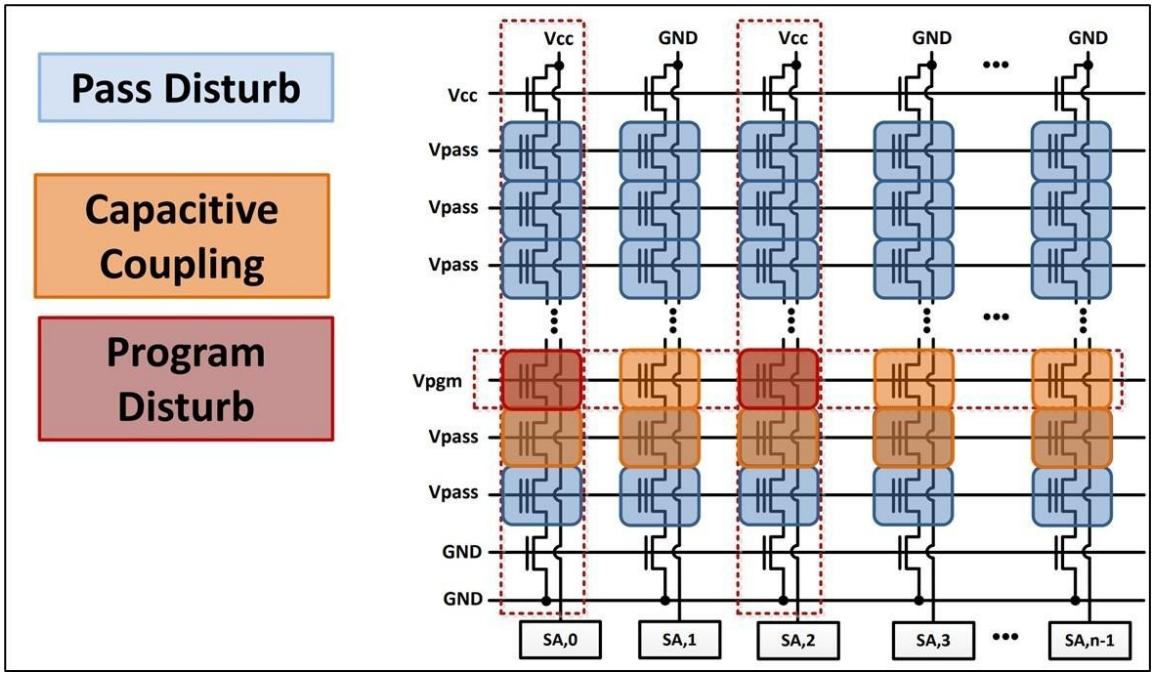

Figura 12

Le celle che ne risultano negativamente influenzate sono quelle non programmate (livello di uscita 1) poiché tendono a passare da una soglia bassa a una soglia alta all'aumentare del numero di cicli di programmazione applicati. Si può quindi dire che la probabilità che un 1 si trasformi in 0 è maggiore della probabilità che uno 0 si trasformi in 1.

Per quanto riguarda i disturbi di lettura, essi sono simili a quelli descritti sopra. Anche qui bisogna considerare che (figura 7) per leggere una determinata cella, tutte le righe precedenti e successive a quella di interesse debbono essere poste a una tensione sufficientemente alta (maggiore della più alta soglia programmata) in modo da poter connettere la cella di interesse al *sense amplifier.* È proprio questa tensione che causa una piccola programmazione indesiderata sulle celle non selezionate. Paradossalmente, sono proprio le celle che non vengono lette ad essere soggette al disturbo, e anche in questo caso le soglie basse risentono negativamente del fenomeno, con una maggiore probabilità di trasformare un 1 in 0. La figura 13 [8] riassume in modo schematico gli effetti dovuti al disturbo di lettura.

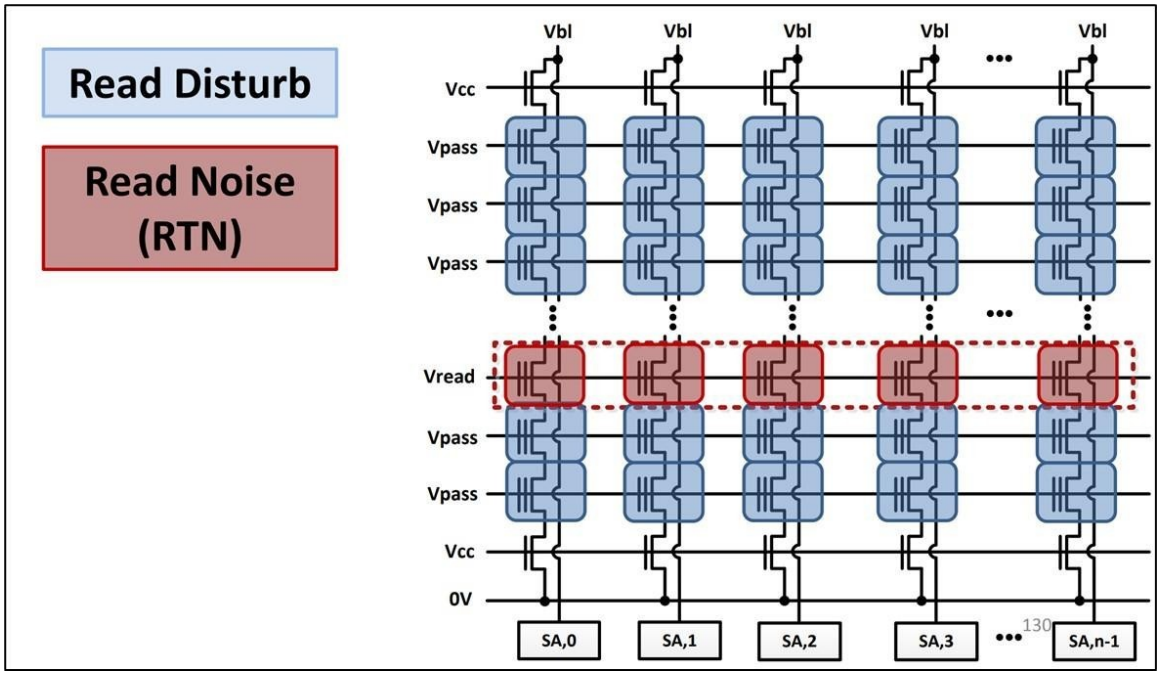

Figura 13

Gli errori causati dai fenomeni appena descritti sono di natura sistematica e dipendente dal *pattern* di dati contenuti nella memoria.

La figura 13 mostra anche che in lettura esiste una componente aleatoria (*read noise*) dovuta alla presenza di *trappole* nell'ossido in prossimità del canale della cella, che possono catturare e rilasciare in maniera casuale elettroni presenti nel silicio sottostante, come mostrato in figura 14 [8].

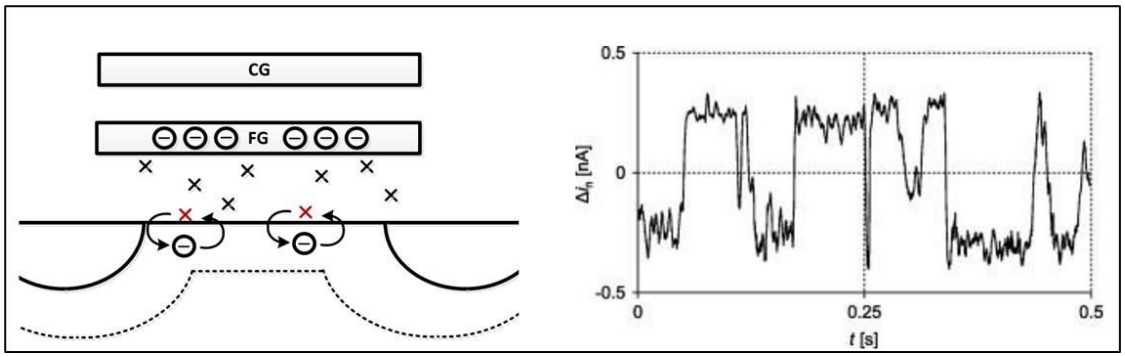

Figura 14

Come evidenziato dalla figura 13 questo disturbo interessa le celle che vengono lette, e fa sì che letture ripetute della stessa cella possano portare a risultati differenti. Questo fenomeno giustifica l'utilizzo di tecniche di *signal processing* a partire da letture ripetute. In particolare, queste tecniche si abbinano molto bene alla codifica di dati LDPC di cui parleremo nel prossimo capitolo.

Per quanto riguarda i disturbi dovuti ai dati precedentemente scritti nell'array, questi non vanno ad alterare la tensione di soglia memorizzata nelle celle, tuttavia influenzano il modo in cui queste tensioni vengono lette. Infatti, quando una cella viene collegata al suo sense amplifier per essere letta, tutte le celle della colonna corrispondente vengono accese e idealmente ciascuna di queste celle dovrebbe essere un corto circuito.

In realtà ciascuna cella presenta una propria resistenza, inoltre le celle non programmate hanno una resistenza più bassa delle celle programmate, poiché la tensione sui gate è uguale per tutte ma la soglia è più alta nelle celle programmate.

La figura 15 [8] mostra la situazione appena descritta che porta al fenomeno di *back pattern effect*.

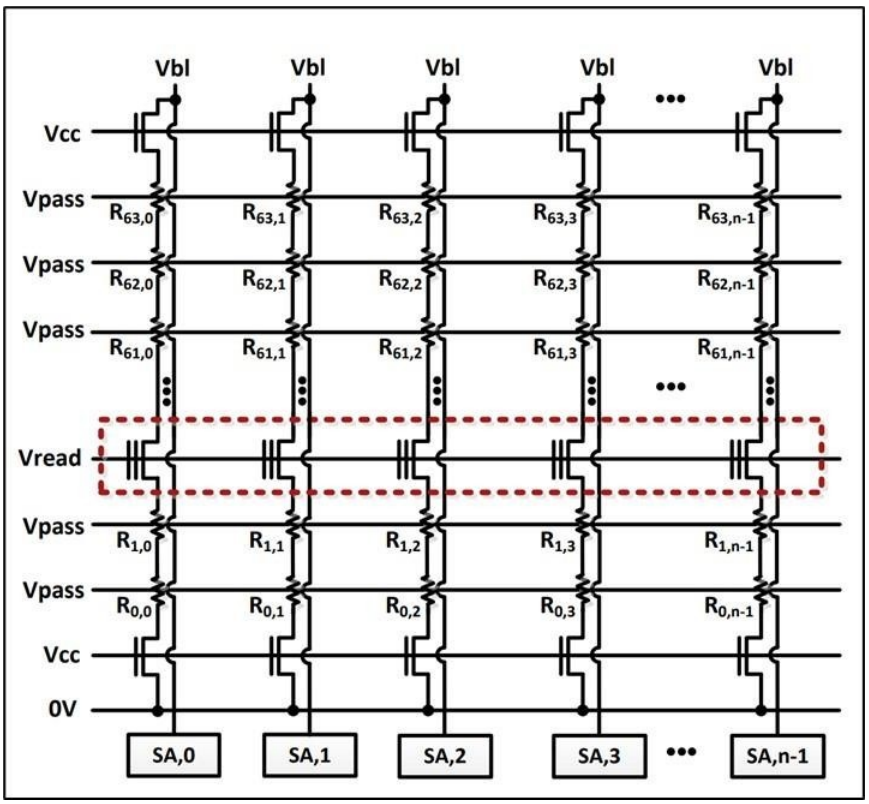

Figura 15

È chiaro che la resistenza complessiva in serie alla cella da leggere è dipendente da quali dati erano stati scritti precedentemente nelle altre celle. Poiché la resistenza in serie, inevitabilmente, altera la corrente condotta dalla cella da rileggere, l'esito della lettura potrebbe essere soggetto a errore a seconda dei dati già programmati (*back pattern*).
In questo caso appare vantaggioso utilizzare delle codifiche di *randomizzazione* dei dati per cui mediamente il numero di bit 0 sia circa uguale al numero di bit 1 in modo da rendere la resistenza serie più o meno costante.

Capitolo II

Codici di correzione di errori in memorie flash

Codici di correzione di errori in memorie flash NAND

Abbiamo già descritto i principi di funzionamento delle memorie flash e le problematiche ad esse relative. È apparso chiaro come sia necessario codificare i dati da scrivere nella flash al fine di minimizzare l'effetto dei disturbi associati alle varie modalità di funzionamento.

Come descritto nel capitolo precedente, al fine di aumentare ulteriormente la densità di memoria nella struttura NAND, da diversi anni viene impiegata una memorizzazione multilivello, vale a dire che, invece di discriminare tra una soglia bassa e una alta e memorizzare un singolo bit per cella, le soglie vengono programmate su più livelli in modo da scrivere più bit per cella (ad esempio 2 b/cella o 3 b/cella). È stato già evidenziato che questo approccio accentua le problematiche descritte nel capitolo precedente, aumentando la probabilità d'errore.

Uno dei più efficienti e più usati codici di correzione di errore (*error correction codes* o ECC) nell'ambito delle memorie flash multilivello è proprio il codice LDPC [9].

II.1 Codici LDPC

I codici *Low-Density Parity-Check*, LDPC, sono codici di canale efficienti che permettono la correzione degli errori di trasmissione o di memorizzazione. La tecnica LDPC fu inventata nel 1961 da Robert Gallager [10], ma rimase relativamente ignorata fino agli anni 90, quando venne sviluppata e proposta [11] come alternativa ai codici Turbo [12]. Sia i codici Turbo sia quelli LDPC sono capaci di avvicinarsi al limite teorico di Shannon [13] (codifiche "capacity-approaching"). Si rappresentano, come spiegato oltre, usando un *grafo di Tanner* ed esistono implementazioni pratiche per cui la soglia di rumore per un canale simmetrico senza memoria può essere molto vicina al massimo teorico, cioè al limite superiore di rumore che può essere presente sul canale in modo che l'informazione persa sia arbitrariamente piccola. Utilizzando delle tecniche iterative di "belief propagation" [14], cioè l'inferenza sul contenuto futuro del messaggio, i codici LDPC possono essere decodificati in un tempo che varia linearmente con la loro lunghezza di blocco.

I codici LDPC stanno trovando sempre più impiego in applicazioni che richiedono un'alta efficienza e affidabilità nel trasferimento di informazioni attraverso canali a banda limitata in presenza di rumore: due esempi di standard di comunicazione che impiegano la codifica LDPC sono il WiMax (IEEE802.16e), reti locali metropolitane, e il DVB-S2, servizio di diffusione digitale terrestre [15].

Lo sviluppo di nuovi codici LDPC (rispetto all'originale implementazione di Gallager) ha permesso di ottenere codici con prestazioni superiori ai codici turbo a parità di "code rate" (rapporto tra numero di bit d'informazione e numero di bit codificati (e quindi trasmessi)).

II.2 Concetti generali

In un codice a correzione d'errore di tutte le parole a n bit che potrebbero essere generate, solo un sottoinsieme costituisce le parole "ammissibili", cosicché se una delle parole non ammissibili viene ricevuta a valle del canale di trasmissione viene rilevato un errore e, in funzione delle capacità correttive del codice, vengono riconosciuti e corretti i bit errati.

Il primo passo nella progettazione di un codice consiste proprio nel generare le parole valide. Nel caso LDPC si parte da una *matrice di parità* **H**, da cui, come mostrato oltre, si deriva una *matrice generatrice* **G**, grazie alla quale generiamo le parole valide effettuando delle moltiplicazioni (equivalente in un campo binario allo XOR logico) con i dati da inviare.

Definiamo ad esempio

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad (2.1)$$

Dalle dimensioni di questa matrice possiamo già dire che le parole del codice saranno lunghe 6 bit, pari al numero di colonne, e che i bit di ridondanza (bit di parità) saranno 3, pari al numero di righe, di conseguenza andremo a trasmettere 3 bit di dato.

In realtà la matrice **H** può essere trasformata nella sua *forma standard* attraverso passi di combinazione lineare tra le righe di partenza in modo da ricavare, per l'esempio, una matrice 3x3 affiancata da una matrice identità. Ad esempio, sommando la riga 1 alla 3, scambiando la 2 con la 3, e sommando di nuovo la 1 con la nuova 3 otteniamo

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad (2.2)$$

la quale, in forma compatta, può essere scritta come segue

$$\boldsymbol{H} = (\boldsymbol{P}^T | \boldsymbol{I}_{n-k}) \qquad (2.3)$$

La *matrice generatrice* **G**, nella sua forma standard è legata alla matrice **H**, potendosi scrivere

$$\boldsymbol{G} = (\boldsymbol{I}_k|\boldsymbol{P}) \qquad (2.4)$$

La matrice **G** così trovata ci permette di ricavare 8 parole di codice valide (*codewords*), corrispondenti a ciascuna delle 8 combinazioni dei 3 bit di dati. Per far questo è sufficiente eseguire la moltiplicazione del vettore di bit di informazione per la matrice **G**

$$(a_1\ a_2\ a_3) \cdot \boldsymbol{G} = (b_1\ b_2 \cdots\ b_6) \qquad (2.5)$$

Dove la moltiplicazione, come già detto, quando applicata al campo binario si traduce in uno XOR logico.

Vale la pena di sottolineare che le *codewords* non sono necessariamente formate in modo che i primi k bit corrispondano ai bit di informazione e che i rimanenti n-k corrispondano a bit di controllo: in realtà questo avviene nell'esempio sopra riportato e si parla, allora, di *codice sistematico*; altrimenti il codice è non sistematico.

Come regola generale, per trasmettere k bit di informazione con parole di n bit complessivi la matrice **H** avrà k righe e n colonne.

La caratteristica dei codici LDPC è che la matrice **H** contiene un numero di simboli 1 rispetto molto minore del numero di simboli 0. In questo senso tale matrice è detta *sparsa*. Dalla bassa densità di 1 nella matrice deriva la prima parte del nome della famiglia di codici (Low Density). Come evidenziato nel seguito, la bassa densità di 1 costituisce un vantaggio nell'elaborazione iterativa della decodifica.

Innanzitutto, a partire dalla matrice **H** si costruisce un *grafo bipartito*, anche detto Grafo di Tanner, Un generico grafo di Tanner è mostrato in figura 1 a titolo di esempio: nella riga superiore sono stati posizionati i bit della parola codificata (*variable nodes*), mentre nella riga inferiore si sono creati dei *check nodes*. La corrispondenza tra matrice **H** e il

grafo della figura 1 è tale per cui quando l'elemento h_{ij} della matrice vale 1, il *variable node* j si collega al *check node* i.

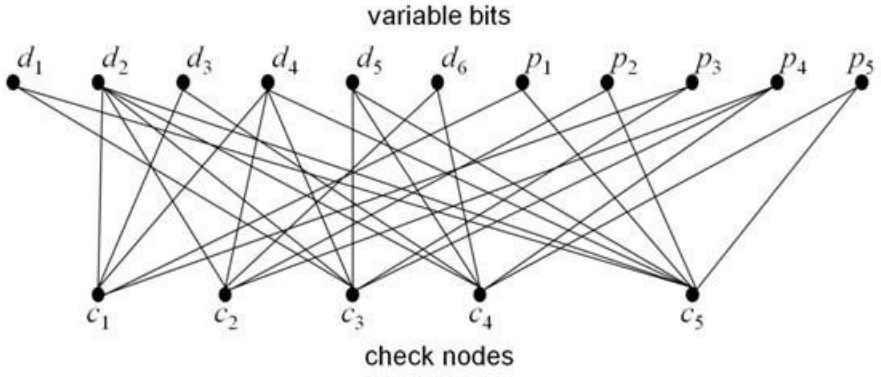

Figura 1

In una codifica LDPC, il grafo di Tanner dovrebbe avere la caratteristica che nessun nodo rimanga isolato, cioè da ciascun nodo debbono partire almeno due rami.
Come regola per generare una codifica LDPC, anche se nessuna distinzione viene fatta tra i data bit e i bit di parità, deve essere garantito che almeno un bit di parità sia coinvolto in ciascuna equazione di parità. Inoltre, i bit di parità devono apparire almeno due volte nel sistema di equazioni di parità in modo da generare un grafo di Tanner con almeno due rami per nodo. D'altro canto, sommando due o più righe della matrice di parità si ottiene una nuova matrice di parità associata ad un codice equivalente a quello iniziale, ma in cui tutti i nodi variabili sono collegati ad (almeno) due nodi di controllo (check). Abbiamo sfruttato questa proprietà per riscrivere la matrice **H** dalla sua forma iniziale (2.1) alla sua forma standard (2.2).
Le equazioni associate alla **H** nella forma standard dell'esempio (2.2) sono

$$\begin{cases} b_1 \oplus b_2 \oplus b_3 \oplus b_4 = 0 \\ b_2 \oplus b_3 \oplus b_5 = 0 \\ b_1 \oplus b_2 \oplus b_6 = 0 \end{cases} \qquad (2.6)$$

Le equazioni (2.6) sono un esempio di codifica sistematica in cui a partire da b_1, b_2, b_3 coincidenti con i bit di informazione, vengono ricavate le incognite b_4, b_5, b_6, rappresentative dei bit di parità.

Come già detto, dalla prospettiva del codificatore di canale, le matrici (2.1) e (2.2) possono essere implementate con lo stesso codificatore, poiché è possibile trasformare l'una nell'altra attraverso la combinazione delle righe. Le parole della codifica di canale possono essere quindi formate con la stessa matrice generatrice. Con l'aiuto dell'algoritmo di Gauss-Jordan [16], è possibile convertire un codice di canale non sistematico in un codice di canale sistematico.

II.3 Decoding

II.3.1 Hard decision

Un primo tipo di decodifica è di tipo *hard decision*, cioè basato su una soglia di decisione fissa in base alla quale il segnale (analogico) proveniente dal canale, attraverso il confronto con una soglia, viene interpretato come 0 o 1. I bit così ricevuti costituiscono la parola di codice **c** da cui vengono estratti i dati.

La matrice di parità **H** deve essere nota al ricevitore, per verificare che **c** sia privo di errori. Dove non c'è alcun errore di trasmissione, **c** = **b** e la *sindrome*, quantità definita come

$$\boldsymbol{s} = \boldsymbol{H} \cdot \mathbf{c}^T \qquad (2.7)$$

è equivalente al vettore nullo. Si potrebbe anche dire in questo caso che tutte le equazioni di parità (2.6) sono soddisfatte.

A partire da una situazione con sindrome nulla, i k bit di informazione possono essere estratti considerando che c'è una corrispondenza biunivoca tra l'insieme delle *codeword* e le possibili stringhe di bit di informazione. Se il codice fosse sistematico la cosa diventerebbe particolarmente facile, in quanto basta prendere i primi k bit della *codeword*. Nel caso di codici non sistematici si potrebbe costruire una *look-up-table* che associa esplicitamente a ciascuna *codeword* la rispettiva stringa di dati. Ovviamente, nel caso di parole estremamente lunghe diventa necessario estrarre i dati per via algoritmica.

Quando uno o più elementi di **s** non sono uguali a zero, vuol dire che c'è almeno un errore di trasmissione. Va notato che anche nel caso di sindrome nulla esiste una probabilità che si siano verificati errori (in questo caso gli errori, combinandosi tra loro, devono essere tali da portare all'annullamento della (2.7)).

Un metodo diffuso per estrarre dalla codeword i bit di informazione è l'algoritmo di *bit flipping* [17].

Consideriamo una memoria flash che viene letta dal sense amplifier. La lettura del bit potrebbe essere incorretta a causa delle non idealità descritte nel capitolo precedente.

Pertanto, la stringa di n bit letta dalla memoria flash potrebbe contenere un certo numero di errori.

Riferiamoci alle figure 2 3 4 5 prese da [18]. Nella figura 2 la stringa letta dalla memoria flash è corretta e, quando presentata allo strato inferiore del grafico bipartito, produce all'uscita dei check nodes tutti valori 0 (**s**=**0**).

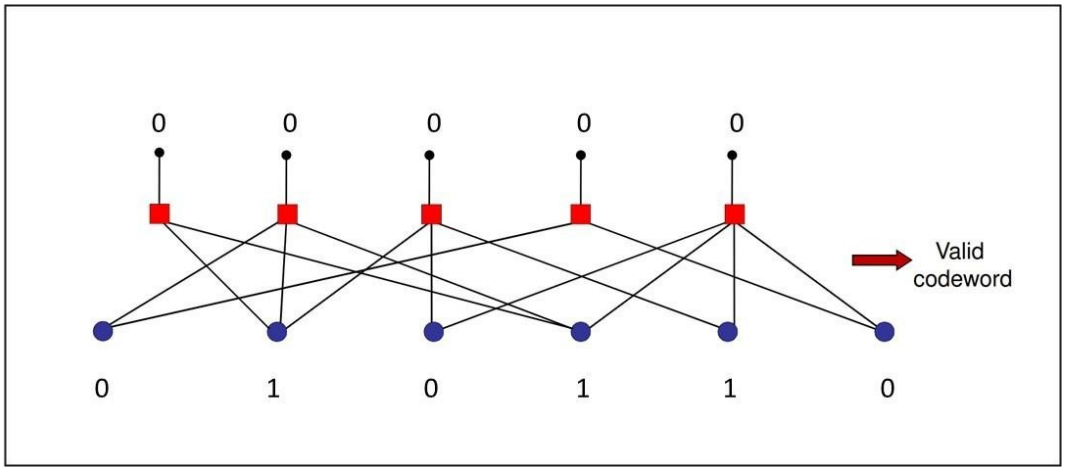

Figura 2

Supponiamo ora (figura 3) che il primo e secondo bit della stringa letta dalla flash siano errati rispetto al valore corretto di figura 2

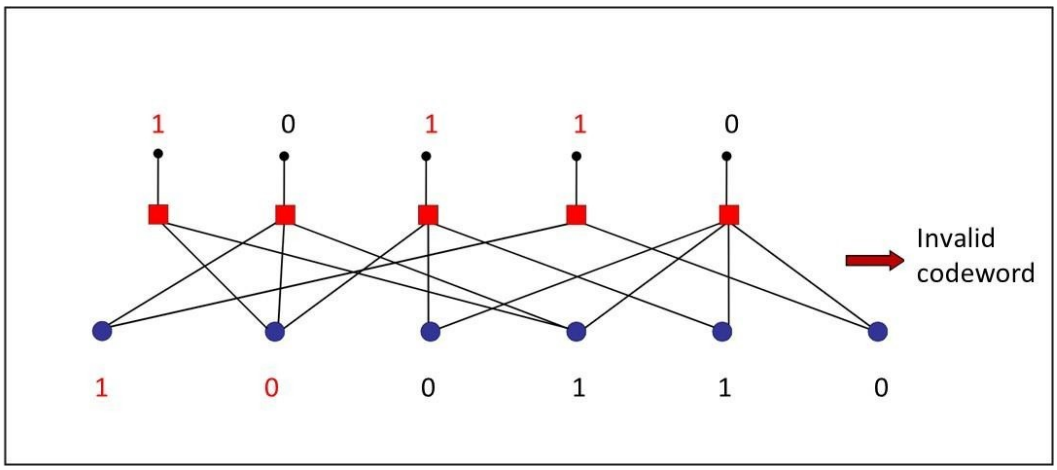

Figura 3

In questo caso il valore assunto dai check nodes non è identicamente nullo (sindrome non nulla). Nella prima iterazione dell'algoritmo è necessario determinare quale dei bit

di ingresso è associato al numero massimo di check nodes invalidi. In questo caso scopriamo che si tratta del secondo bit, che riconosciamo come errato e andiamo pertanto a invertire (*bit flipping*).

Avendo corretto il valore del secondo bit, bisogna ora verificare se il vettore di uscita dai check nodes è nullo. Dalla figura 4 vediamo che ciò non è ancora avvenuto.

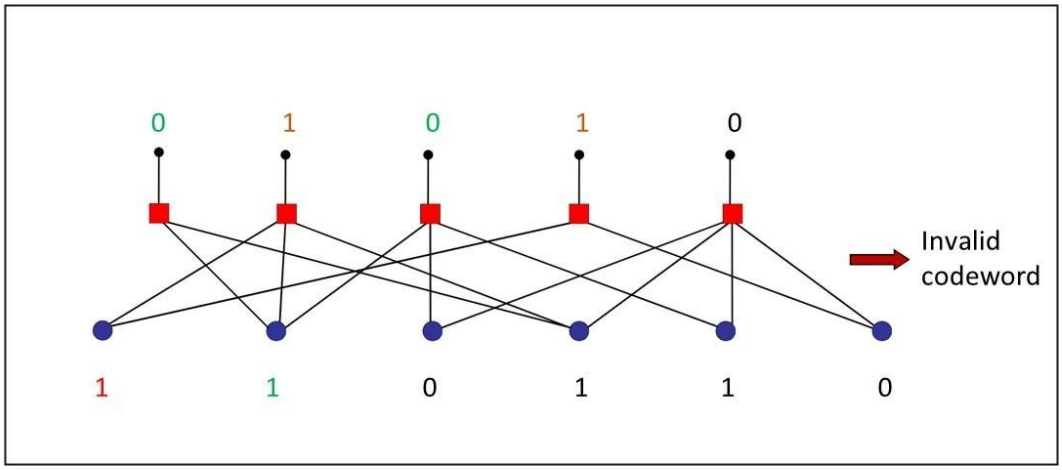

Figura 4

In effetti alcuni check nodes risultano non nulli, e il bit associato al maggior numero di check nodes incorretti risulta essere il primo. Pertanto, ad esso viene applicato il bit flipping e il suo valore viene invertito.

Infine (figura 5), con la correzione apportata tutti i check nodes risultano pari a 0, cioè la sindrome è nulla, la codeword è valida e l'algoritmo può arrestarsi dopo aver corretto due bit errati.

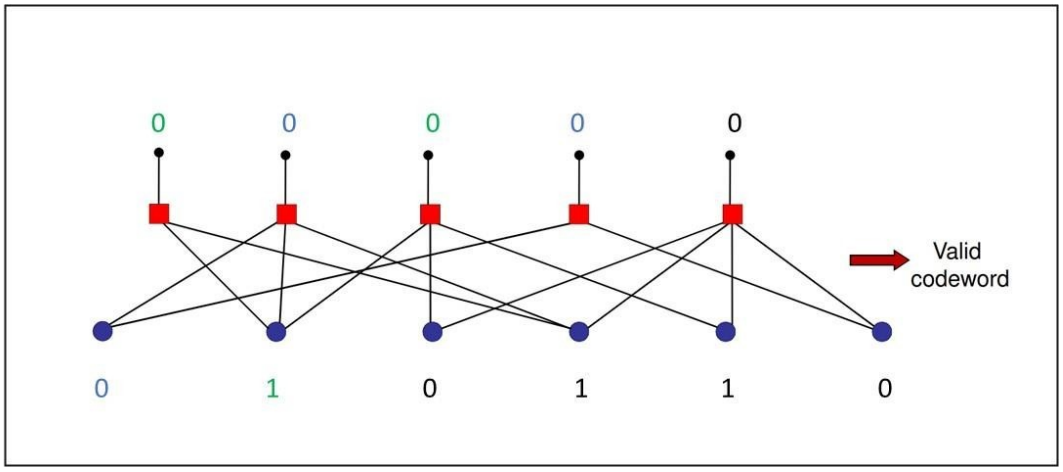

Figura 5

Riassumendo, i bit di informazione vengono ricavati in base a confronti effettuati a maggioranza a partire dalla codeword ricevuta, correggendo eventuali errori con l'inversione (flip) del valore corrente del bit.

Quando viene usata una decodifica hard-decision, **s** fornisce un'indicazione della posizione del bit corrotto, a patto che il numero di bit corrotti non ecceda la capacità di correzione.

Anche una codifica **H** non sistematica, essendo prodotta da una combinazione lineare di **H** sistematica, permette di decodificare un codice di canale generato con **G**.

La figura 6 [18] mostra la capacità correttiva di una decodifica bit flipping: la prima iterazione inizia con un numero consistente di errori, e una sindrome elevata. Nelle iterazioni successive il numero di errori residui cala progressivamente e cala al tempo stesso il peso della sindrome. Alla settima iterazione in questo esempio la sindrome si è annullata e la codeword è diventata valida, situazione equivalente ad aver corretto tutti gli errori iniziali.

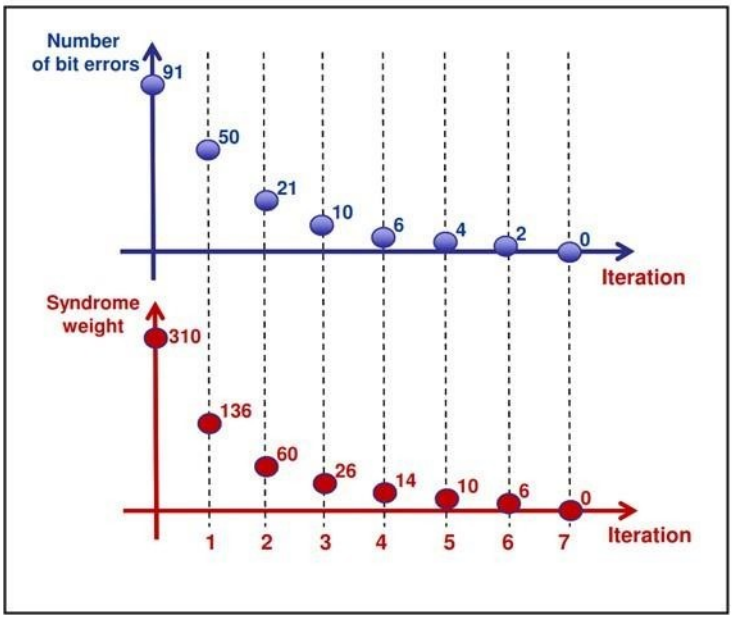

Figura 6

II.3.2 Soft decision

A differenza della decodifica hard, che utilizza la distanza di Hamming ed ha capacità correttive limitate, la decodifica soft utilizza la distanza Euclidea e, al prezzo di una maggiore complessità, consente di ottenere, tipicamente, prestazioni più elevate. Una decodifica soft si avvale del concetto di *Log Likelihood Ratio* (LLR) definito in questo modo [18]

$$LLR(b_i) = \log\left(\frac{P(b_i=0)}{P(b_i=1)}\right) \qquad (2.8)$$

Questa quantità è una misura del "grado di affidabilità" dell'informazione del bit b_i ed è legata alla probabilità che il bit abbia valore 0. Il fatto di non utilizzare direttamente la probabilità è dettato da considerazioni legate alla facilità di implementazione hardware. La relazione tra la probabilità che il bit sia 0 e la sua LLR è mostrata in figura 7. Qualora sia LLR < 0 allora è più alta la probabilità che $b_i=1$, viceversa (LLR > 0) sarà più alta la

probabilità che $b_i=0$. Questa proprietà verrà sfruttata dal decoder in fase di decisione finale sul valore dei bit letti.

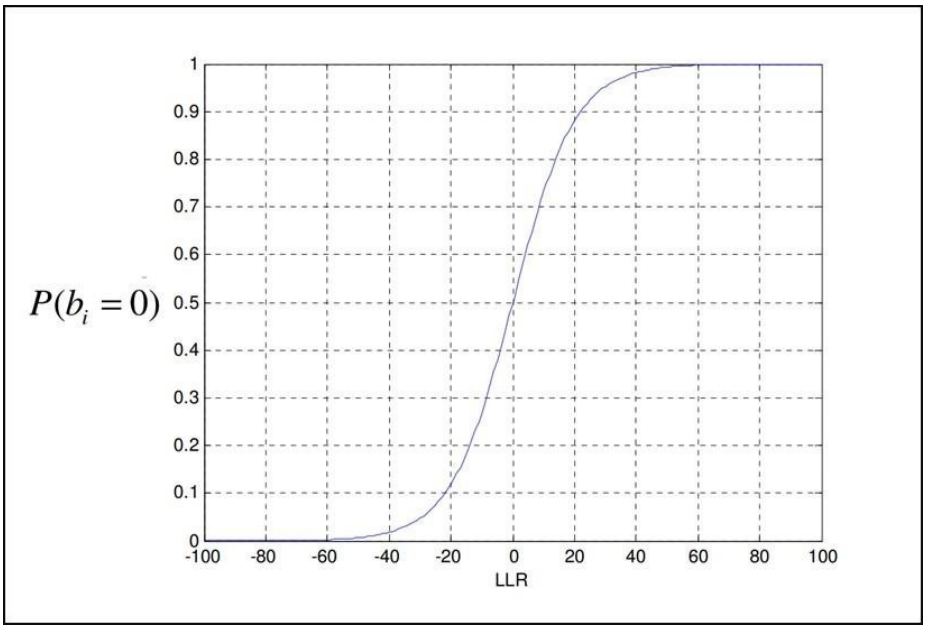

Figura 7

Nella implementazione di memorie flash è effettivamente possibile ricavare la probabilità associata al valore di bit letto da una determinata cella attraverso l'approccio delle letture ripetute.

Per chiarire questo concetto partiamo dal caso semplice di una singola lettura, come mostrato in figura 8 [18]. Nella figura è mostrata la distribuzione statistica delle tensioni di soglia della cella da leggere. Ovviamente la scelta più ragionevole per la tensione di lettura (da applicare al gate della cella per discriminare il livello logico in essa contenuto) è quella mostrata in figura, che minimizza la probabilità di errore complessiva. Tuttavia, al termine della lettura, abbiamo solo una informazione grossolana riguardo al grado di affidabilità del valore letto: lo stesso valore di uscita si avrebbe sia nel caso di soglia molto diversa dal valore di V_{ref} (grado di affidabilità elevato) sia nel caso di valore molto vicino a V_{ref} (grado di affidabilità scarso).

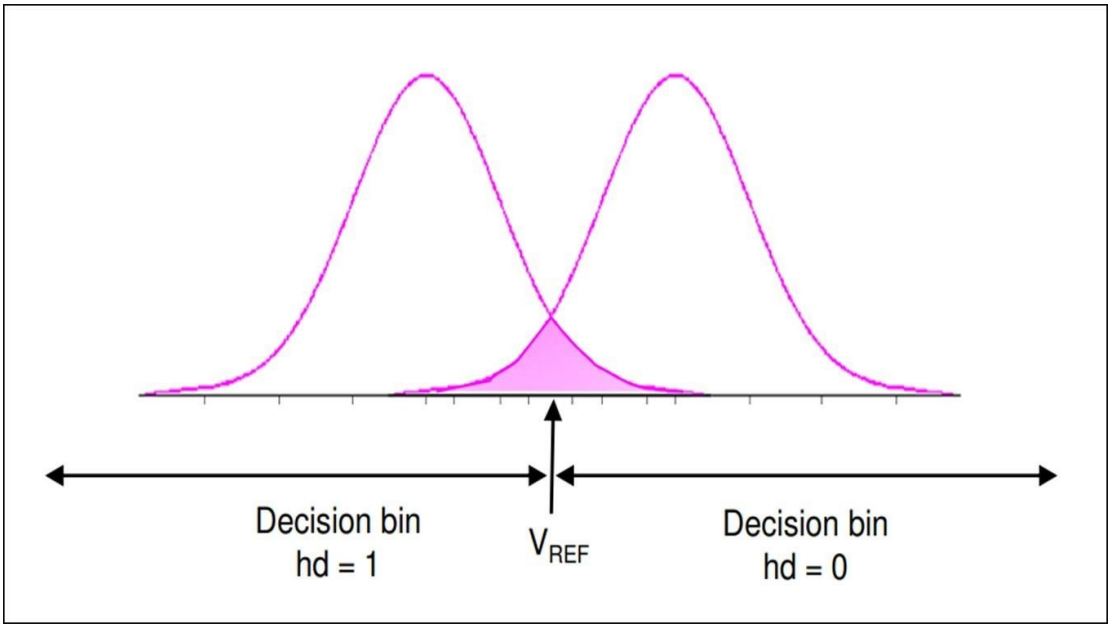

Figura 8

Potremmo però leggere la stessa cella flash più volte utilizzando inoltre diversi valori di tensione di lettura, come mostrato in figura 9 [18].

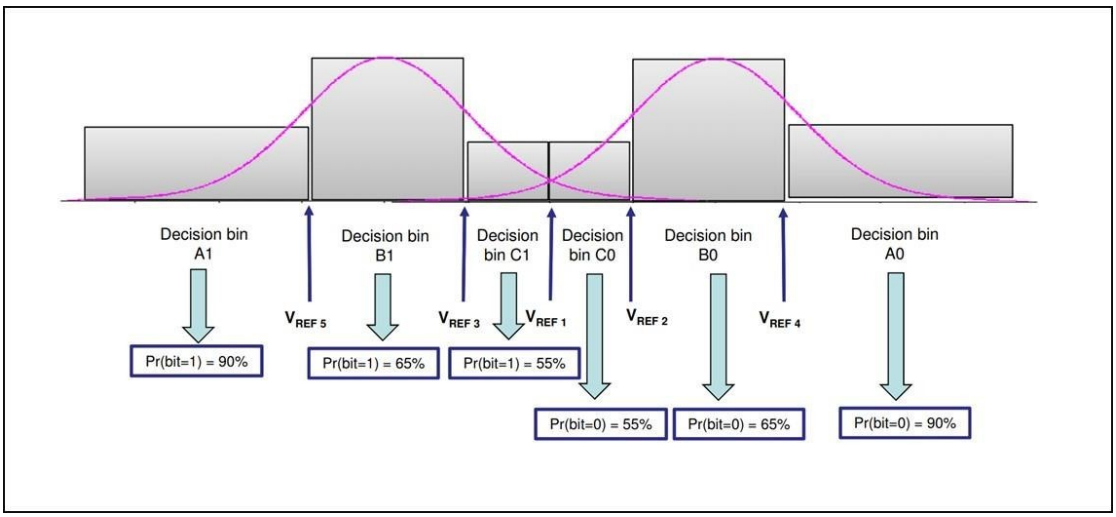

Figura 9

Nell'esempio della figura vengono effettuate 5 letture consecutive con tensioni di riferimento spaziate in modo opportuno così da creare dei *bin* (fasce di decisione) all'interno dei quali possiamo assegnare un grado di probabilità sul valore del bit

associato alla cella. Al termine delle 5 letture la probabilità, e quindi la corrispondente LLR sarà stimata e presentata in ingresso al soft decoder.

La procedura per decodificare la stringa in ingresso si basa sui ragionamenti seguenti.

In figura 10 è riportata la struttura logica del decoder, in forma di grafico bipartito costruito a partire dalla matrice **H**. Come nel caso hard decoding, i dati in ingresso vengono presentati al layer inferiore, solo che in questo caso non si tratta di bit ma di valori LLR associati a ciascun bit riletto.

L'algoritmo prevede una fase in cui dei valori vengono mandati dai bit nodes (strato inferiore) ai check nodes (strato superiore), e una fase in cui dai check nodes vengono restituiti dei valori che aggiornano i bit nodes. La procedura viene iterata fino alla conclusione della decodifica, come spiegheremo nel seguito.

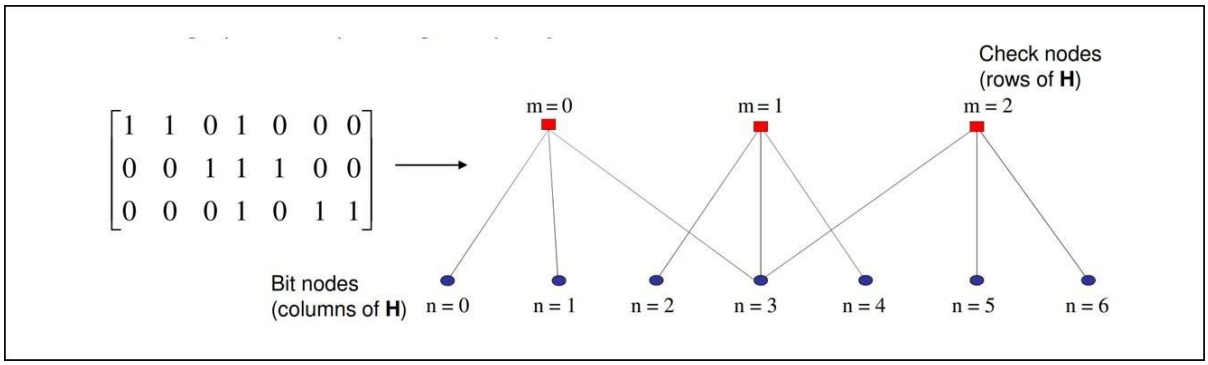

Figura 10

In effetti, esistono 4 categorie di segnali che vengono scambiati tra i nodi del decoder (figura 11):

1) Messaggi dal canale al bit node n-esimo (uscita LLR proveniente dai circuiti di lettura delle celle flash), che chiamiamo L_n;
2) Messaggi dall'n-esimo bit node al m-esimo check node, che indichiamo con $Q_{n\to m}^{(i)}$. In questa notazione il termine (i) indica l'i-esima iterazione dell'algoritmo. È ovvio che i messaggi vengono scambiati solo se esiste una linea che collega il nodo n al nodo m (nodi *adiacenti*, corrispondenti a un valore 1 nella matrice **H**);
3) Messaggi dall'm-esimo check node al n-esimo bit node, che indichiamo con $R_{m\to n}^{(i)}$. Anche in questa notazione il termine (i) indica l'i-esima iterazione

dell'algoritmo. Anche in questo caso è ovvio che i messaggi vengono scambiati solo se esiste una linea che collega il nodo n al nodo m (nodi *adiacenti*, corrispondenti a un valore 1 nella matrice **H**);

4) Informazione relativa all'affidabilità di ogni bit node al termine della i-esima iterazione, che indichiamo con $P_n^{(i)}$. Il nostro obbiettivo al termine della procedura è ottenere dei valori coerenti di questa quantità che permettano l'estrazione della stringa di dati corretta.

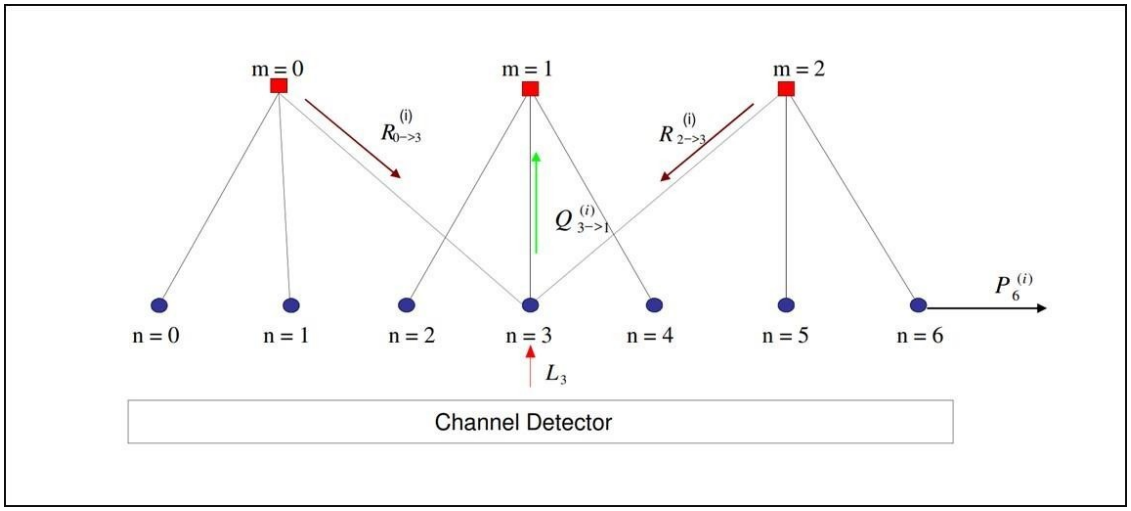

Figura 11

Analizziamo ora le varie fasi che costituiscono l'algoritmo. In figura 12 è mostrata la fase in cui i bit nodes mandano i valori ai check nodes adiacenti. La $Q_{n\to m}^{(i)}$ viene calcolata a partire dai valori della iterazione precedente in questo modo

$$Q_{n\to m}^{(i)} = L_n + \sum_{j\neq m} R_{j\to n}^{(i-1)} \qquad (2.9)$$

Ovviamente all'inizio del processo (i = 1) i valori di $R_{j\to n}^{(0)}$ sono posti tutti nulli. Si può notare che nella sommatoria non viene considerato il termine $R_{m\to n}^{(i)}$.

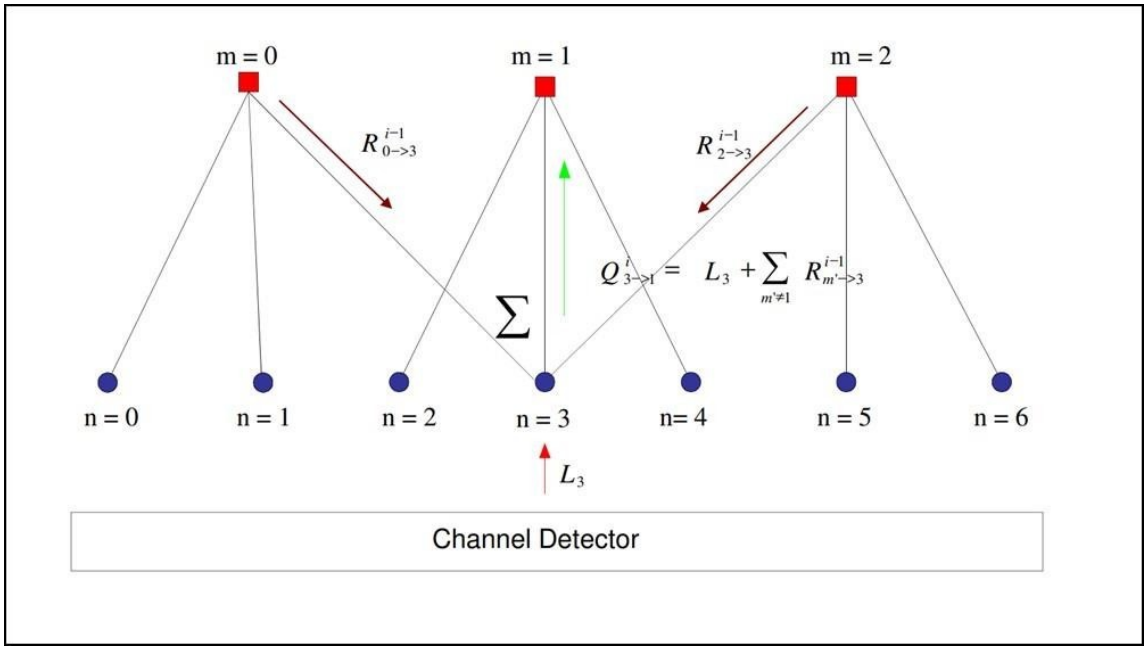

Figura 12

Cerchiamo ora di capire come viene generata la risposta dal check node m-esimo al bit node n-esimo ad esso adiacente $R^{(i)}_{m\to n}$. In base al *Sum-Product Algorithm* tale termine dovrebbe essere calcolato secondo la formula [19]

$$R^{(i)}_{m\to n} = 2\tanh^{-1}\left(\prod_{j\neq n}\tanh Q^{(i)}_{j\to m}\right) \qquad (2.10)$$

In realtà, considerata la complessità che la formula precedente comporta dal punto di vista computazionale, viene utilizzata la forma semplificata

$$R^{(i)}_{m\to n} = \prod_{j\neq n}\operatorname{sign}(Q^{(i)}_{j\to m})\times \min_{j\neq n}|Q^{(i)}_{j\to m}| \qquad (2.11)$$

L'algoritmo che utilizza la (2.11) in luogo della (2.10) viene detto *Min-Sum Algorithm*. Considerando la figura 13 [18] notiamo che i tre bit nodes presentano al check node m 3 valori (-10, -5, 13). Nel "rispondere" a ciascun bit node, l'algoritmo esegue le seguenti operazioni:

1) Esclude il valore relativo al bit node cui deve rispondere;

2) Dei rimanenti, seleziona il termine che aveva il valore assoluto più basso;
3) Questo valore viene restituito al bit node scegliendo il segno in base alla formula (2.11). Ad esempio, nel rispondere al nodo n_1 si deve scegliere tra -5 e 13: la scelta ricade su -5. Il segno deve essere negativo in quanto i due valori sono discordi. Nel rispondere al nodo n_2 si deve scegliere tra -10 e 13. In questo caso la scelta ricade su -10 e il segno è negativo poiché avevamo due valori discordi. Infine, nel rispondere al nodo n_3, la scelta ricade su -5, ma il segno è positivo in quanto i valori in ingresso erano concordi (entrambi negativi).

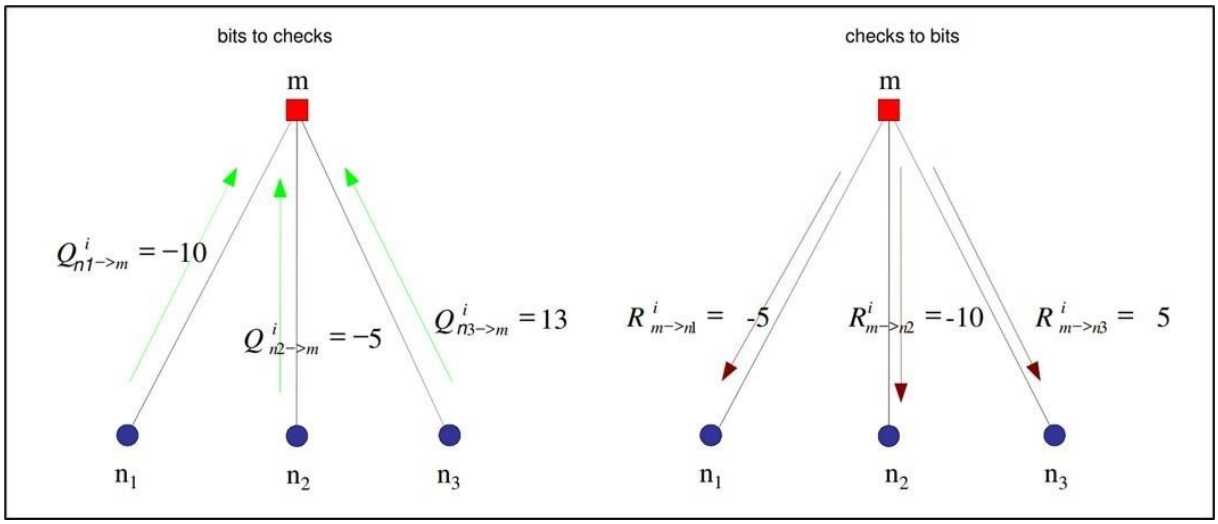

Figura 13

Alla fine di ogni iterazione, il grado di affidabilità associato a ciascun bit node viene aggiornato secondo la formula

$$P_n^{(i)} = L_n + \sum_m R_{m\to n}^{(i)} \qquad (2.12)$$

A questo punto, $P_n^{(i)}$ viene quantizzato per stimare il valore del bit nel seguente modo

$$\hat{x}_n = \begin{cases} 1, se\ P_n^{(i)} < 0 \\ 0,\ altrove \end{cases} \qquad (2.13)$$

Questo risultato potrebbe essere soddisfacente e portare alla conclusione del processo, o non esserlo, e richiedere almeno una ulteriore iterazione.

Il criterio in base al quale l'algoritmo termina può essere così riassunto:

1) Abbiamo raggiunto il massimo numero di iterazioni consentite, OPPURE
2) Tutti i criteri di parity check sono soddisfatti (sindrome nulla).

Per comprendere meglio il meccanismo forniamo ora due esempi tratti da [18]: nel primo l'algoritmo riesce ad arrivare a termine in una sola iterazione, nel secondo sono necessarie due iterazioni.

Per quanto riguarda il primo esempio, riferiamoci alla figura 14. Nella parte alta della figura sono evidenziati i valori originali in ingresso ai bit nodes (-9, 7, -12, +4 …). In rosso è evidenziato che il quarto valore in ingresso corrisponde a una lettura errata.

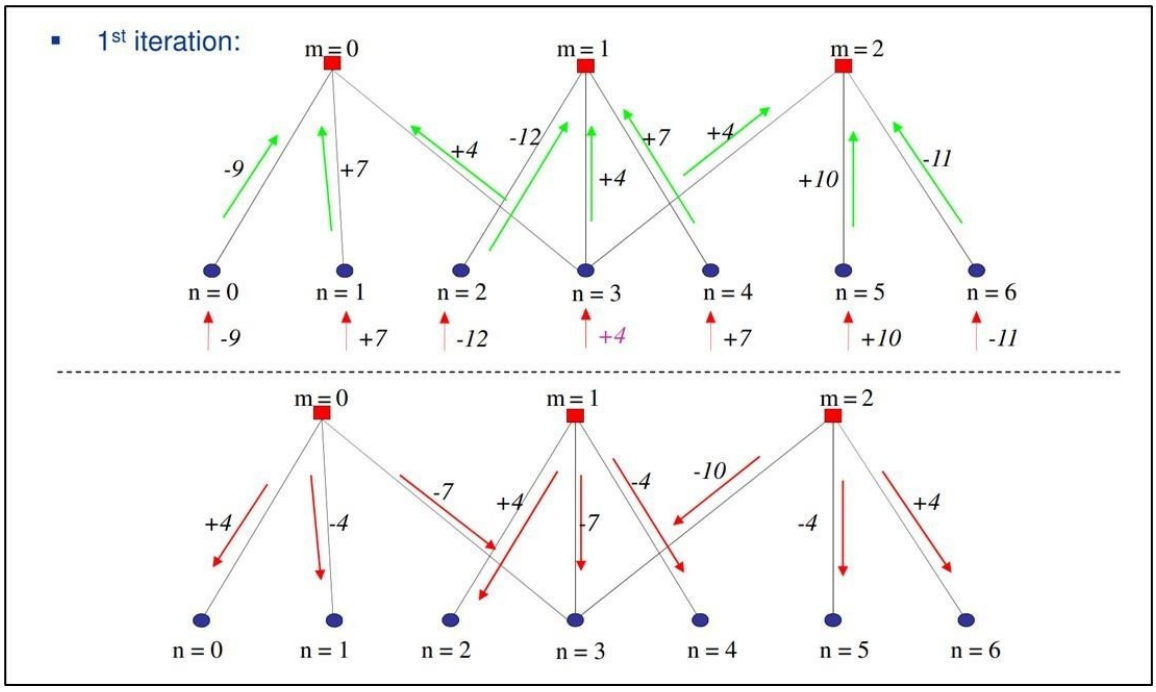

Figura 14

Alla prima iterazione tali valori sono presentati senza alterazione ai nodi adiacenti. Nella parte inferiore della figura possiamo osservare le risposte che i check nodes mandano ai bit nodes adiacenti, calcolate secondo il criterio (2.11).

Nella figura 15 è mostrato il risultato dei gradi di affidabilità ottenuti applicando la formula (2.12) da cui derivano, in base alla (2.13) i valori dei bit riletti. Calcolando la sindrome della stringa così determinata l'algoritmo decide che corrisponde a una codeword ammissibile e termina. In effetti, come mostrato in figura, il bit n = 3 che originariamente veniva letto come 0, ora è diventato 1.

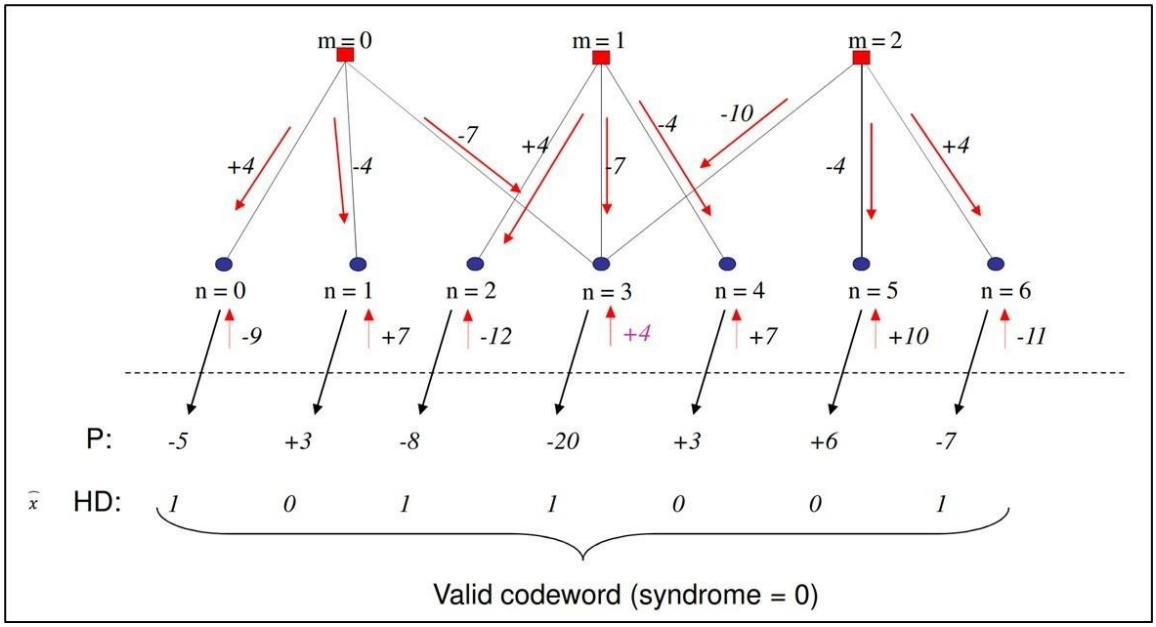

Figura 15

Per quanto riguarda il secondo esempio, consideriamo la figura 16. In questo caso il secondo bit da sinistra corrisponde a una lettura errata. Analogamente a prima, nella parte alta della figura vengono presentati i valori dai bit nodes ai check nodes adiacenti, mentre nella parte bassa sono evidenziate le risposte dai check nodes ai bit nodes adiacenti.

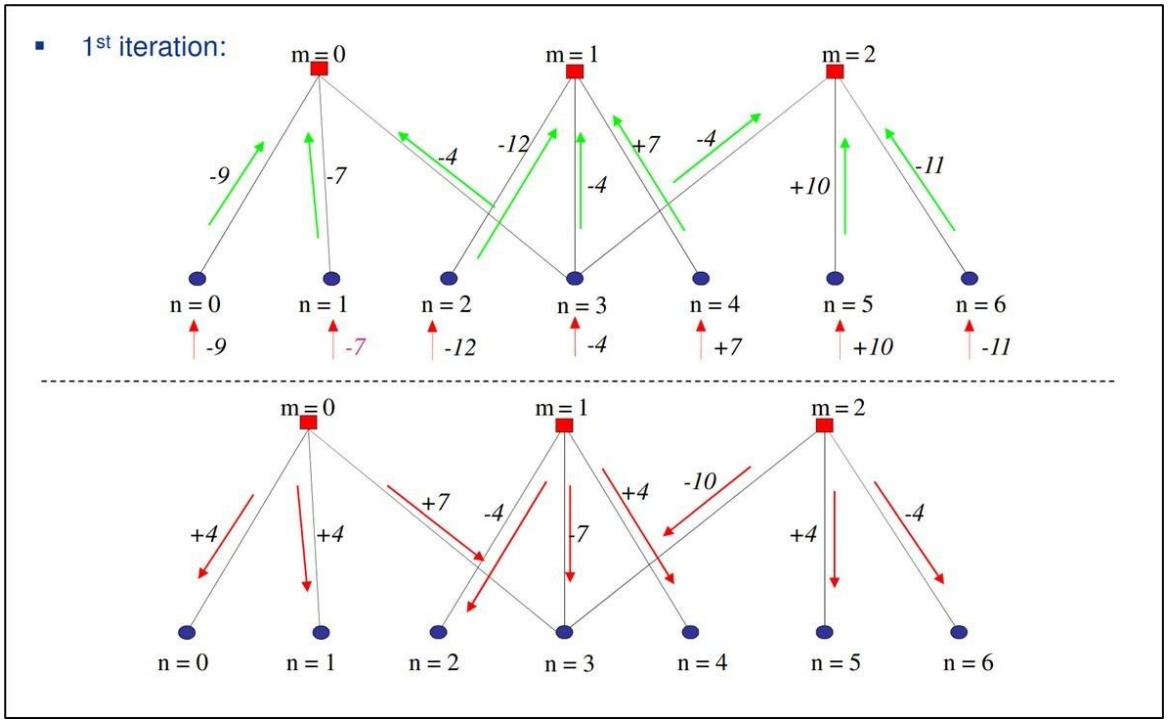

Figura 16

Poiché alla fine della prima iterazione il risultato non è ancora soddisfacente (infatti la $P_1^{(1)}$ risulta ancora negativa mentre dovrebbe essere positiva), è necessario eseguire una ulteriore iterazione, come mostrato in figura 17

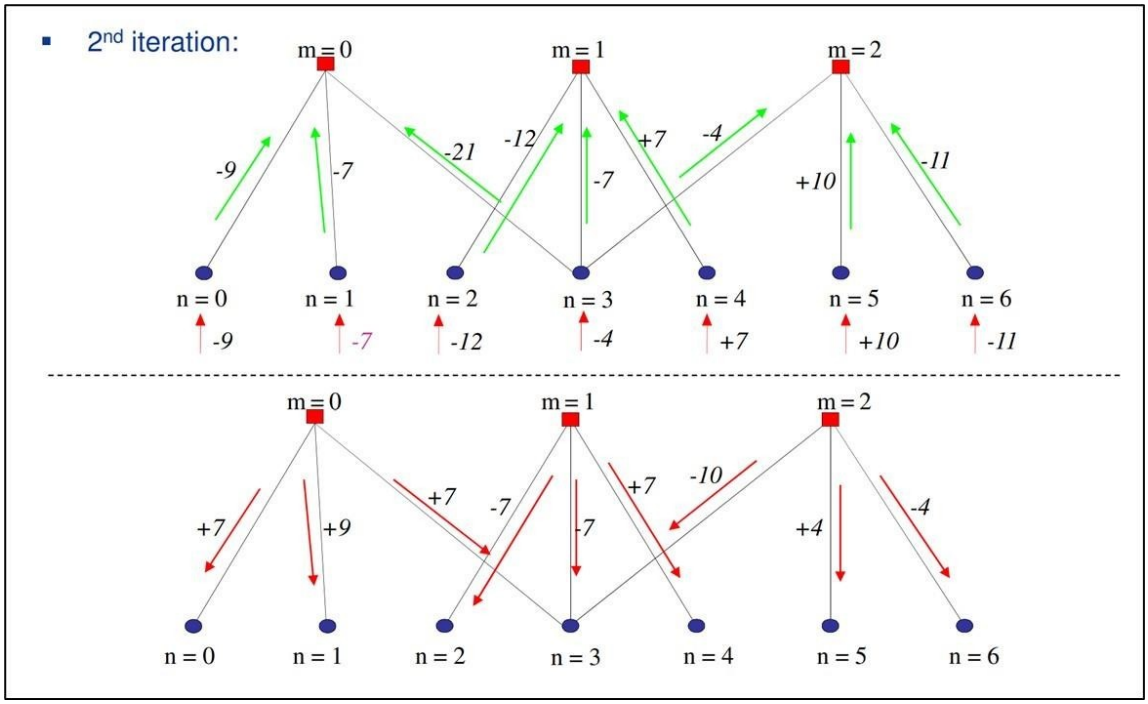

Figura 17

Il risultato alla fine della seconda iterazione è mostrato in figura 18: in questo caso la $P_1^{(2)}$ è diventata positiva, dando origine a una codeword ammissibile (sindrome nulla) e causando la terminazione dell'algoritmo con la correzione del bit n = 1.

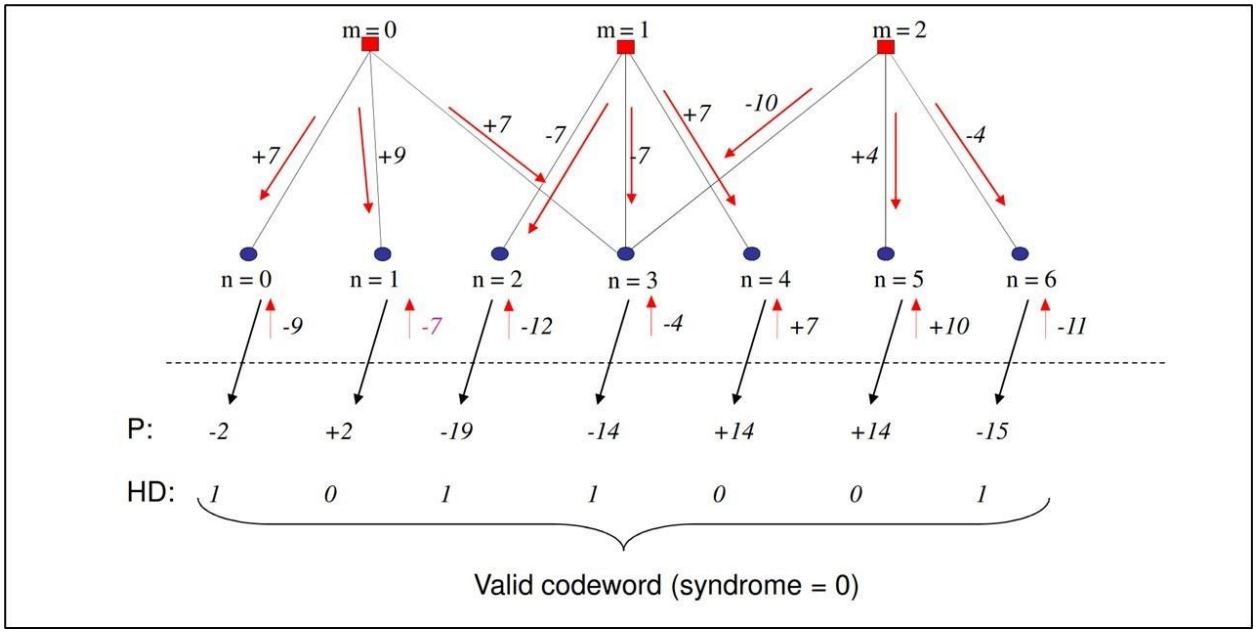

Figura 18

Utilizzando questo tipo di decodifica soft, è possibile partire da una situazione in cui sono presenti molti errori. L'algoritmo, basandosi sulle informazioni soft presenti in ingresso, riesce a inferire un valore più attendibile per l'iterazione successiva. Man mano che si progredisce con le iterazioni diminuisce il numero di errori fino a convergere alla stringa corretta. Un esempio è mostrato in figura 19 [18].

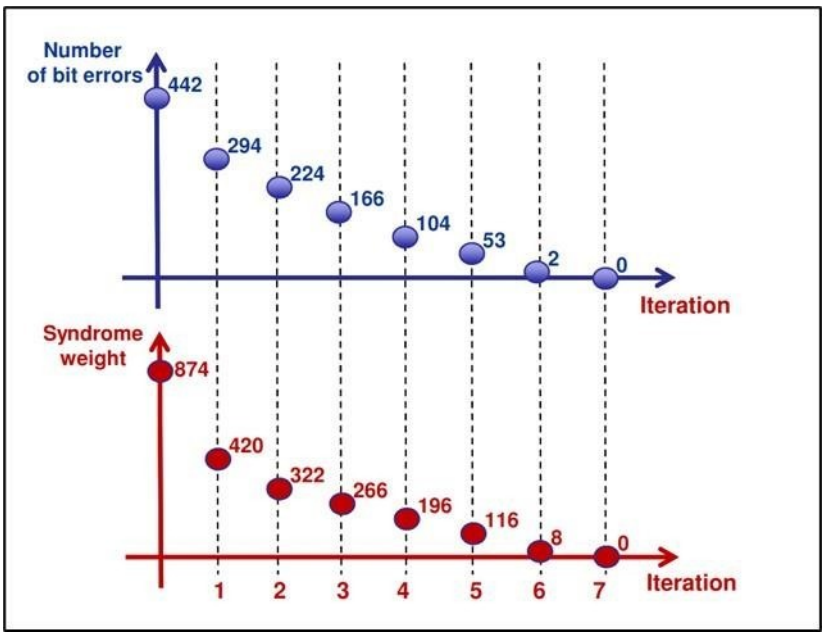

Figura 19

Confrontando questa figura con la figura 6 relativa al decodificatore hard, notiamo che la capacità di correzione del decodificatore soft è notevolmente superiore. Mentre in figura 6 si correggevano 91 errori in 7 iterazioni, nel caso di figura 19 con lo stesso numero di iterazioni se ne correggono 442.

II.4 Prestazioni

Le decodifiche LDPC presentano notevoli vantaggi per la loro capacità di correzione, come abbiamo osservato. D'altra parte, bisogna evidenziare che si tratta di algoritmi *non deterministici*, pertanto non danno una garanzia assoluta sulla correzione di una determinata stringa in ingresso. Paradossalmente, a fronte della capacità di correggere molti errori, non è escluso (pur essendo molto improbabile) che l'algoritmo sia incapace di correggere particolari stringhe contenenti pochi errori.

Le prestazioni di un (generico) codice di canale sono misurate dalla diminuzione della probabilità di errore sul bit (BER) all'aumentare del rapporto segnale-rumore (S/N). Per codici del tipo in oggetto (come per i classici turbo) a partire da un certo valore di S/N il BER diminuisce molto velocemente: la regione in cui ciò avviene è chiamata *waterfall region*. Oltre un certo valore di S/N, però, la curva di BER è di norma caratterizzata da una brusca diminuzione di pendenza, che fa sì che, per ottenere un valore di BER sufficientemente basso, sia necessario un rapporto segnale-rumore significativamente elevato. Il relativo andamento della curva di BER prende il nome di *error floor*.

Ad esempio, in figura 20 [18] è mostrato il confronto tra una codifica LDPC e una codifica RS/BCH. Dal grafico risulta evidente che la codifica LDPC risulta particolarmente vantaggiosa in una zona in cui il rapporto segnale rumore del segnale in ingresso è molto basso, il che è equivalente a dire che funziona in maniera eccellente in condizioni "proibitive": questa è proprio la *waterfall region*. Tuttavia, come già osservato, man mano che il S/N aumenta, le sue prestazioni non migliorano di pari passo, anzi tendono ad appiattirsi nella *error floor region* tanto da diventare inferiori a quanto si potrebbe ottenere con la RS/BCH.

La causa è generalmente da attribuire a bit che sono incorporati in una sola delle equazioni di parità, e quindi hanno una maggiore probabilità di non venire corretti dalla codifica. Siamo in presenza delle *near codewords*: si tratta di combinazioni di bit particolarmente "sfortunate" tali da essere molto simili a delle codeword ammissibili (da cui il nome) e capaci di mettere l'algoritmo di decodifica in una fase oscillatoria in cui le iterazioni potrebbero procedere all'infinito senza mai convergere a una situazione di sindrome nulla. Ovviamente, poiché il numero massimo di iterazioni è limitato,

l’algoritmo arriverà a termine ma non sarà riuscito a correggere la stringa in ingresso: il dato non può essere letto.

Un’altra causa di errore è l’errata interpretazione della stringa letta (*miscorrection*): sostanzialmente viene ottenuta una codeword ammissibile ma essa non corrisponde a quella effettivamente memorizzata nella flash. La figura 21 mostra graficamente questa situazione.

È stato osservato che nelle memorie flash di produzione l’impatto della miscorrection come causa della floor region è molto minore rispetto a quello delle near codewords.

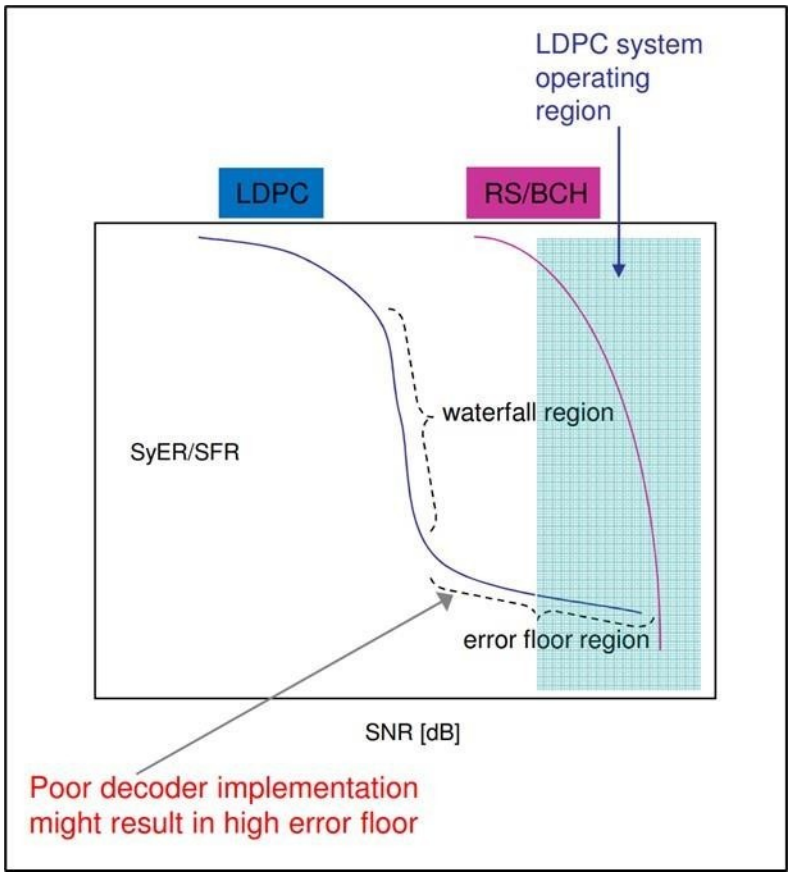

Figura 20

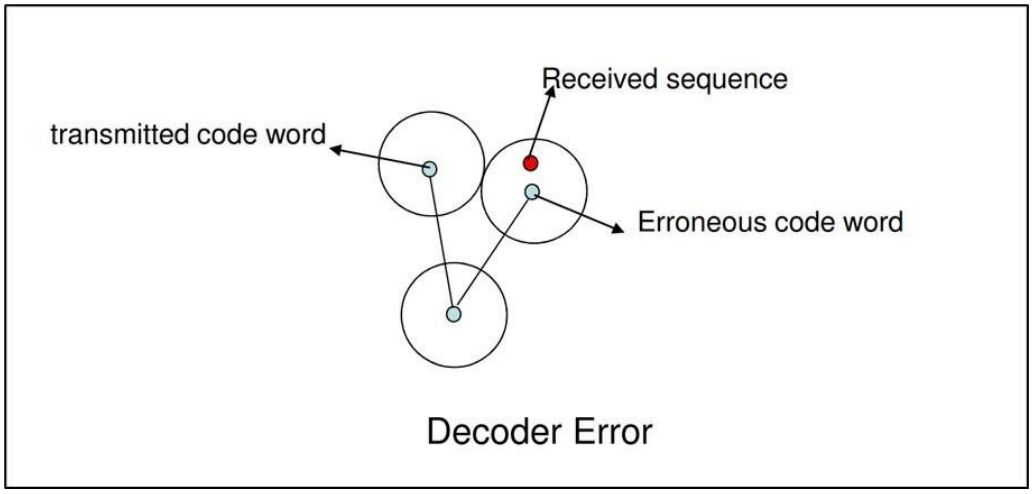

Figura 21

Bisogna inoltre considerare che per costruzione le memorie flash dovrebbero essere il più affidabili possibile, cioè si cerca di progettarle e costruirle in modo da rendere il *raw bit rate* (RBR) molto basso (RBR basso è equivalente a SNR alto). Pertanto, il decoder tenderà tipicamente a lavorare nella error floor region (figura 22).

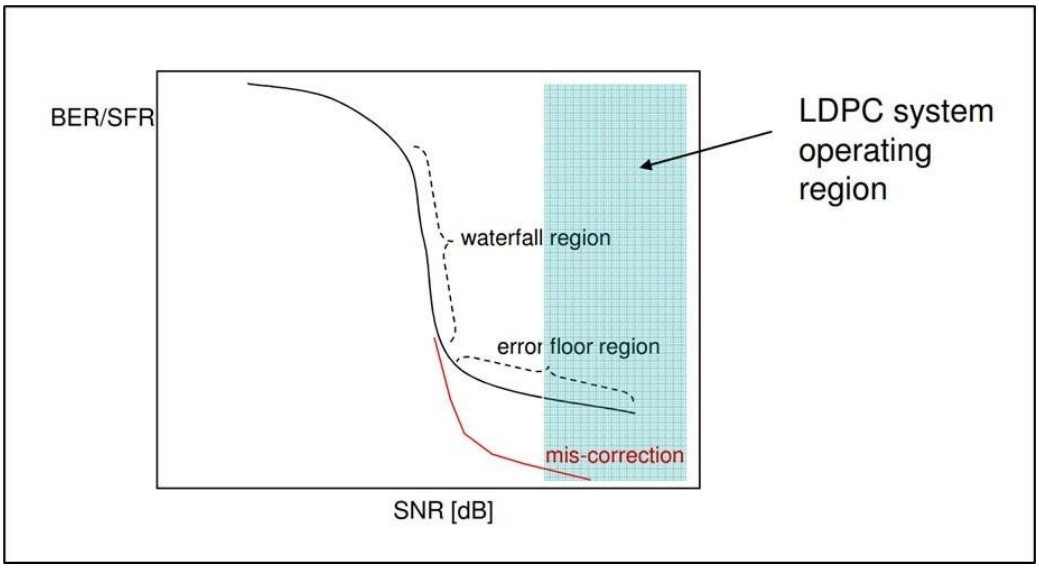

Figura 22

Esistono tecniche per mitigare l'effetto dell'error floor. Per esempio, si ottimizza il codice in maniera da "spingere" l'error floor al di sotto di un livello prestabilito. In

alternativa si potrebbe ricorrere a un *post processing*, cioè una ulteriore elaborazione sul segnale decodificato.

La figura 23 mostra un esempio di risultato ottenibile con questa tecnica.

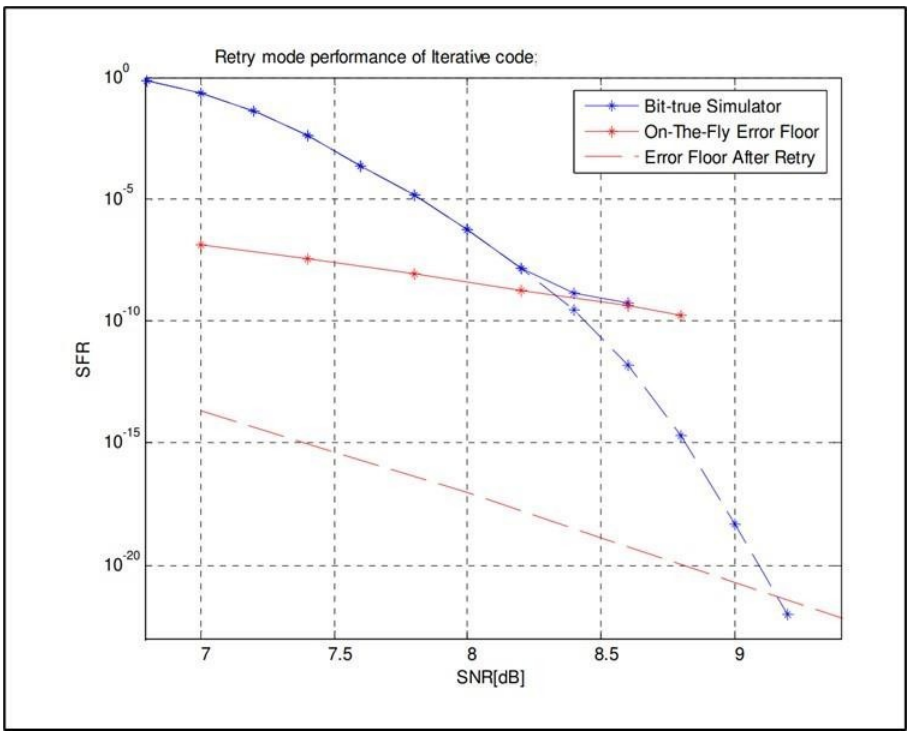

Figura 23

La figura 24 [20] mostra le prestazioni di alcuni codici LDPC con rate 1/2, e al variare della lunghezza del blocco dati (in figura in luogo di S/N viene indicata la quantità E_b/N_0), ovvero il rapporto tra l'energia per bit e la densità spettrale di potenza unilatera del rumore termico). La modulazione utilizzata è la 2-PSK. L'error floor, che dipende dai parametri del codice, non appare per i codici lunghi (per essi, infatti, si verifica per valori di BER più bassi rispetto a quelli riportati in figura, mentre risulta evidente, ad esempio, per il codice (512, 256)).

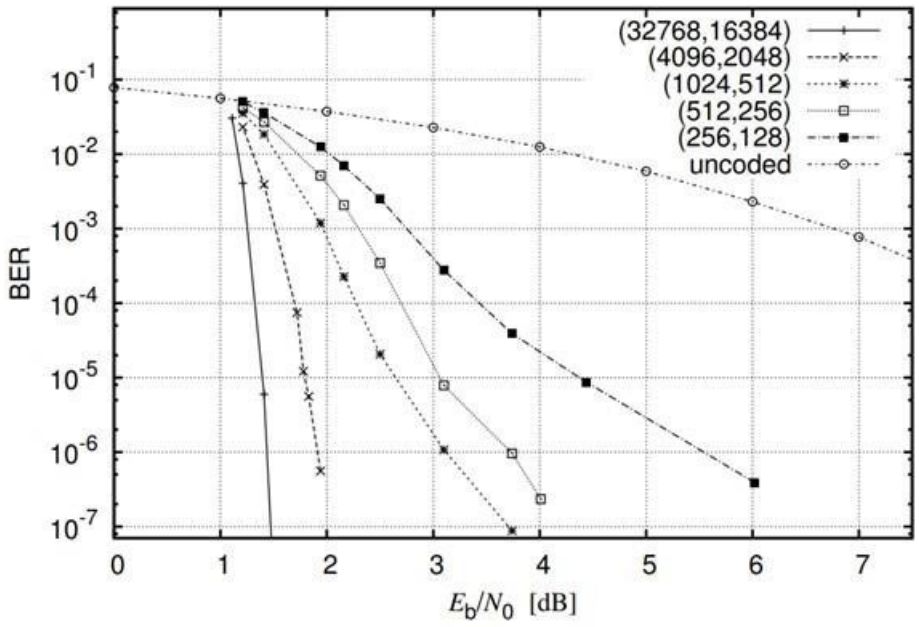

Figura 24

I codici LDPC sono molto impiegati nelle memorie NAND multilivello, e consentono una significativa riduzione del BER: è stato dimostrato che possono avvicinarsi al limite di Shannon.

Le figure 25 e 26 tratte da [21] mostrano un confronto tra limite di Shannon e due diverse codifiche: LDPC vs BCH.

Nell'asse orizzontale viene riportato il code rate R (il rapporto tra i bit di dati e i bit totali) e nell'asse verticale il BER senza codifica (cioè prima della correzione dell'errore). La curva continua in alto rappresenta il limite di Shannon in funzione di R. Più le curve relative a una certa codifica sono spostate verso l'alto migliore è l'efficienza del codice scelto. La figura 25 si riferisce a due decoder LDPC *Hard decision* con R = 0.8 e R = 0.95 rispettivamente, mentre la figura 26 si riferisce a un decoder LDPC *Soft decision* con R = 0.8.

A riprova di quanto già osservato, cioè che le decodifiche *soft* hanno prestazioni superiori alle decodifiche *hard,* osserviamo che nel primo caso la prestazione LDPC è inferiore alla BCH, nel secondo caso invece si verifica un notevole miglioramento.

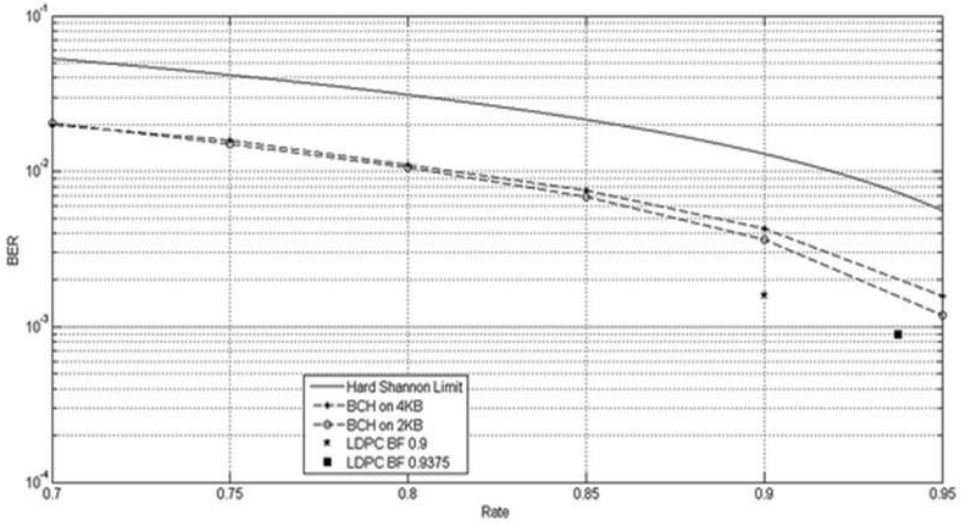

Figura 25

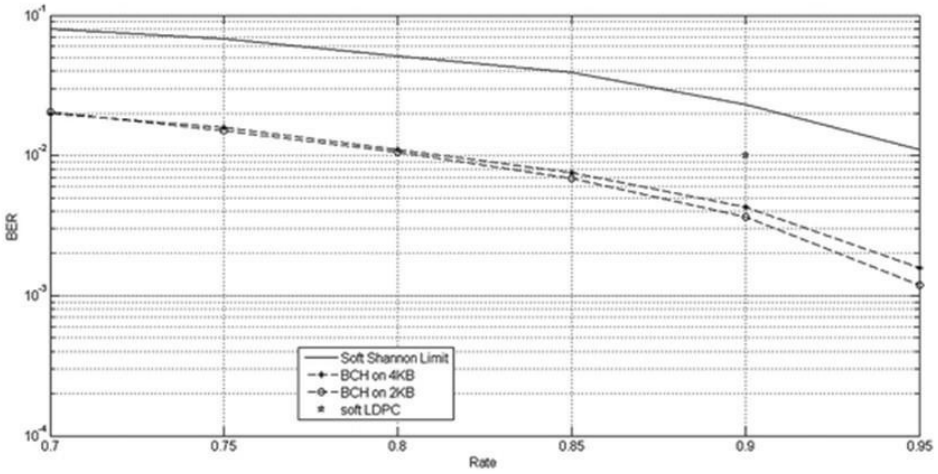

Figura 26

Bibliografia

[1] Yu, K., Beam, A.L. & Kohane, I.S. Artificial intelligence in healthcare. Nat Biomed Eng 2, 719–731 (2018)

[2] Yasuo Tarui; Yutaka Hayashi; Kiyoko Nagai, "Proposal of electrically reprogrammable non-volatile semiconductor memory". Proceedings of the 3rd Conference on Solid State Devices, Tokyo, (1971-09-01). The Japan Society of Applied Physics: pp. 155–162

[3] Masuoka et al, “Semiconductor memory device and method for manufacturing the same”, US patent n. 4531203, 1985

[4] R. Micheloni, A. Marelli, R. Ravasio, “NOR Flash Memories”, in ERROR CORRECTION CODES FOR NON-VOLATILE MEMORIES, Springer 2008, pp. 61-84

[5] R. Micheloni, A. Marelli, R. Ravasio, “NOR Flash Memories”, in ERROR CORRECTION CODES FOR NON-VOLATILE MEMORIES, , Springer 2008, pp. 85-102

[6] Greg Atwood, Al Fazio, Duane Milla, Bill Reaves. "StrataFlash Memory Technology Overview". Intel Technology Journal. Intel Corporation. September 1997

[7] P. L. Rolandi et al., "1 M-cell 6b/cell analog flash memory for digital storage," 1998 IEEE International Solid-State Circuits Conference. Digest of Technical Papers, ISSCC. First Edition (Cat. No.98CH36156), San Francisco, CA, USA, 1998, pp. 334-335

[8] J. Bellorado, “Noise Sources in NAND Flash Memory”, Non-Volatile Memory Workshop, CMRR, University of California, San Diego, March 3rd, 2013

[9] K LeK. Le and F. Ghaffari, "On the Use of Hard-Decision LDPC Decoders on MLC NAND Flash Memory," 2018 15th International Multi-Conference on Systems, Signals & Devices (SSD), Hammamet, 2018, pp. 1453-1458

[10] Robert G. Gallager (1963). Low Density Parity Check Codes. Monograph, M.I.T. Press. Retrieved August 7, 2013

[11] David J.C. MacKay and Radford M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes," Electronics Letters, Electronics Letters, 32 (18): 1645–1646, 1996

[12] France Telecom, “Error-correction coding method with at least two systematic convolutional codings in parallel, corresponding iterative decoding method, decoding module and decoder”. US Patent 5,446,747, filed in 1992

[13] C. E. Shannon "Communication in the presence of noise" Proceedings of the Institute of Radio Engineers. 37 (1): 10–21. Jan. 1949

[14] Judea Pearl "Reverend Bayes on inference engines: A distributed hierarchical approach" Proceedings of the Second National Conference on Artificial Intelligence. AAAI-82: Pittsburgh, PA. Menlo Park, California: AAAI Press. pp. 133–136

[15] T.B. Iliev, G.V. Hristov, P.Z. Zahariev, M.P. Iliev “Application and evaluation of the LDPC codes for the next generation communication systems.” In: Sobh T., Elleithy K., Mahmood A., Karim M.A. (eds) Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics. Springer, Dordrecht

[16] Althoen, Steven C.; McLaughlin, Renate (1987), "Gauss–Jordan reduction: a brief history", The American Mathematical Monthly, Mathematical Association of America, 94 (2): 130–142, doi:10.2307/2322413, ISSN 0002-9890, JSTOR 2322413

[17] L. Shiva Nagender Rao, K.Venkata Sathyajith, Y. Nagaraju Yadav. “Encoding and Decoding of LDPC Codes using Bit Flipping Algorithm in FPGA”. International Journal of Innovative Research in Computer and Communication Engineering Vol. 5, Issue 8, August 2017, pp. 14128-14142

[18] N. Varnica, “LDPC Decoding: VLSI Architectures and Implementations”, Flash Memory Summit 2013

[19] Vikram Arkalgud Chandrasetty, Syed Mahfuzul Aziz, “Resource Efficient LDPC Decoders: From Algorithms to Hardware Architectures”, Academic Press, Dec 5, 2017

[20] Low-Density Parity-Check code – An introduction – Tilo Strutz, 2010-2014, 2016 June 9, 2016. http://www1.hft-leipzig.de/strutz/Kanalcodierung/ldpc_introduction.pdf

[21] Alessia Marelli, Rino Micheloni, “BCH and LDPC Error Correction Codes for NAND Flash Memories”, in 3D FLASH MEMORIES, Ed. Rino Micheloni, Springer