POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Elettronica

Politecnico
di Torino

Tesi di Laurea Magistrale

Design digitale di un Viterbi Detector in tecnologia FinFET a 3 nm

Relatori

Prof. Maurizio MARTINA

Prof. Guido MASERA

Candidato

Francesco GALLARETO

Supervisori aziendali
STMicroelectronics

Piero BONIFACINO

Mauro NATUZZI

Aprile 2023

Sommario

Con la costante crescita di scambio dati nelle comunicazioni, si ha la necessità di aumentare la velocità di trasmissione e ricezione, per questo vengono sviluppati dispositivi SerDes (Serializzatore - Deserializzatore) che possano lavorare a frequenze elevate dove i canali di trasmissione hanno attenuazioni ingenti. Per recuperare queste attenuazioni si introducono nei ricevitori degli algoritmi che stimano la sequenza dei dati a massima verosimiglianza. L'algoritmo Viterbi è uno di questi da usare all'interno dei ricevitori dei sistemi SerDes.
La tesi ha lo scopo di valutare le prestazioni, l'occupazione di area e il consumo di potenza di diverse implementazioni del Viterbi Detector, con lo scopo di avere dati relativi per la progettazione di un SerDes in tecnologia FinFET a 3 nm di TSMC. Inizialmente è stato realizzato un modello matlab, utilizzato per confermare l'effettivo miglioramento delle prestazioni utilizzando il Viterbi rispetto ad altri equalizzatori, e successivamente come riferimento per la verifica delle implementazioni RTL. Nella prima fase di implementazioni RTL sono state studiate alcune ottimizzazioni per la path metric unit (PMU), la parte più complessa del Viterbi, ottenendo informazioni su quale sia la miglior versione sia se l'obiettivo è il risparmio di area sia se è ottenere le migliori prestazioni in termini di frequenza massima raggiungibile. Nella seconda parte la tesi si concentra sulla realizzazione di un Viterbi Detector direttamente utilizzabile all'interno di un ricevitore, dove tipicamente sono presenti più ingressi in parallelo, nel caso specifico 32 campioni sono processati per ogni colpo di clock. In questa fase sono state studiate tre possibili implementazioni, la sintesi è stata effettuata a frequenza costante, analizzando il consumo di potenza e l'area occupata. Delle tre, le prime due utilizzano diversi moduli Viterbi in parallelo, mentre la terza è un singolo Viterbi modificato in modo tale da analizzare più ingressi contemporaneamente. Quest'ultima versione è risultata la migliore, mostrando una riduzione sia di area sia di potenza quasi del 50% rispetto ai casi precedenti.
Il lavoro svolto ha ricercato le possibili implementazioni e tra queste ha trovato l'implementazione più efficace in funzione del parallelismo richiesto. Come appendice si sono comparate le varie tipologie di PMU, studiando il compromesso area-frequenza.

Ringraziamenti

Mi è doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito, con il loro instancabile supporto, alla realizzazione dello stesso.

Vorrei innanzitutto ringraziare i miei relatori, il prof. Martina e il prof. Masera, per la disponibilità e la chiarezza mostrate nel rispondere ai dubbi e alle domande sorte nel corso dell'attività.

Ringrazio Piero Bonifacino e Mauro Natuzzi, per avermi seguito durante tutto il mio percorso in ST. Questi mesi di lavoro sono stati molto stimolanti e il vostro aiuto mi ha permesso di ampliare le mie conoscenze molto più di quanto avrei sperato. Ringrazio anche tutti gli altri colleghi, che grazie ai loro consigli e al loro aiuto mi hanno permesso di lavorare in un ambiente straordinario, che mi ha permesso di mettermi in gioco e fare un'esperienza che sarà preziosa per il mio futuro.

Ringrazio i miei compagni di corso, ma soprattutto amici, con cui ho condiviso questi ultimi anni. In primis ringrazio Carlo e Paola, che a partire dalle infinite videochiamate durante la pandemia mi sono sempre stati accanto. Grazie a Stefano, Federica, Alessandra, Francesco, Sara, Alessandro, Matteo, Gaetano e Cristian per essere stati sempre presenti durante questo percorso.

Ringrazio Luca, che ha avuto il coraggio di condividere una casa con me per più di 5 anni, e tutti gli altri amici su cui ho sempre potuto contare quando tornavo a casa.

Infine ringrazio la mia famiglia, perchè senza di loro tutto questo sarebbe stato impossibile. Grazie per avermi sempre supportato e per aver creduto in me durante tutto questo percorso. Non sono bravo nei ringraziamenti, ma spero sappiate quanto siete importanti per me.

Indice

Elenco delle tabelle

Elenco delle figure

Elenco delle sigle

ACS

Add Compare Select

BER

Bit Error Rate

BMU

Branch Metric Unit

CC

Code Converter

CDR

Clock Data Recovery

CSA

Carry Save Adder

CTLE

Continuous Time Linear Equalizer

DFE

Decision Feedback Equalizer

DTD

Differential Trellis Decoding

FBF

FeedBack Filter

FFF

FeedForward Filter

FIR

Finite Impulse Response

IEA

International Energy Agency

ISI

InterSymbol Interference

LSB

Least Significant Bit

MAP

Maximum A Posteriori probability

MSB

Most Significant Bit

NRZ

No Return to Zero

PAM

Pulse Amplitude Modulation

PCB

Printed Circuit Board

PCI

Peripheral Component Interconnect

PLL

Phase Locked Loop

PMU

Path Metric Unit

QAM

Quadrature Amplitude Modulation

RTL

Register Transfer Level

SerDes

Serializer Deserializer

SNR

Signal to Noise Ratio

TBU

Trace Back Unit

TSMC

Taiwan Semiconductor Manufacturing Company

Capitolo 1

Introduzione

Negli ultimi anni la crescita nella quantità di dati scambiati nelle comunicazioni è stata costante. Per questo motivo è necessario sviluppare dispositivi sempre più performanti che possano far fronte a questo aumento.
Un esempio di questo aumento è visibile nel caso dei data center, che sono strutture utilizzate per l'immagazzinamento e l'elaborazione dei dati. L'aumento dell'importanza della rete e dei sistemi di elaborazione, sia in ambito industriale che in ambito privato, ha fatto sì che i data center e i loro servizi abbiano assunto un ruolo fondamentale nella civiltà moderna.
Come indicato nel report dell'International Energy Agency (IEA) [1], dal 2015 al 2021 il traffico internet è passato da 0.6 ZB a 3.4 ZB, con un aumento del 440% e il consumo di energia dei data center è passato da 200 TWh a 220-320 TWh, con un aumento che va dal 10% al 60%. Per comprendere meglio l'importanza di questa industria a livello globale, un dato importante è quello che indica come il consumo di energia da parte dei data center corrisponda a circa l'1% del consumo di energia mondiale, e questo dato è destinato a crescere nei prossimi anni.
Questi dati rendono evidente la necessità di aumentare la velocità di trasmissione sia all'esterno ma anche all'interno dei data center, e di ridurre il consumo di potenza di tutti i dispositivi che sono coinvolti in queste operazioni.
Uno dei punti più critici all'interno di questa infrastruttura è la comunicazione tra i vari server, che deve avere prestazioni molto elevate per non rappresentare il collo di bottiglia di tutto l'impianto. Essendo le interconnessioni tra i server numerosissime, per limitarne la complessità e il costo è preferibile utilizzare la comunicazione seriale, che nonostante presenti dei limiti sulla quantità di dati che possono essere trasmessi, è la tecnica più semplice ed economica.
Uno dei parametri chiave nella realizzazione di un sistema seriale è il bit error rate (BER), cioè il rapporto tra il numero di bit non ricevuti correttamente e quello dei bit trasmessi. Questo parametro è uno dei principali indicatori della qualità della trasmissione. Una delle tecniche per mantenere il suo valore il più basso possibile

è quella di utilizzare degli equalizzatori in fase di ricezione, che possono essere di diversi tipi: alcuni sono completamente analogici, altri sono digitali o misti. Nel caso in cui sia presente una parte digitale un componente che può essere utilizzato è il Viterbi Detector, che ha lo scopo di decidere quale sia la sequenza di dati che ha la maggior probabilità di essere stata trasmessa, basandosi sui valori ricevuti in uscita dal canale.

1.1 Obiettivi della tesi e struttura

L'obiettivo della tesi è quello di valutare i vantaggi e gli svantaggi derivanti dall'utilizzo di un Viterbi Detector durante la ricezione di un segnale digitale trasmesso a frequenze elevate, quindi soggetto a molti disturbi.
In particolare, sono state descritte in verilog e system verilog diverse implementazioni di questo algoritmo. Da queste, effettuando la sintesi in tecnologia FinFET a 3 nm di TSMC (Taiwan Semiconductor Manufacturing Company), è stato possibile confrontare la frequenza massima a cui possono lavorare, l'area occupata e la potenza consumata. Questi valori hanno permesso di stabilire quale sia la versione migliorare tra quelle studiate nel caso in cui debba essere utilizzata all'interno di un SerDes (Serializer – Deserializer).
Dopo questo capitolo di introduzione, la tesi è strutturata nei seguenti capitoli:

2. L'algoritmo Viterbi: in questo capitolo viene fornita una descrizione dell'algoritmo, in questo modo vengono esposte le basi teoriche su cui si basano le implementazioni sviluppate in seguito.

3. Modello matlab: questo capitolo descrive la fase iniziale della tesi, in cui è stato realizzato un modello matlab che simula il comportamento di un Viterbi Detector. Dopo averne verificato la validità, questo modello sarà utilizzato come riferimento per dimostrare il funzionamento delle implementazioni realizzate nei capitoli successivi.

4. Implementazioni Viterbi singolo: le implementazioni RTL (Register Transfer Level) descrivono diversi modi in cui può essere realizzata una delle parti più complesse all'interno di un Viterbi Detector. Dopo aver effettuato le sintesi delle diverse soluzioni vengono confrontati e discussi i risultati ottenuti riguardo all'area e alla frequenza massima di funzionamento.

5. Implementazioni Viterbi parallelo: in questo capitolo vengono studiate diverse applicazioni che permettono l'analisi di segnali ricevuti in parallelo, permettendo di aumentare la quantità di dati analizzati. Queste sono le implementazioni

che verrebbero effettivamente utilizzate in un SerDes. In questo caso vengono analizzati i risultati ottenuti riguardo all'area a alla potenza.

6. Conclusioni: nell'ultimo capitolo vengono riassunti e confrontati tutti i risultati acquisiti nei capitoli precedenti. Vengono mostrate le implementazioni singole che permettono le prestazioni migliori e quelle parallele che permettono di realizzare un dispositivo che occupi un'area il più possibile ridotta.

1.2 SerDes

All'interno dei data center le tecnologie di trasmissione più utilizzate sono la PCI Express e quella Ethernet ed entrambe si basano sulla comunicazione seriale [2]. Queste tecnologie utilizzano diversi segnali seriali in uscita dal chip per collegarsi alla rete. In particolare, la tecnologia Ethernet richiede nelle interfacce con il silicio dei SerDes, che servono a garantire che la trasmissione venga effettuata con una qualità che permetta di soddisfare i requisiti del sistema.
I SerDes sono sistemi di trasmissione ad alta velocità che mandano il segnale da un chip ad un altro, e nel farlo convertono il segnale da parallelo a seriale e poi di nuovo a parallelo. Questi dispositivi sono composti da due componenti distinti: un trasmettitore e un ricevitore, anche se solitamente sono usati in coppie in modo da fornire una comunicazione bidirezionale.
Una ragione per utilizzare questi dispositivi è che le interfacce dei chip stanno diventando sempre più complesse, a causa dell'aumento del numero di pin necessari a comunicare con l'esterno. Passare ad un'interfaccia seriale aumenta la complessità dei chip, su cui è necessario implementare i SerDes, ma diminuisce notevolmente il numero di pin necessari. Una seconda motivazione è la crescita dei data center; questi, infatti, richiedono un networking in varie forme: dal server al router top of the rack, da rack a rack, e comunicazioni a lungo raggio (long-haul) tra i data center sparsi nel mondo. Tutti questi tipi di interconnessioni diventano il limite nella crescita di queste strutture, infatti se non serializzate diventano impossibili da gestire e mantenere, sia per l'aumento della complessità sia per quello dei costi.
Per aumentare le performance dei SerDes, negli ultimi anni si è iniziata ad usare una codifica PAM4 (Pulse Amplitude Modulation), che, come verrà mostrato in seguito, permette di trasmettere due bit alla volta, pagando però con un aumento della sensibilità ai disturbi, che deve essere gestita per evitare un crollo delle prestazioni.
Inoltre, a seconda del tipo di collegamento, gli obiettivi nel progetto dei SerDes possono essere diversi, ad esempio nelle comunicazioni a corta distanza, dove i disturbi sono più ridotti, non è necessario lavorare per ottenere equalizzatori molto avanzati, ma è possibile concentrarsi sul risparmio di area e potenza. Invece, nel caso in cui siano utilizzati per le comunicazioni a lunga distanza, in cui i disturbi possono essere più importanti, è necessario concentrarsi sulle performance e sulla

riduzione del bit error rate, utilizzando componenti come il Viterbi Detector, lasciando in secondo piano i consumi di area e potenza, anche se questi rimangono comunque fondamentali.

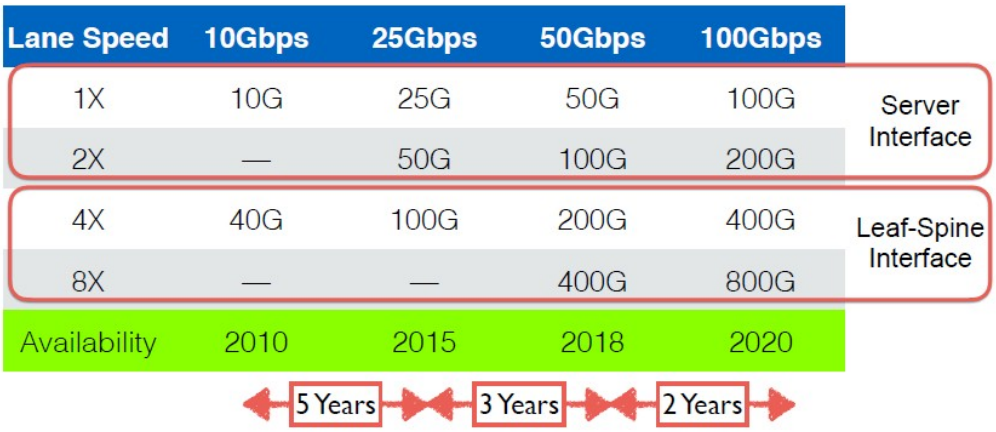

Figura 1.1: Andamento negli anni delle prestazioni richieste ai SerDes [3]

Come mostrato nella Figura 1.1, le prestazioni dei SerDes stanno aumentando esponenzialmente negli anni. Aumentando la velocità della singola linea è possibile raggiungere, con un numero minore di collegamenti, le prestazioni che negli anni precedenti necessitavano di diverse linee per essere raggiunte. Oppure è possibile mantenere costante il numero di linee, ottenendo performance nettamente superiori.

1.3 Codifica del segnale

Nel caso della trasmissione digitale, l'asse dei tempi si divide in intervalli di durata T_S. In ogni intervallo di tempo viene trasmessa una forma d'onda $f(t)$, presa da un insieme M, detta simbolo. A ciascun simbolo viene associata una sequenza di n_{bit} che la identifica, chiamata parola. L'insieme di tutti i simboli che possono essere trasmessi compongono la costellazione. Solitamente il numero di simboli che compone una costellazione è pari a $M = 2^{n_{bit}}$, quindi è un multiplo di 2.
Alcuni dei parametri più importanti che caratterizzano la trasmissione digitale sono:

- Baud Rate (D): numero di simboli trasmessi in un'unità di tempo. $D = \frac{1}{T_s}$
- Bit Rate (Br): numero di bit trasmessi in un'unità di tempo. $Br = \frac{n_{bit}}{T_s}$

Dalle definizioni precedenti può essere ottenuto anche un nuovo parametro, che rappresenta il tempo in cui viene trasmesso un singolo bit, anche se questo potrebbe non avere sempre un significato fisico: $T_b = \frac{T_s}{n_{bit}}$.
Spesso, per semplicità, si utilizza sempre la stessa forma d'onda durante la trasmissione, variando solamente un parametro, come ad esempio l'ampiezza. In questo caso il segnale trasmesso sarà:

$$x(t) = \sum_{n=-\infty}^{+\infty} a_n f(t - nT_S) \tag{1.1}$$

Il caso di trasmissione più semplice è quello in cui si utilizza un sistema NRZ, binario, antipodale, usando una forma d'onda $f(t)$ rettangolare.
Analizzando nel dettaglio i termini utilizzati:

- NRZ (No Return to Zero) significa che il simbolo occupa tutto T_s, senza tornare mai al valore zero.
- Binario indica che la costellazione è composta solamente da due simboli: $M = 2$. Di conseguenza $T_S = T_b$.
- Antipodale significa che $a_n = \pm 1$, per cui i simboli trasmessi potranno assumere solamente i valori $f(t)$ e $-f(t)$.

Questo tipo di modulazione è anche detto PAM2, in quanto il segnale viene modulato variando la sua ampiezza. Il due indica che la costellazione è composta da due simboli (Figura 1.2).

Un altro tipo di modulazione molto utilizzato è quello PAM4, in cui la trasmissione non è più binaria ma multilivello (Figura 1.3). Questo permette di trasmettere più bit per ogni simbolo, nello specifico $n_{bit} = 2$. In questo caso i livelli di ampiezza del segnale saranno quattro:

$$a_n = [-3, -1, +1, +3] \tag{1.2}$$

o nel caso in cui siano normalizzati:

$$a_n = [-1, -\frac{1}{3}, +\frac{1}{3}, +1] \tag{1.3}$$

Tutti i segnali descritti fino a questo momento sono trasmessi in banda base, questo significa che lo spettro del segnale trasmesso è centrato in zero.
Nei sistemi a banda traslata, in cui lo spettro non è centrato in zero ma ad una frequenza f_c, è possibile utilizzare altri tipi di modulazione, in cui i parametri che vengono modificati possono essere la fase o la frequenza della forma d'onda.

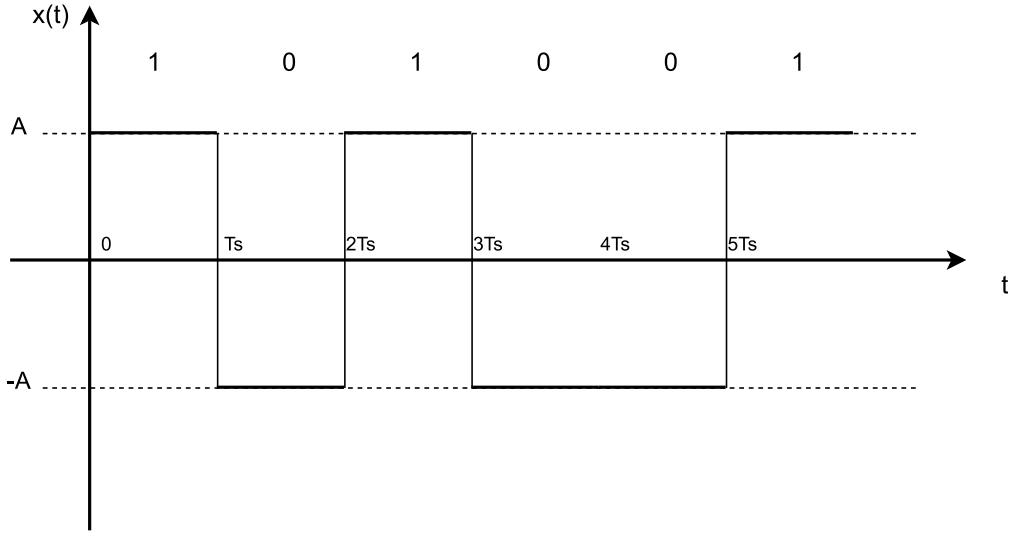

Figura 1.2: Esempio di modulazione PAM2 NRZ

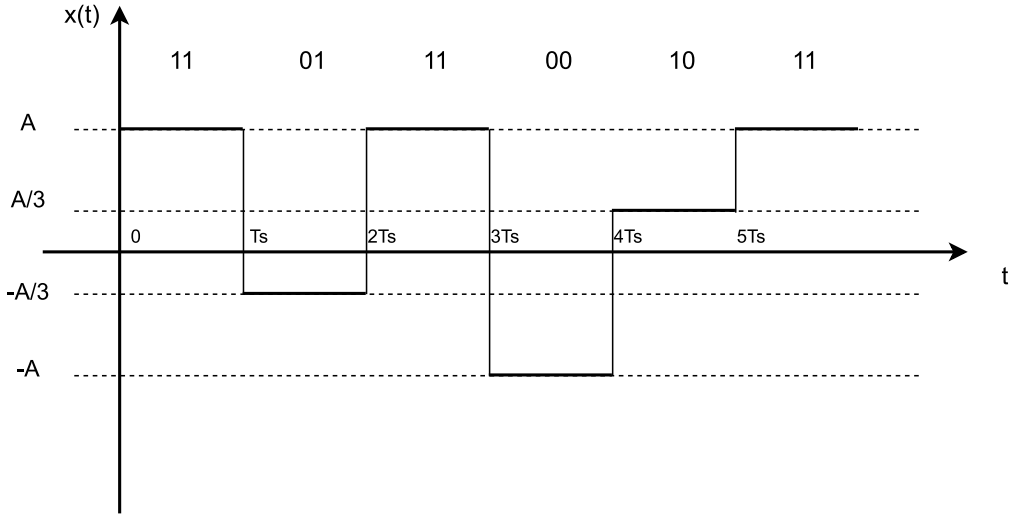

Figura 1.3: Esempio di modulazione PAM4

Un esempio di modulazione in banda traslata è la QAM (Quadrature and Amplitude Modulation), in cui vengono variate sia la fase che l'ampiezza del segnale, permettendo di raggiungere costellazioni composte da un elevato numero di simboli.

1.4 Serial link

Esistono due tipi di trasmissione: quella seriale, in cui ad ogni colpo di clock viene trasmesso un unico segnale, attraverso un singolo collegamento, e quella parallela, in cui il numero di bit trasmessi dipende dal numero di link presenti.
Questo rende la trasmissione parallela più veloce, ma essendo necessario un numero elevato di links anche il costo aumenta notevolmente, soprattutto per le comunicazioni a lunga distanza. Inoltre, la trasmissione parallela comporta un aumento dei disturbi tra i collegamenti vicini, che si influenzano gli uni con gli altri, a causa della presenza del cross-talk. Questo disturbo è causato dal fatto che, trasmettendo un segnale, questo causi una variazione indesiderata anche nei collegamenti vicini, a causa degli accoppiamenti capacitivi ed induttivi presenti tra i fili. Esistono tecniche per ridurre l'effetto del cross-talk, sia nella realizzazione dei collegamenti, ad esempio utilizzando degli schermi per ridurre l'entità degli effetti parassiti, ma il cui utilizzo è molto costoso, sia nell'elaborazione dei segnali ricevuti. Per questi motivi in molti casi viene preferita la comunicazione seriale, nonostante questa offra prestazioni più limitate, ed anche in questo caso sia necessario inserire componenti per l'elaborazione dei segnali, che devono essere convertiti in seriale all'interno del trasmettitore e riconvertiti in parallelo all'interno del ricevitore.
Concentrandosi su quest'ultima tecnica utilizzata per la comunicazione, il sistema è tipicamente composto da tre componenti principali: il trasmettitore, il canale e il ricevitore.

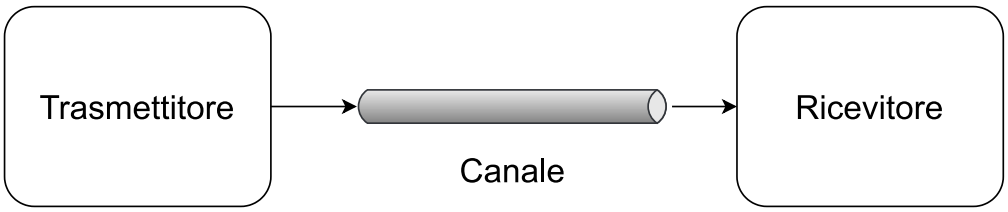

Figura 1.4: Schema di un canale

Il trasmettitore ha il compito di ricevere i dati che devono essere trasmessi, questi ad esempio possono arrivare da un server che deve inviarli ad un altro componente. Questi dati vengono ricevuti con un parallelismo tipicamente elevato, quindi devono essere serializzati prima di poterli trasmettere al ricevitore tramite il canale. Questo componente è tipicamente formato da un serializzatore, un phase locked loop (PLL) e un driver, come mostrato in Figura 1.5.

Il serializzatore serve a convertire il segnale da parallelo a seriale, inoltre il segnale viene elaborato tramite componenti analogici e digitali per migliorare la

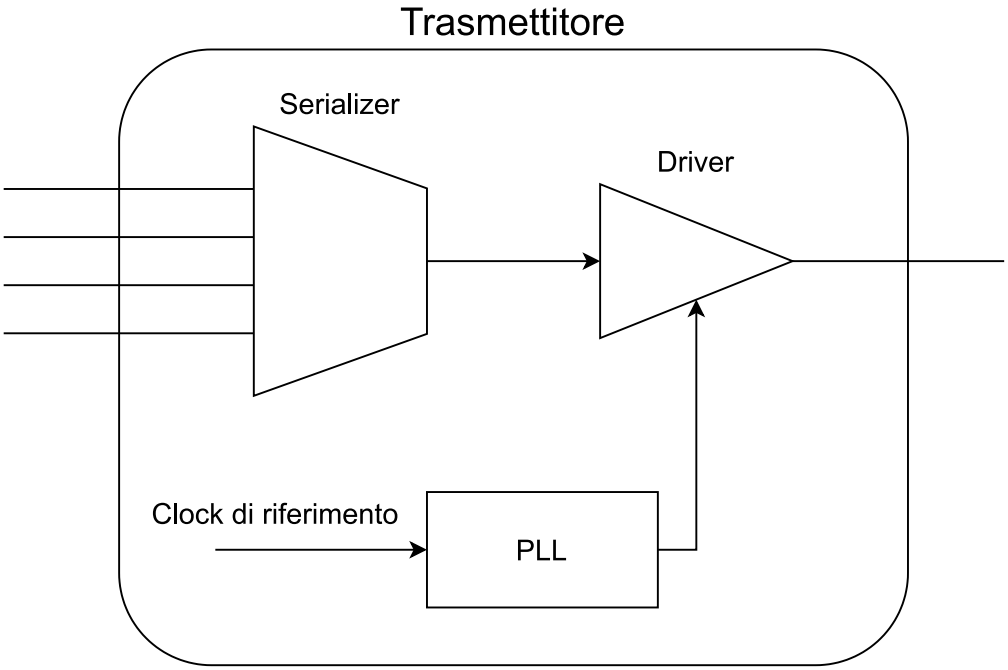

Figura 1.5: Schema del trasmettitore

sua resistenza ai disturbi; il PLL riceve il clock dal sistema e ne crea uno nuovo ad una frequenza multipla di quello originale, per poter raggiungere la velocità necessaria per la trasmissione seriale, che dovendo utilizzare un unico collegamento deve trasmettere i dati ad una frequenza molto superiore a quella a cui li riceve. Infine, è presente un driver per fornire la potenza necessaria al segnale che deve essere trasmesso, infatti, soprattutto nel caso di comunicazioni a lunga distanza, questa potrebbe aumentare, dato che maggiore è la potenza del segnale maggiore è la probabilità che venga ricevuto correttamente. Come verrà analizzato in seguito, uno dei fattori più importanti in questo caso è infatti il rapporto segnale-rumore (SNR: Signal to Noise Ratio) che indica il valore in decibel del rapporto tra la potenza del segnale e quella dei disturbi.
L'utilizzo dei decibel permette di avere una rappresentazione logaritmica di questa grandezza, ottenuta come:

$$SNR_{dB} = 10 \cdot \log(\frac{P_{segnale}}{P_{rumore}}) \tag{1.4}$$

Il ricevitore ha il compito opposto rispetto al trasmettitore, deve infatti ricevere i dati in formato seriale dopo che hanno percorso il canale, venendo alterati dal rumore e dai disturbi, e riportarli in formato parallelo. I suoi componenti principali sono l'equalizzatore, il deserializzatore e il Clock Data Recovery (CDR), mostrati

in Figura 1.6.

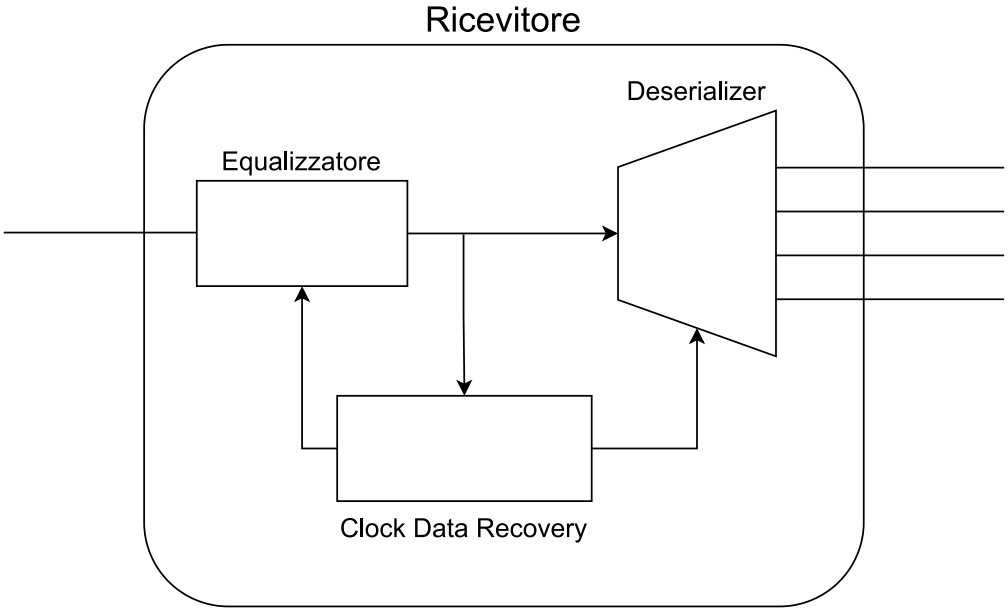

Figura 1.6: Schema del ricevitore

Il compito dell'equalizzatore è quello di compensare gli effetti introdotti dai disturbi del canale, rendendo più facile la sua ricezione e diminuendo la probabilità che vengano commessi errori. Questa operazione è effettuata da una parte analogica, dove sono presenti i primi filtri, ed una parte digitale. Il segnale deve essere quindi convertito e reso nuovamente parallelo, questo passaggio è effettuato dal deserializzatore, che spesso non è l'ultimo componente del ricevitore dato che le sue uscite possono venire ulteriormente elaborate da componenti digitali. Nel caso in cui per mantenere limitata la complessità della trasmissione il clock non venisse trasmesso, questo deve essere ricostruito lavorando solamente sul segnale ricevuto. Questa è un'operazione molto complessa ma fondamentale, che viene svolta dal clock data recovery, un componente analogico che permette la ricostruzione del clock e la sincronizzazione del segnale d'ingresso.
Il canale può essere di diversi tipi, che variano notevolmente a seconda della distanza a cui deve essere trasmesso il segnale e alla banda necessaria. Si può infatti passare dalla fibra ottica, principalmente usata per comunicazioni a lunga distanza, alle piste di una PCB, utilizzate per trasmissioni molto più brevi su un'unica scheda. Ovviamente modificando il tipo di canale e la sua lunghezza cambia completamente la natura e l'entità dei disturbi e delle distorsioni, che se non corretti possono causare degli errori duranti la ricezione. Normalmente i principali disturbi introdotti dal canale sono il rumore e l'interferenza intersimbolica.

Il moto termico degli elettroni genera una tensione di rumore, che non sarà mai nulla tranne che allo zero assoluto. Questa è la causa del rumore gaussiano bianco, che non dipende dalla frequenza e per questo avrà uno spettro piatto. In genere il rumore termico ha statistica gaussiana, con valore atteso nullo, ed è quindi possibile ottenere la sua distribuzione di probabilità:

$$f_V(V) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{V^2}{2\sigma^2}} \tag{1.5}$$

La deviazione standard σ influisce sulla spanciatura della distribuzione, più la deviazione standard diminuisce, più la distribuzione statistica si restringe. La deviazione standard è definita dal valore efficace della tensione, che dipende direttamente dalla temperatura. La potenza di questo rumore, nonostante sia teoricamente infinita essendo lo spettro costante in frequenza, viene definita dalla banda del canale, che è limitata. Per quanto questo termine sia un vantaggio nel limitare la potenza del rumore, in altri casi può essere uno svantaggio. Questa limitazione fa sì che il canale si comporti come un filtro passa basso, allargando la forma dei simboli trasmessi e causando la loro sovrapposizione con quelli adiacenti, questo effetto è chiamato interferenza intersimbolica (ISI: intersymbol interference).
Per ottenere la forma che assumerà un simbolo in uscita è necessario conoscere il comportamento in frequenza del canale. Conoscendo la risposta all'impulso $h(t)$ è possibile effettuare la convoluzione con il simbolo trasmesso $x(t)$. Il risultato che si ottiene corrisponde al simbolo in uscita dal canale $y(t)$.

$$y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(t-\tau) \cdot h(\tau) d\tau \tag{1.6}$$

Calcolando l'ampiezza ad intervalli discreti di larghezza pari all'unità di tempo di riferimento del simbolo in uscita si ottengono i cursori. Il cursore corrispondente all'ampiezza massima del segnale è chiamato c_0, i cursori precedenti a questo riferimento sono detti "precursors" (c_n con $n < 0$) e quelli successivi "postcursors" (c_n con $n > 0$) [4].
In ultimo, la resistenza associata al canale causa un'attenuazione del segnale, facendo sì che il segnale in uscita abbia un'ampiezza ridotta rispetto a quello originale, rendendo più complicata la sua analisi, a causa della maggiore precisione richiesta. Questo effetto è tanto più importante quanto più è lungo il canale, rendendolo uno dei principali problemi per la comunicazione a lunga distanza.
Conoscendo la funzione di trasferimento del canale, idealmente sarebbe possibile realizzare un equalizzatore che abbia una funzione di trasferimento che corrisponde alla sua inversa:

$$H_{EQ}(s) = H_C^{-1}(s) \tag{1.7}$$

Dove H_{EQ} è la funzione di trasferimento dell'equalizzatore e H_C quella del canale.
Utilizzando questo equalizzatore ideale, la sua uscita corrisponderebbe esattamente al segnale trasmesso, annullando completamente gli effetti del canale.
Gli equalizzatori si dividono tra quelli puramente analogici, quelli digitali e quelli misti, quindi con una componente analogica e una digitale. Quelli analogici hanno il vantaggio di non avere bisogno del clock, ma possono essere utilizzati solo per canali con una funzione di trasferimento molto regolare.
L'equalizzazione può essere effettuata sia nel trasmettitore sia nel ricevitore: nel primo caso si ha il vantaggio che, nel caso in cui sia sufficiente, è più semplice di quella al ricevitore; nel secondo caso una delle possibilità è quella di utilizzare un Continuous Time Linear Equalizer (CTLE), un equalizzatore analogico che fornisce un guadagno ad alta frequenza, contrastando le perdite e le distorsioni del canale. I vantaggi principali derivanti dall'utilizzo di un CTLE sono che consuma poca potenza e poca area. Nel caso in cui si realizzi un equalizzatore misto, in cui è presente anche una parte digitale, può essere utilizzato anche un filtro FIR (Finite Impulse Response) digitale, per contrastare allo stesso modo gli effetti negativi del canale. Questo filtro inoltre permette di rimuovere gli effetti dei precursori dovuti all'interferenza intersimbolica e di implementare semplici algoritmi adattativi.
Nel caso in cui le distorsioni siano limitate gli equalizzatori lineari sono i migliori, perché sono semplici da implementare e occupano poca area. Nel caso in cui gli effetti dei disturbi siano però più accentuati, per raggiungere le prestazioni richieste può essere necessario utilizzare equalizzatori digitali non lineari, che permettono di migliorare le performance senza aumentare il rumore. Infatti, gli equalizzatori lineari hanno bisogno di amplificare il segnale ricevuto alle frequenze a cui questo viene attenuato dal canale, ma così facendo viene amplificato anche il rumore. Per questo motivo il rumore finale non sarà più un rumore gaussiano bianco, perché avrà valori differenti in base alla frequenza, e questo rende più complicata la realizzazione di un "optimum detector".

1.5 Decision Feedback Equalizer

Il Decision Feedback Equalizer (DFE) è un ricevitore sub-ottimo non lineare che fornisce vantaggi a livello di performance rispetto agli equalizzatori lineari, senza la complessità delle soluzioni ottime [5].

Il funzionamento di questo equalizzatore si basa sul tentativo di ridurre l'effetto dell'interferenza intersimbolica, prevedendo l'effetto che i simboli trasmessi in precedenza avrebbero su quello attuale e annullandolo.

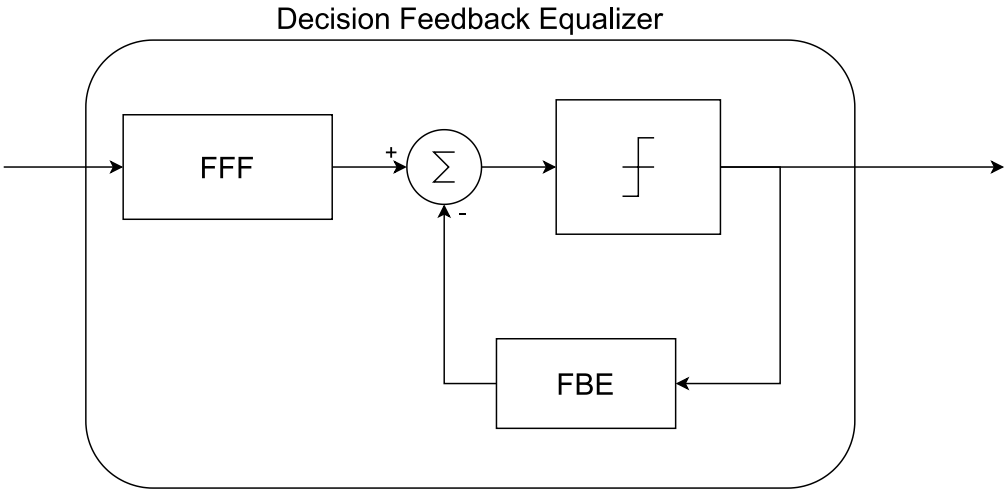

Figura 1.7: Schema del DFE

La struttura classica del DFE è composta principalmente da un filtro lineare (feed-forward filter, FFF), la cui uscita è quindi la somma di una combinazione lineare di N simboli trasmessi e del rumore. Il compito di questo filtro è quello di ridurre l'effetto dei precursors. Nel momento in cui deve essere presa una decisione sul k-esimo simbolo trasmesso il filtro in retroazione (feedback filter, FBF) fornisce una combinazione lineare pesata delle precedenti decisioni sui simboli, assumendo che siano corrette, e cancella l'effetto dell'interferenza intersimbolica prodotta da questi, in questo caso l'interferenza è generata dai postcursors. In questo modo viene ridotto l'effetto sia dei simboli trasmessi in precedenza sia di quelli futuri, il risultato è quindi fornito ad un comparatore di soglia per determinare la decisione sul simbolo corrente. Il problema principale di questo equalizzatore è dovuto alla propagazione degli errori: nel caso in cui una decisione presa sia sbagliata a causa della presenza del rumore, questa influenzerà negativamente anche le decisioni successive, aumentando significativamente la probabilità di commettere altri errori che si propagheranno potenzialmente fino a quando non verranno prese nuovamente decisioni corrette.

1.6 Ricevitore a massima verosimiglianza

I ricevitori a massima verosimiglianza permettono di ottenere, dal vettore di segnali ottenuto in ricezione $\boldsymbol{y}$, la sequenza originale trasmessa $\boldsymbol{x}$. L'obiettivo di questo tipo di ricevitore è massimizzare la probabilità che le decisioni prese basandosi su $\boldsymbol{y}$ siano corrette, per farlo è necessario considerare una regola basata sulla *probabilità*

a posteriori: $P(\boldsymbol{x}|\boldsymbol{y})$.
Questo criterio di decisione è detto *massima probabilità a posteriori* (MAP: maximum a posteriori probability) ed è stato dimostrato che minimizza la probabilità di errore.
Ipotizzando che, nel caso in cui siano presenti M simboli, ognuno abbia la stessa probabilità $p = \frac{1}{M}$ di essere stato trasmesso, trovare il massimo della probabilità equivale a trovare la sequenza trasmessa che per ogni segnale ricevuto minimizza la distanza euclidea [6]:

$$D(\boldsymbol{y}, \boldsymbol{x}) = \sum_{k=1}^{N} (y_k - x_k)^2 \tag{1.8}$$

dove N è la lunghezza della sequenza.
Una possibilità per realizzare questo tipo di ricevitore consiste nell'utilizzare un Viterbi Detector.

Capitolo 2

L'algoritmo Viterbi

2.1 Descrizione teorica dell'algoritmo

Un processo markoviano è un modello stocastico che descrive una sequenza di possibili eventi, in cui la probabilità di ognuno dipende solo dallo stato in cui si trovava l'evento precedente. Una sequenza di stati infinita, in cui il processo avanza in stati discreti, dà luogo a un processo markoviano a tempo discreto.
L'algoritmo Viterbi è una soluzione ricorsiva ottima al problema di stimare la sequenza di stati finiti di un processo markoviano a tempo discreto, fornendo a posteriori la stima della sequenza di stati più probabile, chiamata Viterbi path, che risulta in una sequenza di eventi osservati.
L'algoritmo è stato presentato per la prima volta nel 1967 da Andrew Viterbi [7], come un metodo per decodificare codici convoluzionali e da quel momento è stato riconosciuto come una soluzione valida per un'ampia varietà di problemi nell'ambito digitale.
Nel 1973 è stata pubblicata da Forney una descrizione dell'algoritmo, dove per la prima volta vengono presentate nel dettaglio le sue applicazioni pratiche [8].
Il processo Markoviano è caratterizzato nel modo seguente:

- È a tempo discreto;
- Lo stato x_k al tempo k fa parte di un numero finito M di stati $m : 1 \leq m \leq M$;

Assumendo inizialmente che il processo vada solo dal tempo 0 al tempo K e che gli stati iniziali e finali x_0 e x_K siano noti, la sequenza di stati è rappresentata da un vettore

$$\boldsymbol{x} = (x_0, \cdots, x_K) \tag{2.1}$$

Essendo il processo markoviano, la probabilità di essere nello stato x_{k+1} al tempo $k+1$, dati tutti gli stati precedenti, dipende solo dallo stato x_k all'istante k:

$$P(x_{k+1}|x_0, x_1, \cdots, x_k) = P(x_{k+1}|x_k) \tag{2.2}$$

La transizione tra una coppia di stati (x_{k+1}, x_k) può essere definita come:

$$\xi_k \triangleq (x_{k+1}, x_k) \tag{2.3}$$

Ed essendo Ξ il set di transizioni $\xi_k = (x_{k+1}, x_k)$ per cui la probabilità $P(x_{k+1} \mid x_k) \neq 0$, e $|\Xi|$ il loro numero. Chiaramente $|\Xi| \leq M^2$.
La probabilità di transizione tra due stati, anche se per semplicità non viene mostrato nella notazione, può essere tempo variante.
Assumendo che il processo sia osservato con rumore gaussiano bianco, in una sequenza $\boldsymbol{z}$ di osservazioni, z_k dipende dal punto di vista probabilistico solo dalla transizione ξ_k al tempo k:

$$P(\boldsymbol{z}|\boldsymbol{x}) = P(\boldsymbol{z}|\boldsymbol{\xi}) = \prod_{k=0}^{K-1} P(z_k|\xi_k) \tag{2.4}$$

Quindi, $\boldsymbol{z}$ può essere descritta come l'output di un canale senza memoria che ha come input la sequenza $\boldsymbol{\xi}$.
La Figura 2.1 mostra il diagramma degli stati di un processo markoviano, in cui

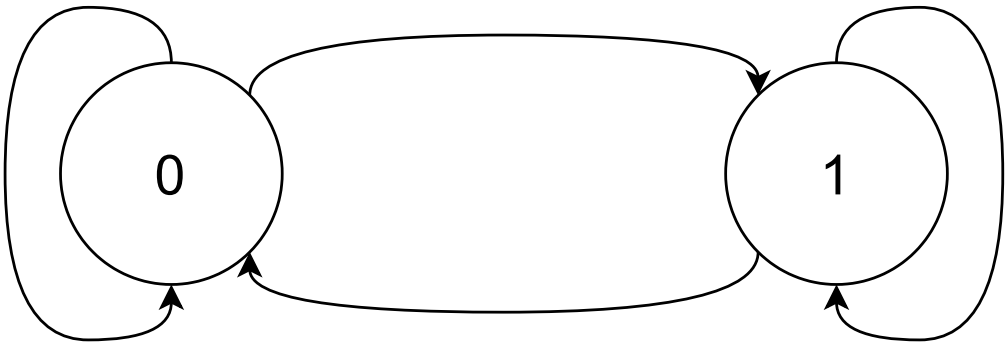

Figura 2.1: diagramma a stati di un processo a due stati

gli stati possibili sono solo 0 ed 1.
I nodi del grafico rappresentano gli stati, mentre le frecce (branch) le transizioni.
Per mostrare più chiaramente la variazione degli stati, mostrando il loro andamento nel tempo, è possibile realizzare un altro grafico, detto trellis (Figura 2.2), dove ogni nodo rappresenta uno stato in un preciso istante temporale.
È infine possibile associare ad ogni transizione un peso $bm(\xi_k)$, che dipende dalla probabilità che ha la transizione di avvenire: se una transizione è molto probabile

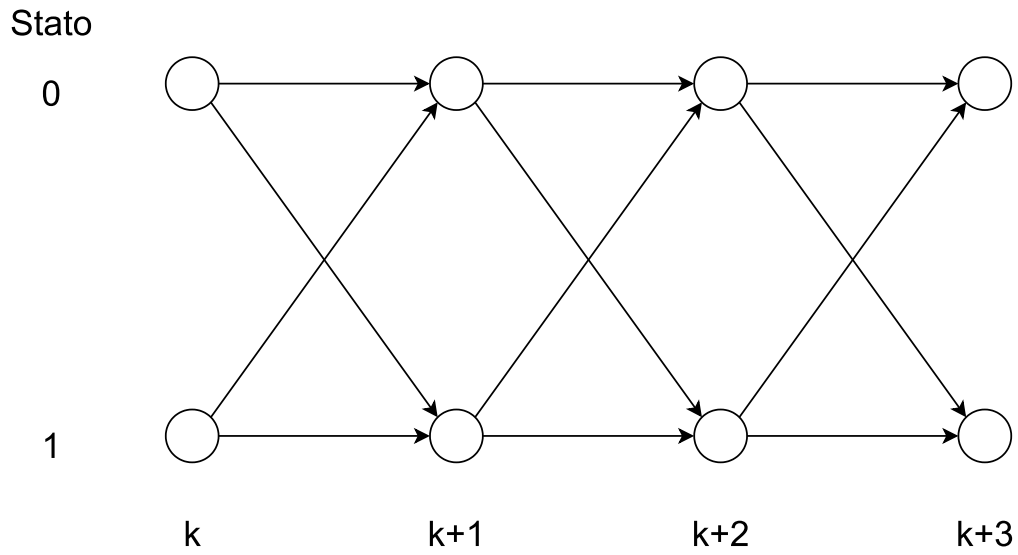

Figura 2.2: trellis di un processo a due stati

avrà un peso maggiore di una poco probabile. Per questo motivo il suo valore sarà più basso. Questi valori rappresentano le "branch metrics".
La somma di tutte le branch metrics corrispondenti ad una sequenza di stati $\boldsymbol{x}$ viene definita "path metric".
Lo scopo dell'algoritmo è trovare la sequenza di stati che abbia il peso minore tra tutte quelle possibili. Trovare il path più corto non è però un problema banale: nel 1957 è stata proposta una soluzione da Minty [9], ma il suo metodo non era facilmente adattabile in implementazioni hardware. Per questo motivo è stato necessario utilizzare altre tecniche, come quella mostrata in seguito.
Definendo x_0^k una sequenza di stati $(x_0, x_1, \cdots, x_k)$, è possibile rappresentare il path come:

$$pm(x_0^k) = \sum_{i=0}^{k-1} bm(\xi_i) \tag{2.5}$$

Il path più corto, cioè quello in cui la somma di tutte le branch metrics è minore, viene definito survivor path. Ad ogni istante di tempo ci sono M survivor paths, uno per ogni stato x_k, identificati come $\hat{\boldsymbol{x}}(x_k)$. Ripetendo l'operazione per un istante di tempo successivo, si otterranno M nuovi risultati. È però possibile osservare che la parte che identifica gli stati da 0 a k deve coincidere obbligatoriamente con uno dei paths trovati in precedenza, altrimenti significherebbe che il path da 0 a k identificato in questo istante ha un valore di path metric minore di quello precedente, avendo quindi una contraddizione.
Per questo motivo ad ogni istante k è sufficiente memorizzare gli M survivor paths $\hat{\boldsymbol{x}}(x_k)$ e i loro pesi $pm(x_k) \triangleq pm[\hat{\boldsymbol{x}}(x_k)]$. Per arrivare al tempo $k+1$, è necessario

estendere tutti i survivors di un'unità di tempo, calcolare la lunghezza dei nuovi paths e per ogni nodo x_{k+1} selezionare il path con peso minore, che diventerà il nuovo survivor.
È quindi possibile procedere in modo ricorsivo, senza che il numero di survivors superi mai M.
In seguito, viene proposto l'algoritmo in modo formale:

Algoritmo Viterbi:

Variabili:

k		indice temporale
$\hat{\boldsymbol{x}}(x_k)$,	$1 \leq x_k \leq M$	survivor che termina in x_k
$pm(x_k)$,	$1 \leq x_k \leq M$	path metric del survivor

Inizializzazione:

$k = 0$;

$\hat{\boldsymbol{x}}(x_0) = x_0$;	$\hat{\boldsymbol{x}}(x_m)$ arbitrario,	$m \neq x_0$;
$pm(x_0) = 0$;	$pm(m) = \infty$,	$m \neq x_0$.

Ricorsione:

$$pm(x_{k+1}, x_k) \triangleq pm(x_k) + bm[\xi_k = (x_{k+1}, x_k)] \quad \forall \xi_k = (x_{k+1}, x_k) \tag{2.6}$$

Trovare:

$$pm(x_{k+1}) = \min_{x_k}(pm(x_{k+1}, x_k)) \quad \forall x_{k+1}; \tag{2.7}$$

Memorizzare: (x_{k+1}) e il survivor corrispondente $\hat{\boldsymbol{x}}(x_{k+1})$.

$k = k + 1$;

Ripetere: finché $k = K$.

All'istante di tempo K il percorso con peso minore terminante in x_K sarà memorizzato nel survivor $\hat{\boldsymbol{x}}(x_K)$.
In pratica però sono necessarie alcune modifiche per rendere implementabile l'algoritmo. Se la sequenza di stati è molto lunga, per memorizzare i survivor paths servirebbe una memoria troppo elevata, è quindi necessario troncare i survivors ad una lunghezza gestibile δ, che rappresenta la lunghezza della path history, cioè il

numero di decisioni precedenti attualmente memorizzate dall'algoritmo. L'algoritmo quindi al tempo k deve avere una decisione definitiva per tutti i nodi fino al tempo $k - \delta$.
Più la lunghezza della path history è elevata, maggiore è la probabilità che tutti i survivors attraversino lo stesso nodo al tempo $k - \delta$. Nel caso in cui all'istante $k - \delta$ la decisione non sia univoca è possibile procedere in vari modi, ad esempio scegliendo un nodo in modo arbitrario o scegliendo quello che corrisponde alla path metric più bassa [10] [11], questa tecnica è detta best selection.

2.2 Applicazione pratica

Nel caso in cui si abbia una trasmissione digitale attraverso un canale analogico, il segnale ricevuto può essere descritto come:

$$z_k = y_k + n_k \tag{2.8}$$

dove y_k è una funzione deterministica ottenuta da un numero finito di ingressi, $y_k = f(x_k, \cdots, x_{k-\nu})$, e da un rumore gaussiano bianco, n_k. Nel caso studiato, in cui è l'interferenza intersimbolica a causare la variazione del segnale ricevuto rispetto a quello trasmesso, conoscendo i coefficienti associati al canale è possibile prevedere il valore di y_k conoscendo il valore degli ingressi x_k:

$$y_k = x_k \cdot h_0 + x_{k-1} \cdot h_1 + \cdots + x_{k-\nu} \cdot h_\nu \tag{2.9}$$

In generale quindi all'interno del ricevitore possono esserci m^ν possibili stati, dove m rappresenta il numero dei segnali possibili ad ogni istante di tempo, che nel caso del canale viene definito dalla modulazione (2 nel caso PAM2, 4 nel caso PAM4):

$$s_k \triangleq (x_{k-1}, x_{k-2}, \cdots, x_{k-\nu}) \tag{2.10}$$

dove per convenzione $x_k = 0$ per $k < 0$.
La mappatura tra la sequenza di ingresso x, la sequenza degli stati s e la sequenza dei segnali ricevuti y è uno a uno, quindi è invertibile.
Utilizzando questo parallelismo tra la sequenza di stati e la sequenza dei segnali è possibile utilizzare l'algoritmo Viterbi all'interno del ricevitore per rimuovere gli effetti dovuti all'interferenza intersimbolica [8][12][13].
Nell'applicazione dell'algoritmo un problema è dovuto al fatto che i valori delle path metrics ad ogni iterazione vengono incrementati, ma per procedere con l'implementazione tutti i valori devono avere dei limiti massimi definiti, perché deve essere stabilito il numero di bit necessari per rappresentarli correttamente, senza che si verifichi overflow [14]. In seguito, verranno mostrate delle soluzioni, ma per farlo è necessario prima dimostrare due proprietà dell'algoritmo:

1. Le uscite dell'algoritmo Viterbi dipendono solamente dalla differenza delle metriche;

2. La differenza tra le metriche è limitata.

La prima proprietà si dimostra sapendo che la selezione dei survivor path, ottenuta dall'Equazione 2.6 e dall'Equazione 2.7, coinvolge solo confronti fra metriche $pm(x_{k-1}) + bm(x_{k-1}, x_k)$ e quindi dipende solo dalle differenze della metrica.
Dimostrazione seconda proprietà:
Chiamando $bm_{st}(k)$ la branch metric che indica il passaggio dallo stato s allo stato t al tempo k, ed assumendo che B sia il suo limite superiore:

$$|bm_{st}(k)| \leq B, (s,t) \in T, k \in N \tag{2.11}$$

Dove T rappresenta la matrice di transizione del processo markoviano.
Sia S un insieme finito, da qui in poi, T viene considerata in modo interscambiabile sia come un sottoinsieme di $S \times S$ sia come una matrice di $0-1$ su $S \times S$. Una condizione sufficiente perché la proprietà sia valida è che T, quando elevata ad una potenza finita n, abbia tutti i suoi componenti strettamente positivi. In questo caso T sarà definita irriducibile e aperiodica.
Nel caso studiato, dato che l'applicazione nel caso del canale coincide con quella di un decoder convoluzionale, il valore di n coincide con l'ordine della memoria del codice, come mostrato in [15].
Questo valore è gia stato utilizzato in precedenza, in particolare nell'Equazione 2.9, e coincide con il valore di ν, cioè 1 nel nostro caso.

Dimostrazione:
Siano t_1 e t_2 stati arbitrari al tempo k. Inoltre, viene definita $\boldsymbol{x}_1$ l'abbreviazione per $\hat{\boldsymbol{x}}(t_1)$ al tempo k il path sopravvissuto di t_1. Senza perdere generalità, $pm_{t_1}(k) \leq pm_{t_2}(k)$. Nel caso in cui k sia minore di n, q_2 è uguale al survivor path di t_2 al tempo k. Altrimenti esiste uno stato s al tempo $(k-n)$ sul survivor path di t_1. Il segmento di p_1 che inizia al tempo 0 e finisce al tempo $(k-n)$ è detto q. Dalle assunzioni su T, esiste un'estensione di q, tra le branch metrics da t, ad un path che finisce in t_2 al tempo k. Se $k \geq n$, questo path sarà q_2. In entrambi i casi il path q_2 termina in t_2, quindi ha una metrica che non è inferiore a $pm_{t_2}(k)$. La differenza $pm_{t_2}(k) - pm_{t_1}(k)$ è quindi limitata superiormente dalla metrica di q_2 meno la metrica del survivor path di t_1. In un'espansione delle metriche di q_2 e p_1 come somma di branch metrics, tutti tranne le ultime $\min\{k, n\}$ coppie di termini cancellano le loro differenze, in modo tale che $m = 2n$.

$$|pm_{t_1}(k) - pm_{t_2}(k)| \leq mB, t_1 \in S, t_2 \in S, k \in N \tag{2.12}$$

Dalla proprietà 1, la sottrazione di una costante dalle path metrics $pm_s(k)|s \in S$, non modifica l'output dell'algoritmo. Questo permette di mantenere il valore delle path metrics sempre sotto ad un valore massimo, e quindi di sapere a priori il numero di bit necessario a rappresentarle senza errori.
Questa tecnica è detta rescaling, è molto semplice da implementare, ma ha lo svantaggio di richiedere dell'hardware aggiuntivo, per eseguire le sottrazioni.
La soluzione migliore è invece quella di usare l'approccio dell'aritmetica in complemento a 2, in cui invece di cercare un modo per evitare l'overflow, questo viene gestito evitando che influisca sulla correttezza del risultato. L'aritmetica in complemento a 2, su c bit, si riferisce al gruppo:

$$F_c \coloneqq \{-2^{c-1}, 1 - 2^{c-1}, \cdots, 2^{c-1} - 1\} \tag{2.13}$$

Dove l'addizione è definita in modulo 2^c. L'operatore del modulo che riduce un numero ad un elemento dell'intervallo F_c è definito "mod 2^c". La modifica dell'algoritmo con lo scopo di adottare un'implementazione basata sull'aritmetica in complemento a 2 consiste nel sostituire le metriche $pm_s(k)$ con i loro residui $pm_s(k) \mod s^2, s \in S, k = 0,1,\cdots$.
Come dimostrato dalla proprietà 1, il calcolo del minimo nell'Equazione 2.7 è fatto confrontando gli elementi. Sono i segni delle differenze

$$pm_s(k-1) + bm_{st}(k) - (pm_{s'}(k-1) + bm_{s't}(k)), s \in S, s' \in S, t \in S, k \in N \tag{2.14}$$

che contano. Sostituendo con le metriche ridotte in modulo-2^c porta alla valutazione delle differenze

$$pm_s(k-1) + bm_{st}(k) - (pm_{s'}(k-1) + bm_{s't}(k)) \mod 2^c, s \in S, s' \in S, t \in S, k \in N \tag{2.15}$$

Dalla seconda proprietà (2.14) non supera $(m+2)B$ ($(m+1)B$ se le branch metrics sono non negative). Quindi, se il range di valori è almeno $\{-(m+2)B, \cdots, (m+2)B\}$,

$$2^{c-1} - 1 \geq (m+2)B \tag{2.16}$$

e la differenza ridotta coincide con quella reale. Quindi la proprietà 1 implica la correttezza della modifica in complemento a 2.
Nel caso studiato i valori delle branch metrics sono positivi, quindi la 2.16 diventa:

$$2^{c-1} - 1 \geq (m+1)B \tag{2.17}$$

Il vantaggio di questa tecnica è che l'operazione di modulo corrisponde al meccanismo di overflow dell'aritmetica in complemento a 2, e quindi non ha nessun costo dal punto di vista dell'hardware.
Queste considerazioni sulla possibilità di troncare la lunghezza della path history e

di mantenere limitato il valore massimo delle path metrics permettono di passare ad un'implementazione hardware dell'algoritmo, nella quale ovviamente non potrebbero essere memorizzate sequenze infinite o valori troppo elevati.

Capitolo 3

Modello matlab

3.1 Descrizione generale del modello

La prima parte del lavoro di tesi, oltre allo studio teorico dell'algoritmo, è consistita nella realizzazione di un modello matlab che simula il comportamento di un sistema di trasmissione e ricezione dati in presenza di rumore gaussiano bianco e di distorsione dovuta al canale.
Questo ha permesso di verificare le prestazioni del sistema in presenza del Viterbi e di poterle confrontare con quelle in altre condizioni di funzionamento.
La parte iniziale del modello matlab, che rappresenta la trasmissione dei segnali, è comune a tutte le condizioni; infatti, sono presenti la generazione degli ingressi e la simulazione del canale.
Gli ingressi sono ottenuti da un generatore di numeri pseudocasuali, la quantità di valori possibili per ogni ingresso è decisa dall'utilizzatore del modello. Infatti, questo è pensato per simulare un canale sia nel caso di trasmissione PAM2 NRZ, in cui il segnale può assumere due valori, normalizzati a +1 e -1, sia nel caso di trasmissione PAM4, in cui il segnale può avere i valori normalizzati a -3, -1, +1, +3, permettendo di trasmettere due bit di informazione per ogni simbolo.
Nel primo caso i valori generati saranno 0 e 1, mentre nel secondo 0, 1, 2, 3.
Questi valori sono dovuti al generatore utilizzato, ma verranno processati dalla funzione `pammod` che restituisce il segnale modulato che verrà effettivamente trasmesso.
Si ottengono così i valori forniti dal trasmettitore, ancora assoluti per la mancanza di disturbi.
La parte successiva invece rappresenta la simulazione del canale: qui ai valori iniziali vengono aggiunti il rumore e l'effetto dell'interferenza intersimbolica.
Il rumore viene generato da una funzione matlab, e la potenza di questo può essere definita selezionando il valore del rapporto segnale rumore. Il SNR indica il valore

in decibel del rapporto tra la potenza del segnale e quella del rumore, permettendo quindi di poter eseguire l'analisi usando i rapporti tra le potenze e non il valore assoluto, mantenendola il più generale possibile.
Al rumore si aggiunge poi l'effetto dell'interferenza intersimbolica, che può essere simulato da un filtro con coefficienti h_0 e h_1, in cui per rispettare la fisicità del canale la somma dei coefficienti deve sempre dare come valore 1.

$$y(k) = h_0 \cdot x(k) + h_1 \cdot x(k-1) \tag{3.1}$$

Nel caso in cui la coppia abbia i valori $\{1, 0\}$, il valore di $y(k)$ sarà identico a quello di $x(k)$, avendo quindi una distorsione nulla, ma con l'aumentare del valore di h_1 (e quindi il diminuire di h_0), il peso che ha il simbolo precedente aumenta, avendo quindi un incremento dell'interferenza.
Come si vedrà in seguito, il Viterbi ha un'utilità maggiore nel caso in cui la distorsione è elevata (0.6, 0.4).
Il valore ottenuto rappresenta il valore del segnale in uscita dal canale, il valore quindi che arriva al ricevitore.
Mentre il trasmettitore ed il canale sono fissi, il modello permette di utilizzare diversi tipi di ricevitore, permettendo di ottenere informazioni sul guadagno che si ha utilizzando un sistema invece di un altro.
I ricevitori utilizzabili sono 3:

- Convertitore da PAM a binario;
- Decision feedback equalizer;
- Viterbi decoder.

Il primo è il più semplice, rappresenta infatti un ricevitore in cui il segnale non viene elaborato, ma viene solo deciso il valore del simbolo in base a delle soglie fissate a priori.
Il decision feedback equalizer, come visto in precedenza, permette di ridurre l'effetto della distorsione, infatti in base al valore del simbolo precedente, prevede l'effetto dell'ISI sul simbolo corrente e lo annulla.
Il Viterbi come il DFE riduce l'effetto dell'interferenza intersimbolica, ma in questo caso calcolando la sequenza di stati a massima verosimiglianza.
Mentre per simulare il DFE è stata utilizzata una funzione matlab, la funzione che simula il Viterbi è stata realizzata a parte.

3.2 Funzione matlab Viterbi

La funzione creata per simulare la presenza di un Viterbi Detector è molto simile sia nel caso in cui si utilizzi una codifica PAM2 sia nel caso in cui la codifica sia

PAM4. La differenza principale è l'aumento della complessità nel secondo caso, ma le operazioni da effettuare sono le stesse.
Il primo passaggio è il calcolo dei valori delle branch metrics, i cui valori sono ottenuti effettuando il quadrato della differenza tra il valore atteso dato dalla transizione a cui si riferisce, ed il valore dell'ingresso. Nel caso della transizione dallo stato j allo stato i:

$$bm(i,j) = (sym(i) \cdot h_0 + sym(j) \cdot h_1 - x(k))^2 \tag{3.2}$$

Dove i valori rappresentati da *sym* indicano i valori che compongono la costellazione scelta in base alla codifica e $x(k)$ il valore dell'ingresso. Questa operazione può essere effettuata un'unica volta prima di iniziare la simulazione, salvando in una matrice tutti i valori delle branch metrics che possono essere ottenuti in base ai limiti degli ingressi.
Dopo questa fase iniziale si passa a quella ricorsiva, in cui vengono effettivamente ottenute le sequenze degli stati.
In questa fase, come spiegato nella descrizione teorica dell'algoritmo, per ogni stato possibile vengono calcolati i valori delle path metrics. Questi sono ottenuti effettuando la somma tra la path metric dello stato di partenza e il valore della branch metric relativo alla transizione necessaria per portare allo stato attuale.
In questo modo nel caso di modulazione PAM2 si avranno, per ogni stato, due valori di path metrics possibili, mentre saranno quattro nel caso di modulazione PAM4. Per scegliere quello corretto è necessario calcolare il minimo, che rappresenta il valore della path metric definitiva relativa allo stato. La decisione presa nella scelta del minimo inoltre deve essere memorizzata, perché permette di risalire allo stato di partenza. Per questo il suo valore viene inviato alla path history dove viene utilizzato per selezionare la sequenza delle decisioni precedenti, mantenendo in memoria un numero di decisioni pari alla path history length. I valori delle path metrics invece devono essere memorizzati per poter essere utilizzati durante i calcoli relativi all'ingresso successivo.
Il valore dell'uscita viene ottenuto dai valori presenti nell'ultima posizione della path history, in questo caso quello relativo allo stato 0, dato che non vengono utilizzati algoritmi particolari per la scelta.
Questa operazione viene effettuata in modo ricorsivo per tutti gli ingressi ricevuti dal canale, a cui viene aggiunta una sequenza di ingressi nulli necessari ad ottenere in uscita tutti i valori presenti nella path history.
La sequenza ottenuta può quindi essere confrontata con quella generata dal trasmettitore, in modo da ottenere il valore del bit error rate, che rappresenta il rapporto tra il numero di bit ricevuti errati e il numero di bit totali trasmessi.

3.3 Modello floating point

Inizialmente nel modello matlab i dati utilizzati sono stati salvati con la rappresentazione floating point. Il vantaggio di questa rappresentazione è che permette di rappresentare i dati con una precisione molto elevata, rendendo l'errore associato a questa trascurabile.
Ciò permette di vedere esattamente l'effetto del rumore e della distorsione sul bit error rate, senza che questo sia condizionato dalla presenza dell'errore di quantizzazione.
Per ottenere dei valori di ber validi a livello statistico è necessario effettuare le simulazioni utilizzando in ingresso sequenze di dati molto lunghe, idealmente infinite. Questo è però ovviamente impossibile, a causa del tempo richiesto dalla simulazione.
Durante le simulazioni per avere risultati validi è stato utilizzato un numero di ingressi minimo pari a 10^6, perché questo valore è abbastanza elevato da mostrare differenze significative fra le varie tipologie di ricevitore ed i tempi di simulazione non sono eccessivamente lunghi.
Nella Figura 3.1 e nella Figura 3.2 sono riportati i valori di BER ottenuti in funzio-

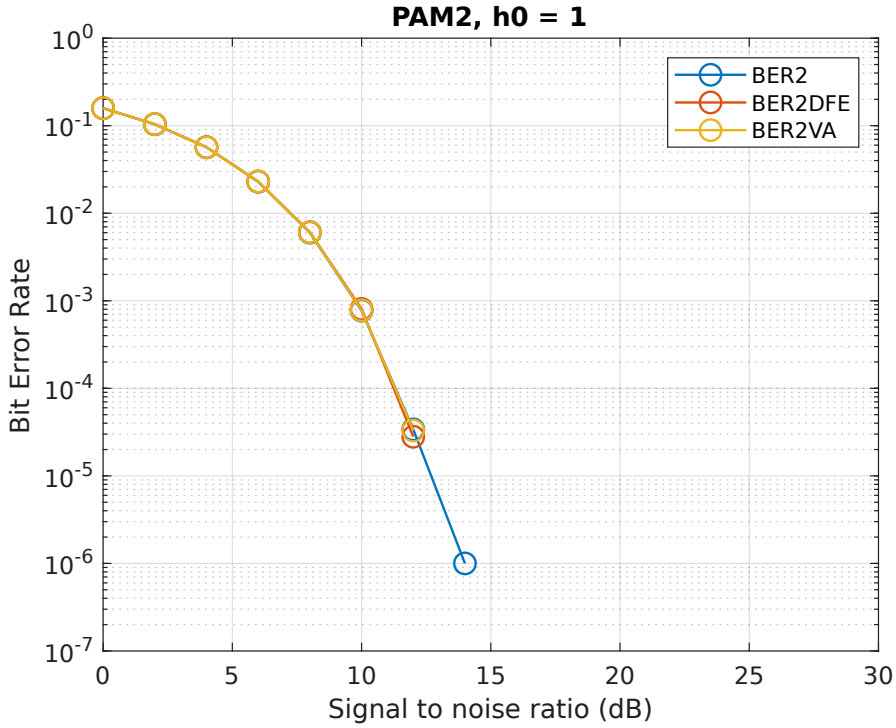

Figura 3.1: BER per modulazione PAM2 senza distorsione

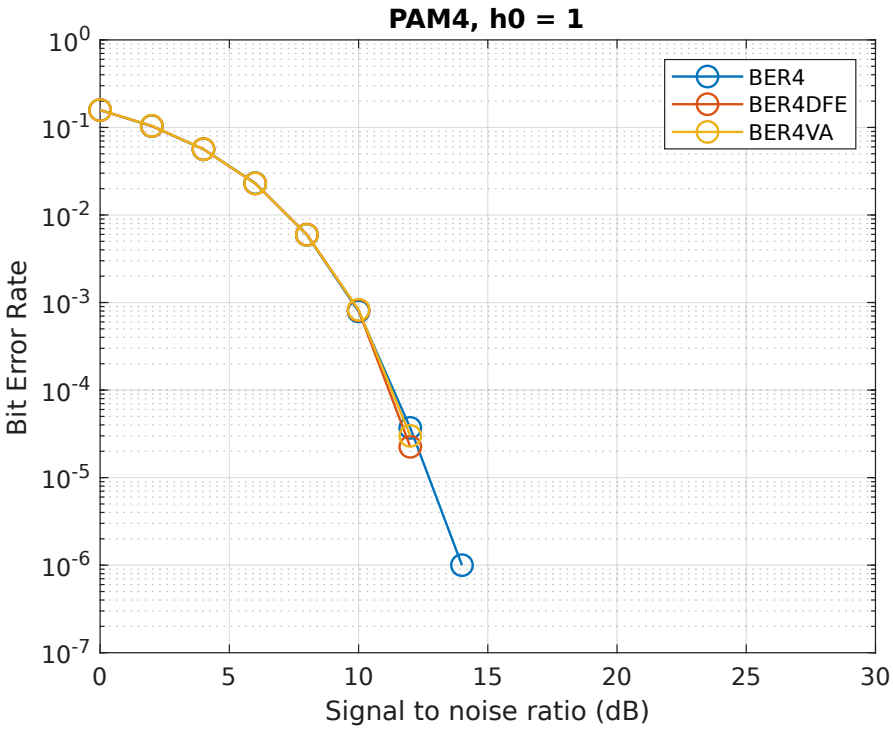

Figura 3.2: BER per modulazione PAM4 senza distorsione

ne del SNR per la modulazione PAM2 e per quella PAM4, in assenza di distorsione di canale. Come si può vedere nei grafici, il valore del BER è indipendente dal tipo di ricevitore per entrambe le modulazioni, questo era prevedibile dato che sia il DFE sia il Viterbi migliorano le prestazioni riducendo l'effetto dell'interferenza intersimbolica, ma non possono fare nulla per ridurre l'effetto del rumore.

In presenza di distorsione invece il contributo dei ricevitori diventa fondamentale, come mostrato nella Figura 3.3 e nella Figura 3.4.
Nel caso di modulazione PAM2 l'effetto dell'interferenza intersimbolica è significativo, ma l'utilizzo di ricevitori diversi permette di ottenere netti miglioramenti, come mostrato nella Figura 3.3 e nella Tabella 3.1.

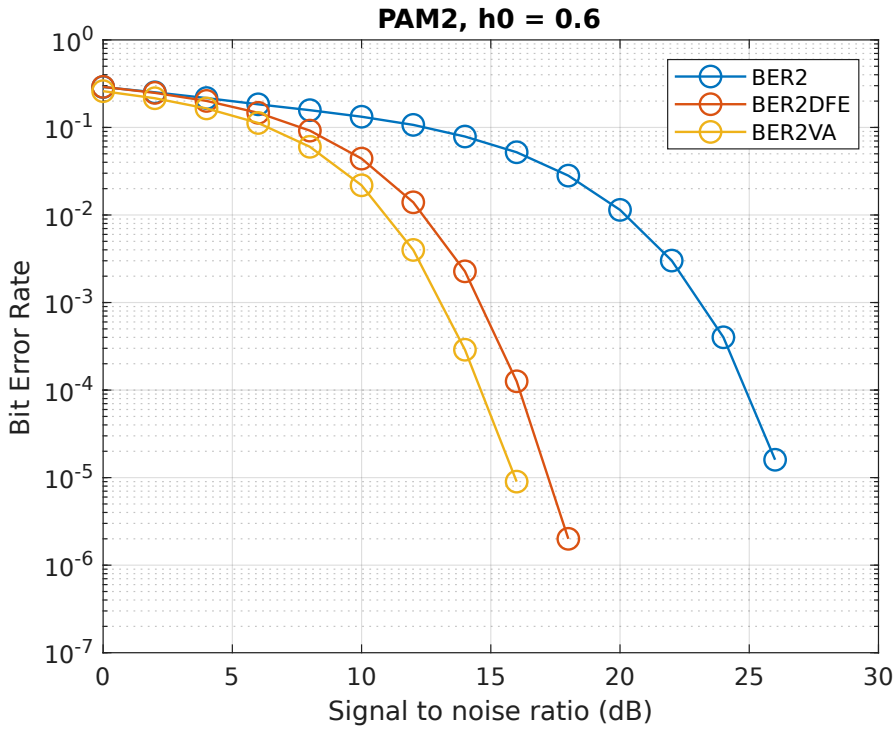

Figura 3.3: BER per modulazione PAM2 e distorsione ($h_0 = 0.6$, $h_1 = 0.4$)

SNR (dB)	0	4	8	12	16	20
BER (no correzione)	0.2896	0.2161	0.1569	0.1064	0.0519	0.0113
BER DFE	0.2905	0.2002	0.0916	0.0138	0.0001	0
BER Viterbi	0.2594	0.1654	0.0600	0.0041	0	0

Tabella 3.1: BER modulazione PAM2, $h_0 = 0.6$, $h_1 = 0.4$

Nel caso di modulazione PAM4 l'effetto dell'interferenza è ancora più importante. In questo caso, infatti, questa distorsione è sufficiente a causare errori anche in assenza di rumore. Per questo motivo l'utilizzo di un DFE o di un Viterbi diventa fondamentale.
Gli effetti dei ricevitori sono mostrati nella Figura 3.4 e nella Tabella 3.2. Con alti livelli di rumore il guadagno è ridotto, anche se non trascurabile, mentre quando il valore del SNR supera gli 8 dB diventa fondamentale il contributo dei ricevitori.

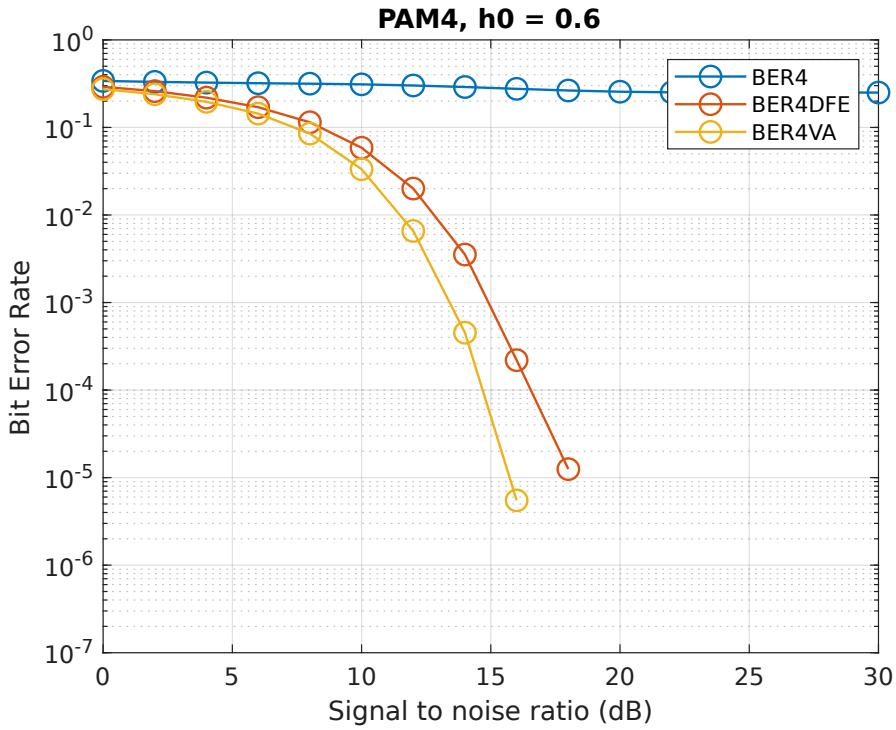

Figura 3.4: BER per modulazione PAM4 e distorsione ($h_0 = 0.6$, $h_1 = 0.4$)

SNR (dB)	0	4	8	12	16	20
BER (no correzione)	0.3398	0.3248	0.3163	0.3018	0.2761	0.2559
BER DFE	0.2917	0.2188	0.1137	0.0201	0.0002	0
BER Viterbi	0.2746	0.1963	0.0852	0.0066	0.0000	0

Tabella 3.2: BER modulazione PAM4, $h_0 = 0.6$, $h_1 = 0.4$

La Figura 3.5 mostra l'andamento del BER in funzione del SNR (asse y) e del valore di h_0 (asse x) per i vari ricevitori, rispettivamente con modulazione PAM2 e PAM4. In questo caso è meno evidente la differenza di prestazioni tra il DFE e il Viterbi, ma è evidenziata quella con il caso in cui non sia effettuata nessuna correzione.

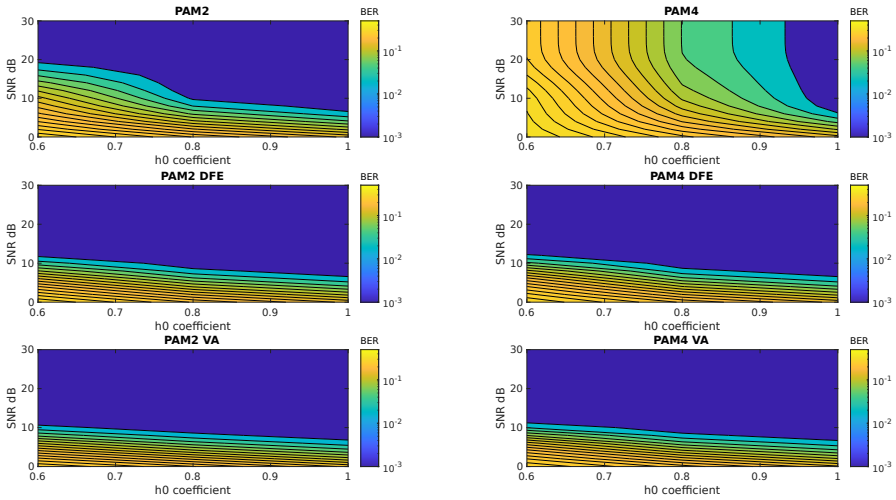

Figura 3.5: variazione del BER in funzione di SNR e distorsione

3.4 Modello quantizzato

Per poter utilizzare il modello matlab come riferimento per le successive implementazioni RTL, è però necessario tenere in considerazione il fatto che i dati non potranno essere rappresentati in formato floating point, dato che l'occupazione di area sarebbe troppo elevata e anche le prestazioni, riferendosi alla frequenza massima a cui è possibile lavorare, non sarebbero sufficienti. Per questo motivo tutti i dati devono essere rappresentati come interi, simulando il comportamento che avrebbero effettuando le operazioni in complemento a 2.
Per poter utilizzare questa diversa rappresentazione dei dati è stato necessario modificare alcune funzioni del modello matlab.
Innanzitutto, è stato necessario implementare delle nuove funzioni per effettuare le operazioni di somma e sottrazione, che nel caso di superamento dei valori massimi e minimi definiti dal numero di bit devono simulare il comportamento del complemento a due. In questi casi, infatti superando il valore massimo $(2^{n_{bit}} - 1)$ il valore successivo ripartirebbe dal valore minimo negativo. Per esempio, se la rappresentazione fosse a 4 bit, il valore massimo positivo sarebbe 7 (0b0111), tentando di incrementarlo ulteriormente si ripartirebbe da -8 (0b1111).
Questa aggiunta ha permesso di implementare il nuovo modello in modo molto simile a quello precedente, cambiando solamente la rappresentazione dei dati e utilizzando le nuove funzioni in sostituzione a quelle classiche utilizzate in precedenza.
Con questo nuovo modello i valori di bit error rate sono più alti di quelli ottenuti in precedenza, soprattutto se il numero di bit utilizzato per rappresentare gli ingressi

e le branch metrics è molto piccolo (minore di 5). Questo è dovuto all'aggiunta dell'errore di quantizzazione, che diventa un'ulteriore causa di errore. Utilizzando un numero di bit sufficientemente elevato è però possibile ridurlo, al punto di avere risultati quasi identici al caso in cui sia utilizzata la rappresentazione floating point. Lo svantaggio in questo caso è però che, come si potrà vedere nei risultati delle implementazioni RTL, aumentare il numero di bit causa un aumento di area e di potenza consumata e una diminuzione della frequenza massima utilizzabile.

Come si può vedere dalla Figura 3.6, nella versione PAM2 solo con un parallelismo

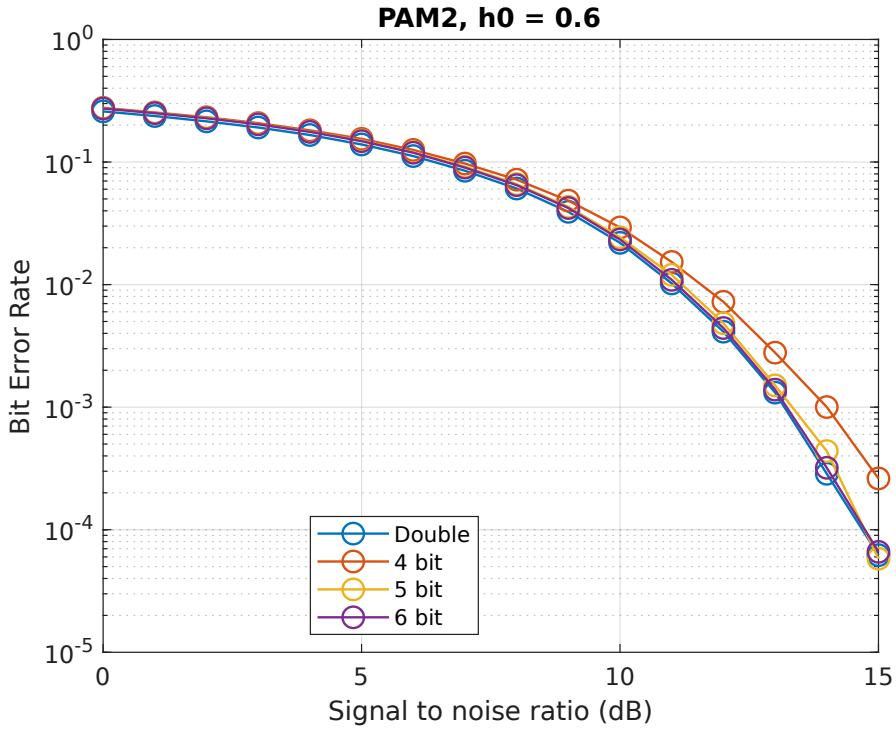

Figura 3.6: BER al variare del parallelismo delle branch metrics, PAM2

di 4 bit per le branch metrics si notano variazioni significative di bit error rate rispetto alla versione in cui i dati sono rappresentati con la massima precisione, mentre con un numero superiore l'errore dovuto alla quantizzazione è quasi nullo.

Nella Figura 3.7 è mostrata la variazione del bit error rate nel caso in cui la codifica sia PAM4. In questo caso si può notare che per avere risultati più vicini a quelli ottenibili nel caso in cui l'errore di quantizzazione sia nullo è necessario utilizzare un numero maggiore di bit. Questo è dovuto al fatto che gli intervalli che separano un simbolo dall'altro sono molto più stretti, ed è quindi sufficiente una variazione minore a causare un errore.

Per avere prestazioni migliori è possibile utilizzare tecniche per il calcolo delle

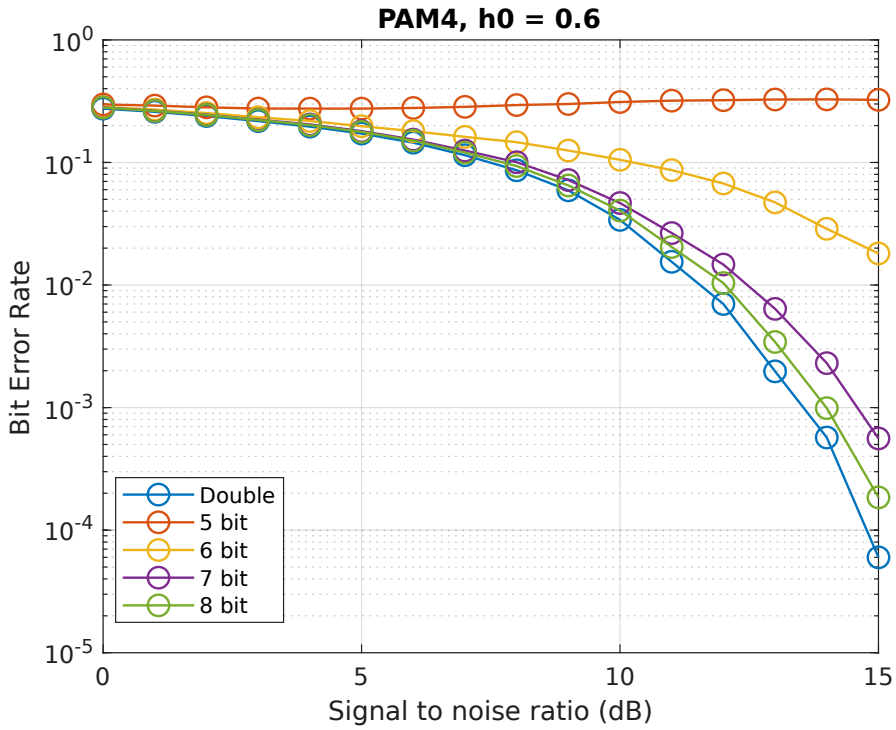

Figura 3.7: BER al variare del parallelismo delle branch metrics, PAM4

branch metrics diverse rispetto a quella utilizzata e descritta in precedenza, in cui i loro valori quantizzati vengono modificati in modo da sfruttare al massimo l'ampiezza data dal numero di bit a disposizione. Queste permettono di avere risultati più accurati, ma non sono state approfondite dato che nella fase in cui è stata effettuata l'implementazione RTL le branch metrics vengono fornite ad una memoria e non sono calcolate volta per volta, rendendo l'implementazione indipendente dal modo in cui sono ottenute.

Capitolo 4

Implementazioni Viterbi singolo

Dopo aver completato i modelli di riferimento su matlab, è stato possibile procedere con l'implementazione del Viterbi Detector in verilog e system verilog. Da questo momento tutti i casi si riferiscono ad implementazioni per la ricezione di dati con codifica PAM4, quindi con 4 possibili stati.
L'implementazione è stata composta da due fasi: la prima è consistita nella realizzazione di un modulo che implementasse il Viterbi ricevendo un ingresso per volta e fornendo una sola uscita; la seconda fase invece è servita a realizzare un modulo a più alto livello, che permettesse di ricevere più dati in ingresso contemporaneamente (32) e che fornisse quindi lo stesso numero di uscite.
Per realizzare il singolo Viterbi è stato necessario a sua volta procedere nella

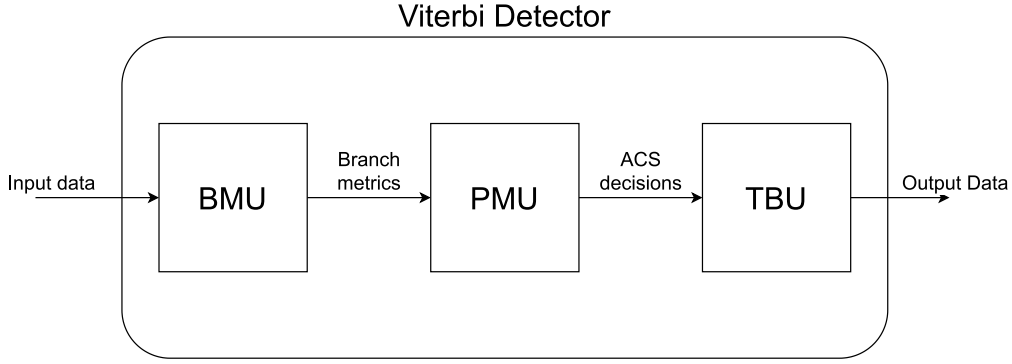

Figura 4.1: schema del Viterbi Detector

realizzazione dei blocchi principali che lo compongono, rispettivamente:

- BMU: la branch metric unit riceve in ingresso il valore che il Viterbi riceve

dal componente che lo precede, solitamente un FIR, e lo usa come indirizzo di una memoria che restituisce le branch metrics corrispondenti;

- PMU: la path metric unit utilizza le branch metrics ottenute dal blocco precedente: le somma alle path metrics memorizzate al suo interno per calcolare i valori aggiornati, e trasmette le decisioni prese al blocco successivo;
- TBU: la traceback unit salva i valori delle decisioni prese dalla PMU in quattro registri, dove viene composta la path history.

4.1 Branch metric unit

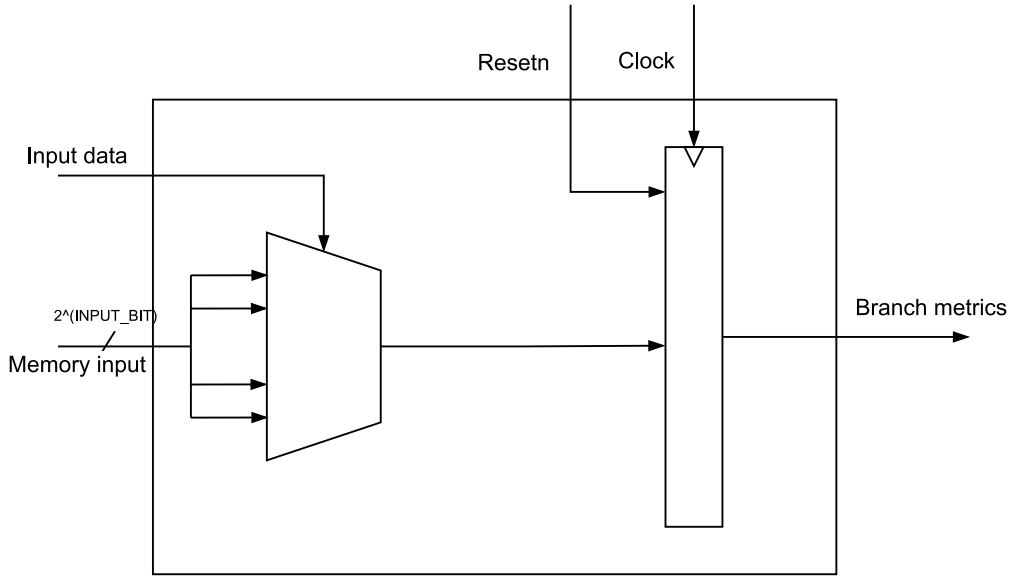

Figura 4.2: schema della branch metric unit

Nel caso studiato, la branch metric unit riceve in ingresso i dati ottenuti da un FIR, anche lui parte dell'equalizzatore, oltre ovviamente ai segnali di clock e di reset.
La memoria è descritta come un vettore di 2^n valori, dove n è il numero di bit dell'ingresso, e utilizzando il dato ricevuto dal FIR come indirizzo del vettore, seleziona i valori delle branch metrics corrispondenti.
Il Viterbi lavora su 4 stati diversi, quindi per tenere conto di tutte le possibili transizioni sono necessarie 16 branch metrics, che vengono trasmesse alla PMU.
La BMU è quindi composta da un multiplexer che seleziona i valori dalla memoria, mandando in uscita le branch metrics relative al dato ricevuto in ingresso, che viene utilizzato come selettore. I loro valori sono salvati con un parallelismo definito

dalla variabile BM_BIT. Le branch metrics scelte sono raggruppate in un unico bus, di parallelismo $16 \cdot BM\_BIT$ e mandate in uscita dopo essere state salvate in un registro, che campionandole evita che la memoria entri a far parte del percorso critico (a livello di timing) presente nella PMU.

4.2 Path metric unit

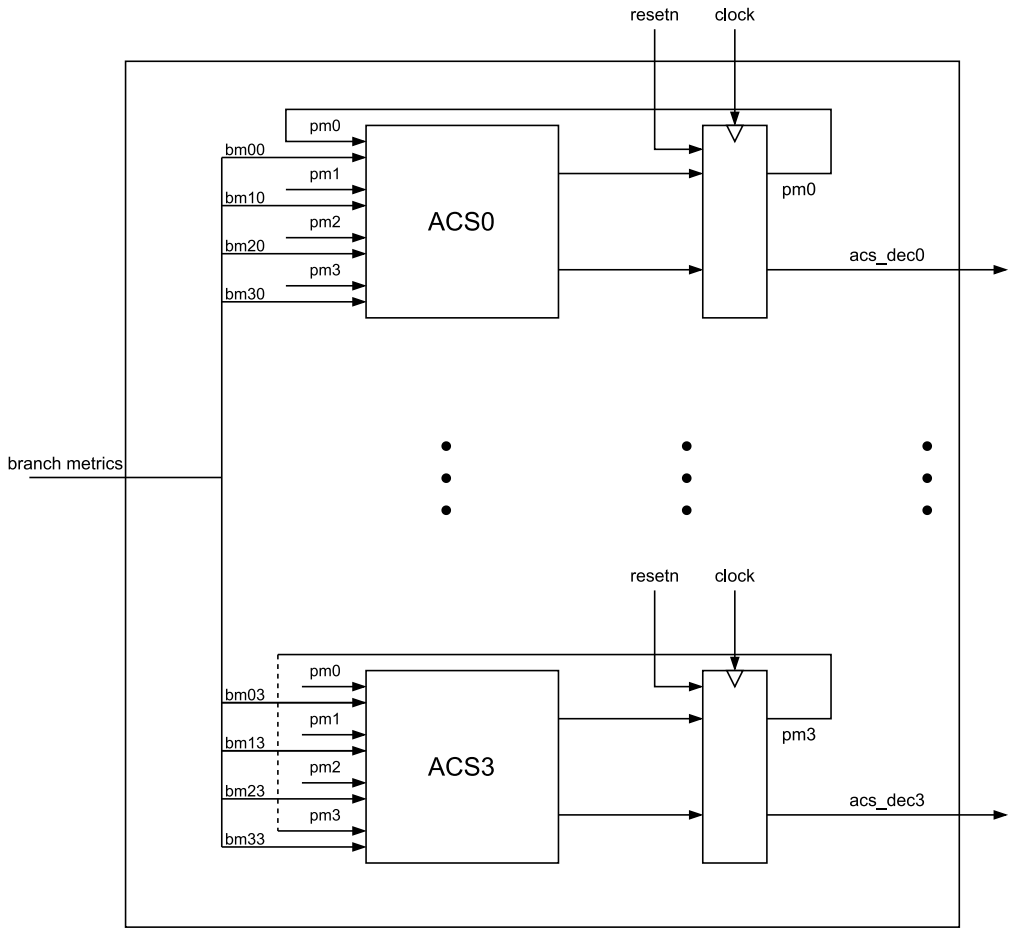

Figura 4.3: schema della path metric unit

La PMU è il blocco più complesso del Viterbi. Utilizzando le path metrics calcolate nel colpo di clock precedente e le branch metrics ottenute dalla BMU, deve calcolare il valore delle nuove path metrics e mandare alla TBU le decisioni prese. Le operazioni principali vengono svolte da unità chiamate ACS (Add-Compare-Select) che hanno il compito di sommare le path metrics in ingresso con le rispettive

branch metrics. In ogni Viterbi Detector tipicamente sono presenti quattro ACS, ad ognuno dei quali è associato uno stato. Viene quindi aggiunto, ad ogni path metric dello stato precedente, il valore della branch metrics relativa alla transizione che porta dallo stato rappresentato dalla path metric a quello dell'ACS. All'interno degli ACS questi valori vengono sommati, dopodiché viene eseguito il confronto per trovare il valore minore, che viene mandato all'esterno insieme alla decisione presa. All'interno della PMU i valori delle path metrics vengono memorizzati in un registro, in modo tale che al colpo di clock successivo possano essere usati come ingressi degli ACS. Anche le decisioni attraversano i registri, in modo da spezzare il loro percorso critico, e successivamente vengono mandate all'esterno del blocco, dove verranno utilizzate dalla TBU.
Il fattore che rende più complessa questa unità è la presenza del feedback che, tramite i registri, porta all'ingresso degli ACS i valori delle path metric calcolati nel colpo di clock precedente. Questo fa sì che sia molto più complesso trovare ottimizzazioni per diminuire il percorso critico, perché il loop rende impossibile, a meno di accorgimenti particolari, l'utilizzo di ottimizzazioni come la pipeline, che porterebbero alla generazione di errori. Per questo motivo una delle parti più importanti della tesi è stata appunto quella di analizzare alcune tecniche che permettono la riduzione del critical path, studiando il trade-off tra il miglioramento delle performance e l'aumento dell'area occupata.

4.2.1 Add – Compare – Select

Gli ACS sono le unità in cui vengono eseguite tutte le principali operazioni matematiche presenti nel Viterbi, e a causa dell'impossibilità di usare la pipeline, è al loro interno che è presente il percorso critico che limita le prestazioni di tutto il dispositivo.
Le operazioni principali svolte al suo interno, mostrate in Figura 4.4, sono la somma delle branch metrics con le path metrics e il confronto dei risultati.
La somma iniziale viene svolta da normali sommatori, descritti ad alto livello in modo che, in fase di sintesi, il sintetizzatore possa scegliere l'architettura migliore, a seconda che il fine sia avere sommatori più veloci o sommatori più piccoli. Il metodo utilizzato nell'immagine per effettuare la comparazione è quello più semplice, mentre quelli più complessi, ma che garantiscono prestazioni migliori, verranno descritti insieme alle ottimizzazioni della PMU.

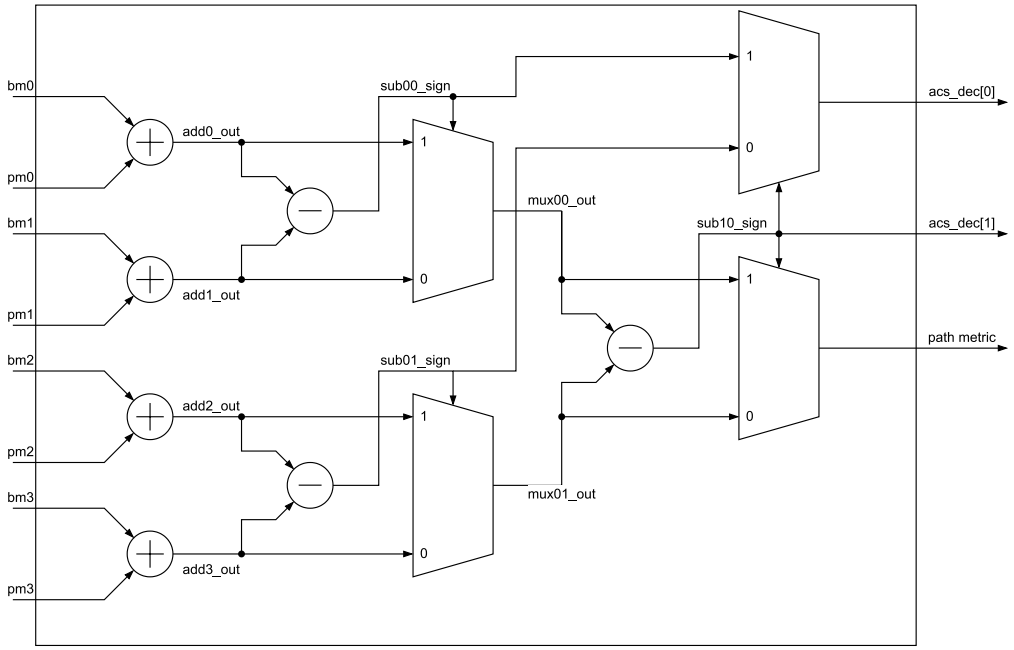

Figura 4.4: schema Add-Compare-Select

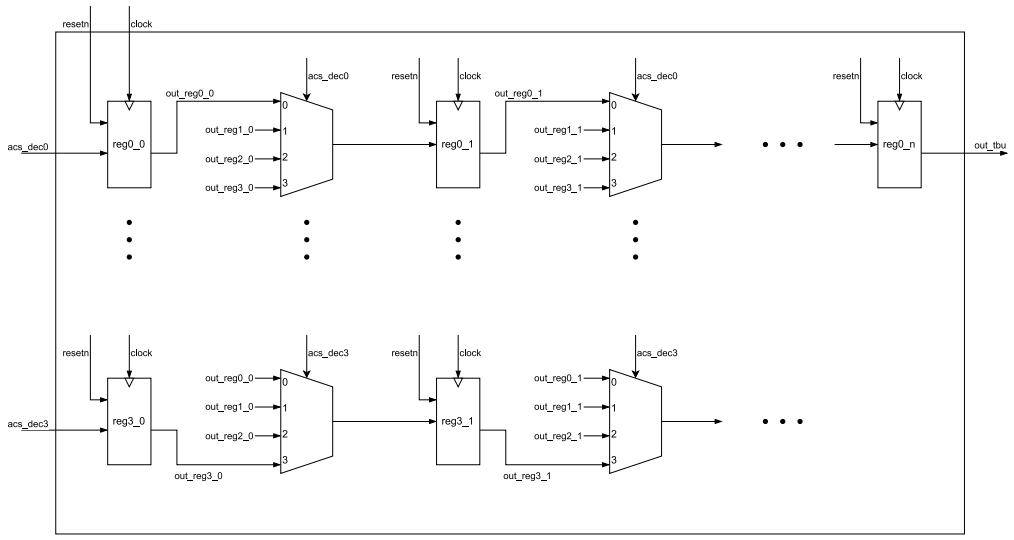

Figura 4.5: schema della traceback unit

4.3 Traceback unit

La traceback unit (TBU) è l'ultima unità all'interno del Viterbi Detector. Questa unità ha il compito di memorizzare le sequenze di decisioni prese dalla PMU per

arrivare ad ognuno dei quattro stati finali. Queste sequenze di decisioni prendono il nome di path history e la loro lunghezza è limitata, come visto nella parte teorica, ad un valore tale che le ultime decisioni coincidano per tutti gli stati. La lunghezza, detta path history length, è stata limitata a 20.
Ogni riga della Figura 4.5 rappresenta uno stato, ad esempio osservando la riga più in alto, questa rappresenta la sequenza di decisioni prese nel tempo per arrivare alla fine allo stato 0.
Le decisioni, come si può vedere nello schema, vengono inizialmente salvate nei registri iniziali, dato che rappresentano l'ultima decisione necessaria per arrivare in ognuno degli stati da essi rappresentati. Dopodiché, sempre le stesse decisioni vengono utilizzate come selettori per i multiplexer che selezionano l'ingresso di tutti i registri successivi.
Questo è dovuto al fatto che la decisione indica quale sia lo stato precedente a quello indicato dalla riga, per cui il resto della sequenza deve essere presa dalla riga associata alla decisione presa, prima che i suoi valori vengano aggiornati. Tutte le decisioni vengono quindi memorizzate nei registri, e nel farlo vengono shiftate di una posizione verso l'uscita del Viterbi.
Dato che, come detto in precedenza, le decisioni prese negli stati più vicini all'uscita tendono a coincidere, si assume che tutte le decisioni relative all'ultimo stato siano uguali. Se così non fosse, significherebbe che il valore scelto per la path history length è troppo basso, e che dovrebbe quindi essere aumentato. Per questo è possibile scegliere in modo arbitrario da quale riga scegliere il valore dell'uscita.
Per risparmiare area, inoltre, è possibile rimuovere i registri relativi agli stati non utilizzati, cioè quelli le cui decisioni non verranno trasmesse, dato che occuperebbero area inutilmente.
La versione della TBU appena descritta e mostrata è quella più semplice, in seguito verrà mostrata una tecnica, chiamata best selection, che permette di ridurre il valore della path history length.

4.4 Ottimizzazioni path metric unit

4.4.1 Comparatore a tre sottrattori

La versione dell'ACS già mostrata in Figura 4.4, rappresenta l'implementazione più semplice. In questo caso, infatti, dopo alla somma tra le path metric e le branch metric, i valori in uscita dai sommatori devono essere confrontati per trovare il minimo. Per farlo vengono divisi a coppie, e vengono effettuate le sottrazioni tra il primo e il secondo risultato, e tra il terzo e il quarto. Il risultato delle sottrazioni permette di trovare il minimo osservando il valore del bit di segno, il most significant bit (MSB). Se il valore dell'MSB è 0, significa che il risultato della sottrazione è

positivo, quindi il primo ingresso del sottrattore è maggiore del secondo, viceversa se è 1, significa che il risultato è negativo, quindi il secondo ingresso è più grande del primo.
I bit di segno dei risultati dei sottrattori vengono quindi utilizzati come selettori per due multiplexer, che hanno in ingresso due dei risultati ottenuti dagli adder. I multiplexer scelgono i due più piccoli, che vengono quindi confrontati nello stesso modo, per trovare il valore più piccolo fra i quattro, che verrà mandato in uscita all'ACS. Il bit di segno dell'ultima sottrazione rappresenta il segnale `acs_dec[1]` (Figura 4.4), e viene anche utilizzato come selettore per un ulteriore multiplexer che sceglie quale dei bit di segno ottenuti nel primo livello rappresenti il segnale `acs_dec[0]`.
Lo svantaggio più evidente di questa implementazione è la presenza nel percorso critico di un sommatore e due sottrattori, se l'ulteriore ritardo introdotto dai multiplexer viene considerato trascurabile. Per questo, a causa dell'impossibilità di inserire livelli di pipe tra questi componenti, la frequenza massima a cui può lavorare il Viterbi viene ridotta notevolmente. Inoltre, come sarà visibile più chiaramente dai risultati delle sintesi, all'aumentare del numero di bit con cui sono rappresentate le branch metrics e le path metrics le performance peggiorano ulteriormente, a causa del tempo maggiore richiesto per completare le operazioni di somma e sottrazione.
Questa implementazione, nonostante la sua semplicità, presenta dei vantaggi rispetto alle altre studiate; infatti, ottenendo il valore corrispondente al minimo utilizzando solamente quattro sommatori e tre sottrattori, l'occupazione di area risulta molto ridotta.

4.4.2 Comparatore a sei sottrattori

Nel lavoro di Black e Meng [16] viene mostrata un'implementazione alternativa del comparatore presente all'interno dell'ACS. Nella versione descritta in precedenza il principale lato negativo è la presenza dell'ultimo sottrattore, in serie a quelli precedenti, che aumenta notevolmente la lunghezza del critical path, limitando le prestazioni.
L'alternativa proposta, mostrata in Figura 4.6, riesce invece a migliorare il critical path, pagando però con un aumento dell'area occupata.
Utilizzando sei sottrattori in parallelo, che eseguono la differenza tra i valori di tutte le possibili coppie di dati relativi agli stati e con l'aggiunta di una parte di logica combinatoria che elabora i risultati, è possibile stabilire immediatamente quale dei 4 valori ottenuti dai sommatori sia quello inferiore. Il vantaggio, in questo caso, è che non sono presenti sottrattori in serie, ma lavorano tutti nello stesso momento, garantendo una maggiore velocità.

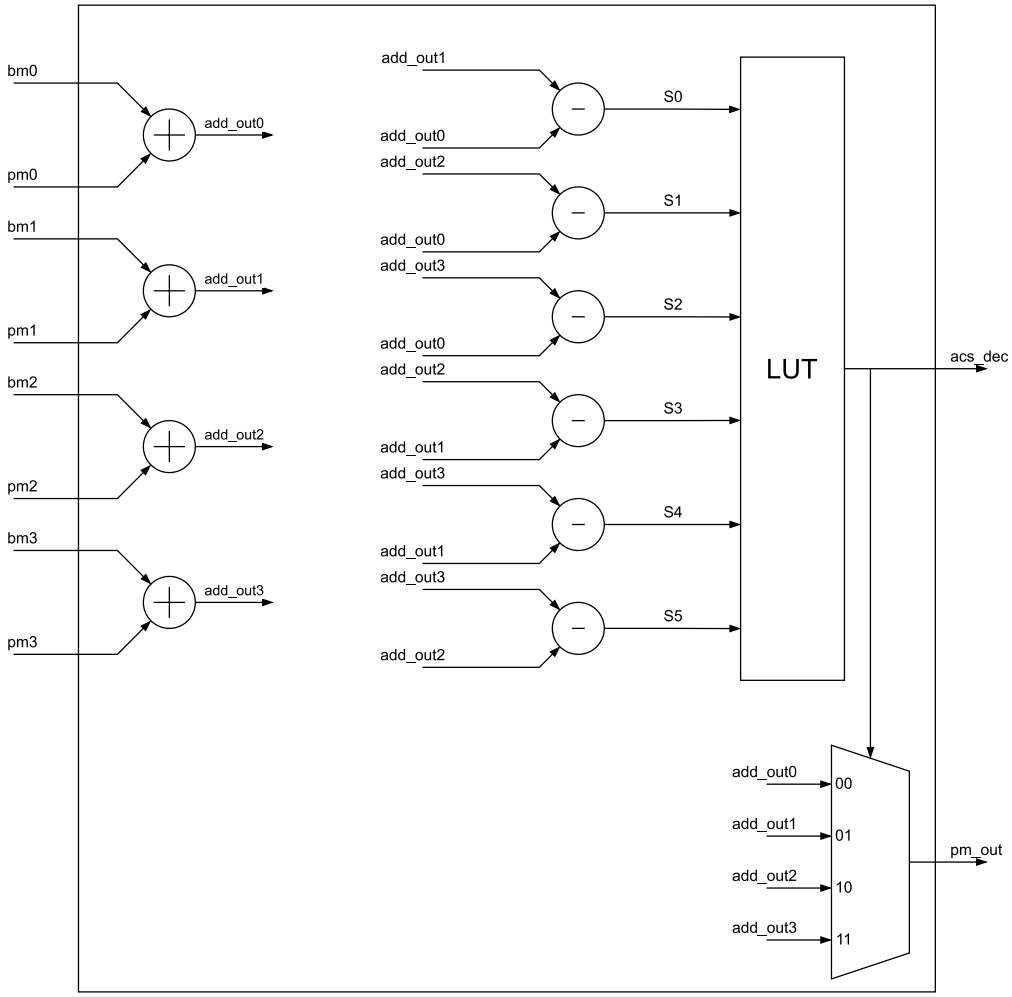

Figura 4.6: schema comparatore ottimizzato con 6 sottrattori

Indicando con S_n il most significant bit in uscita all'ennesimo sottrattore, nella Tabella 4.1 viene mostrato il comportamento della look up table che elabora i risultati ottenuti. Nel caso in cui nella tabella un ingresso sia indicato con "X", significa che il suo valore non è rilevante nel caso in cui si verifichino le altre condizioni indicate. L'uscita della look up table corrisponde alla decisione dell'ACS e viene utilizzata per controllare il multiplexer che seleziona il valore della path metric minore. Il valore utilizzato viene anche mandato all'esterno, dove la decisione verrà salvata nella TBU.

S_0	S_1	S_2	S_3	S_4	S_5	*out*
0	0	0	X	X	X	00
1	X	X	0	0	X	01
X	1	X	1	X	0	10
			ELSE			11

Tabella 4.1: LUT per scegliere la path metric minore

Esaminando più nel dettaglio la Tabella 4.1, è facile capire quale sia la logica su cui si basa. Nella prima riga, in cui la decisione presa è quella che indica lo stato 0, si vede che viene ottenuta quando il bit di segno relativo alle prime 3 differenze è 0: questo risultato si ottiene quando la differenza tra i valori di una coppia è positiva, avendo il primo ingresso più grande del secondo. Nelle prime 3 coppie il secondo ingresso è sempre relativo allo stato 0, che quindi genera un valore della path metric minore rispetto a tutti gli altri. Il valore dell'uscita può quindi essere fissato, senza aver bisogno di analizzare i risultati delle altre sottrazioni. Nelle altre righe si utilizza la stessa logica, si osservano i risultati di tutte le differenze che riguardano uno stato, e solamente se tutte indicano che lui è quello minore viene presa la decisione.
La decisione presa viene quindi mandata alla TBU e allo stesso tempo utilizzata come selettore per il multiplexer che ha come ingressi i risultati dei sommatori, che sono i possibili valori della path metric. Il multiplexer seleziona quello inferiore, che diventa ufficialmente la nuova path metric dello stato a cui si riferisce l'ACS.

4.4.3 Double State

Nelle tecniche viste in precedenza cambia solamente la soluzione scelta per la realizzazione del comparatore. Aumentando la complessità della PMU è possibile però modificare completamente la struttura dell'Add-Compare-Select, in modo tale da permettere che la somma delle path metrics con le rispettive branch metrics venga realizzata in parallelo alla comparazione. Una soluzione è quella di utilizzare la tecnica del double state, descritta da Lee e Sonntag [17].
Il loro lavoro mostra come possa essere vantaggioso descrivere uno stato non solo con il valore dello stato finale, ma considerando anche lo stato precedente, così facendo il numero di stati aumenta, nel caso PAM2 come $2 \cdot N$, mentre come $4 \cdot N$ nel caso PAM4.
Considerando per semplicità il caso della trasmissione binaria, se lo stato di arrivo utilizzando le altre tecniche era inizialmente 0, adesso potrà essere descritto sia come 00 sia come 10, dove il primo numero rappresenta lo stato di partenza. Nel

caso in cui il numero di stati non sia 2 come nel caso descritto, ma 4 come in quello della trasmissione PAM4, il procedimento è esattamente lo stesso.
La Figura 4.7 mostra come varia il trellis per il caso di trasmissione binaria.

Anche in questo caso, per ognuno degli stati di arrivo, è necessario calcolare il

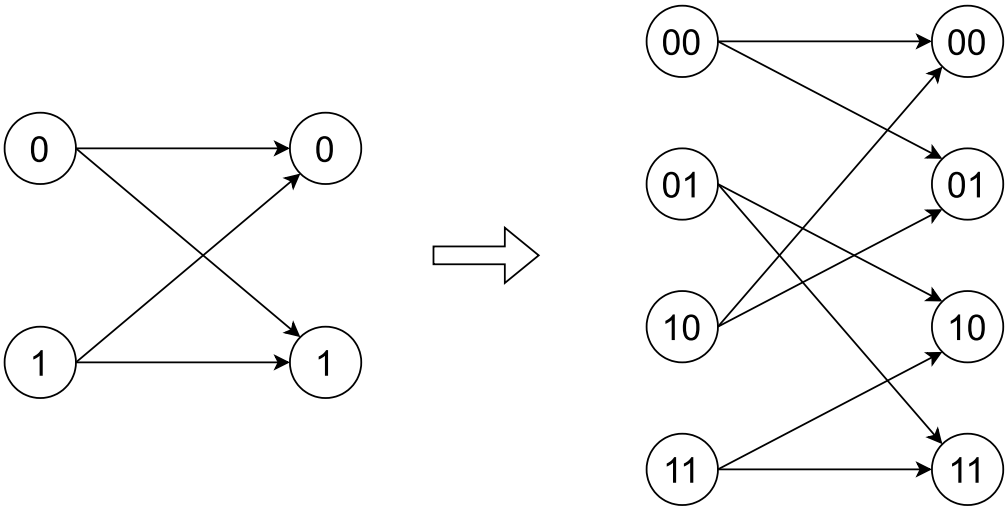

Figura 4.7: trellis per segnale binario nel caso normale e in quello double state

valore della path metric. Essendo aumentato il numero di stati, aumentano anche le unità di ACS necessarie, che dovranno essere una per ogni stato.
Il funzionamento dei singoli ACS però non aumenta di complessità, anzi diminuisce. Il valore dei dati dipende da quello delle path metrics degli stati di partenza, che possono essere diverse tra di loro, e da quello delle branch metrics. È però possibile notare dalla figura come, per ogni stato, nella nuova configurazione gli archi che lo raggiungono rappresentano sempre la stessa transizione, per cui il valore della branch metric è sempre lo stesso. Essendo questo valore costante, può essere escluso dalla comparazione, che può essere effettuata solo sui valori delle path metrics di partenza.
Per questo, nella struttura double state, l'operazione di "Add" che calcola il nuovo valore della path metric può essere eseguita in parallelo a quella di "Compare". La struttura del nuovo ACS è mostrata nella Figura 4.8.

Evitando di avere in serie le operazioni di Add e Compare è possibile diminuire notevolmente il percorso critico, ma la nuova struttura, per come è stata descritta nella Figura 4.8, ha bisogno di 4 ACS invece di 2, causando un aumento notevole di area, che sarebbe ancora più elevato nel caso di trasmissione PAM4, dove gli ACS necessari da 4 diventerebbero 16. Osservando la figura è però possibile vedere come per le coppie di stati finali (00, 01) e (10, 11), le path metric di partenza da confrontare siano le stesse. In questo caso è quindi possibile modificare la struttura

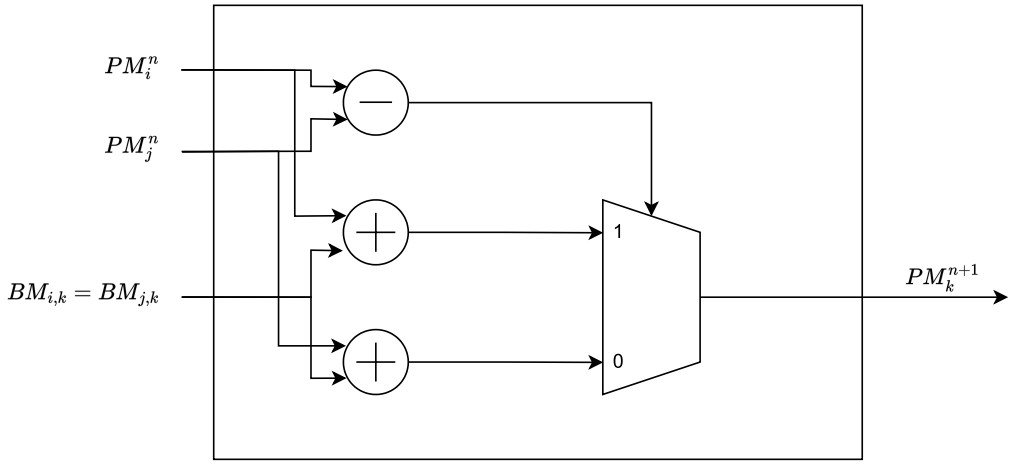

Figura 4.8: ACS per architettura double state

degli ACS, in modo da raggruppare gli stati che hanno la stessa decisione in comune. Grazie a questa modifica il numero totale di comparazioni richieste rimane lo stesso delle versioni precedenti, anche se rimane un aumento di area causato dall'aumento dei sommatori, che non può essere evitato.

La Figura 4.9 mostra la versione definitiva di uno dei due ACS nella versione

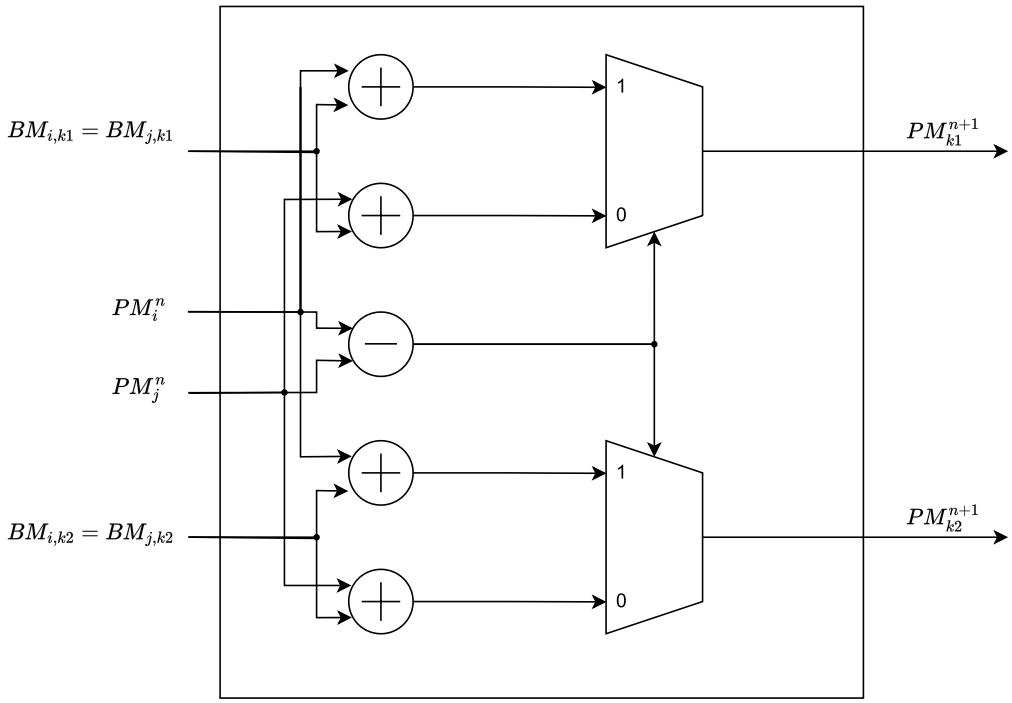

Figura 4.9: ACS per architettura double state ottimizzato

PAM2. La versione implementata resta quella in PAM4, dove il numero elevato di stati rende poco chiara la rappresentazione grafica. Nel caso PAM4 gli stati possibili aumentano da 4 a 16, con le ottimizzazioni appena descritte è possibile mantenere costante il numero dei comparatori, ma per ogni ACS adesso il numero di sommatori diventa 16, variando nello stesso modo del numero degli stati. Questo incremento, oltre a quello dei registri necessari per memorizzare i valori di tutte le path metrics, è la causa principale dell'aumento dell'area.
L'architettura proposta può essere spiegata come una normale implementazione della tecnica lookahead nel caso in cui siano presenti non linearità all'interno del loop [18].
In pratica questa architettura trasforma l'ACS, da un "Add-Compare-Select" in un "Compare-Add-Select". In [19] è stato proposto un algoritmo diverso, detto differential trellis decoding (DTD), basato su un metodo di "Compare-Select-Add". Questo metodo riduce il numero di addizioni aumentando il numero di operazioni di pre-processing del trellis, al contrario dell'approccio double state che aumenta la velocità aumentando il numero di stati, ma non è stato approfondito a livello implementativo in questa tesi.
Le variazioni di area e prestazioni effettivamente ottenuti durante le sintesi verranno analizzati in seguito e confrontati con quelli delle altre ottimizzazioni.

4.4.4 Bit level pipeline

L'ultima ottimizzazione analizzata per migliorare le prestazioni del Viterbi Detector, senza ricorrere all'utilizzo di più Viterbi in parallelo, è quella della bit level pipeline. Questa architettura viene proposta per la prima volta da Fettweis e Meyr [20], nel 1990, e tenta di risolvere il problema dovuto al fatto che, per aumentare la precisione del Viterbi e aumentare la qualità dei suoi risultati, sia necessario aumentare il parallelismo interno. L'aumento del numero di bit utilizzati permette di avere risultati più accurati, ma questo miglioramento viene pagato con un aumento delle dimensioni di tutti i componenti dell'hardware, e di conseguenza anche una diminuzione della frequenza massima a cui è possibile lavorare e ad un maggior consumo di potenza. Per evitare questo peggioramento delle prestazioni è stata studiata una tecnica che permette di implementare in modo diverso tutta la parte di ACS, in modo da rendere possibile l'utilizzo della pipeline.
Nei comparatori visti in precedenza, tranne che nel caso del double state, inizialmente vengono sommati i valori delle branch metrics e delle path metrics e successivamente per trovare il valore minimo vengono usati dei sottrattori in cui il percorso critico parte dall'LSB (Least Significant Bit) e si propaga fino al bit di segno, che viene utilizzato per sapere quale degli ingressi è maggiore e quale è minore. Come già detto con questa implementazione non è possibile utilizzare la

pipeline, a causa del loop che rende necessario avere il risultato della path metric minore pronto al colpo di clock successivo.
In questa nuova implementazione, per motivi che saranno chiari in seguito, si farà riferimento ad un tipo di comparatore diverso da quelli utilizzati in precedenza: invece che usare un sottrattore che fa i confronti ottenendo il risultato a partire dall'LSB e propagandolo fino all'MSB, verrà ipotizzato l'utilizzo di un comparatore che ottiene il minimo partendo dall'MSB.
In questo caso, il percorso critico attraversa tutti i full adders del sommatore, a causa della propagazione del carry, e poi attraversa tutti i blocchi che compongono il comparatore, dove un flag si propaga a partire dell'MSB fino all'LSB, dove viene definito il risultato.
In questo caso, mantenendo questa struttura non è possibile utilizzare la pipeline, rendendo impossibile ridurre il critical path. È però possibile spezzare il percorso critico dovuto alla propagazione del carry, utilizzando dei sommatori detti carry save adders (CSA).
Se il bit di carry c_i non venisse dato immediatamente al full adder di peso maggiore, ma venisse salvato come parte del risultato, la somma avrebbe 2 bit di peso 2^i, rispettivamente s_i e c_i:

$$S = \sum_i (s_i + c_i)2^i = \sum_i \nu_i 2^i, \nu_i \in \{0, 1, 2\} \tag{4.1}$$

Il grande vantaggio di questa somma carry save è che non esiste il carry ripple, limitando il percorso critico. Quindi, se fosse possibile realizzare anche un comparatore carry save, questa potrebbe essere la soluzione per il loop dell'ACS.
Come è possibile vedere dalla Figura 4.11, mentre nella vecchia versione il percorso critico attraversava tutti i full adder dal basso verso l'alto, per poi ridiscendere fino in fondo attraverso tutti i comparatori, nella nuova versione attraversa solo un sommatore, quello più in alto nello schema, e poi attraversa tutti i comparatori.
Nonostante il miglioramento, resta ancora un problema: il percorso critico è ancora definito dalla propagazione del flag dei comparatori, che si propaga dal blocco che analizza i bit di peso maggiore, quello che compara gli MSB, a quello riferito agli LSB. Questo fa sì che il critical path sia ancora molto lungo e soprattutto dipende dal numero di bit; infatti, come in tutti i casi precedenti, dipende direttamente dal numero di bit delle path metrics. Questo però può adesso essere risolto in modo molto semplice, nella nuova struttura infatti è possibile aggiungere la pipeline.
La Figura 4.12 mostra gli step che permettono l'inserimento della pipeline. Analizzando la figura è possibile vedere come l'utilizzo della pipeline riduca notevolmente il percorso critico, che viene ridotto all'attraversamento di due full adders e di due blocchi che effettuano la ricerca del minimo. Rispetto alle versioni precedenti questa occupa una quantità di area maggiore, ed aumenta anche la latenza sul risultato finale a causa dell'utilizzo della pipeline, ma questi svantaggi sono bilanciati da un

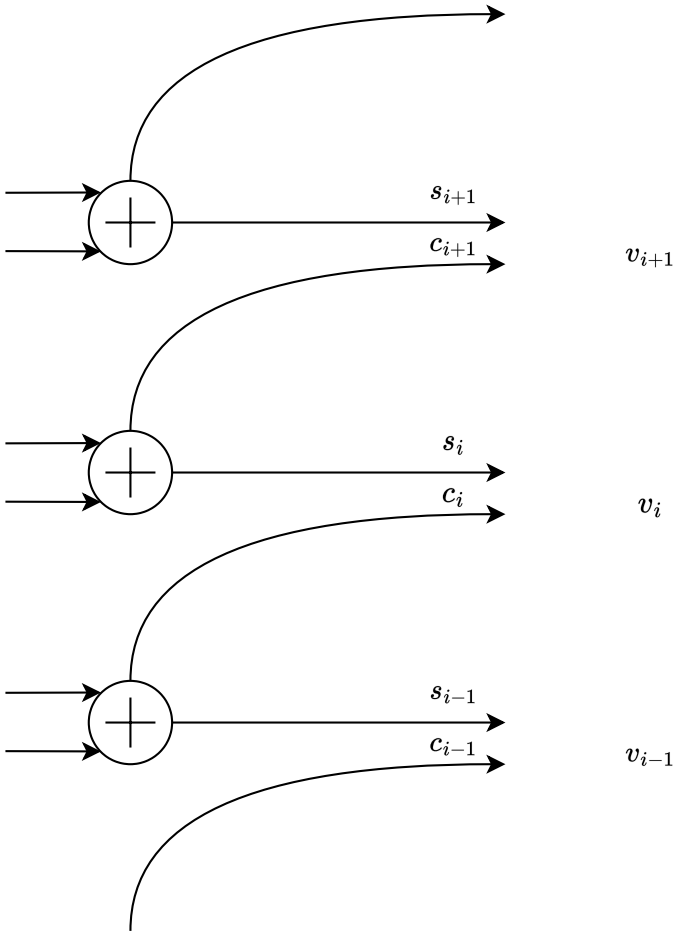

Figura 4.10: schema di un sommatore Carry Save

notevole miglioramento delle performance. Il vantaggio più importante di questa ottimizzazione è che, per la prima volta, la lunghezza del percorso critico viene resa completamente indipendente rispetto al numero di bit utilizzati per rappresentare le path metrics. Questo permette di poter utilizzare branch metrics con un numero di bit elevato, in modo da aumentarne la precisione, senza però aumentare il percorso critico, anche se questo comporta un ulteriore aumento dell'area.

Questa versione però deve ancora essere approfondita, infatti in questo caso non può essere effettuata la normalizzazione sfruttando le proprietà del complemento a due, a causa della presenza della pipeline e dell'utilizzo dei carry save adders, ma deve essere utilizzata un'altra tecnica. Inoltre, il blocco utilizzato per effettuare la comparazione non è molto semplice ed è necessario approfondire la sua implementazione.

Ad ogni colpo di clock i valori delle branch metrics vengono sommati a quelli delle

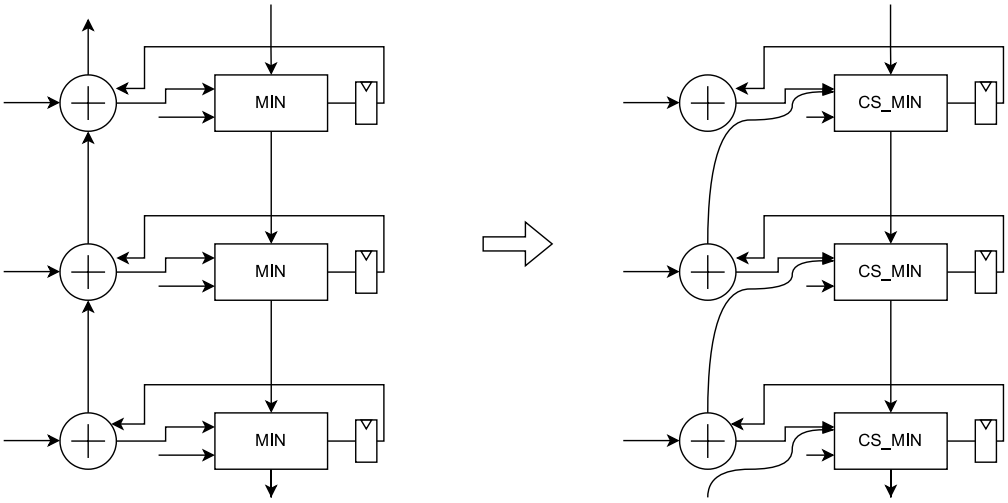

Figura 4.11: ACS con full adder e versione con i CS adder

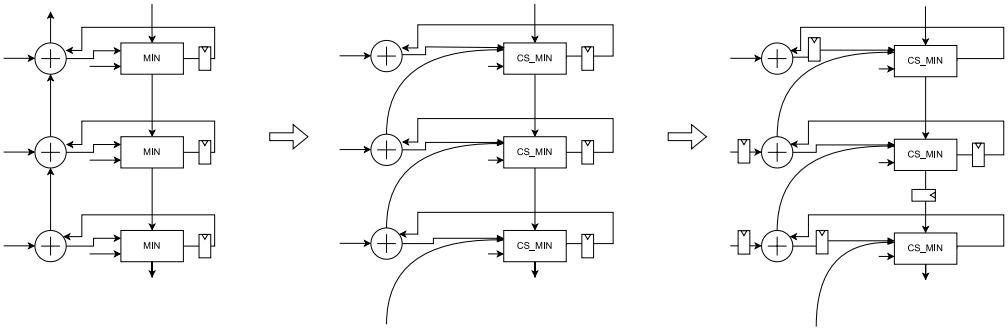

Figura 4.12: inserimento della pipeline usando i CSA

path metrics, che quindi tendono a crescere all'infinito. Per evitare l'overflow, una volta ogni tanto a tutti i valori delle path metrics deve essere sottratto un valore costante. Dimostrando che la differenza massima tra le path metrics ha un valore massimo limitato, è possibile semplificare l'operazione utilizzando solamente i valori ottenuti dal CSA relativo al most significant bit. Ogni carry save ha 3 possibili valori di uscita: 0, 1, 2. Quando in uscita si ha un 2, al colpo di clock successivo si verificherebbe l'overflow. Se però si riesce a garantire che tutti gli altri carry save relativi agli MSB abbiano come uscita 1 o 2, e non valori inferiori, è possibile sottrarre 1 da tutti questi, effettuando la normalizzazione su un unico blocco, evitando di dover lavorare sui bit di peso inferiore.
La differenza massima tra le path metrics può essere definita sapendo che, se la differenza tra due di queste è maggiore del valore massimo delle branch metrics, nella decisione successiva la path metric con il valore maggiore non sarà presa in considerazione perché il suo valore sarà sicuramente superiore a quello di almeno

un'altra, qualsiasi sia il valore di branch metrics che le viene sommato [21]. Da questo si può dedurre che, al livello delle comparazioni, la differenza massima può essere descritta come:

$$\Delta pm_{max} = 2 \cdot \max(bm) \tag{4.2}$$

considerando anche i valori che verranno sicuramente esclusi.
Come mostrato sempre nello stesso lavoro da Fettweis and Meyr [20], chiamando E_{min} il valore più piccolo di una path metric che inizializza la normalizzazione avendo il valore 2 all'MSB, questo valore deve essere più grande del valore massimo che può assumere una path metric mantenendo l'uscita dell'MSB a 0 (F_{max}), di almeno $\Delta pm_{max} + 1$:

$$\Delta pm_{max} + 1 \leq E_{min} - F_{max} \tag{4.3}$$

In questo modo viene garantito che le uscite di tutti i CSA relativi al most significant bit abbiano almeno il valore 1 e possano quindi essere normalizzate.
È chiaro che al numero di bit delle branch metrics sia necessario aggiungere un certo numero di bit per poter rappresentare le path metrics in modo tale che si verifichi la proprietà (4.3), questo valore viene identificato con h:

$$E_{min} = 2^{bm\_bit+h} \tag{4.4}$$

Nel paper a cui viene fatto riferimento è presente una tabella che contiene i valori che deve assumere h affinché si verifichi la condizione dell'Equazione 4.3. In questo caso il valore necessario è $h = 3$.
Oltre alla normalizzazione, l'altro problema è l'implementazione del blocco che ha il compito di calcolare il minimo partendo dall'MSB.
Come punto di partenza è stato utilizzato il lavoro di Parhi [22], che mostra come realizzare un blocco utilizzabile nel caso in cui si desideri calcolare il massimo, questo è stato quindi modificato in modo da permettergli di calcolare il minimo.

Come mostrato in Figura 4.13, il blocco riceve in ingresso i bit ottenuti dai carry save adders da confrontare: 4 bit di somma, dai sommatori allo stesso livello, e 4 di carry, dai sommatori relativi agli ingressi di peso immediatamente inferiore. Gli ingressi sul lato superiore dello schema sono flag utilizzati per portare indicazioni sulle decisioni prese nei blocchi ai livelli superiori, nel caso ad esempio un ingresso sia già stato escluso perché più grande degli altri, i suoi bit non verranno confrontati.
Le uscite del blocco contengono i bit che rappresentano il minimo dei valori ricevuti in ingresso, tra gli ingressi considerati validi, e le informazioni sulle decisioni prese che verranno utilizzate dal blocco al livello inferiore.
All'interno del blocco, inizialmente è necessario utilizzare un code converter (CC)

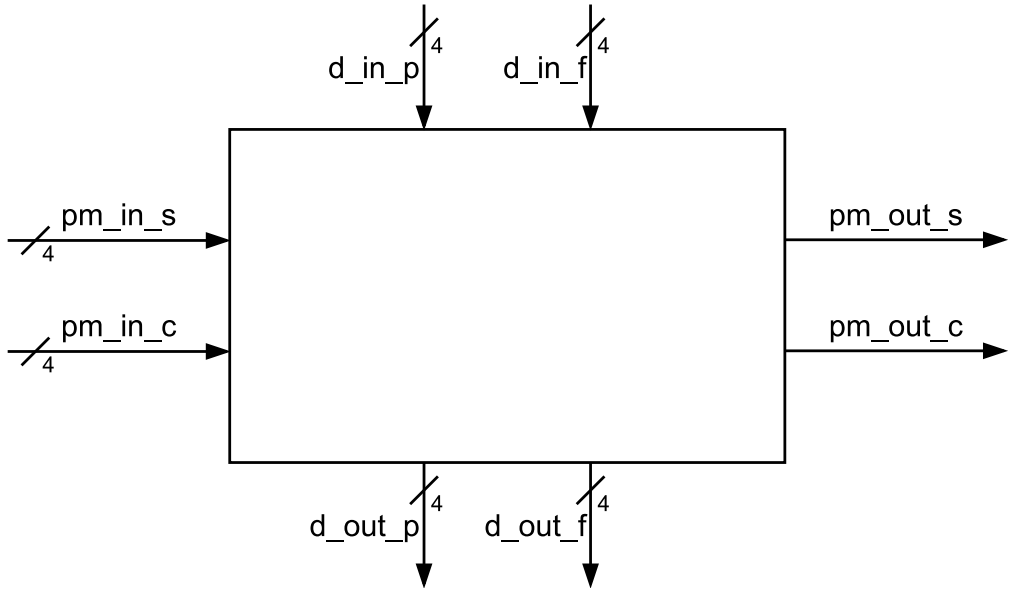

Figura 4.13: interfaccia del blocco per il calcolo del minimo

[23], per rimuovere la complessità dovuta alla ridondanza associata all'uscita 1 del carry save, che si può avere avendo ad 1 sia il bit del carry sia quello della somma (ma non entrambi). Per farlo si utilizza il circuito in Figura 4.14 che implementa la truth table rappresentata dalla Tabella 4.2.

pm_c	pm_s	pm_c^r	pm_s^r	Digit
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	2

Tabella 4.2: truth table del code converter

Osservando più nel dettaglio le decisioni, queste hanno bisogno di essere rappresentate su due bit a causa della ridondanza della rappresentazione dei numeri. Ad ogni ingresso è associato un bit relativo alla decisione preliminare (d_in_p) e uno relativo a quella finale (d_in_f).
Se entrambe le decisioni sono a 0 significa che la metrica è sicuramente maggiore della minima delle altre, quindi i suoi ingressi devono essere esclusi dal confronto, mentre se entrambi valgono 1 la metrica è valida e può essere confrontata normalmente con le altre.
Nel caso in cui il bit della decisione preliminare sia 0 e quello della decisione finale

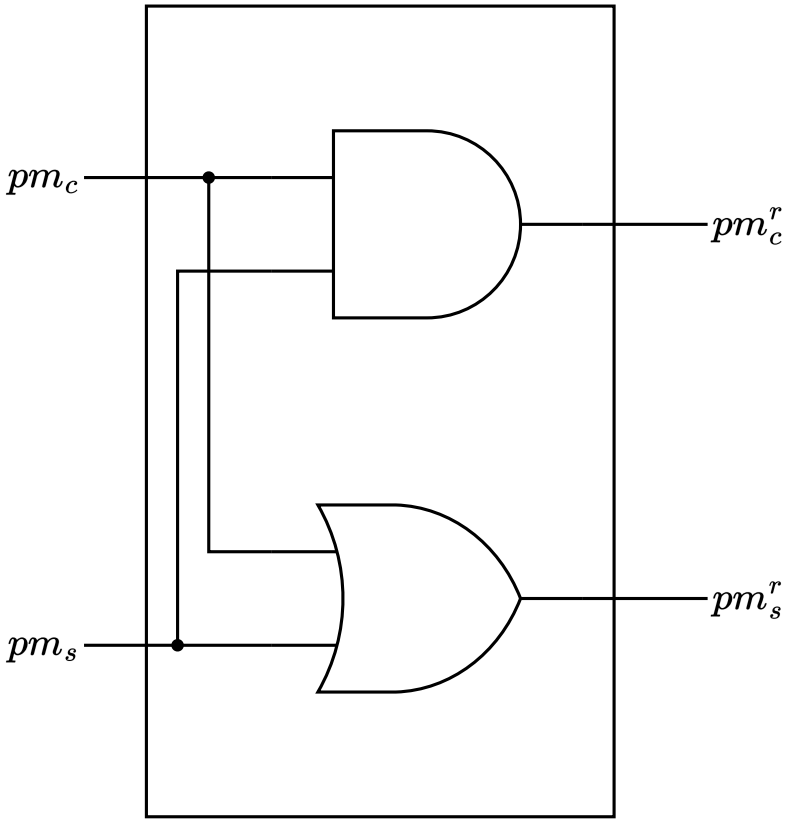

Figura 4.14: code converter

sia 1 significa che c'è una differenza $\delta^i = +1$ tra la metrica attuale di peso 2^i e il numero corrispondente al valore minimo [24]. Nel caso in cui, al livello inferiore questo bit fosse inferiore agli altri ingressi validi con una differenza $\delta^{i-1} = -2$, significherebbe che anche questo valore è ancora valido, perché le differenze si bilanciano e indicano che fino a quel livello non è ancora possibile stabilire quale sia il minimo.
Dopo a questa fase iniziale i valori delle decisioni prese al livello superiore vengono usati per generare due nuovi gruppi di segnali. Questi gruppi sono composti dai valori in uscita dal code converter, nel primo gruppo questi valori vengono soppressi in base ai valori delle decisioni provvisorie, nel secondo in base a quelli delle decisioni finali. Con soppressione si intende che entrambi i bit vengono portati a 1, in questo modo il valore è sicuramente più grande di tutti quelli validi con cui viene confrontato, e per farlo vengono utilizzate delle porte OR. I valori ottenuti dalla soppressione in base alle decisioni provvisorie vengono utilizzati per calcolare i minimi con cui devono essere confrontati i dati, per esempio, i dati derivanti dall'ingresso 0 devono essere confrontati con il minimo ottenuto tra tutti gli altri

ingressi. Questa tecnica è stata provata più efficiente dal punto di vista dell'area rispetto ad ottenere il valore del minimo confrontando i dati a coppie [24] [25]. Per trovare il valore del minimo vengono utilizzate le uscite di due porte AND a 3 ingressi, una delle quali ha come ingressi i 3 valori di somma mentre l'altra ha i valori di carry.
Il valore del minimo che uscirà invece dal blocco deve essere ottenuto a parte, dato che deve essere calcolato confrontando tutti i quattro ingressi. Anche in questo caso l'operazione viene effettuata tramite porte AND, che però devono avere quattro ingressi.
I valori delle decisioni da trasmettere al livello successivo vengono ottenute tramite un sistema di multiplexer, che tiene in considerazione le decisioni precedenti, il valore del minimo con cui ogni ingresso deve essere confrontato e i valori degli ingressi soppressi dalle decisioni finali precedenti. Il suo funzionamento viene mostrato nella Figura 4.15.

Il principale vantaggio di questa configurazione è il netto aumento delle prestazioni che è possibile avere rispetto alle soluzioni viste in precedenza. La possibilità di utilizzare la pipeline permette di avere percorsi critici dalla lunghezza controllata, pagando solamente con un aumento della latenza, potendo aumentare idealmente all'infinito il numero di bit senza che la frequenza massima utilizzabile venga ridotta. Questa implementazione però presenta anche degli svantaggi, infatti i blocchi utilizzati per calcolare i minimi sono molto più complessi dei sottrattori usati in precedenza, questo comporta un notevole aumento dell'area utilizzata, peggiorato ulteriormente dall'aumento dei registri.

4.5 Risultati Viterbi singolo (area, performance)

Tutte le ottimizzazioni descritte fino a questo momento sono state implementate a livello RTL in verilog o system verilog. In tutte le descrizioni, per mantenere la massima flessibilità, tutti i valori relativi al parallelismo o al numero di blocchi necessari per una certa implementazione sono stati definiti in modo parametrico. In questo modo tutti i valori possono essere fissati tramite il testbench, come la lunghezza della path history e il numero di bit delle branch metrics o degli ingressi, rendendo molto più semplice realizzare versioni in cui questi parametri sono diversi per verificarne le differenze.
Dopo alla realizzazione del codice relativo alle descrizioni RTL si è passati alla fase di simulazione e verifica. Questo processo è stato realizzato in modo che non presentasse differenze durante le simulazioni delle diverse implementazioni, in modo da poterlo utilizzare per verificare tutti i diversi modelli senza doverlo modificare di

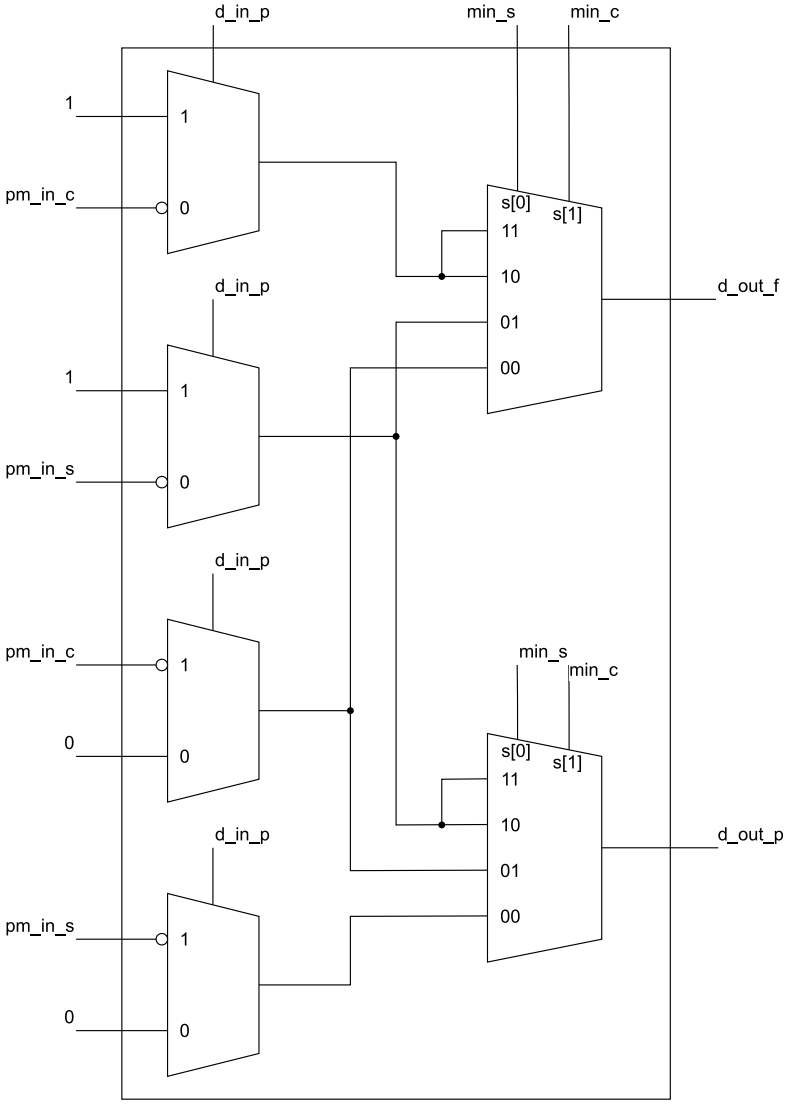

Figura 4.15: schema del blocco per la selezione delle uscite del comparatore

volta in volta. Anche il testbench utilizzato è sempre lo stesso, dato che l'interfaccia di tutte le ottimizzazioni del Viterbi Detector è sempre la stessa.
Nel testbench è necessario fissare i valori di tutti i parametri, che verranno utilizzati in tutti i blocchi contenuti in esso. L'ottimizzazione invece viene scelta in fase di compilazione. Questa fase viene eseguita tramite degli script, ognuno dei quali permette di compilare solamente i file utilizzati da una determinata ottimizzazione del dispositivo. Questi script sono realizzati come file in formato .sh, cioè file eseguibili in ambiente unix da linea di comando. Dopo che tutti i file richiesti sono stati compilati è possibile procedere con le fasi successive della simulazione.

I passaggi principali sono i seguenti:

- Creazione della sequenza di ingressi tramite script matlab;
- Salvataggio su file della sequenza di ingressi;
- Simulazione utilizzando il modello matlab;
- Salvataggio su file dei risultati ottenuti tramite il modello;
- Simulazione RTL utilizzando gli ingressi generati da matlab;
- Salvataggio dei risultati della simulazione su file;
- Confronto file dei risultati.

Tutti i passaggi appena elencati verranno approfonditi e descritti più nel dettaglio.
La sequenza di ingressi viene generata all'interno di uno script matlab. Per ottenerla viene utilizzato il generatore di numeri pseudocasuali. In questa fase è disponibile il modello di riferimento, che è stato dimostrato essere corretto, non è quindi necessario avere degli ingressi che simulino in modo realistico quelli che si trovano in uscita ad un canale, ma vengono ottenuti dei numeri casuali che hanno dei valori definiti dal range $[0, 2^{fir\_bit} - 1]$, dove fir_bit indica il parallelismo degli ingressi. In questo modo si perde la possibilità di calcolare valori come il bit error rate, ma dato che questo non è necessario, viene privilegiata la semplicità. Inoltre, utilizzando valori completamente casuali e indipendenti gli uni dagli altri è possibile coprire molte più possibilità di errore che nel caso in cui gli ingressi siano realistici. Ad esempio, se la potenza associata al rumore è molto bassa, i valori delle path metrics crescono molto lentamente, perché la presenza di pochi disturbi fa sì che sia molto più semplice trovare la transizione corretta e questo porta ad un aumento della probabilità di usare branch metrics con valori vicini allo 0. In questa situazione, nel caso in cui ci fosse un errore nella normalizzazione delle path metrics o nella gestione del loro overflow, con una simulazione non sufficientemente lunga sarebbe impossibile da rilevare.
La sequenza generata avrà una lunghezza definibile dall'utente, solitamente viene mantenuta tra i 1000 e i 10000 campioni, in modo da coprire abbastanza casi da essere certi dell'effettivo funzionamento del circuito ma senza rendere la simulazione troppo lenta.
Questa sequenza viene sia utilizzata come ingresso per il modello matlab, sia salvata su un file di testo, dopo che tutti i numeri che la compongono sono stati convertiti a valori binari. Vengono quindi calcolati i valori di uscita di riferimento utilizzando il modello. Prima di eseguire la simulazione è però necessario verificare che tutti i

parametri inseriti nello script siano corretti, è in questa fase, infatti, che vengono definiti i vari parallelismi e altri valori come la lunghezza della path history.
Il parallelismo delle branch metrics deve essere di 6 o 7 bit, in quanto verranno utilizzati dei valori di riferimento disponibili per questi casi.
Al termine della simulazione del modello i risultati vengono salvati su un file, in attesa che vengano confrontati con quelli ottenuti dalla simulazione dell'RTL.
Terminata la simulazione matlab si passa alla simulazione del circuito descritto in verilog, che è stato nel frattempo compilato. La simulazione viene effettuata tramite il software xcelium, sviluppato da Cadence. All'interno del testbench viene generato un clock, la durata del suo periodo, fissata arbitrariamente a 2 ns, al momento non è importante dato che non verranno calcolati i valori di potenza e che i ritardi dei blocchi sono considerati nulli durante la simulazione. Oltre al clock viene anche generato il reset, fondamentale dato che tutti i registri sono sensibili ad un reset asincrono attivo basso. Quando inizia la simulazione, il testbench legge sia il file con la sequenza di ingressi sia quello dove sono contenute le branch metrics. Le branch metrics vengono fornite tutte contemporaneamente alla branch metric unit, dove verrà simulato il comportamento di una memoria, mentre gli ingressi vengono forniti uno ad ogni colpo di clock, in modo che siano ricevuti dal Viterbi in modo sequenziale e possano quindi essere elaborati correttamente, simulando la normale ricezione.
L'ultimo parametro da definire nel testbench è la latenza del Viterbi utilizzato, che dipende dall'ottimizzazione scelta, in questo modo è possibile sapere dopo quanti colpi di clock le uscite del Viterbi sono valide. Dopo che è passato questo intervallo di tempo, il valore presente in uscita al Viterbi viene letto ad ogni colpo di clock e viene salvato su un file.
Tutto il lavoro viene eseguito in ambiente unix, per confrontare i file dei risultati è quindi sufficiente utilizzare il comando `diff file_matlab.txt file_RTL.txt` da linea di comando per confrontare il contenuto del file generato dal modello matlab e quello generato dal testbench. Nel caso in cui non ci siano errori il comando non deve riportare nessuna differenza.
Dopo aver verificato il corretto funzionamento di un'ottimizzazione, si passa alla sua sintesi in tecnologia FinFET a 3 nm di TSMC. Questa tecnologia garantisce un'area minima e delle performance molto elevate, essendo una delle migliori al momento disponibili.
Tutte le ottimizzazioni descritte sono state simulate e sintetizzate per i due valori consentiti di parallelismo delle branch metrics, cioè 6 e 7 bit. Gli altri parametri sono stati mantenuti costanti, in particolare la lunghezza della path history è fissata a 20 e gli ingressi hanno un parallelismo di 6 bit. In questo modo è stato possibile studiare le variazioni di area e prestazioni causate dall'utilizzo di branch metrics con parallelismo diverso. Gli effetti causati dalle variazioni di questo parametro sono tra i più importanti da conoscere, perché il suo valore determina i risultati

ottenuti dal Viterbi Detector anche per quanto riguarda il bit error rate.
Il grafico riportato in Figura 4.16 riporta i risultati ottenuti per tutte le imple-

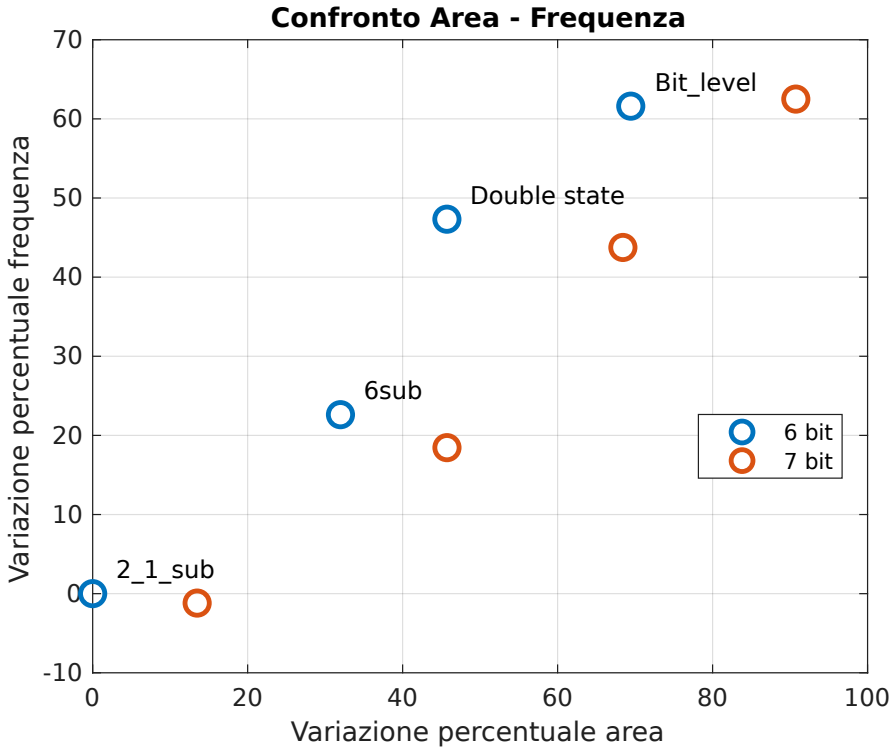

Figura 4.16: variazione percentuale di area e frequenza rispetto al modello 2_1_sub a 6 bit

mentazioni. Come era previsto l'implementazione iniziale a tre sottrattori è quella che occupa meno area ma che ha anche le prestazioni peggiori; infatti, non può lavorare a frequenze molto elevate. Le altre implementazioni presentano tutte un aumento della frequenza massima a cui possono essere utilizzate, accompagnato sempre anche da un aumento dell'area occupata.
Un'altra importante informazione che può essere ottenuta dal grafico è l'effetto causato dall'aumento del parallelismo delle branch metrics. Come si può vedere nei casi dove sono utilizzate le implementazioni a 3 sottrattori, a 6 sottrattori o la double state questo causa sia un aumento di area sia un peggioramento delle prestazioni, causato dall'aumento delle dimensioni dei sommatori e dal conseguente aumento della lunghezza del percorso critico. Nel caso in cui viene utilizzata la tecnica della bit level pipeline come previsto le prestazioni rimangono costanti, grazie appunto all'utilizzo della pipeline che fa si che il percorso critico rimanga

costante. Dal grafico si vede che con il passaggio da 6 a 7 bit le prestazioni addirittura migliorano, ma questo è probabilmente causato dal sintetizzatore che potrebbe aver applicato ottimizzazioni leggermente diverse. Rimane invece presente, come negli altri casi, l'aumento dell'area.

Capitolo 5

Implementazioni Viterbi parallelo

Dato che, come visto in precedenza, i SerDes trasmettono e ricevono dati a velocità molto elevate, con la tecnologia attuale è impossibile pensare che un singolo Viterbi Detector possa elaborare correttamente tutti i dati ricevuti. Inoltre, la maggior parte dei SerDes, per raggiungere le prestazioni necessarie, parallelizza internamente il segnale, per permettere alla parte digitale dell'equalizzazione di lavorare a frequenze ragionevoli. In particolare, nel caso studiato dopo la conversione analogico-digitale il segnale ha un parallelismo pari a 32, questo permette di lavorare ad una frequenza pari ad 1/32 rispetto a quella a cui il ricevitore riceve in ingresso. I Viterbi Detector analizzati fino a questo momento, che per semplicità verranno chiamati Viterbi singoli, non hanno la possibilità di ricevere più di un ingresso per volta, per questo è stato necessario studiare soluzioni alternative.

5.1 Viterbi parallelo

La prima possibilità è stata quella di parallelizzare il Viterbi a livello algoritmico [26]. Questa versione permette di elaborare una grande quantità di dati utilizzando più di un Viterbi in parallelo.
Questo ha però causato un notevole aumento della complessità principalmente per due motivi:

1. È necessario utilizzare un buffer in ingresso, per memorizzare tutti i dati in attesa che siano smistati al Viterbi singolo corretto.

2. Anche i dati in uscita devono essere memorizzati, questo perché è necessario attendere che siano validi in blocchi da 32 prima di essere trasmessi, in modo da avere un flusso in uscita pari a quello in ingresso.

Quando un Viterbi singolo riceve i primi dati in ingresso, i valori delle sue path metrics non sono validi, dato che gli ingressi precedenti sono stati elaborati da un'altra unità. Per questa ragione è necessario che gli venga fornito un certo numero di ingressi, detti di training, per far sì che le differenze tra i valori associati ai vari stati siano congruenti con quelle che si avrebbero nel caso in cui tutta la sequenza di dati fosse analizzata dallo stesso Viterbi. Come dimostrato da Viterbi e Omura in [21], perché questa condizione abbia una probabilità sufficientemente elevata di verificarsi, è necessario che il Viterbi elabori D dati prima che le uscite siano considerate valide, dove il valore di D è pari alla lunghezza della path history.

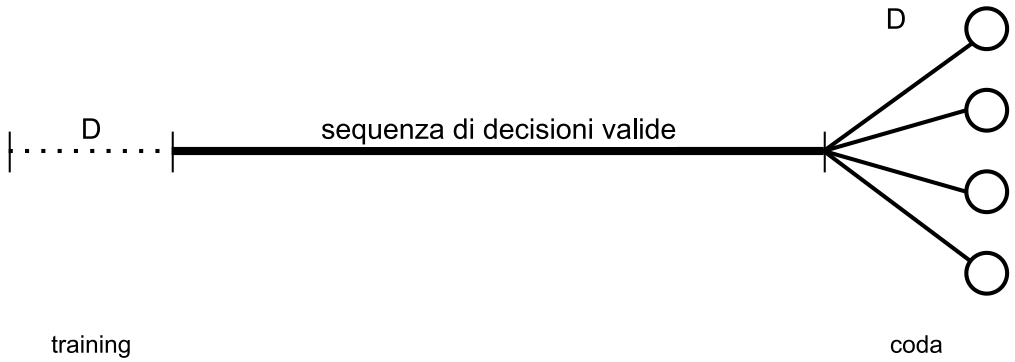

Figura 5.1: schema del path in uscita da un Viterbi singolo

Inoltre, quando il Viterbi elabora l'ultimo dato valido, le ultime D decisioni fanno ancora parte della path history all'interno della traceback unit. Per questo motivo il Viterbi ha bisogno di altri D ingressi, presi dai dati che verranno elaborati dall'unità successiva, per permettere che tutte le decisioni valide all'interno della path history raggiungano l'uscita e vengano lette. Le decisioni che rimangono all'interno della path history sono quelle che divergono portando ai 4 stati finali, per questo non è necessario che siano estratti.
Nell'implementazione realizzata ogni unità restituisce 64 valori validi, quindi quelli relativi agli ingressi ricevuti in due colpi di clock. Essendo il valore di D pari a 20, ogni Viterbi deve però analizzare 104 dati.

È possibile associare ai Viterbi un valore di efficienza:

$$\eta = \frac{ingressi\ elaborati}{ingressi\ totali} \tag{5.1}$$

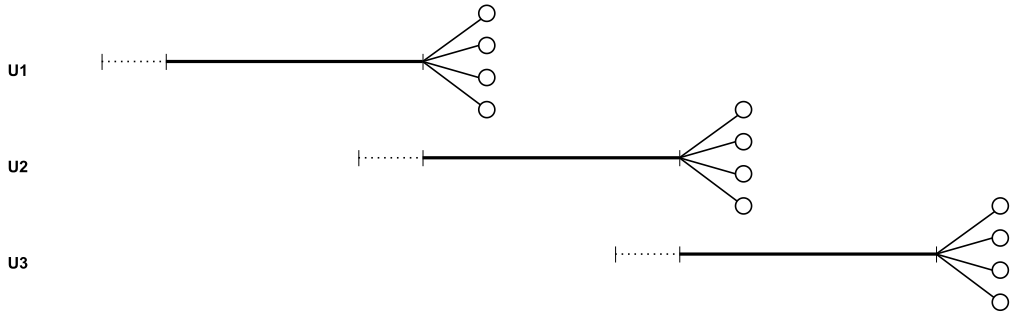

Figura 5.2: schema che mostra come le varie unità si dividono la sequenza da analizzare

Che nel caso studiato assume un valore di 0.615. Questo valore può essere incrementato facendo in modo che ogni Viterbi analizzi sequenze di dati validi più lunghe, ma questo porterebbe ad un incremento nella latenza complessiva, che è già molto elevata.
Il numero di Viterbi singoli necessari, considerando che ognuno di essi ha bisogno di 104 colpi di clock per elaborare i dati che vengono ricevuti in 2 cicli, è quindi di 52 unità. Questo numero può anche essere ottenuto dalla seguente formula, che restituisce il numero di unità necessarie:

$$NV = dati\ ricevuti \cdot \eta^{-1} \tag{5.2}$$

Perché tutti i Viterbi funzionino correttamente senza interruzioni è necessario un sistema di buffer, sia per la gestione degli ingressi sia per quella delle uscite.

Come mostrato schematicamente in Figura 5.3, gli ingressi vengono salvati in un buffer. Questo è composto da 104 unità, ognuna delle quali a sua volta è formata da 32 registri ottenuti da un numero di flip-flop pari al numero di bit degli ingressi. Ognuno dei registri ha in ingresso un bit di enable, che permette di abilitare o disabilitare la scrittura al suo interno. La scelta viene effettuata utilizzando un contatore, che incrementa il suo valore ad ogni periodo di tempo, raggiungendo il valore massimo di 103 prima di ripartire da 0. Questo viene utilizzato per selezionare la sezione corretta del buffer in cui salvare gli ingressi. Questa configurazione fa sì che gli ingressi siano salvati nella stessa unità del buffer a distanza di 104 colpi di clock, lasciando ad ogni Viterbi singolo il tempo necessario per elaborare 64 dati validi.

Ad ognuno dei 52 Viterbi è associato un multiplexer che seleziona gli ingressi corretti dal buffer. La sua struttura viene mostrata nella Figura 5.4. I primi 20 valori corrispondono agli ultimi 20 valori validi che riceve il Viterbi precedente,

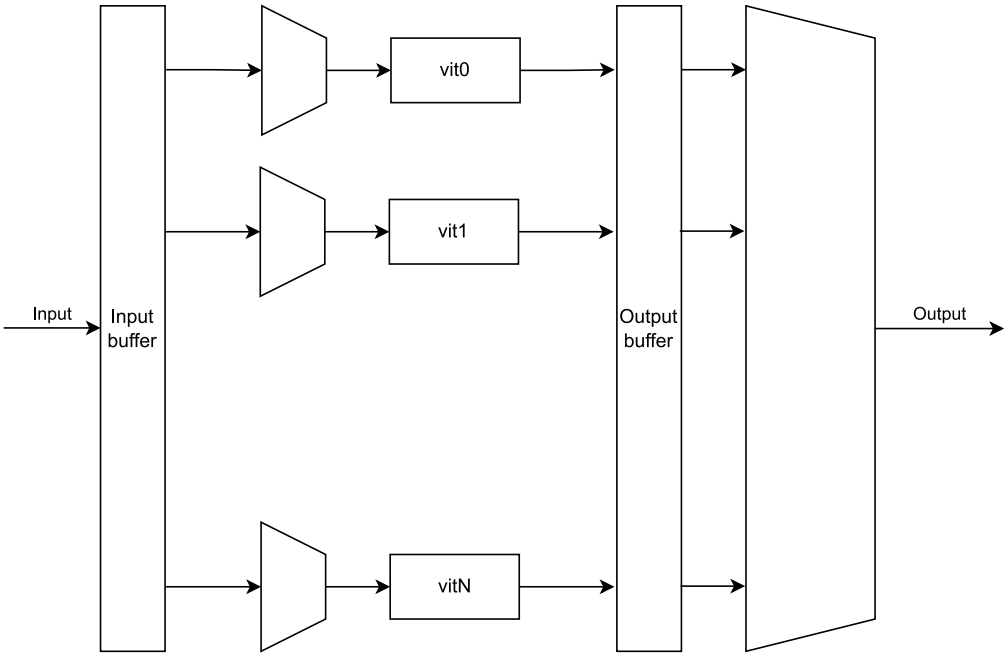

Figura 5.3: schema funzionamento del Viterbi in parallelo

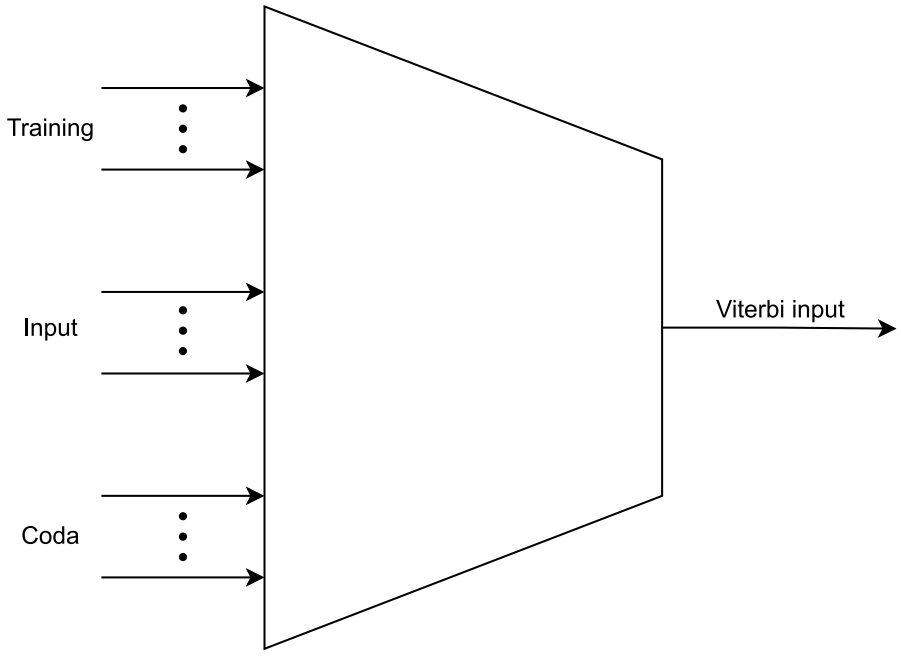

Figura 5.4: multiplexer per la selezione degli ingressi del singolo Viterbi

dopo di questi vengono presi i 64 ingressi validi dalle unità associate al Viterbi in questione, infine vengono presi gli ultimi 20 ingressi dalla prima unità relativa al Viterbi successivo. Tutti i multiplexer hanno bisogno di un contatore che controlli il selettore, permettendo di scegliere l'ingresso corretto ad ogni colpo di clock. Dato che ogni Viterbi ha bisogno di partire con 2 colpi di clock di ritardo rispetto al precedente, sarebbe estremamente complicato e consumerebbe moltissima area associare un contatore ad ognuno di essi, per semplificare il processo è stato utilizzato un unico contatore e gli ingressi sono stati collegati ai multiplexer in modo da avere uno scalamento di 2 posizioni per ogni Viterbi. In questo modo, il secondo multiplexer ha il primo ingresso valido per il training non collegato all'ingresso relativo all'indirizzo 0 del selettore ma a quello relativo all'indirizzo 2, permettendo al Viterbi di ricevere il primo dato utile con 2 colpi di clock di ritardo rispetto all'unità precedente.
In modo simile è presente un buffer anche per la gestione delle uscite, che devono essere memorizzate in attesa di ricevere tutti i risultati da un Viterbi, prima che questi possano essere mandati in uscita. In questo caso la gestione del buffer è ancora più complessa, dato che sono presenti le stesse unità composte da registri del buffer in ingresso, ma in questo caso è necessario utilizzare un sistema diverso, perché deve essere abilitato il singolo registro in cui è necessario scrivere all'interno del blocco, dato che viene ricevuto un unico valore ad ogni colpo di clock. Inoltre, non devono essere presi tutti i dati in uscita dal Viterbi, ma solo quelli corretti, escludendo quelli che derivano dagli ingressi relativi al training. Per questo motivo anche in questo caso è stato utilizzato un contatore, ed ogni unità del buffer di uscita seleziona quale registro abilitare in scrittura in base al valore ricevuto dal contatore.
Le uscite del sistema devono avere lo stesso parallelismo degli ingressi, quindi, vengono forniti 32 risultati per ogni colpo di clock. Per farlo viene utilizzato un multiplexer, controllato da un terzo contatore, che ha come ingressi i valori memorizzati nelle unità del buffer d'uscita. In questo modo ad ogni colpo di clock vengono selezionate tutte le decisioni memorizzate all'interno di un'unità e vengono mandate all'uscita.
Utilizzando 52 Viterbi in parallelo le prestazioni richieste ad ognuno di essi non sono molto elevate, riuscendo a rimanere all'interno dei vincoli temporali senza troppe difficoltà. Questo permette di utilizzare i Viterbi singoli con la prima ottimizzazione analizzata nel capitolo precedente, che utilizza solamente 3 sottrattori all'interno della PMU per effettuare la comparazione. Come visto dai risultati precedenti questa implementazione non garantisce prestazioni molto elevate, ma essendo comunque sufficienti l'obiettivo diventa quello di occupare meno area possibile, e l'ottimizzazione scelta è la migliore per questo scopo.
Come visto in precedenza, l'efficienza di utilizzo di ogni Viterbi con questa tecnica non è molto elevata, ma potrebbe aumentare facendo sì che ogni Viterbi singolo

analizzi sequenze di dati validi più lunghe. Dato che il sistema riceve ad ogni colpo di clock 32 ingressi, e che al momento le sequenze ne elaborano 64, si potrebbe pensare di aumentare la lunghezza a 128. Con una sequenza di dati validi in ingresso al Viterbi pari a 128, la formula dell'efficienza fornirebbe:

$$\eta = \frac{128}{128 + 20 + 20} = 0.762 \tag{5.3}$$

L'aumento è importante, ma questa soluzione non è la migliore, perché nonostante in questo caso il numero di Viterbi singoli venga ridotto da 52 a 42, le dimensioni dei buffer di ingresso e di uscita incrementerebbero notevolmente. In questa versione ogni Viterbi impiega 168 colpi di clock da quando riceve il primo dato a quando è nuovamente disponibile; questo significa che il buffer di ingresso deve contenere abbastanza registri per memorizzare i dati ricevuti in 168 colpi di clock, prima che questi possano essere sovrascritti nei registri già utilizzati in precedenza. L'aumento delle dimensioni dei buffer fa sì che, nonostante la riduzione del numero dei Viterbi, non ci sia un risparmio di area. Come detto in precedenza, un aumento della quantità di dati elaborati da ogni Viterbi causa anche un aumento della latenza complessiva. Questi fattori hanno fatto sì che sia stata preferita un'implementazione in cui ogni unità restituisce solamente sequenze di 64 risultati validi.

5.2 Viterbi parallelo con best selection

Come appena visto, la dimensione del sistema composto dai buffer e dai Viterbi singoli in parallelo dipende direttamente dal valore dell'efficienza η, ma migliorandolo aumentando solamente la lunghezza della sequenza di dati validi non porta ad effettivi vantaggi, anzi causa un incremento sia dell'area occupata sia della latenza necessaria per avere dati validi in uscita. Un'alternativa è quella di ridurre il numero di dati che compongono gli ingressi di training o della coda. Purtroppo, non è possibile ridurre il numero di ingressi di training, senza causare un peggioramento delle prestazioni del sistema che non sarebbe accettabile, ma è possibile diminuire la lunghezza della path history, rendendo necessaria una quantità di dati che compongono la coda inferiore.

Per farlo è possibile utilizzare la tecnica detta best selection (Figura 5.5), che permette di ridurre la lunghezza della path history, quindi la path history length, senza causare un peggioramento delle prestazioni. Invece che scegliere in modo arbitrario il registro dal quale viene presa la decisione finale, questo può essere scelto in base ai valori delle path metric. Per farlo è necessario modificare la path metric unit, che oltre alle decisioni prese deve fornire alla traceback unit anche i

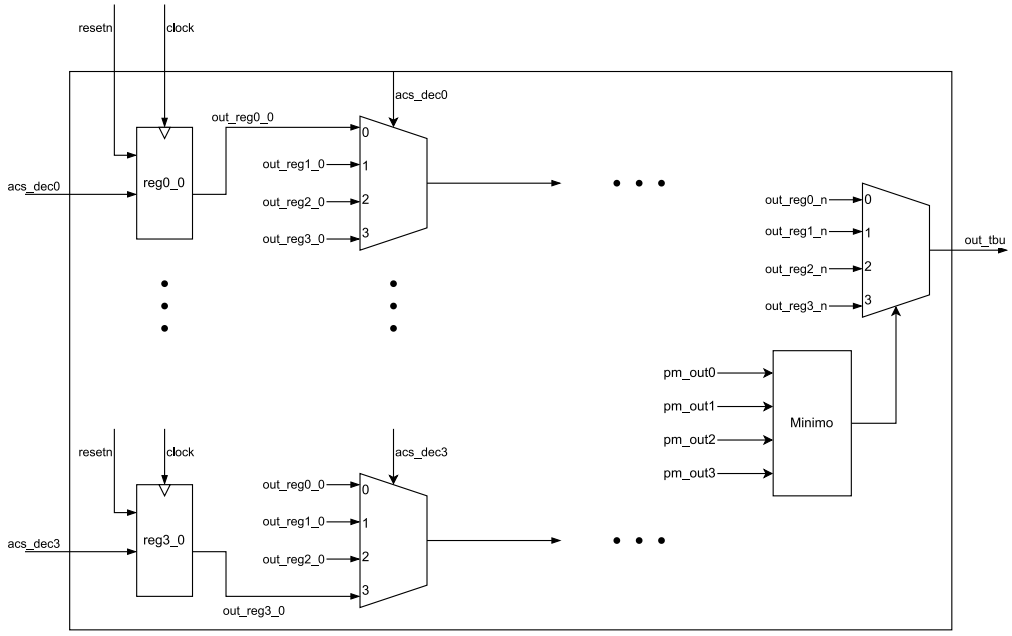

Figura 5.5: inserimento del comparatore all'interno della TBU

valori delle path metrics. Utilizzando un comparatore, all'interno della path metric unit viene cercata la path metric con valore minore, questo significa che il suo stato è quello che ha la maggior probabilità di essere corretto. Anche se tutte le decisioni nell'ultimo registro della path history non sono ancora uguali tra di loro, quella relativa allo stato scelto dal comparatore è quella più probabilmente corretta.
Perché la versione con best selection funzioni correttamente è necessario ritardare di un colpo di clock le decisioni, prima che queste vengano salvate nella path history. In questo modo le decisioni salvate e la decisione del comparatore sono correttamente allineate dal punto di vista temporale.
È stato dimostrato che l'utilizzo della tecnica della best permette di ridurre il valore della path history length da 20 a 16, senza causare un peggioramento delle prestazioni. La riduzione non è molto elevata, ma, oltre a permettere di risparmiare area sui singoli Viterbi, permette la riduzione della lunghezza della coda necessaria a svuotare la path history. In questo modo varia la quantità di dati necessari in ingresso ai singoli Viterbi, aumentandone leggermente l'efficienza, che passa ad un valore 0.64. Con questo nuovo valore di efficienza il numero necessario di Viterbi viene ridotto da 52 a 50, ed è quindi possibile ridurre le dimensioni dei buffer. Infatti, in questo caso, la riduzione del numero di Viterbi permette di utilizzare buffer formati da 100 unità, recuperando quasi il 4% di area. Oltre al vantaggio relativo all'area la nuova versione presenta anche miglioramenti per quanto riguarda la latenza complessiva, che viene ridotta.

5.3 Tecniche alternative

Per migliorare l'efficienza delle soluzioni appena descritte esistono molte possibilità. Una di queste, studiata in [27], permette di aumentare l'efficienza evitando di sprecare i valori calcolati dai Viterbi utilizzando gli ingressi che corrispondono alla coda.
A differenza delle decisioni generate dagli ingressi di training, che non sono affidabili poiché i valori delle path metrics non sono validi, quelli generati dagli ultimi ingressi sono ottenuti da path metrics corrette. In questo intervallo vengono ottenuti 4 path diversi, uno per ogni stato finale e uno di questi sarebbe il path ottimo se il Viterbi continuasse a decodificare la parte successiva del trellis. L'unica informazione che manca è quale dei 4 path sia corretto. Questa informazione può però essere ottenuta dal Viterbi singolo successivo, che la ottiene alla fine della sua fase di training.

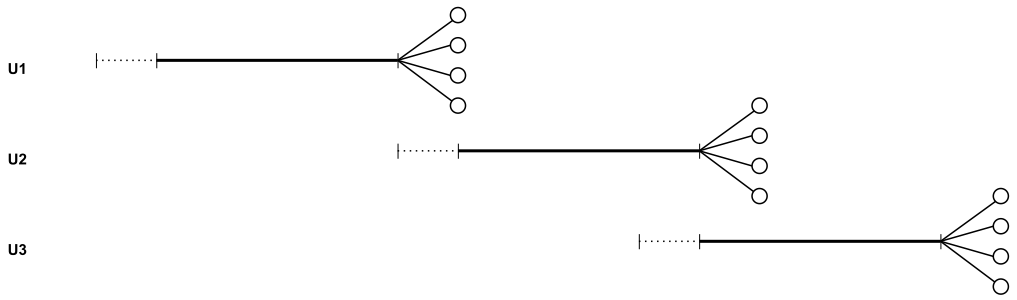

Figura 5.6: schema che mostra la divisione della sequenza per la nuova soluzione

Utilizzando gli stessi valori dei casi precedenti, con 104 ingressi ad ogni Viterbi si otterrebbero 84 uscite valide, aumentando notevolmente il valore dell'efficienza che diventa pari a:

$$\frac{84}{104} = 0.807 \tag{5.4}$$

Questa soluzione però comporta un aumento della complessità della struttura finale, dato che necessita collegamenti tra i vari Viterbi singoli. Inoltre, dato che le varie unità lavorano in parallelo, la decisione ottenuta da un Viterbi che permette di selezionare il path finale dell'unità precedente viene ottenuta con molti colpi di clock di anticipo, rispetto al momento in cui potrà essere utilizzata, rendendo necessario gestire anche la sua memorizzazione. Questi aspetti negativi hanno fatto sì che questa soluzione non sia stata implementata, preferendo concentrare gli sforzi sulla soluzione mostrata in seguito, che sembrava più promettente.

5.4 32 step Viterbi

Utilizzare i Viterbi in parallelo, come mostrato nei casi precedenti, presenta degli svantaggi principalmente dovuti alla non elevata efficienza, facendo sì che sia necessario utilizzare molti più Viterbi singoli di quelli necessari nel caso ideale. Inoltre, solo per il buffer in ingresso, nella versione descritta in precedenza, è necessario un numero di registri tale da poter memorizzare 3328 ingressi contemporaneamente. Questo evidentemente richiede un'occupazione di area e un consumo di potenza notevoli. Per questo motivo si è pensato ad una soluzione completamente diversa, che si basa sull'utilizzo di un unico Viterbi, evitando la necessità di avere ingressi di training o di coda ed eliminando completamente la necessità di utilizzare buffer in ingresso e in uscita.
Questa nuova implementazione si basa sull'idea che la PMU del Viterbi non debba necessariamente ricevere le branch metrics che contengono informazioni solamente sulle transizioni relative al passaggio tra stati in istanti di tempo consecutivi, ma possa ricevere le branch metrics relative a transizioni tra stati distanti tra loro nel tempo.
Questa soluzione si basa su quella proposta nel 1991 da Fettweis e Meyr [26].

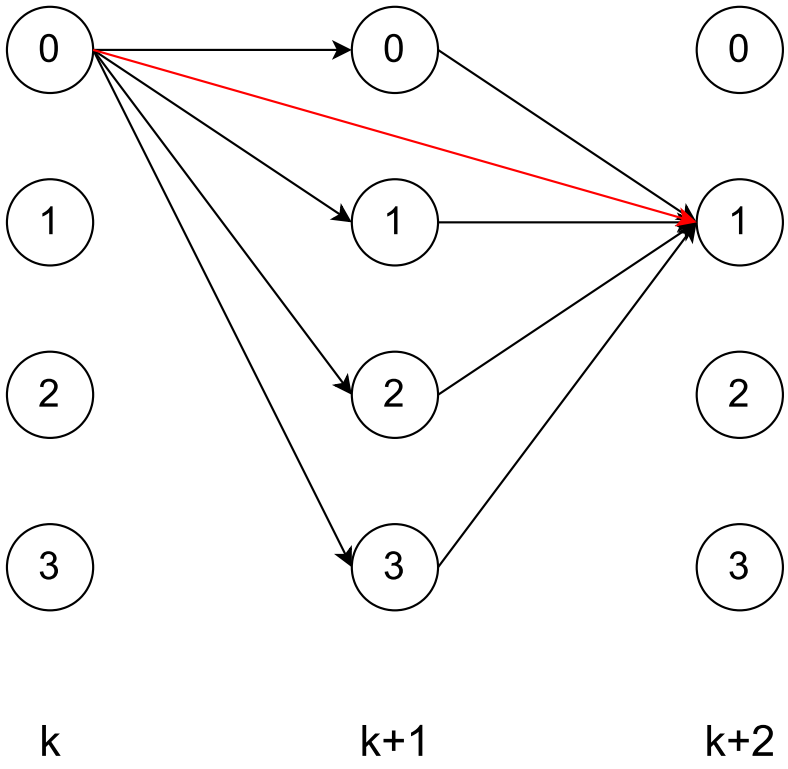

Figura 5.7: calcolo dei nuovi valori di branch metrics

Osservando gli ACS si può notare come siano effettuate solo due operazioni, la somma e la selezione del minimo. A queste è possibile applicare la proprietà distributiva nel modo seguente:

$$\min(a + c, b + c) = \min(a, b) + c \tag{5.5}$$

Dal punto di vista algebrico quindi la somma corrisponde ad una moltiplicazione e la selezione del minimo ad una somma. L'Equazione 5.5 può essere riscritta utilizzando i simboli $\otimes$ per la somma e $\oplus$ per la ricerca del minimo, in modo da rendere la notazione più semplice ed enfatizzare il fatto che sia applicata la proprietà distributiva:

$$a \otimes c \oplus b \otimes c = (a \oplus b) \otimes c \tag{5.6}$$

Utilizzando questi operatori viene formata una struttura detta semi-ring (o semianello) [28][29][30].
L'equazione che descrive l'aggiornamento delle path metrics, nel caso in cui per semplicità la sequenza sia composta solamente da due stati, è:

$$\begin{aligned} pm_{0,k+1} &= \min(bm_{00,k} + pm_{0,k}, bm_{01,k} + pm_{1,k}) \\ pm_{1,k+1} &= \min(bm_{10,k} + pm_{0,k}, bm_{11,k} + pm_{1,k}) \end{aligned} \tag{5.7}$$

Utilizzando la nuova notazione, la (5.7) diventa:

$$\begin{aligned} pm_{0,k+1} &= bm_{00,k} \otimes pm_{0,k} \oplus bm_{01,k} \otimes pm_{1,k} \\ pm_{1,k+1} &= bm_{10,k} \otimes pm_{0,k} \oplus bm_{11,k} \otimes pm_{1,k} \end{aligned} \tag{5.8}$$

L'Equazione 5.8 rappresenta una ricorsione algebrica lineare convenzionale. Due fattori permettono di riscriverla come una ricorsione vettore-matrice:

1. $\oplus$ e $\otimes$ formano un semi-ring su tutte le matrici $N \times N$, dove N è il numero degli stati;

2. Le equazioni formate dalle operazioni del semi-ring sono lineari.

Queste condizioni permettono di riscrivere l'Equazione 5.8 come:

$$\Gamma_{k+1} = \Lambda_k \otimes \Gamma_k \tag{5.9}$$

Dove Γ_k è il vettore di tutte le N path metrics al tempo $k : \Gamma_k \coloneqq (pm_{1,k}, \cdots, pm_{N,k})^T|$ e la matrice Λ_k comprende tutte le $N \times N$ branch metrics relative alla transizione $(k, k + 1)$.
L'Equazione 5.9 può essere riscritta come:

$$\begin{pmatrix} pm_0 \\ pm_1 \end{pmatrix}_{k+1} = \begin{pmatrix} bm_{00} & bm_{01} \\ bm_{10} & bm_{11} \end{pmatrix}_k \otimes \begin{pmatrix} pm_0 \\ pm_1 \end{pmatrix}_k = \begin{pmatrix} bm_{00} \otimes pm_0 \oplus bm_{01} \otimes pm_1 \\ bm_{10} \otimes pm_0 \oplus bm_{11} \otimes pm_1 \end{pmatrix}_k \quad (5.10)$$

Il vantaggio di questa notazione semi-ring è che mostra che la ricorsione dell'ACS è lineare. Questo permette di gestire l'Equazione 5.10 come un'equazione lineare. Riscrivendo l'Equazione 5.9 per l'istante $k+2$, ed inserendo al suo interno quella dell'istante $k+1$ si ottiene:

$$\Gamma_{k+2} = \Lambda_{k+1} \otimes \Gamma_{k+1} = \Lambda_{k+1} \otimes (\Lambda_k \otimes \Gamma_k) = (\Lambda_{k+1} \otimes \Lambda_k) \otimes \Gamma_k \quad (5.11)$$

Questo risultato è eccezionale, permette infatti di calcolare Γ_{k+2} direttamente da Γ_k, senza aver bisogno di conoscere Γ_{k+1}.
L'operazione fatta sull'Equazione 5.11 può essere ripetuta, arrivando al risultato più generale:

$$\Gamma_{k+M} = {}_M\Lambda_k \otimes \Gamma_k \quad (5.12)$$

Con la matrice di transizione per M step definita come:

$${}_M\Lambda_k \coloneqq \Lambda_{k+M-1} \otimes \cdots \otimes \Lambda_{k+1} \otimes \Lambda_k \quad (5.13)$$

La possibilità di effettuare una parallelizzazione algebrica è stata trovata in modo indipendente in [28] e in [31].
Utilizzando il risultato ottenuto in (5.13), la ricorsione dell'ACS è utilizzata per calcolare in un unico passaggio M step.
Si ottiene così un nuovo trellis, ottenuto combinando M step di quello originale in uno unico, come mostrato in Figura 5.8. La matrice ${}_M\Lambda_k$ descrive la transizione ottima dallo stato al tempo k a quello al tempo $k+M$.

Il calcolo delle metriche necessario per ottenere i valori finali relativi agli M step può essere effettuato sia con una struttura pipeline (Figura 5.9) sia con una struttura ad albero (Figura 5.10). Per limitare la latenza totale del sistema, nel lavoro svolto per la tesi è stata scelta la struttura ad albero. Essendo la struttura utilizzata un semianello a livello algebrico, tutte le strutture note per le ricorrenze lineari possono essere applicate (per esempio: [32]). Inoltre, la struttura è stata realizzata per il valore $M = 32$, dato che questo è il numero di ingressi ricevuti in parallelo dal sistema.

Ognuno dei blocchi mostrati in Figura 5.10 calcola i valori delle branch metrics ottenuti dalla somma di due diverse transizioni. Il suo scopo è quello di trovare il path ottimo relativo a tutte le combinazioni possibili per passare da uno stato iniziale ad uno finale. Per farlo, essendo che per ogni transizione sono possibili

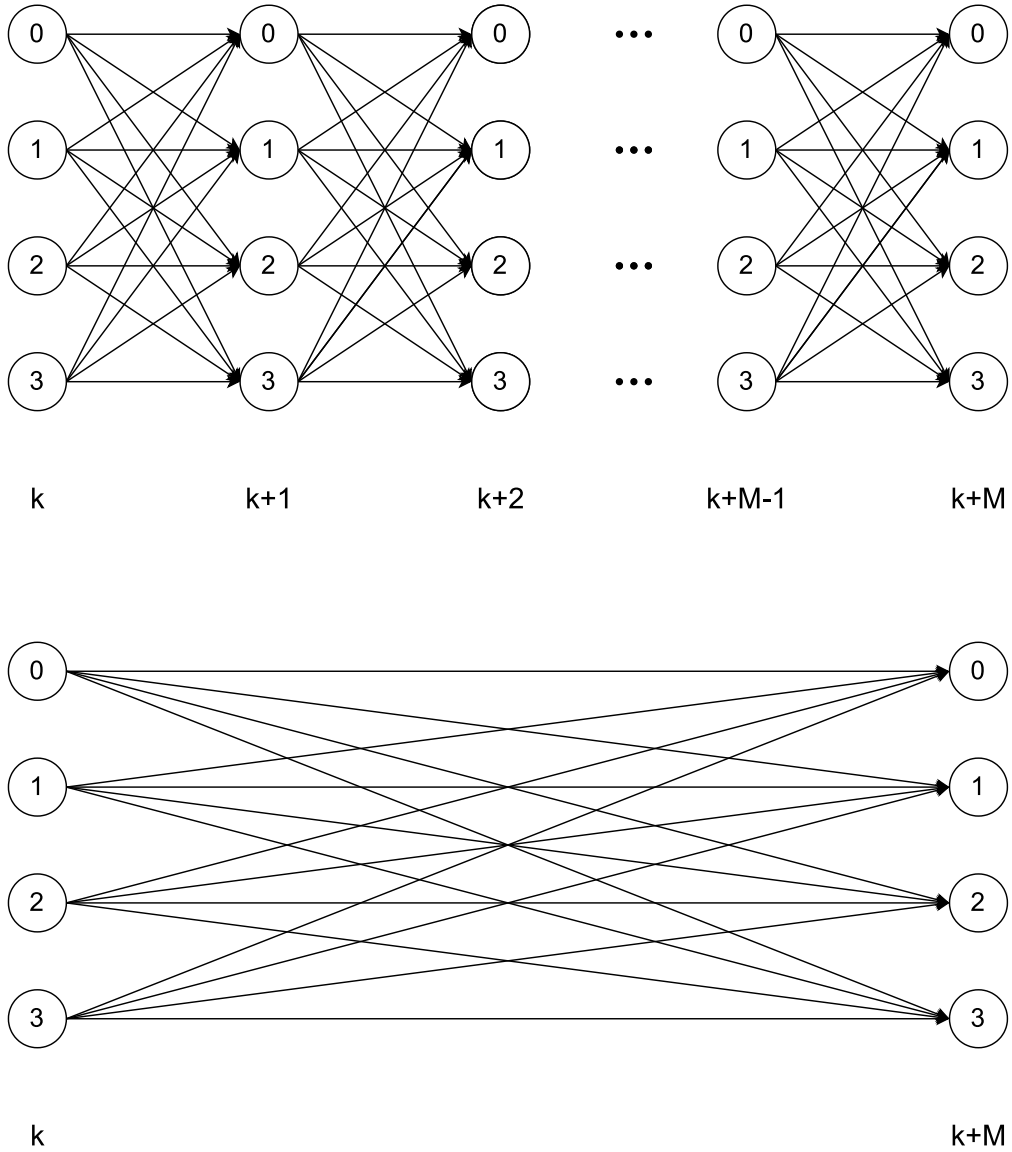

Figura 5.8: trellis originale e trellis 32 step

quattro percorsi diversi, come mostrato nella Figura 5.7, esegue il calcolo del minimo tra questi, ottenendo infine i 16 valori relativi a tutte le transizioni.
La PMU in questo caso riceve i valori delle branch metrics relative alle 16 transizioni possibili per il passaggio da uno stato a quello successivo, e ha a disposizione un colpo di clock per sommarle alle path metrics e prendere le decisioni, esattamente come nel caso classico, ma essendo questa architettura 32-step, senza ulteriori elaborazioni verrebbero perse le 31 decisioni intermedie prese durante il calcolo delle branch metrics. Per questo è importante che tutte le decisioni intermedie

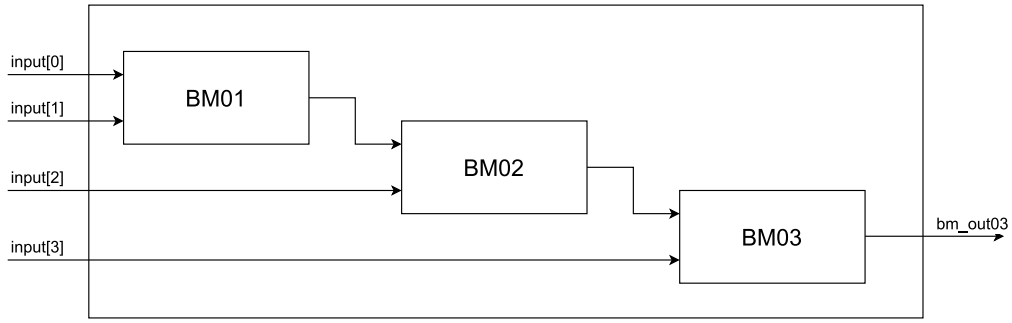

Figura 5.9: schema pipeline per il calcolo delle branch metrics

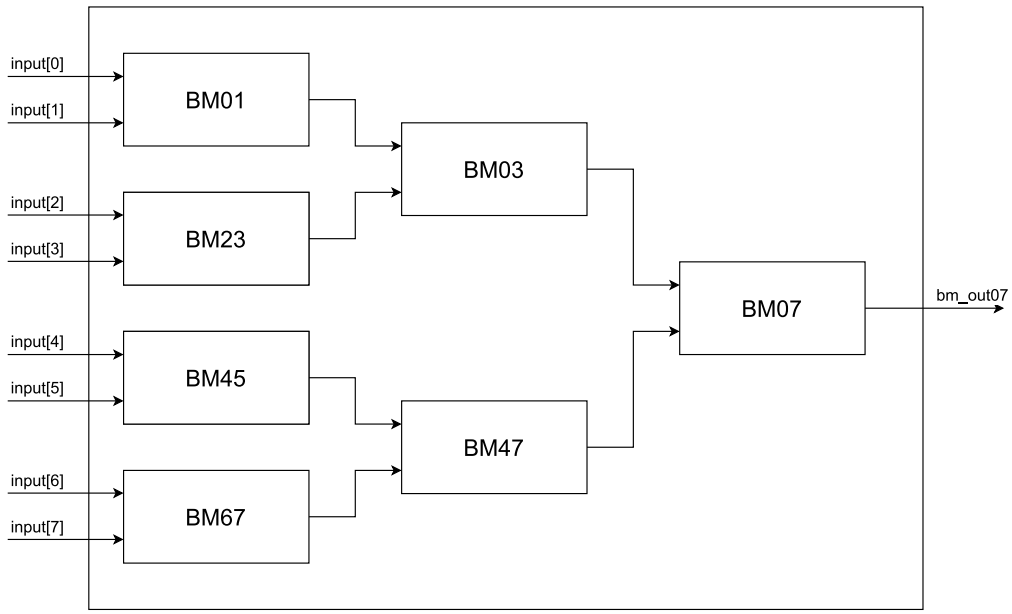

Figura 5.10: schema ad albero per il calcolo delle branch metrics

prese dalla BMU siano memorizzate; infatti, queste devono essere mandate alla traceback unit dove verranno utilizzate per ricostruire tutta la sequenza di scelte, senza che vengano perse.
All'interno della traceback unit vengono ricevute tutte le decisioni intermedie, oltre a quelle prese dalla PMU. Queste ultime possono essere utilizzate per sapere, per ogni stato finale, quale è stato scelto come stato di partenza. Conoscendo lo stato all'istante k e quello all'istante $k + 32$, è possibile ottenere lo stato intermedio $(k + 16)$ osservando la decisione che ha preso il blocco nell'ultima colonna della BMU, dove sono state calcolate le branch metrics finali usate dalla PMU, che indica lo stato intermedio della transizione. A questo punto è possibile procedere nello stesso modo e identificare gli stati intermedi tra l'istante k e $k + 16$, e tra k e

$k + 32$ dalle decisioni prese dai blocchi della colonna precedente in uno schema ad albero. Procedendo fino alle decisioni della prima colonna si ottengono le 4 sequenze di 32 decisioni che portano ai 4 stati finali. Scegliendo, tra le quattro a disposizione, la sequenza relativa allo stato 0 per ottenere primi 12 stati, questi vengono mandati direttamente all'uscita. Per lo stesso principio utilizzato nei Viterbi singoli, le prime 12 decisioni non fanno parte della path history, essendo ad una distanza dall'ultimo stato calcolato maggiore della path history length; quindi, le loro sequenze dovrebbero coincidere e quella da mandare in uscita può essere scelta in modo arbitrario. Le altre sequenze da 20 decisioni vengono memorizzate all'interno della traceback unit e vanno a formare la path history. Con questa struttura ad ogni colpo di clock vengono fornite in uscita 32 decisioni: 20 ottenute dalla path history e 12 prese direttamente da quelle appena calcolate.

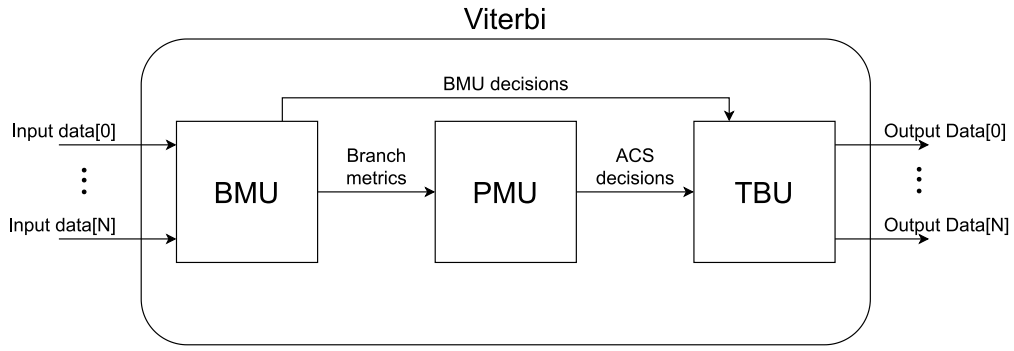

Figura 5.11: schema Viterbi Detector per implementazione 32-step

Un notevole vantaggio è dato dal fatto che, nonostante le decisioni all'interno della BMU siano prese da comparatori uguali a quelli all'interno della PMU, in questo caso non è presente il feedback, quindi è possibile utilizzare la pipeline, permettendo di avere la lunghezza del percorso critico molto simile a quella che era stata calcolata per i Viterbi singoli. La latenza di questa struttura inoltre è molto simile a quella di un Viterbi singolo, dato che funziona nello stesso modo, pagando solamente un aumento nella parte iniziale dovuto alla pipeline inserita per calcolare le branch metrics.

5.5 Confronto risultati (area, power)

Anche in questo caso, come durante lo studio dei Viterbi singoli, la procedura per la simulazione e la verifica è stata la stessa. Infatti, i nuovi modelli, anche se ricevono 32 dati allo stesso colpo di clock, li devono comunque considerare come

fossero una sequenza; per questo motivo i risultati devono essere gli stessi che vengono ottenuti analizzando i dati uno per volta. Questo ha permesso di utilizzare lo stesso modello matlab realizzato in precedenza, senza dover modificare né la generazione degli ingressi né il calcolo dei risultati. L'unica differenza, che però non ha modificato nulla nel processo, è il fatto che la lunghezza minima delle sequenze di ingressi, soprattutto per la simulazione delle implementazioni con i Viterbi in parallelo, doveva essere molto più elevata. Utilizzare sequenze da 1000 ingressi, come fatto in alcuni casi con i Viterbi singoli, non sarebbe sufficiente nemmeno a riempire il buffer d'ingresso, facendo sì che alcuni Viterbi non ricevano dati validi nella simulazione.
Per le simulazioni è stato necessario realizzare un nuovo testbench, dato che in questo caso c'è stata una variazione nell'interfaccia del Viterbi, che presenta un parallelismo diverso per gli ingressi e le uscite. Il funzionamento del nuovo testbench è però molto simile a quello precedente, con l'unica differenza che durante la lettura del file contenente gli ingressi non viene letto solo un dato ma ne vengono letti 32, allo stesso modo viene eseguita la fase di lettura dei risultati e il relativo salvataggio su file.
Le simulazioni, nel caso di utilizzo delle implementazioni con i Viterbi in parallelo, non hanno mostrato nessuna differenza nei risultati rispetto al caso in cui sia utilizzato un singolo Viterbi, mostrando che l'utilizzo della sequenza di training funziona e che la sua lunghezza è sufficiente.
Durante la verifica dell'implementazione a 32 step, il confronto dei risultati ha mostrato delle differenze rispetto a quelli ottenuti tramite il modello matlab di un Viterbi singolo. Questo è dovuto al fatto, soprattutto nel caso in cui si effettuino simulazioni di canali con un rumore molto elevato o con ingressi casuali, che c'è la possibilità che ci siano percorsi diversi con lo stesso peso. Nel caso dei Viterbi in parallelo le decisioni vengono sempre prese in modo sequenziale, uno stato per volta, nello stesso modo utilizzato da un Viterbi singolo. Per questo motivo nel caso in cui ci siano due percorsi diversi che terminano con lo stesso valore di path metrics, la decisione viene presa nello stesso modo e non si creano differenze, non è neanche possibile sapere che un altro percorso con la stessa probabilità esista. Nel caso del calcolo delle BMU per 32 step, invece, la sequenza viene generata ricostruendo la storia delle decisioni non in modo sequenziale. In questo caso, come visto in precedenza, la sequenza viene sempre divisa in due sequenze più corte, ottenendo il valore dello stato intermedio, fino a quando sono state ottenute tutte le decisioni. In questo modo però per arrivare allo stato finale, viene prima ottenuto lo stato di partenza, poi lo stato in posizione 16, poi quelli in 8 e 24, e così via. Questa differenza fa sì che, in presenza di percorsi diversi con lo stesso peso, non ci sia una garanzia che venga scelto lo stesso del Viterbi singolo.
Questi però non sono veri e propri errori, dato che la sequenza finale ottenuta è comunque una sequenza a massima verosimiglianza. Per verificare l'effettivo

funzionamento del Viterbi ed avere la certezza che alcune differenze non siano causate da errori, sono stati modificati sia il testbench che il modello matlab.
La nuova versione del modello matlab è molto simile alla precedente, ma aggiunge la possibilità di salvare su un file anche i valori delle path metrics, la stessa cosa viene fatta anche dal testbench con i valori ottenuti dalla PMU durante la simulazione. Non è possibile confrontare direttamente i file ottenuti, dato che il modello matlab salva le path metrics relative ad ogni istante temporale, mentre il testbench le ottiene solo per i valori ottenuti dalla PMU, che si aggiornano ad intervalli di 32 step. Confrontando i valori ottenuti dalla simulazione RTL con quelli corrispondenti del modello matlab è stato possibile verificare l'effettivo funzionamento di questa implementazione.
Per avere un'ulteriore conferma è stato effettuato un altro test tramite matlab. Utilizzando uno script vengono lette le sequenze di decisioni prese sia dal modello matlab sia dalla simulazione RTL; avendo a disposizione anche la sequenza di ingressi su cui è stata basata la simulazione, è possibile ottenere il valore della branch metric relativa ad ogni transizione. Sommando tutti i valori delle branch metrics relative alle sequenze che non coincidono è possibile ottenere il valore del peso associato alla verosimiglianza di entrambi i casi, se questo è lo stesso quando i percorsi tornano a coincidere significa che la verosimiglianza è la stessa e che quindi entrambi sono validi.
Grazie anche a quest'ultima verifica è stato possibile confermare l'effettivo funzionamento di tutte le implementazioni.
A differenza delle implementazioni dei Viterbi singoli, dove sono stati ottenuti i valori di area e le massime performance raggiungibili, in questo caso è stato verificato che i dispositivi riuscissero a lavorare alla frequenza imposta dal resto del SerDes, dopodiché è stato mantenuto questo valore costante procedendo per ottenere, tramite le sintesi, i valori di area occupata.
Infine, utilizzando delle sequenze di ingressi con valori realistici per un canale con coefficienti $h0 = 0.6$, $h1 = 0.4$ ed elevato rumore, sono stati generati i file in formato `vcd` (value change dump), che contengono tutte le variazioni dei segnali che si sono verificate durante la simulazione. Questi file vengono utilizzati all'interno del software Joules, sempre sviluppato da Cadence, per ottenere una stima della potenza consumata dai dispositivi precedentemente sintetizzati.
In questo modo è stato possibile effettuare un confronto per visualizzare, a parità di prestazioni, l'area occupata e il consumo di potenza delle diverse implementazioni.

La Figura 5.12 mostra la variazione percentuale dell'area e della potenza, mantenendo come riferimento i valori ottenuti per la versione con i Viterbi in parallelo con branch metrics a 6 bit, senza utilizzare la best selection. Come si può vedere dal grafico, per tutte le implementazioni un aumento del parallelismo interno comporta un aumento sia dell'area sia della potenza. Questo risultato era prevedibile, dato

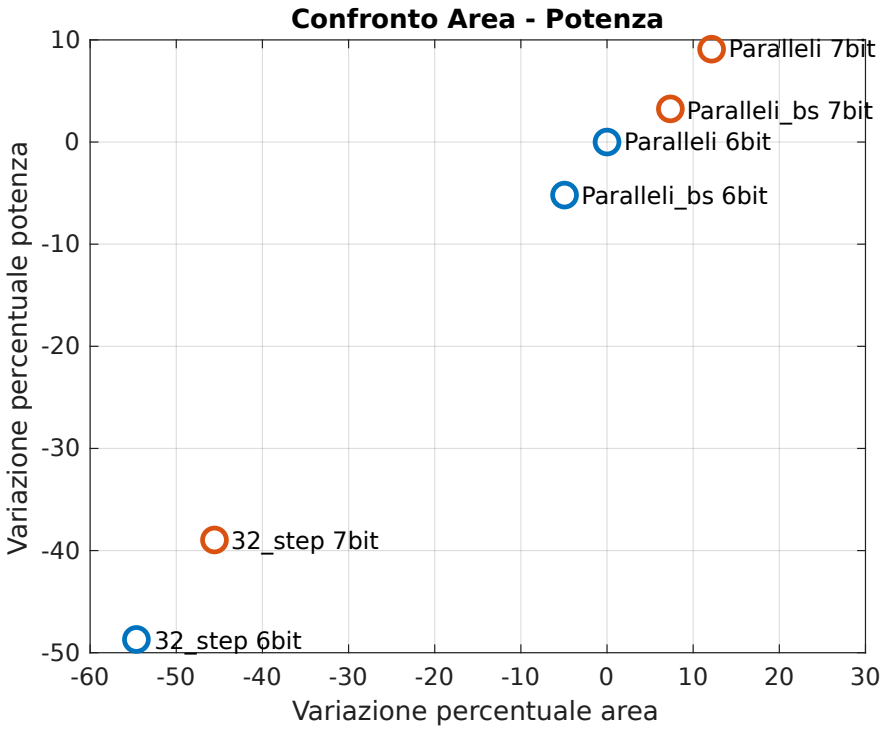

Figura 5.12: variazione percentuale di area e potenza rispetto al modello parallelo a 6 bit

che è in linea con quelli ottenuti per i Viterbi singoli. Nel caso in cui si utilizzi la best selection, la riduzione delle dimensioni dei buffer e della lunghezza della path history delle singole unità permette una riduzione di area e di potenza. Non essendo però possibile diminuire di molto il numero degli ingressi, potendo ridurre solo quelli necessari a svuotare la path history, l'efficienza del circuito non aumenta di molto. Come si può vedere dal grafico, infatti, i guadagni sono abbastanza ridotti, raggiungendo riduzioni inferiori al 10% sia per l'area che per la potenza. Utilizzando la versione 32 step invece si può vedere che i vantaggi sono notevoli. Nonostante l'incremento di complessità necessario a calcolare le branch metrics, grazie alla possibilità di rimuovere completamente sia i buffer d'ingresso che quelli d'uscita e grazie al fatto che in questo caso si utilizza un unico Viterbi, la riduzione sia dell'area occupata che della potenza consumata raggiunge quasi il 50%.

Capitolo 6

Conclusioni

I risultati ottenuti dagli studi fatti per questa tesi sono molto promettenti. Come prima cosa i modelli matlab hanno confermato i miglioramenti che comporta l'utilizzo del Viterbi Detector rispetto agli equalizzatori lineari a livello di bit error rate. Questi lo rendono un dispositivo molto valido da utilizzare all'interno dei SerDes, soprattutto nel caso in cui siano realizzati per comunicazioni a lunga distanza, dove la ricostruzione del segnale originale è più critica. In questi casi, nonostante il Viterbi occupi una quantità di area più elevata rispetto agli equalizzatori lineari più classici, il miglioramento nelle prestazioni che comporta il suo utilizzo può essere fondamentale.
A livello di design dei singoli Viterbi i risultati ottenuti mostrano che non ci sia un'implementazione migliore delle altre a livello assoluto. Le quattro possibilità studiate possono tutte essere valide a seconda delle prestazioni richieste. Per questo motivo i risultati ottenuti possono essere molto utili nel momento in cui sia necessario decidere quale implementazione usare in base alle limitazioni di area o performance richieste dal progetto.
Nell'ultima fase, cioè nella realizzazione di un Viterbi Detector con un'interfaccia in cui non viene ricevuto un unico ingresso ma 32 in parallelo, a differenza che nel caso delle implementazioni del Viterbi singolo, i risultati hanno mostrato come un'implementazione sia nettamente migliore delle altre. La versione 32 step, infatti, permette di risparmiare sia area sia potenza rispetto alle altre possibilità mostrate. Questa implementazione si è dimostrata la migliore qualsiasi sia l'obiettivo finale del progetto. La versione 32 step inoltre è molto flessibile, infatti i risultati attuali sono stati ottenuti considerando di lavorare ad una frequenza definita, ma nel caso in cui sia necessario utilizzarla a frequenze più elevate, anche se la struttura attuale non potesse supportarle, sarebbe possibile migliorarla abbastanza facilmente. Infatti, nella fase del calcolo delle branch metrics è possibile sia aggiungere livelli di pipeline intermedi ai singoli blocchi, non essendoci in quel caso un feedback, sia utilizzare comparatori che garantiscono prestazioni superiori. A livello di PMU invece le

ottimizzazioni che possono essere utilizzate sono le stesse che sono state descritte nel capitolo relativo ai Viterbi singoli.
In conclusione, i risultati ottenuti mostrano come il Viterbi Detector possa essere una soluzione valida da utilizzare all'interno dei SerDes, grazie al fatto che può garantire prestazioni migliori rispetto alle altre soluzioni. È pero necessario continuare a studiare nuove soluzioni che permettano di ottenere miglioramenti ulteriori a livello di area e consumo di potenza, per rendere il suo utilizzo ancora più vantaggioso.

Bibliografia

[1] IEA. *Data Centres and Data Transmission Networks*. 2022. URL: https://www.iea.org/reports/data-centres-and-data-transmission-networks (cit. a p. 1).

[2] Paul McLellan. *How to Build a Data Center: It's All About the SerDes...and Thermal*. 2021. URL: https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/builddatacenter (cit. a p. 3).

[3] Timothy Prickett Morgan. *The road to 400G ethernet is paved with Bechtolsheim's intentions*. 2018. URL: https://www.nextplatform.com/2018/02/21/road-400g-ethernet-paved-bechtolsheims-intentions/ (cit. a p. 4).

[4] D. R. Stauffer, J. T. Mechler, M. Sorna, K. Dramstad, C. R. Ogilvie, A. Mohammad e J. Rockrohr. *High Speed Serdes Devices and Applications*. Springer, 2008 (cit. a p. 10).

[5] C.A. Belfiore e J.H. Park. «Decision feedback equalization». In: *Proceedings of the IEEE* 67.8 (1979), pp. 1143–1156. DOI: 10.1109/PROC.1979.11409 (cit. a p. 11).

[6] J.G. Proakis. *Digital Communications*. Electrical engineering series. McGraw-Hill, 2001. ISBN: 9780072321111. URL: https://books.google.it/books?id=sbr8QwAACAAJ (cit. a p. 13).

[7] A. Viterbi. «Error bounds for convolutional codes and an asymptotically optimum decoding algorithm». In: *IEEE Transactions on Information Theory* 13.2 (1967), pp. 260–269. DOI: 10.1109/TIT.1967.1054010 (cit. a p. 14).

[8] G.D. Forney. «The viterbi algorithm». In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278. DOI: 10.1109/PROC.1973.9030 (cit. alle pp. 14, 18).

[9] «Solutions of the Shortest-Route Problem—A Review». In: *Oper. Res.* 8.2 (apr. 1960), pp. 224–230. ISSN: 0030-364X. DOI: 10.1287/opre.8.2.224. URL: https://doi.org/10.1287/opre.8.2.224 (cit. a p. 16).

[10] K. S. Gilhousen, J. A. Heller, I. M. Jacobs e A. J. Viterbi. *Coding systems study for high data rate telemetry links*. Rapp. tecn. NAS2-6024. San Diego, California: AMES research center NASA, gen. 1971 (cit. a p. 18).

[11] Jr. Clark George C. *Implementation of Maximum Likelihood Decoders for Convolutional Codes*. 1971. URL: http://hdl.handle.net/10150/607058 (cit. a p. 18).

[12] G. Forney. «Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference». In: *IEEE Transactions on Information Theory* 18.3 (1972), pp. 363–378. DOI: 10.1109/TIT.1972.1054829 (cit. a p. 18).

[13] G. David Forney. «Convolutional codes II. Maximum-likelihood decoding». In: *Information and Control* 25.3 (lug. 1974), pp. 222–266. DOI: 10.1016/s0019-9958(74)90870-5. URL: https://doi.org/10.1016/s0019-9958(74)90870-5 (cit. a p. 18).

[14] A.P. Hekstra. «An alternative to metric rescaling in Viterbi decoders». In: *IEEE Transactions on Communications* 37.11 (1989), pp. 1220–1222. DOI: 10.1109/26.46516 (cit. a p. 18).

[15] S. Lin e D.J. Costello. *Error Control Coding: Fundamentals and Applications*. Computer applications in electrical engineering series. Prentice-Hall, 1983. ISBN: 9780132837965. URL: https://books.google.it/books?id=autQAAAAMAAJ (cit. a p. 19).

[16] P.J. Black e T.H. Meng. «A 140-Mb/s, 32-state, radix-4 Viterbi decoder». In: *IEEE Journal of Solid-State Circuits* 27.12 (1992), pp. 1877–1885. DOI: 10.1109/4.173118 (cit. a p. 38).

[17] Inkyu Lee e J.L. Sonntag. «A new architecture for the fast Viterbi algorithm». In: *Globecom '00 - IEEE. Global Telecommunications Conference. Conference Record (Cat. No.00CH37137)*. Vol. 3. 2000, 1664–1668 vol.3. DOI: 10.1109/GLOCOM.2000.891920 (cit. a p. 40).

[18] K.K. Parhi. «High-speed architectures for algorithms with quantizer loops». In: *IEEE International Symposium on Circuits and Systems*. 1990, 2357–2360 vol.3. DOI: 10.1109/ISCAS.1990.112483 (cit. a p. 43).

[19] M.P.C. Fossorier e Shu Lin. «Differential trellis decoding of convolutional codes». In: *IEEE Transactions on Information Theory* 46.3 (2000), pp. 1046–1053. DOI: 10.1109/18.841183 (cit. a p. 43).

[20] G. Fettweis e H. Meyr. «A 100 Mbit/s Viterbi decoder chip: novel architecture and its realization». In: *IEEE International Conference on Communications, Including Supercomm Technical Sessions*. 1990, 463–467 vol.2. DOI: 10.1109/ICC.1990.117124 (cit. alle pp. 43, 47).

[21] A. J. Viterbi e J. K. Omura. *Principles of digital communication and coding*. McGraw-Hill, 1979 (cit. alle pp. 47, 57).

[22] K.K. Parhi. «An improved pipelined MSB-first add-compare select unit structure for Viterbi decoders». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 51.3 (2004), pp. 504–511. DOI: 10.1109/TCSI.2004.823657 (cit. a p. 47).

[23] A.K. Yeung e J.M. Rabaey. «A 210 Mb/s radix-4 bit-level pipelined Viterbi decoder». In: *Proceedings ISSCC '95 - International Solid-State Circuits Conference*. 1995, pp. 88–89. DOI: 10.1109/ISSCC.1995.535288 (cit. a p. 48).

[24] T. Gemmeke, M. Gansen e T.G. Noll. «Implementation of scalable power and area efficient high-throughput Viterbi decoders». In: *IEEE Journal of Solid-State Circuits* 37.7 (2002), pp. 941–948. DOI: 10.1109/JSSC.2002.1015694 (cit. alle pp. 49, 50).

[25] V.S. Gierenz, O. Weiss, T.G. Noll, I. Carew, J. Ashley e R. Karabed. «A 550 Mb/s radix-4 bit-level pipelined 16-state 0.25-/spl mu/m CMOS Viterbi decoder». In: *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors*. 2000, pp. 195–201. DOI: 10.1109/ASAP.2000.862390 (cit. a p. 50).

[26] G. Fettweis e H. Meyr. «High-speed parallel Viterbi decoding: algorithm and VLSI-architecture». In: *IEEE Communications Magazine* 29.5 (1991), pp. 46–55. DOI: 10.1109/35.79382 (cit. alle pp. 56, 64).

[27] G. Fettweis e H. Meyr. «Feedforward architectures for parallel viterbi decoding». In: *Journal of VLSI signal processing systems for signal, image and video technology* 3.1-2 (giu. 1991), pp. 105–119. DOI: 10.1007/bf00927838. URL: https://doi.org/10.1007/bf00927838 (cit. a p. 63).

[28] G. Fettweis e H. Meyr. «High-rate Viterbi processor: a systolic array solution». In: *IEEE Journal on Selected Areas in Communications* 8.8 (1990), pp. 1520–1534. DOI: 10.1109/49.62830 (cit. alle pp. 65, 66).

[29] G. Fettweis, L. Thiele e G. Meyr. «Algorithm transformations for unlimited parallelism». In: *IEEE International Symposium on Circuits and Systems*. 1990, 1756–1759 vol.3. DOI: 10.1109/ISCAS.1990.111973 (cit. a p. 65).

[30] Michael Yoeli. «A Note on a Generalization of Boolean Matrix Theory». In: *The American Mathematical Monthly* 68.6 (giu. 1961), p. 552. DOI: 10.2307/2311149. URL: https://doi.org/10.2307/2311149 (cit. a p. 65).

[31] Kuei Ann Wen e Jau Yien Lee. «Parallel processing for Viterbi algorithm». In: *Electronics Letters* 24.17 (1988), p. 1098. DOI: 10.1049/el:19880745. URL: https://doi.org/10.1049/el:19880745 (cit. a p. 66).

[32] K. Parhi e D. Messerschmitt. «Concurrent cellular VLSI adaptive filter architectures». In: *IEEE Transactions on Circuits and Systems* 34.10 (1987), pp. 1141–1151. DOI: 10.1109/TCS.1987.1086048 (cit. a p. 66).