Università degli studi di Firenze
Facoltà di Scienze Matematiche Fisiche e Naturali

Tesi di laurea in Fisica

Algoritmi di compressione per il rivelatore tracciante dello spettrometro PAMELA

di
Massimiliano Badi

Relatore: dott. Oscar Adriani

Anno accademico 1999/2000
24 Aprile 2001

Indice

Introduzione

Questa tesi tratta l'analisi dei vari metodi di compressione dati che possono essere applicati allo spettrometro magnetico PAMELA.

PAMELA è un esperimento della durata di tre anni volto all'analisi dei raggi cosmici e si terrà a bordo di un satellite in orbita polare. Verranno effettuate in particolare misure di spettro energetico delle diverse componenti dei raggi cosmici. Si ricercheranno inoltre eventuali tracce di antimateria presente in essi.

La necessità di comprimere i dati è dovuta alla mole di dati provenienti dalla rivelazione delle particelle che attraversano il telecopio (eventi) in rapporto alle possibilità di trasmissione a terra del satellite. Essendo quest'ultima limitata non si può che ridurre la quantità di dati da trasmettere: questa operazione deve essere fatta direttamente a bordo del satellite. Il contenuto dei vari capitoli di questa tesi è il seguente:

- **Capitolo 1**: PAMELA. In questo capitolo vengono illustrate le caratteristiche del telescopio PAMELA, con particolare riguardo al rivelatore a microstrisce di silicio, che è il rivelatore piú importante ai fini di questa tesi.

- **Capitolo 2**: Algoritmi di compressione. Qui viene descritta la compressione dei dati in generale e vengono presi in considerazione alcuni algoritmi in particolare (sia distruttivi che non distruttivi) rilevanti ai fini di questa tesi.

- **Capitolo 3**: Applicazioni allo spettrometro tracciante PAMELA. Qui si prende in considerazione l'implementazione degli algoritmi di compressione nei processori di segnali digitali di PAMELA. Essendo questo tipo di processori ad operare la compressione dati per il rivelatore a microstrisce di silicio, in questo capitolo se ne analizza l'architettura interna e le caratteristiche salienti.

- **Capitolo 4**: Risultati utilizzando i rivelatori di PAMELA. In questo capitolo si descrivono le prove effettuate su fascio nel luglio 2000 al CERN di Ginevra e dei laboratori a Firenze. Inoltre vengono descritti gli apparati sperimentali utilizzati nei due casi ed i risultati ottenuti.

Capitolo 1

PAMELA

Il progetto PAMELA, nato come parte del progetto WIZARD [1], prevede la costruzione di un telescopio magnetico per lo studio dei raggi cosmici nello spazio da lanciare in orbita a bordo di un satellite di costruzione russa. Esso è stato preceduto da numerosi esperimenti a bordo di pallone.

Gli obiettivi principali di PAMELA sono la ricerca di antimateria nei raggi cosmici e la misura dello spettro energetico delle varie componenti dei raggi cosmici per lo studio di propagazione e accelerazione dei raggi cosmici.

In particolare lo studio della componente di antimateria nei raggi cosmici è utile per determinare l'esistenza di eventuali sorgenti primarie galattiche o cosmologiche.

Più in dettaglio gli obiettivi principali del telescopio PAMELA sono: la misura dello spettro di antiprotoni da 80 MeV a 190 GeV e dei positroni da 50 MeV fino ad oltre 270 GeV, la ricerca di antinuclei di elio con una sensibilità di circa 10^{-7} nel rapporto antielio/elio ed il monitoraggio dell'attività solare ed in particolare dei *flare* [2] [3].

Il telescopio PAMELA (fig. 1.1) è costituito da un sistema di tempo di volo (*time of flight* o TOF), un rilevatore di radiazione di transizione (*transition radiation detector* o TRD), uno spettrometro magnetico ed un calorimetro. È poi presente un sistema di anticoincidenze a scintillatore per la reiezione delle particelle che non entrano nell'accettanza del telescopio.

L'alimentazione del telescopio è data da un sistema di pannelli solari disposto all'esterno del satellite che, oltre al telescopio PAMELA, deve rifornire gli altri sistemi del satellite. Questo limita la disponibilità di potenza per l'esperimento ad un valore di 345 W.

Conviene fissare delle convenzioni per le direzioni all'interno del telescopio. Il campo magnetico viene generato da un magnete permanente a torre con al centro una cavità a sezione rettangolare. Come asse Z prendiamo l'asse del telescopio, dove il TRD viene considerato "in alto", mentre il calorimetro viene considerato "in basso" (vedere fig. 1.1).

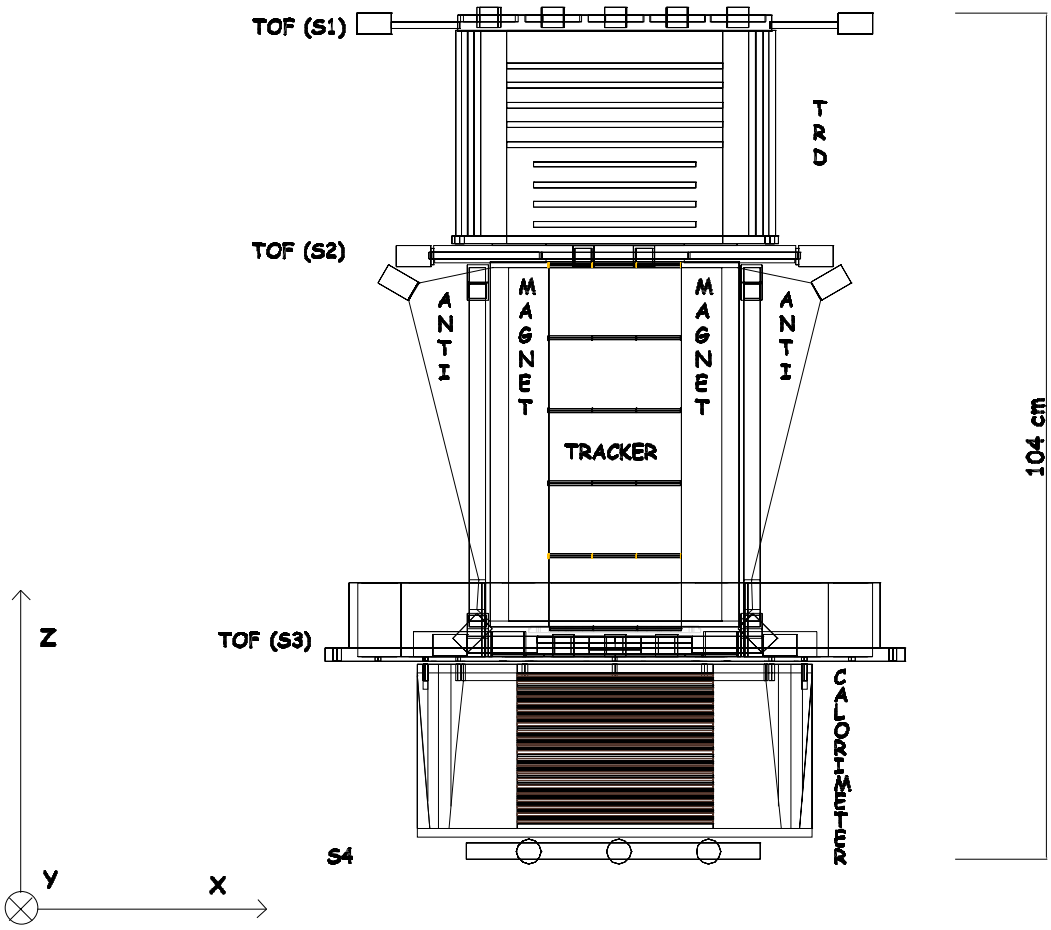


Figura 1.1: Rilevatori all'interno di PAMELA. Esso è costituito da: un sistema di tempi di volo (TOF), un rivelatore a radiazione di transizione (TRD), un *tracker* al silicio dotato di sei piani di rilevazione ed un calorimetro a campionamento.

L'asse Y è scelto con lo stesso verso e la stessa direzione del campo magnetico al centro della cavità nel magnete. Nella figura 1.1 è la direzione perpendicolare entrante all'immagine. L'asse X viene poi fissato perpendicolare agli assi Y e Z in modo da soddisfare la regola della mano destra.

Vediamo adesso in dettaglio i vari rivelatori. In particolare sarà descritto più approfonditamente il rilevatore a microstrisce di silicio, in quanto è il rilevatore di maggior interesse ai fini di questa tesi.

1.1 TOF

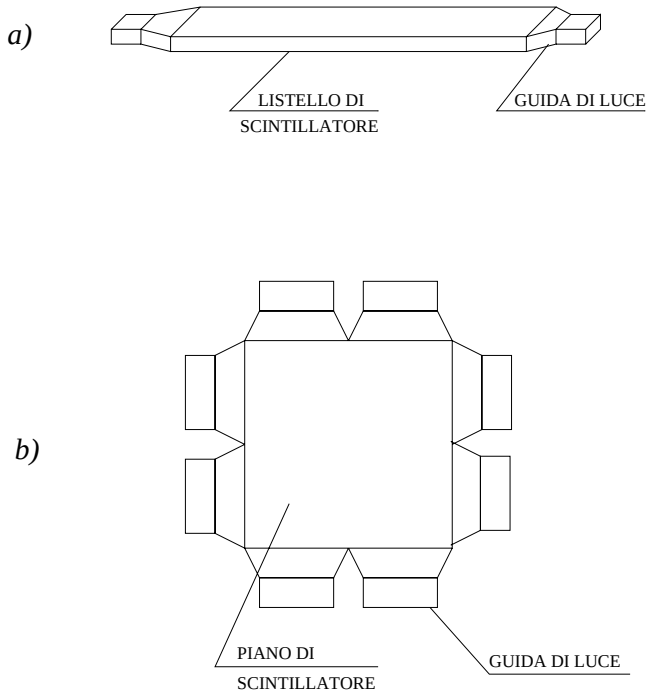


Figura 1.2: Scintillatori del TOF di PAMELA (non in scala) nei sistemi S1 e S3 (a) e nel sistema S2 (b). Ad essi sono agganciate le guide di luce.

Il TOF è un sistema per la misura del tempo di volo delle particelle all'interno del rivelatore. Tramite la misura del tempo di volo è possibile risalire alla velocità della particella. Inoltre è il TOF a fornire il segnale di inizio dell'acquisizione (o *trigger*).

Esso è costituito da cinque piani di scintillatore divisi in tre gruppi.

Il primo, posto in cima al telescopio, è composto da due piani a scintillazione (detti S11 e S12) 40×45 cm^2 divisi in 6 e 8 listelli posti l'uno accanto all'altro e dello spessore di 1 cm (fig. 1.2a). I due piani di scintillatore sono posti uno sopra l'altro. I listelli dei due piani sono posti perpendicolarmente tra di loro.

Il secondo gruppo è composto da un singolo piano di scintillatore (S2) 17×15 cm^2 e dello spessore di 0.7 mm posto tra il TRD e lo spettrometro magnetico (fig. 1.2*b*).

Il terzo gruppo si trova sotto lo spettrometro magnetico, sopra il calorimetro. Esso è composto da due piani di scintillatore (S31 e S32) delle dimensioni di 17×15 cm^2, ognuno dei quali è diviso in due listelli dello spessore di 1 cm (fig. 1.2*a*). Analogamente al gruppo S11 e S12 descritto sopra, i due piani sono posti uno sopra l'altro, con i listelli perpendicolari.

Alle due estremitá più corte dei listelli di S11, S12, S31 e S32 sono poste delle guide di luce che, a loro volta, sono accoppiate a due fotomoltiplicatori. Nel caso di S2 il piano di scintillatore è accoppiato in ogni lato a due fotomoltiplicatori per un totale di 8 fotomoltiplicatori (fig. 1.2*b*). Ogni fotomoltiplicatore è dotato di uno schermo magnetico per limitare l'influenza del campo magnetico disperso dal magnete.

1.2 TRD

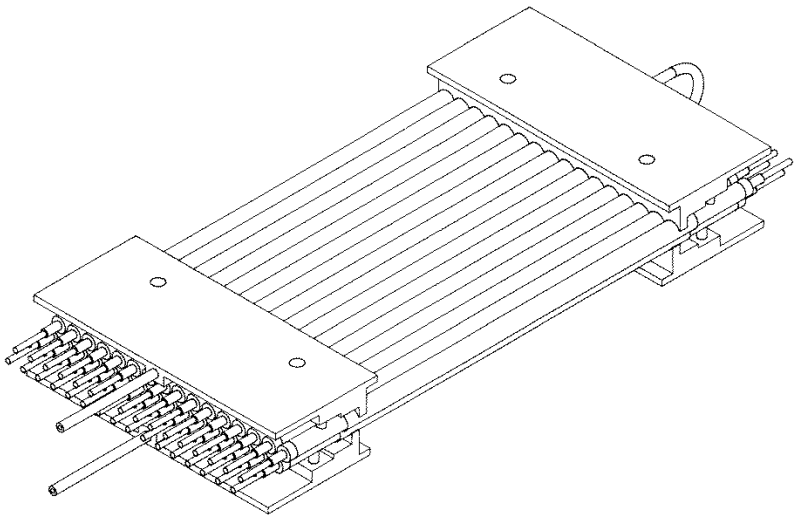


Figura 1.3: Strato di tubi proporzionali per la rivelazione della radiazione di transizione all'interno del TRD di PAMELA. Figura presa da [3]

Lo scopo del TRD è l'identificazione di particelle con velocità superiori ad una certa soglia (tipicamente con un fattore di Lorentz $\gamma \geqslant 1000$). Tale rivelatore viene utilizzato per distinguere protoni da positroni ed antiprotoni da elettroni nell'intervallo di energie compreso tra $\approx$ 1 GeV e $\approx$ 1000 GeV. Inoltre mediante la misura del $\frac{dE}{dx}$ può dare indicazioni sul valore assoluto della carica delle particelle.

Esso sfrutta la radiazione emessa da particelle ad alta energia che attraversano l'interfaccia tra mezzi diversi. Questa radiazione verrà poi rilevata con appositi dispositivi. Per una buona efficienza è importante che il materiale radiativo sia il più possibile trasparente alla luce di radiazione da esso stesso emessa. Inoltre il dispositivo di rivelazione deve avere una buona efficienza per le lunghezze d'onda interessate. Infine, dovendo essere la misura il più possibile non perturbativa, il TRD deve frapporre la minor quantità di materiale possibile sul percorso delle particelle.

Per quanto riguarda il materiale radiativo è stato scelto la lana di fibra di carbonio, che emette radiazione di transizione nella banda dei raggi X (approssimativamente attorno ad un'energia di 5 KeV). La lana di fibra di carbonio è stata scelta in quanto composta da numerosi filamenti che comportano frequenti passaggi attraverso superfici di interfaccia. Inoltre il materiale è caratterizzato da un basso numero atomico Z per una buona trasparenza ai raggi X.

Dovendo essere il TRD meno denso possibile, sono stati scartati rivelatori solidi e si è optato per rivelatori a gas a "cannuccia" (*straw tubes*, vedi fig. 1.3). Questi sono dei tubi di kapton del diametro di 4mm e di lunghezza variabile, riempiti di gas Xenon per un efficace assorbimento dei raggi X emessi, con il 20% di anidride carbonica, con un filo anodico lungo l'asse. Il rivelatore opera in regime proporzionale.

I tubi possono sopportare una differenza di pressione con l'esterno fino a 4 atmosfere. Questi tubi sono organizzati in 9 strati intervallati dai radiatori in lana di fibra di carbonio. Il tutto è incapsulato in una struttura di alluminio.

1.3 Calorimetro

Il principale scopo del calorimetro di PAMELA è l'identificazione delle particelle sulla base dello studio delle caratteristiche degli sciami prodotti da esse. In questo modo esso può separare gli antiprotoni dal fondo di elettroni con un potere di reiezione di 10^4 ed una efficienza del 90%.

Esso è un calorimetro elettromagnetico a campionamento (fig. 1.4) composto da strati alternati di 22 piastre di tungsteno e rilevatori a strisce di silicio. Questi ultimi sono a loro volta alternati con le strisce in direzione X ed in direzione Y.

I rilevatori al silicio permettono di avere informazioni sulla forma degli sciami. Ogni strato di rilevatore al silicio è composto da 3×3 moduli, ognuno dei quali ha un'area di 8×8 cm^2 con una segmentazione delle strisce di 2.5 mm.

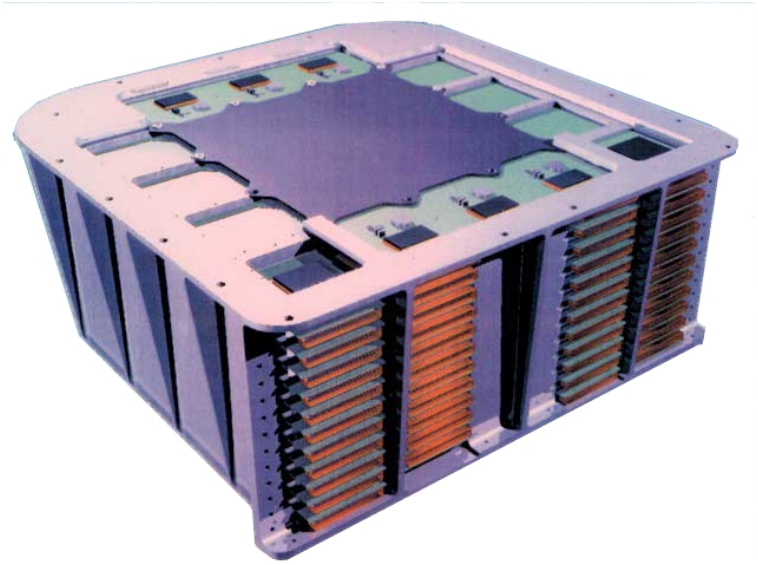


Figura 1.4: Struttura meccanica del calorimetro di PAMELA. Figura presa da [4].

Lo spessore totale del calorimetro è di 16 lunghezze di radiazione e 0.9 lunghezze di interazione.

Sotto il calorimetro è posto un piano di scintillatore (S4) allo scopo di rilevare eventuali particelle che riescano a attraversare completamente il calorimetro.

1.4 Lo spettrometro magnetico

Lo scopo dello spettrometro magnetico di PAMELA è la misura dell'impulso e del segno della carica elettrica delle particelle cariche che lo attraversano, osservandone la traiettoria, curvata da un campo magnetico approssimativamente uniforme.

Esso è formato da un sistema magnetico a magnete permanente e la traiettoria delle particelle viene ricostruita misurando i punti di passaggio in sei piani di rivelatori a microstrisce di silicio.

Il magnete

La scelta di un magnete permanente rispetto ad un elettromagnete è dettata dalla necessità di un basso consumo energetico, visto che la potenza a disposizione per l'esperimento è limitata.

Il materiale magnetico usato è una lega di Nd-Fe-B con una magnetizzazione residua di circa 1.31 T.

La forma del magnete è quella di una torre di sezione rettangolare 228×240 mm^2 alta 445 mm, con al centro una cavità disposta lungo l'asse, pure a sezione rettangolare

Figura 1.5: Prototipo dello spettrometro magnetico di PAMELA. Figura presa da [4]

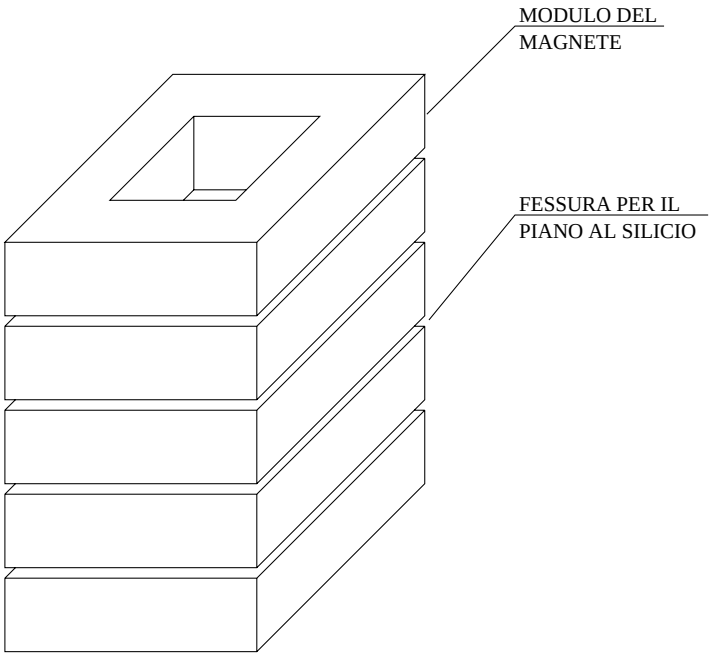


Figura 1.6: Disposizione dei 5 moduli del magnete in cui sono rappresentate le fessure per l'introduzione dei piani di rilevazione.

132×162 mm^2 come si vede in figura 1.5. La torre è composta da cinque moduli uguali alti 81 mm intervallati da quattro fessure alte 8 mm nelle quali possono venir inseriti i piani del rilevatore al silicio (fig. 1.6). Questo permette di mettere e togliere i piani con una certa libertà.

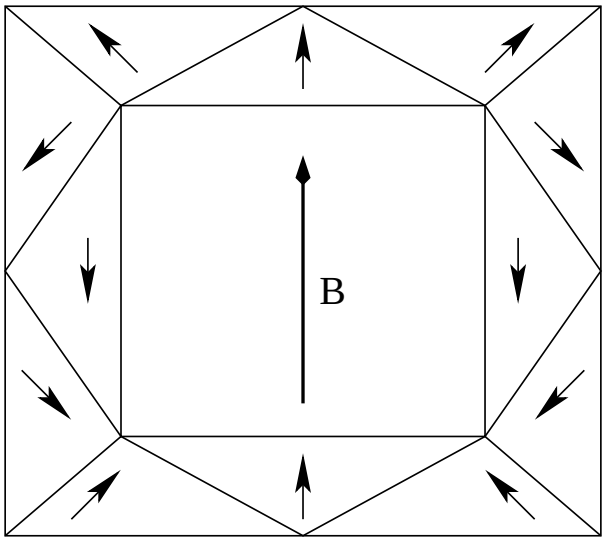


Figura 1.7: Sezione di un modulo del magnete (le frecce indicano la direzione delle magnetizzazioni residue all'interno del materiale magnetico).

Ogni modulo magnetico è composto da dodici prismi magnetici a sezione triangolare, incollati tra di loro come in figura 1.7.

Ogni modulo è magnetizzato in maniera appropriata, così che all'interno della cavità si formi un campo magnetico più intenso ed uniforme possibile e con il minor campo disperso all'esterno. Il tutto è inserito in una gabbia di alluminio a formare la torre.

Nel centro dello spettrometro il campo magnetico è di 0.48 T.

È utile definire ora il fattore geometrico che è una grandezza dipendente soltanto dalle caratteristiche geometriche dell'esperimento e normalmente utilizzata nella fisica dei raggi cosmici [5]. Nel caso di PAMELA la grandezza geometrica dominante è la cavità del magnete (i rivelatori esterni sono costruiti in modo da coprire l'angolo solido sotteso da essa).

Se prendiamo come riferimento la sezione della cavità in corrispondenza del lato inferiore del magnete (fig. 1.8), il fattore geometrico F è definito come l'angolo solido sotteso dalla cavità integrato su tutta la sezione, secondo la formula :

$$F = \int_S dS \cos\theta \int_\omega d\omega \tag{1.1}$$

dove ω è l'angolo solido che sottende la cavità da un punto del piano di riferimento, dω l'elemento di angolo solido in ω, S la superficie della sezione della cavità del magnete e θ l'angolo tra la normale al piano di riferimento e la direzione in cui punta l'elemento di angolo solido dω centrato sull'elemento di superficie dS del piano di riferimento .

Il fattore geometrico dello spettrometro magnetico, e quindi di PAMELA, è di 20.5 $\mathrm{cm}^2\mathrm{sr}$.

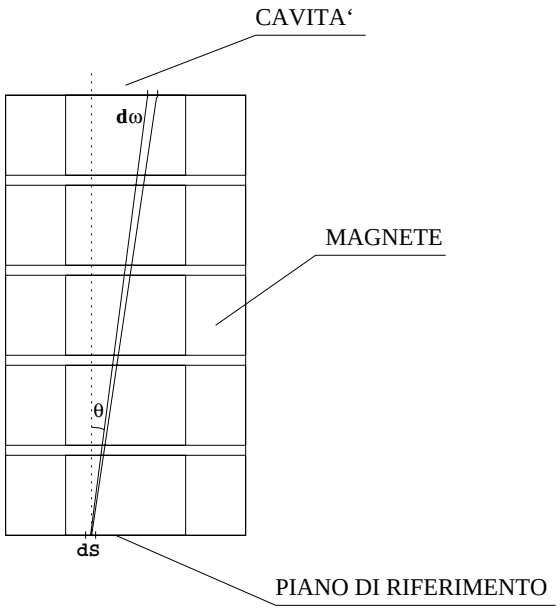


Figura 1.8: Descrizione del calcolo del fattore geometrico: dS è l'elemento di superficie del piano di riferimento, dω l'elemento dell'angolo solido che da dS sottende la cavità del magnete e θ l'angolo tra direzione dell'elemento dell'angolo solido dω e la normale al piano di riferimento.

Sulle pareti esterne del magnete sono poste delle piastre di scintillatore che agiscono come unità di anticoincidenze per l'identificazione delle particelle che colpiscono esternamente il magnete. È inoltre presente, all'esterno del magnete, uno schermo magnetico che ne limita il campo disperso.

Il rivelatore a microstrisce di silicio

Il tracciatore è costituito da sei piani di rivelazione a microstrisce di silicio.

I sei piani sono disposti parallelamente alla distanza di 8.9 cm l'uno dall'altro. Il primo e l'ultimo piano si trovano immediatamente sopra e sotto il magnete, mentre i quattro piani intermedi sono all'interno del magnete inseriti in apposite fessure (fig. 1.5). Questo permette la ricostruzione della curvatura delle traiettorie delle particelle cariche nel campo magnetico.

Ogni piano di rivelatore è formato da tre unità rivelatrici, ognuna formata da due

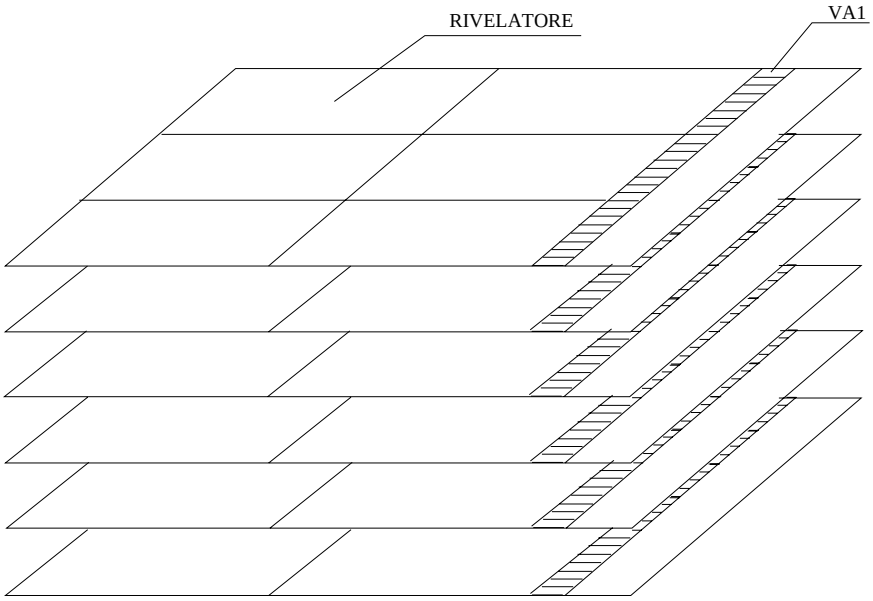


Figura 1.9: Schema dei sei piani del rivelatore a microstrisce di silicio. Attaccati ai rivelatori sono indicati i circuiti di *front end* in cui sono evidenziati i *chip* VA1.

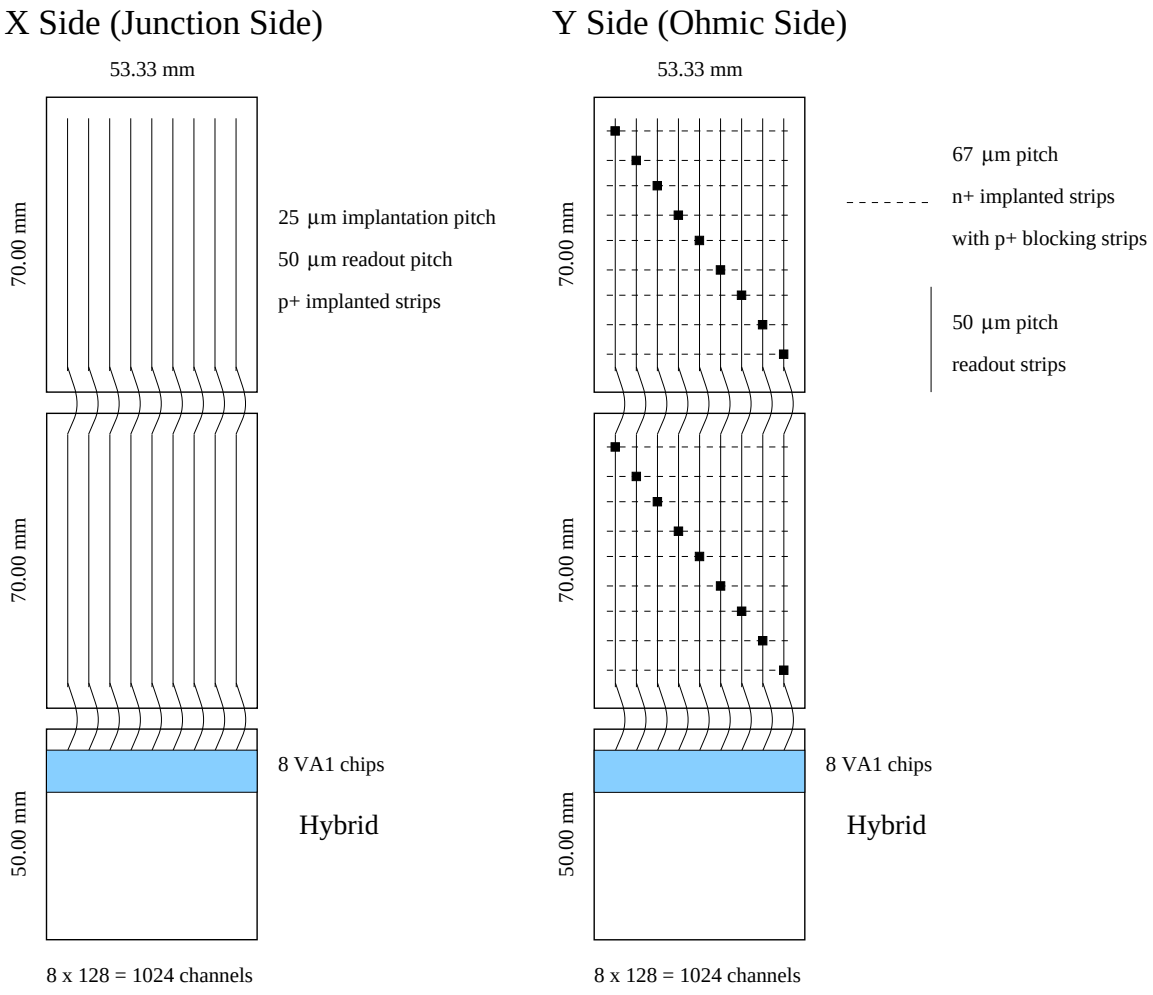


Figura 1.10: Unità a microstrisce di silicio formata da due sensori (figura presa da [3]).

sensori di forma rettangolare collegati tra di loro e dello spessore di 300μm. A loro volta le unità rivelatrici sono collegate all'elettronica di *front end* (figure 1.9 e 1.10).

Le unità rivelatrici sono sostenute da sbarrette di fibra di carbonio molto leggere incollate lateralmente. In questo modo il materiale attraversato dalle particelle è dato solo dai piani di silicio, cioè da uno spessore di 3.2×10^{-3} X_0 per ogni piano attraversato.

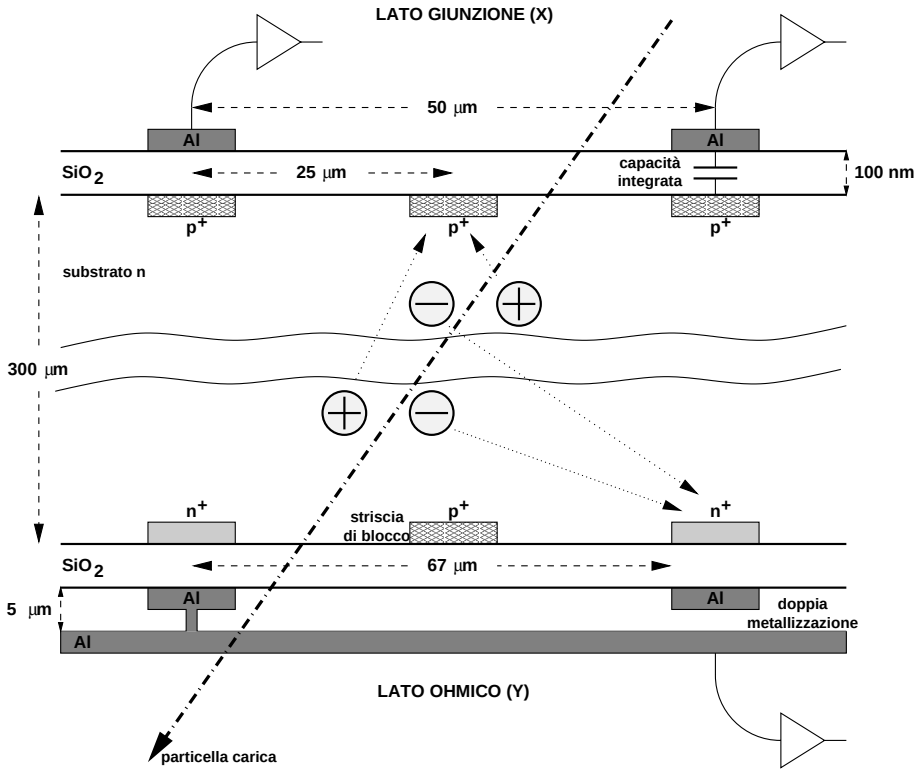


Figura 1.11: Struttura dei sensori al silicio di PAMELA.

Ogni sensore, di dimensione 70×53.33 mm^2 è costituito da un cristallo di silicio drogato di tipo n. Su di una faccia, detta lato giunzione, sono impiantate 2048 strisce di silicio drogato p^+ parallelamente al lato più lungo del rettangolo con una distanza tra le strisce di 25.5μm. Sulla faccia opposta, detta lato ohmico, sono impiantate 1024 strisce di silicio drogato n^+ e 1024 strisce di tipo p^+ alternate a quelle di tipo n^+, parallele al lato più corto del sensore (fig. 1.11). La distanza tra le strisce di tipo n^+ sul lato ohmico è di 67μm.

I due sensori di una stessa unità di rivelazione sono collegati striscia per striscia in modo che l'unità si comporti come se avesse le strisce di lunghezza doppia (il collegamento tra le strisce è rappresentato in figura 1.10). Questo comporta un'ambiguità nel lato ohmico. Infatti nel lato giunzione le strisce di una unità sono la continuazione di quella dell'altra, mentre sul lato ohmico sono disposte parallelamente. Di conseguenza la rivelazione di un segnale potrebbe arrivare da una striscia di un sensore o dalla corrispondente

dell'altro sensore a 70 mm di distanza. Ma questa ambiguità non produce problemi in quanto è facile distinguere tra due coordinate così distanti sulla base degli altri rivelatori di PAMELA.

Sul lato giunzione le strisce vengono lette una sì ed una no in modo che il passo di lettura sia di 51μm. Quando una particella ionizzante passa attraverso il rivelatore provoca la generazione di coppie elettrone-lacuna all'interno del silicio. Quando la giunzione è contropolarizzata, si crea una zona svuotata da cariche di conduzione al cui interno si forma un campo elettrico. Le coppie elettrone-lacuna vengono trasportate dal campo elettrico verso il lato ohmico e quello giunzione, dove vengono raccolte dalle strisce e rivelate dall'elettronica di *front end*. Normalmente la carica prodotta dalla particella non viene raccolta da una sola striscia, ma da più strisce adiacenti. La coordinata del passaggio della particella sarà poi determinata mediante un processo di analisi pesato con le cariche raccolte dalle strisce interessate. Sul lato ohmico sono presenti due tipi di strisce: le strisce di lettura vere e proprie di tipo n^+ ed alternate ad esse delle strisce di tipo p^+ che invece non vengono lette. Sulla superficie del sensore si forma infatti un accumulo di cariche elettrostatiche tra il silicio e l'ossido di silicio che abbasserebbe la resistenza elettrica tra due strisce adiacenti. Per evitare questo problema vengono impiegate delle strisce p^+ (dette di blocco) tra due strisce n^+, che permettono il corretto funzionamento del rivelatore.

La risoluzione spaziale nel lato giunzione ($\approx 3\mu$m) è migliore di quella nel lato ohmico ($\approx 15\mu$m). Questo perché è molto più importante la misura della coordinata X di quella della coordinata Y, in quanto la direzione del campo magnetico è lungo la coordinata Y e quindi la coordinata che più ci interessa è la X per misurare la curvatura della traiettoria delle particelle. Si stima che il massimo impulso misurabile dallo spettrometro per particelle con Z=1 sia di 740 GeV/c [6] [7].

Poiché le microstrisce di silicio sono perpendicolari nelle due viste, sarebbe necessario disporre schede elettroniche di lettura su almeno due lati di ogni vista. Questo avrebbe occupato spazio prezioso ed inoltre sarebbe stato necessario praticare ulteriori fessure nel magnete, peggiorando l'integrità del campo magnetico interno. La tecnica della doppia metallizzazione ha permesso in PAMELA di poter sistemare l'elettronica di lettura per entrambe le viste in corrispondenza di un solo lato dei piani. La tecnica della doppia metallizzazione consiste nell'applicare, sul lato ohmico, una metallizzazione in direzione delle strisce (ovvero nella direzione X) e successivamente una seconda metallizzazione in direzione perpendicolare alla prima, separata da essa da uno strato di ossido spesso 5μm.

Ogni striscia metallizzata in direzione X è collegata ad una striscia conduttrice in direzione Y da un contatto metallico che attraversa l'ossido. In questo modo entrambe le viste di un piano rivelatore possono essere lette da schede di elettronica situate sullo stesso lato del piano (fig. 1.10). Il rischio di questa tecnica risiede nel pericolo di corto circuiti e di strisce scollegate, ma con le attuali tecniche si possono costruire doppie metallizzazioni con meno di 1% di difetti.

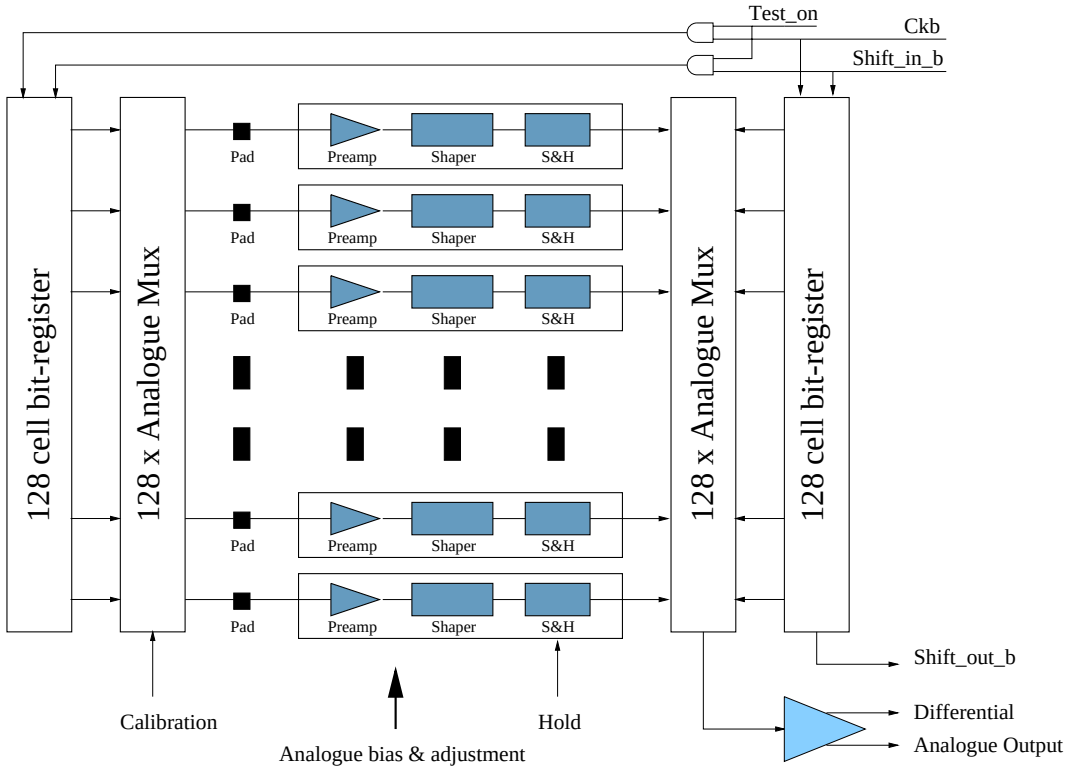


Figura 1.12: Il *chip* VA1, che costituisce l'elettronica di *front end* del *tracker* al silicio di PAMELA (figura presa da [3])

La lettura dei segnali delle strisce viene effettuata mediante schede di elettronica dette ibridi. Qui avviene la lettura e la formazione dei segnali. A questo scopo si utilizzano dei *chip* VA1 ognuno dei quali può gestire fino a 128 canali di lettura (di conseguenza per poter analizzare le 1024 strisce di una vista di una singola unità di rivelazione occorrono 8 *chip* VA1). Ogni *chip* VA1 opera una preamplificazione ed una formazione dei segnali, che verranno poi memorizzati con un meccanismo di *sample&hold* descritto nel seguito, in attesa di essere passati ad un ADC per la conversione in segnali digitali (vedere figura 1.12).

Gli ADC convertono i segnali in parole a dodici *bit* che vengono scritte in appositi spazi di memoria all'interno di alcuni processori di segnali digitali (*digital signal processor* o DSP). Questi ultimi tratteranno i dati prima di essere trasmessi a terra, ovvero opereranno

una compressione sui medesimi. Il modo in cui i DSP eseguono la compressione dei dati è l'argomento principale della tesi e viene descritto nei capitoli successivi.

Riassumendo, ogni piano del rivelatore a microstrisce di silicio è diviso in tre unità contenenti ognuna 1024 strisce lette per vista, quindi contiene $1024 \times 3 \times 2 = 6144$ strisce lette. Questo significa che i sei piani del telescopio contengono un totale di 36864 strisce lette. Ogni unità viene letta da *chip* VA1 ognuno dei quali gestisce 128 strisce, di conseguenza occorrono 8 VA1 per unità (48 VA1 per piano e 288 per l'intero telescopio). Ogni unità viene servita da un ADC per vista (che riceve dati da 8 VA1), quindi ogni piano utilizza $3 \times 2 = 6$ ADC, per un totale di 36 ADC per i sei piani. I dati di ogni vista di un piano vengono trattati da un DSP (che quindi analizza i segnali di 3072 strisce) per un totale di $2 \times 6 = 12$ DSP.

1.5 Il flusso dei dati

Il satellite artificiale a bordo del quale il telescopio PAMELA sarà alloggiato seguirà un'orbita ellittica quasi polare (latitudine $\approx 70°$), con quota che varia tra 350 km e 600 km. Di conseguenza il flusso di raggi cosmici varierà con la latitudine e la quota a cui il satellite si troverà momento per momento, poiché il campo magnetico terrestre farà da schermo più efficacemente alle basse latitudini che a quelle alte.

Si stima che la frequenza di eventi raccolti da PAMELA sia compresa fra 0.2 e 8 Hz (a seconda della posizione del satellite lungo l'orbita) con una media di $\approx 3 \times 10^5$ eventi al giorno, costituiti principalmente da protoni. Questo valore non tiene conto del flusso di particelle corrispondente all' Anomalia del Sud Atlantico [1]; inoltre esiste una dipendenza del flusso di particelle dall'attività solare. L'andamento della frequenza di acquisizione per PAMELA è mostrato in figura 1.13.

Stime più dettagliate delle frequenze di rivelazione si trovano in tabella 1.1

Data l'orbita si prevede che il satellite sarà in posizione utile per la trasmissione a terra da una a quattro volte al giorno con una finestra temporale utile di 10 minuti circa. Con la telemetria a disposizione è possibile trasmettere circa 15 Mbit/sec. Questo significa che ogni giorno si possono tramettere a terra da 1.1 Gbyte/giorno a 4.5 Gbyte/giorno.

Ogni evento comporta un certo quantitativo di dati da trasmettere a terra. La maggior mole di essi viene dal rivelatore a microstrisce di silicio che, con le sue 38864 strisce

[1] Si tratta di una zona, situata fra il Sud America e l'Africa, in cui la diminuzione di intensità magnetica rispetto al campo di dipolo è particolarmente marcata. Questa differenza consente a molte particelle cosmiche di penetrare più profondamente nell'atmosfera.

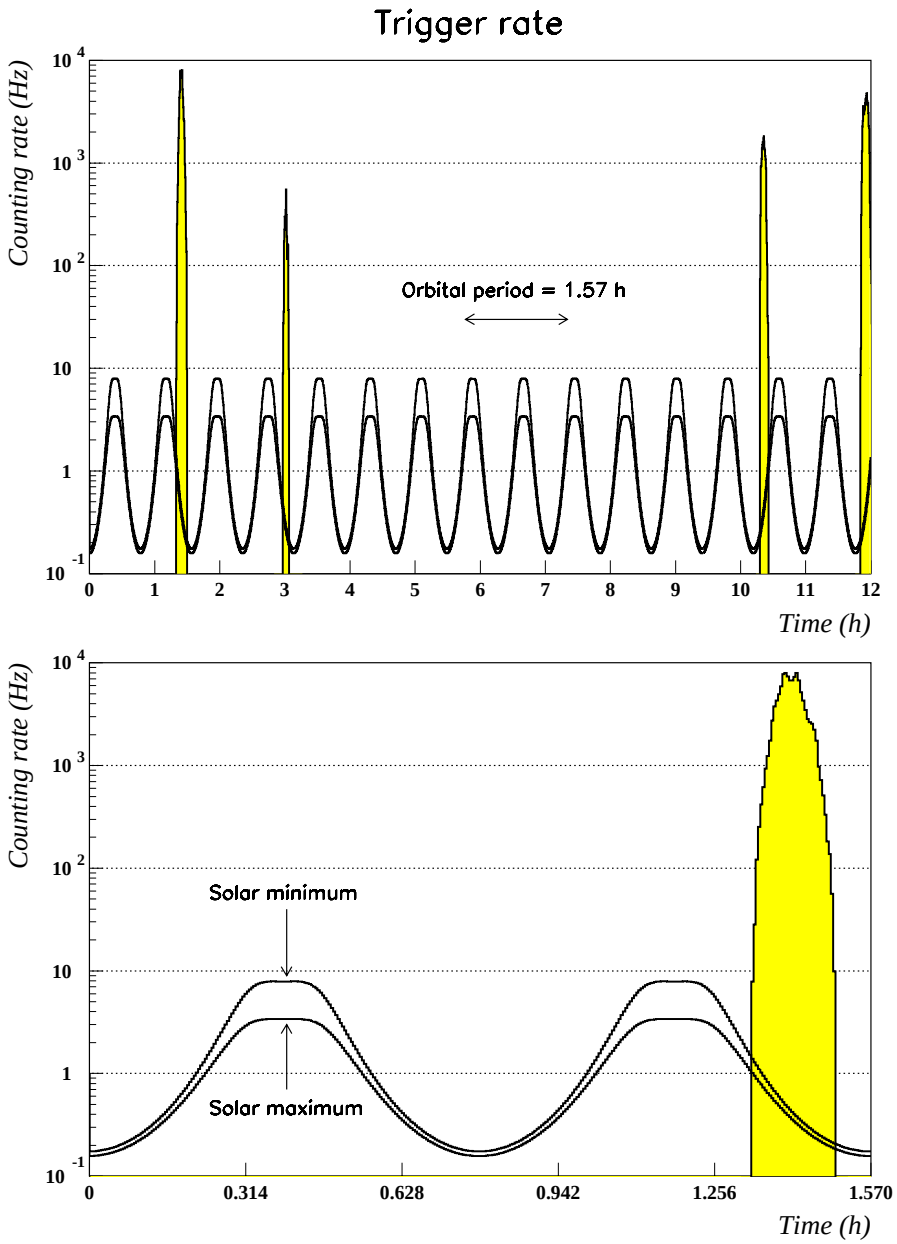


Figura 1.13: Andamento previsto della frequenza di acquisizione per PAMELA durante un periodo di 12 ore (*sopra*) e lungo un'orbita (*sotto*). Si nota il passaggio del satellite all'interno della Anomalia del Sud Atlantico, dove la frequenza di particelle acquisite cresce di vari ordini di grandezza. Si nota anche che il flusso di particelle cosmiche è anticorrelato all'attività del Sole, nel senso che a un periodo di massima attività solare corrisponde un flusso più basso di raggi cosmici galattici e viceversa.

Particelle	**Eventi in 3 anni**
Protoni	3×10^8
Antiprotoni	$\geq 3 \times 10^4$
Elettroni	6×10^6
Positroni	$\geq 3 \times 10^5$
Nuclei di elio	4×10^7
Nuclei di berillio	4×10^4
Nuclei di carbonio	4×10^5

Tabella 1.1: Eventi previsti in 3 anni riguardanti alcune particelle contenute nei raggi cosmici.

(ognuna delle quali trasmette parole a 12 *bit*) comporterebbe la trasmissione di circa 58 Kbyte/evento, contro i 9 Kbyte/evento del calorimetro e i 2 Kbyte/evento del TRD.

Considerato che il numero medio di eventi quotidiani è $3 \cdot 10^5$ il solo sprettrometro dovrebbe trasmettere a terra circa 18 Gbyte/giorno di dati, cosa completamente al di fuori delle possibilità della telemetria a disposizione, di qui la necessità di comprimere i dati (specialmente quelli dello spettrometro magnetico) che è l'argomento di questa tesi.

Data la mole di lavoro che comporta la compressione dei dati non è realistico farla compiere dalla CPU principale del satellite. Si è preferito quindi affidare il compito a delle CPU dedicate allo scopo: i già nominati DSP.

I DSP sono processori molto veloci ed a basso consumo di potenza. In essi viene implementato un programma che provvede, una volta che i dati di un evento sono stati scritti nelle loro memorie interne, a eseguire la compressione. Questa tesi riguarda appunto i programmi di compressione da implementare nei DSP.

Elettronica di acquisizione

Descriviamo ora brevemente i vari processi che vengono attivati nel momento in cui si acquisiscono i dati.

Come già detto il segnale di *trigger* viene generato dal TOF. La richiesta affinché il segnale venga abilitato è che la particella produca un segnale in tutti i tre sistemi di scintillazione S1, S2, S3 (si tratta in pratica di fare un AND logico dei segnali provenienti da essi).

Quando una particella attraversa il rivelatore al silicio, genera coppie elettrone-lacuna che vengono raccolte nelle strisce adiacenti in tempi rapidi (< 30 ns). Ogni striscia è connessa elettricamente ad un canale del *chip* VA1 che provvede a preamplificare il segnale in carica e formare il segnale in tensione, con tempi tipici di formazione di 1μs. Inoltre, lo

stadio di formazione di ogni canale è connesso ad un circuito di *sample&hold* costituito da un condensatore ed un interruttore utilizzato per "congelare" il livello in tensione presente all'uscita del circuito di formazione al momento dell'apertura dell'interruttore.

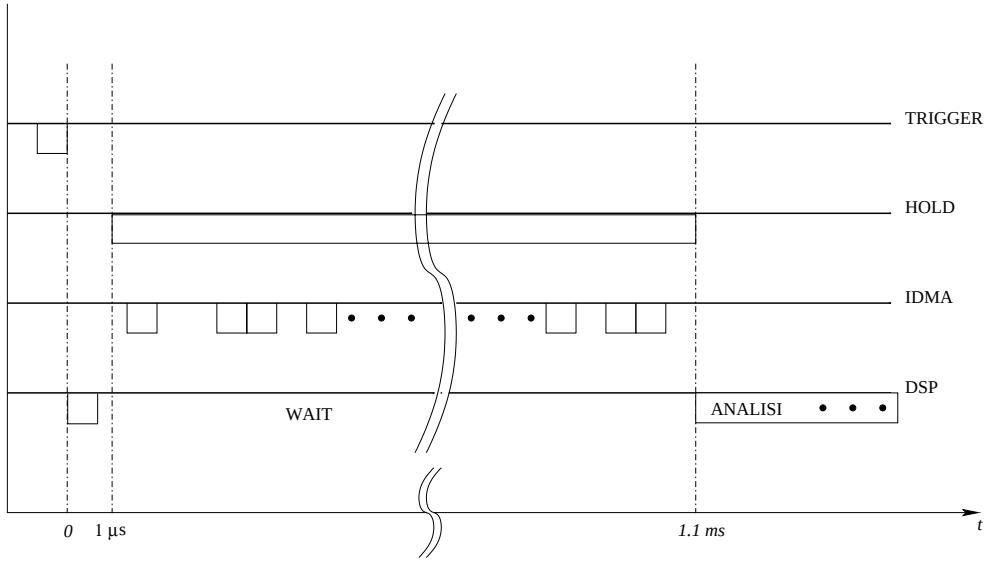


Figura 1.14: Schema dei tempi durante la lettura/scrittura dei dati.

Dopo circa 1μs dal passaggio della particella è quindi necessario inviare al VA1 un segnale di *hold*, che viene utilizzato per aprire i 128 interruttori del *chip*. In seguito un circuito di *multiplex* viene utilizzato per presentare serialmente (con tempi di scansione di 1μs) sull'unica uscita analogica del *chip* i 128 livelli di tensione che sono stati congelati al momento dell'arrivo del segnale di hold.

Nel caso di PAMELA, 8 VA1 vengono letti serialmente e sono quindi necessari $1\mu\text{s}\times 8\times 128 = 1.1$ ms per effettuare la scansione completa dei 1024 canali di un lato dell'ibrido. Gli ADC vengono utilizzati per convertire digitalmente le informazioni analogiche presentate sequenzialmente all'uscita dei VA1.

Anche il DSP riceve il segnale di trigger e attiva il programma di analisi. Poiché occorre circa 1 ms ($1\mu\text{s}\times 1024$) per scrivere tutti i dati nelle memorie dei DSP, il programma di analisi si mette in attesa per 1.1 ms, dopodiché inizia l'elaborazione vera e propria. La scrittura dei dati nel DSP avviene mediante un'apposita porta (IDMA), attraverso la quale si può scrivere direttamente nelle memorie del DSP in maniera asincrona con il programma eseguito dal DSP. In figura 1.14 sono riassunti i tempi di elaborazione.

Il DSP provvederà a processare i dati ed a trasmetterli ad una memoria comune dove vengono accumulati tutti i dati provenienti dai vari rivelatori, in attesa di essere trasmessi a terra dalla telemetria.

Capitolo 2

Algoritmi di compressione

I vari algoritmi di compressione dati si possono dividere in non reversibili (*lossy*) e reversibili (*lossless*). I primi sono algoritmi che comportano la perdita di una parte delle informazioni nel processo di compressione. I secondi invece mantengono intatta tutta l'informazione del messaggio che, una volta decompresso, sarà in tutto e per tutto uguale all'originale.

L'obiettivo di questa tesi è l'analisi delle varie tecniche di compressione dati per poi implementarle nei DSP che analizzano i dati provenienti dal tracker al silicio di PAMELA.

Nel seguito si utilizzerà la quantità detta Rapporto di Compressione, o RdC. Essa viene definita da:

$$RdC = 1 - \frac{Comp}{NonComp}\,, \tag{2.1}$$

dove $Comp$ è la dimensione del messaggio una volta compresso e $NonComp$ la dimensione del messaggio originale. Quindi un valore vicino ad uno significa che il messaggio è stato fortemente compresso mentre un valore di RdC vicino allo zero significa una scarsa compressione. Il rapporto di compressione è un numero minore o uguale ad uno (può essere anche negativo) ma spesso viene riportato sotto forma di percentuale.

2.1 Algoritmi non reversibili

Tipicamente gli algoritmi di compressione non reversibili sono diffusi soprattutto in campo audio e video, in cui la perdita di informazione non si nota facilmente, oltretutto la sorgente è spesso analogica e la sua digitalizzazione è una approssimazione già in partenza.

Un metodo tipico di compressione non reversibile è quello di passare al dominio delle frequenze tramite trasformate quali ad esempio la F.F.T. (*Fast Fourier Trasform*) o la D.C.I. (*Discrete Cosine Transformation*), e poi applicare una maschera alle frequenze. Per

maschera si intende una serie di tagli applicati a vari intervalli di frequenza. Successivamente si possono applicare ulteriori algoritmi di compressione magari reversibili. Seguono questo schema ad esempio l'algoritmo di compressione per immagini fisse J.P.E.G. (*Joint Photographic Experts Group*) o mobili M.P.E.G. (*Moving Pictures Experts Group*).

Nel seguito verrà descritto il funzionamento di una particolare classe di algoritmi non reversibili: i predittori.

Predittori

I predittori sono un tipo di algoritmi di compressione basati su di una predizione a priori o a posteriori. Il metodo di compressione consiste nel codificare soltanto quella porzione dei dati che si discosta significativamente dalla predizione.

Un esempio è lo *zero suppression like*, nel quale si fa una predizione a priori dei dati. Ammettiamo di essere in presenza di un flusso di dati oscillante in modo casuale (rumore) attorno ad un segnale noto, ad esempio una costante (che può essere nulla) o una sinusoide (fig. 2.1).

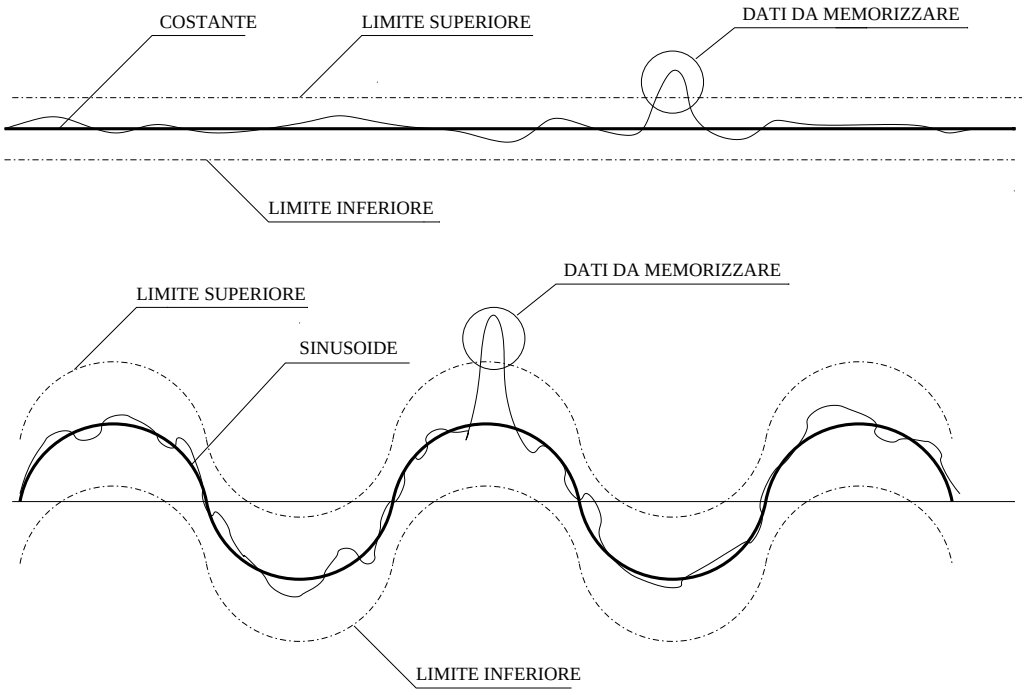


Figura 2.1: Esempio di *zero suppression like*: soltanto i dati che escono dalla banda delimitata dalle linee tratteggiate vengono memorizzati. Le linee in grassetto rappresentano le predizioni del modello.

Possiamo prevedere che per la maggior parte del tempo il flusso dei dati non si discosti troppo dal nostro modello, entro limiti che dipendono dal rumore. Se, come spesso

avviene, non siamo interessati al rumore presente nel flusso dei dati possiamo provare ad eliminarlo da esso. Possiamo quindi analizzare i dati sottraendoci la predizione del modello e dare una stima delle fluttuazioni dovute al rumore, attraverso la valutazione della varianza del segnale. Successivamente possiamo definire un taglio proporzionale al valore della varianza e sottoporre i dati ad un confronto con il modello. Se il dato sotto analisi si discosta dal modello (la costante o la sinusoide) di una quantità superiore al taglio definito, viene memorizzato, altrimenti no. Una schematizzazione del metodo è riportata in tabella 2.1

| DATO - MODELLO |	$\leq$ T $\times$ VAR	Il dato non viene memorizzato
	$>$ T $\times$ VAR	Il dato viene memorizzato

Tabella 2.1: Condizioni dello *zero suppression like*, dove T è una costante di taglio e VAR è la stima della varianza del rumore dei dati, DATO è il dato in esame e MODELLO la sua previsione.

In questo modo si ottiene spesso un buon rapporto di compressione, ad esempio in campo audio.

Un altro esempio è il predittore di ordine zero (*Zero Order Predictor* , ZOP). Questo compie una predizione sulla base dei dati incontrati in precedenza e non su di un modello a priori. In pratica esso è un algoritmo che trasmette un dato solo se la differenza in valore assoluto rispetto all'ultimo dato memorizzato supera una certo taglio (che può essere definito in modo analogo allo *zero suppression like* (tab. 2.2)). Questo significa che la banda fissa dello *zero suppression like* in figura 2.1 è sostituita ora da una banda mobile come è rappresentato in figura 2.2.

|DATO - DATO PRECEDENTE|	$\leq$ T $\times$ VAR	Il dato non viene memorizzato
	$>$ T $\times$ VAR	Il dato viene memorizzato

Tabella 2.2: Condizioni dello *zero order predictor* , dove T è una costante di taglio e VAR è la stima della varianza del rumore dei dati, DATO è il dato in esame e DATO PRECEDENTE è l'ultimo dato memorizzato.

Algoritmi di compressione dello stesso tipo dello ZOP, ma più raffinati, sono il predittore al primo ordine (*First Order Predictor* , F.O.P.), il predittore al secondo ordine (*Second Order Predictor* , S.O.P.), ecc. Questi algoritmi si basano su predizioni via via più raffinate. Se lo ZOP si basa su una previsione del prossimo valore in un flusso dati basato solo sull'ultimo dato trasmesso, il FOP fa una previsione considerando anche la derivata prima nel flusso dei dati analizzando gli ultimi due dati trasmessi, il SOP tiene conto anche della derivata seconda e così via.

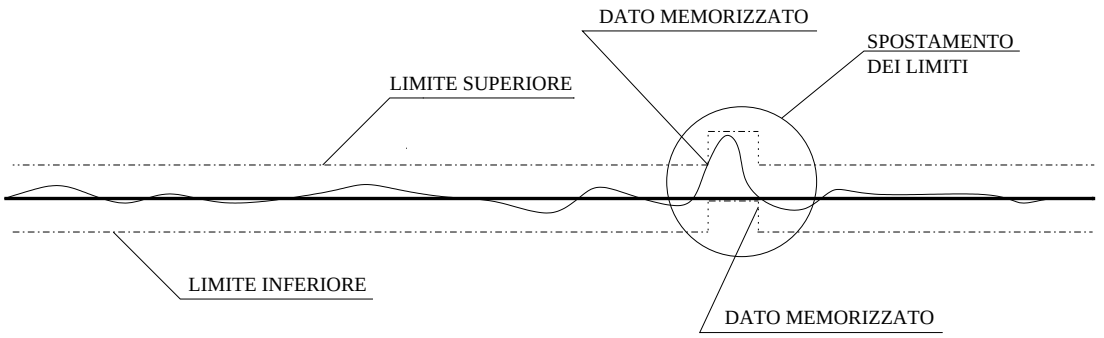


Figura 2.2: Esempio di ZOP: se il dato esce dalla banda viene memorizzato e diventa il nuovo riferimento (la banda viene centrata su di esso).

2.2 Algoritmi reversibili

Più conosciuti di quelli non reversibili, questi sono algoritmi finalizzati all'eliminazione della ridondanza dell'informazione in un messaggio.

Lo scopo di un messaggio è quello di trasmettere un certo quantitativo di informazioni. Il modo in cui queste informazioni vengono codificate, sia questo un foglio di carta o un calcolatore, non cambia la quantità di esse. Nei calcolatori le informazioni si registrano per mezzo di sequenze di *bit*.

I messaggi di testo vengono spesso codificati per mezzo di codici *ascii* . Questi sono costituiti da 256 caratteri, tra cui si trovano anche le lettere dell'alfabeto, contenuti in una tabella che associa un numero ad ognuno di essi. Nella codifica a codici *ascii* ad ogni simbolo del messaggio viene associato il numero ad esso relativo che viene memorizzato mediante una parola a 8 *bit* (sufficiente a contenere numeri fino a 256 in codice binario). I codici *ascii* sono molto diffusi e quindi comodi per trasmettere e memorizzare informazioni, ma non sono il metodo di codifica più efficiente dal punto di vista dello spazio di memoria utilizzato.

Ad esempio: se il messaggio non fa uso di tutti i 256 caratteri, perché utilizzare 8 *bit* per codificare i caratteri ? Ed inoltre, se anche li utilizzasse tutti, perché non assegnare ai caratteri più frequenti codifiche con un numero inferiore di *bit* ?

Queste domande vennero sollevate già alla fine degli anni quaranta da Claude Shannon [8].

Il processo di compressione dati consiste in pratica nel sostituire ai simboli che costituiscono il messaggio, dei codici che occupino meno spazio rispetto alla loro codifica originaria.

La compressione viene spesso impropriamente chiamata "codifica". In realtà la codi-

fica è solo un passaggio della compressione. In genere, almeno nei casi più semplici, la compressione è composta dalla creazione del modello e dalla codifica.

La creazione del modello consiste nell'analisi dei dati da comprimere e quindi nella costruzione di un modello di essi. Un modello altro non è che un insieme di dati e regole che vengono passate al codificatore allo scopo di creare dei codici da sostituire ai dati da comprimere.

La decompressione funziona in maniera inversa alla compressione. Uno schema abbastanza generale è dato nella seguente figura 2.3, in cui è rappresentato sia lo schema di compressione (fig. 2.3*a*), che lo schema di decompressione (fig. 2.3*b*):

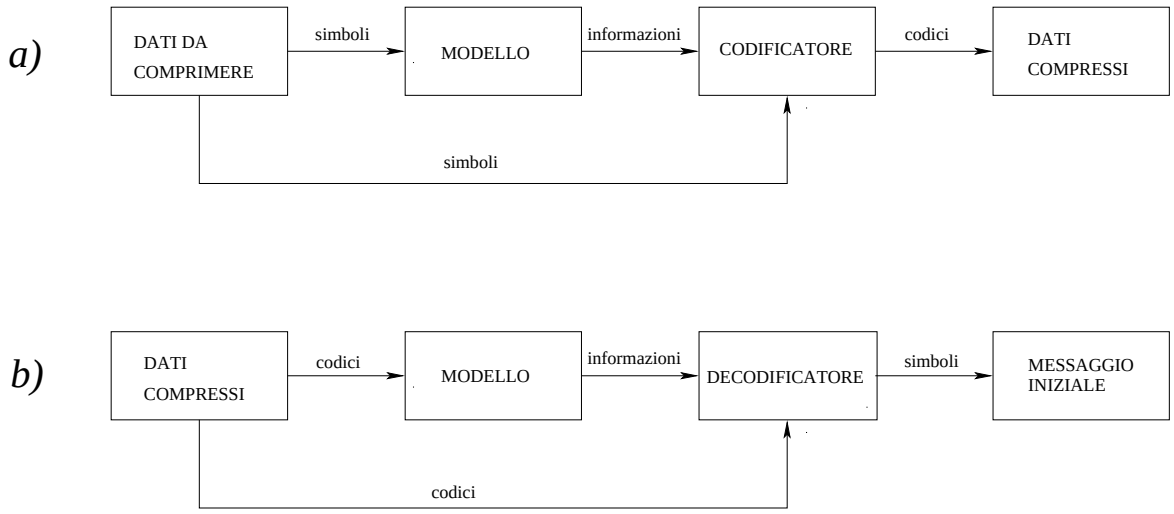


Figura 2.3: Schema generale della compressione (*a*) e della decompressione (*b*)

Va notato che esistono molti tipi diversi di modelli come anche di codifiche e non sono necessariamente legati tra di loro: ad un certo modello possono essere associati diversi codificatori e viceversa.

Vediamo ora alcuni tipi di modello.

Modelli

Una prima divisione che è possibile fare è tra modelli statistici e modelli basati su di un dizionario.

I **modelli statistici** analizzano i dati alla ricerca di simboli o gruppi di simboli. Successivamente creano una tabella in cui ad ogni gruppo di simboli viene associato il numero di volte in cui esso appare nel messaggio. Questa operazione si può effettuare anche solo su di un campione di dati in modo che, una volta normalizzato con la dimensione del

campione, i conteggi rappresentino la probabilità che esso appaia anche in altri messaggi dello stesso tipo. Si dice "ordine" del modello statistico la dimensione meno uno del gruppo di simboli analizzato: un modello di ordine zero analizza un simbolo alla volta, un modello di ordine uno ne analizza due, uno di ordine due ne analizza tre a così via.

Ad esempio un modello di ordine zero applicato ad un testo conteggia il numero di volte in cui compare la lettera "a" e passa al codificatore l'informazione. Quest'ultimo assocerà poi alla lettera "a" il relativo codice compresso basandosi sul conteggio e sostituirà ad ogni "a" del messaggio questo codice compresso. Un modello di ordine uno conteggerebbe non solo la lettera "a" ma anche il gruppo di lettere "aa" "ab" "ac" ecc. L'utilità di fare questo risiede soprattutto nell'analisi di messaggi che contengono schemi ripetuti di simboli come i testi: la probabilità di incontrare la lettera "q" nella maggior parte delle lingue occidentali è bassa ma la probabilità che una "q" sia seguita da una "u" è altissima, quindi codificare "qu" con un unico simbolo è più efficiente.

I modelli statistici dovrebbero ottenere una migliore compressione all'aumentare dell'ordine, almeno se applicati a flussi di dati che contengono schemi ripetuti di simboli. Il prezzo di questa maggior efficienza è una maggior richiesta di memoria: nel caso dei caratteri *ascii* la tabella di un modello di ordine zero contiene 256 simboli (uno per simbolo *ascii*). Un modello di ordine uno crea una tabella di $256^2 = 65536$ simboli e così via per gli ordini superiori. Inoltre la tabella dei simboli va trasmessa insieme al messaggio compresso in modo che il decodificatore possa invertire il processo di compressione. Questo controbilancia in parte la miglior efficienza di compressione.

Nella compressione mediante **modelli a dizionario** ci si concentra soprattutto nella costruzione del modello. La codifica riveste un ruolo molto più limitato rispetto a quella associata a modelli statistici. In questo caso il modello consiste nella ricerca nel flusso dei dati di gruppi di simboli presenti in un dizionario. Quando vengono trovati al loro posto viene trasmesso un codice. Il programma di decompressione, avendo a disposizione lo stesso dizionario, risostituisce ai codici il gruppo di simboli corrispondente. Ci si rende conto che questo sistema dipende dal fatto che nel flusso dei dati ci siano gruppi di simboli corrispondenti al dizionario e che quindi la costruzione di quest'ultimo è critica per un buon rapporto di compressione.

Modelli Adattativi

Agli albori dell'era dei calcolatori il costo del tempo macchina dei medesimi era molto alto e quindi gli algoritmi di compressione dovevano essere più veloci possibile. Nella creazione

dei modelli statistici si preferiva utilizzare un campione rappresentativo del messaggio ed utilizzare la tabella di conteggi ottenuta per comprimere tutto il messaggio. Questo poteva essere pericoloso in quanto se il flusso dei dati fosse cambiato ed il campione avesse cessato di essere rappresentativo la compressione avrebbe potuto calare drasticamente.

Successivamente il costo del tempo macchina dei calcolatori calò sensibilmente e nacquero nuovi approcci per affrontare il problema: i nuovi modelli furono chiamati **adattativi**, mentre i modelli utilizzati in precedenza furono chiamati **statici**.

I modelli adattativi si basano su uno schema del tipo riportato in figura 2.4. In fig. 2.4*a* è riportato lo schema di compressione e in fig. 2.4*b* il relativo schema di decompressione.

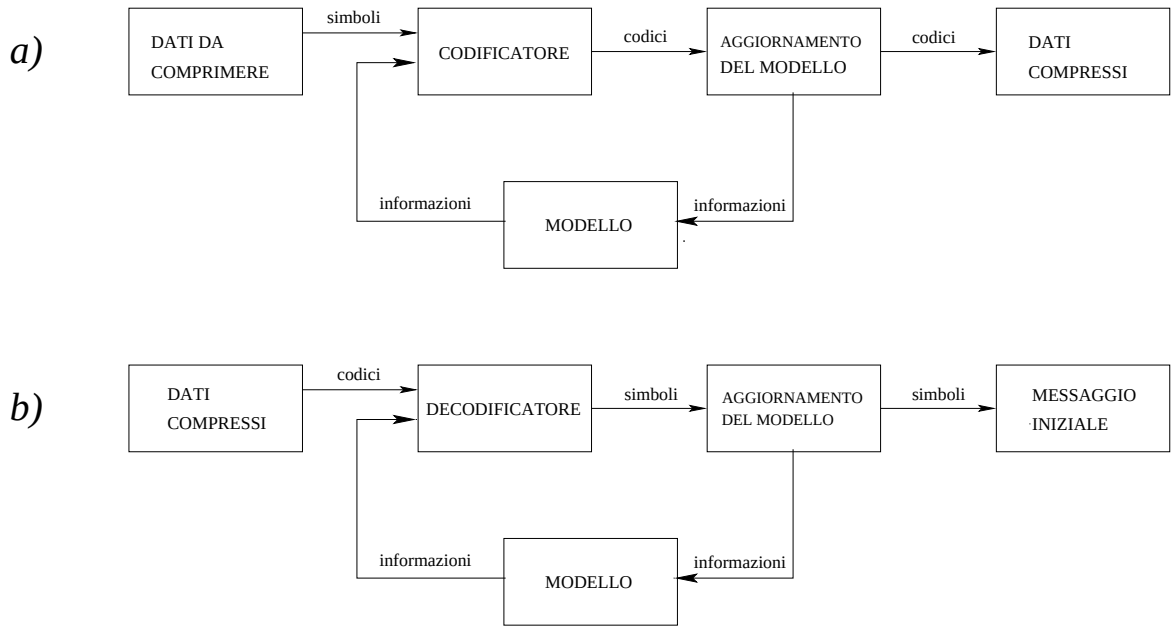


Figura 2.4: Schema della compressione adattativa (*a*) e della decompressione adattativa (*b*).

In pratica, se nel caso dei modelli statici il messaggio viene prima analizzato per farne un modello e poi codificato, nel caso dei modelli adattativi il modello viene costruito durante la codifica.

Questo significa che durante la compressione il modo di codificare lo stesso simbolo o gruppo di simboli cambia al variare del modello e quindi del messaggio stesso. Ad esempio, nel caso di un algoritmo di compressione con modello adattativo statistico di ordine zero applicato ad un testo, la lettera "b" potrebbe venir trasmessa con il relativo carattere *ascii* la prima volta che viene incontrato (questo dipende dal codificatore). La seconda volta potrebbe essere codificato con il relativo codice compresso, la terza con un altro codice compresso che potrbbe non essere lo stesso di prima e così via. Idealmente

tanto più spesso la lettera "b" viene incontrata tanto più corto sarà il codice relativo e viceversa.

I vantaggi dei modelli adattativi sono molteplici: anzitutto il modello si adatta al flusso dati. Se la natura dei dati cambia nel procedere dell'analisi il modello si conformerà fino a ritrovare l'optimum. Un secondo vantaggio è che non è più necessario trasmettere tavole di simboli o dizionari: il processo di decompressione ricostruirà il modello a partire dai dati compressi in modo perfettamente identico al programma di compressione.

Riprendendo l'esempio sopra, il programma di decompressione "sa" che ogni volta che incontra un simbolo non compresso, ad asempio la lettera "b", questa è incontrata per la prima volta, quindi aggiorna il modello in modo identico a quanto fatto a suo tempo dal progamma di compressione, creando un codice per "b". Ora ogni nuovo simbolo provoca un aggiornamento del modello e quindi del codice relativo a "b". Se il programma di decompressione incontrerà questo codice, esso "saprà" che corrisponde al simbolo "b", sostituirà ad esso il codice *ascii* per "b" ed aggiornerà il modello.

L'uso dei modelli adattativi ha anche degli svantaggi. Anzitutto richiede una potenza di calcolo maggiore rispetto a quelli statici: il modello deve venire continuamente aggiornato ed il codificatore adattarsi a sua volta. Inoltre il modello adattativo parte senza informazioni iniziali sul flusso di dati, quindi all'inizio produrrà tipicamente una compressione peggiore rispetto ad un modello statico (se si esclude in quest'ultimo la trasmissione della tabella dei conteggi).

Codifica

Fin dalla fine degli anni '40 , quando Claude Shannon cominciò ad occuparsi dell'entropia di un messaggio [8], il panorama della compressione dati fu dominata dall'idea di una codifica a minima ridondanza. Una codifica, cioè, che cerca di eliminare le ridondanze di un messaggio.

Alla normale codifica dei simboli viene sostituita un'altra codifica più efficiente, che utilizza un numero non fisso di *bit*. Si noti che le locazioni di memoria nei calcolatori hanno invece lunghezza fissa, quindi una serie di simboli codificati in modo compresso possono venire inseriti in una singola locazione di memoria o spezzata in diverse locazioni.

Algoritmo di Shannon-Fano

Shannon e Fano svilupparono un algoritmo basato su di un modello statistico di ordine zero. Il metodo è il seguente:

- Una volta ottenuta la tabella dei conteggi dal modello, la si riorganizza in ordine crescente o decrescente di conteggio.

- La si divide in due parti in modo che la somma dei conteggi delle due parti sia il più possibile uguale.

- Si assegna ad una delle due sottotabelle il *bit* 0 ed all'altra il *bit* 1.

- Si ripete la procedura sopra iterativamente fino a che ognuna delle sottotabelle contiene un solo simbolo.

A questo punto, partendo da un simbolo e seguendo l'albero a ritroso si ha quindi che ad ogni simbolo della lista è associato un gruppo di *bit*, che verranno utilizzati come codice per comprimere il messaggio. Ci si rende conto che, poiché si divide ogni volta una sottotabella in modo che la somma delle due parti sia il più possibile uguale, la parte di tabella che contiene i simboli più "frequenti" subirà meno suddivisioni e quindi ad essi verranno assegnati gruppi di *bit* più piccoli.

A questo punto, per operare la compressione, non resta che sostituire ad ogni simbolo del messaggio il rispettivo codice a *bit*. Per chiarire le procedure facciamo un esempio: ammettiamo che ad un certo messaggio, composto da 39 simboli, corrisponda la tabella 2.3, ottenuta con un modello statistico di ordine zero.

Nelle tabelle da 2.4 a 2.6 si possono osservare i vari passaggi per ottenere il codice finale. Nella tabella 2.4 viene operata la prima divisione, che cade tra B e C. Infatti la

Simbolo	**Conteggio**
A	15
B	7
C	6
D	6
E	5

Tabella 2.3: Esempio di tabella dei conteggi di un messaggio di 39 simboli.

somma dei conteggi della sottotabella superiore (che contiene i simboli A e B) è pari a 22 mentre è 17 per la sottotabella inferiore (che contiene i simboli C, D ed E). Nessun'altra divisione ottiene somme così vicine. Alla tabella superiore viene assegnato il *bit* 0 ed all'altra il *bit* 1.

Nella tabella 2.5 si continuano le suddivisioni delle sottotabelle. La tabella che contiene i simboli (A e B) vengono divise in sottotabelle contenenti i soli simboli A e B

Simbolo	Conteggio	Somma	*bit* assegnato
A	15	22	0
B	7		
C	6	17	1
D	6		
E	5		

Tabella 2.4: Prima divisione.

ripettivamente. La tabella che contiene i simboli C, D ed E viene divisa in due sottotabelle, una contente il simbolo C (con somma pari a 6), l'altra i simboli D ed E (con somma pari a 11). Si noti che se si fosse diviso la seconda tabella tra D ed E, le somme sarebbero state 12 e 5, con una differenza maggiore rispetto a 6 e 11 che è la divisione corretta. Ancora una volta si assegna alle sottotabelle superiori il *bit* 0 ed alle sottotabelle inferiori il *bit* 1.

Simbolo	Conteggio	Somma	*bit* assegnati
A	15	15	00
B	7	7	01
C	6	6	10
D	6	11	11
E	5		

Tabella 2.5: Seconda divisione.

Simbolo	Conteggio	Somma	*bit* assegnati
A	15	15	00
B	7	7	01
C	6	6	10
D	6	6	110
E	5	5	111

Tabella 2.6: Terza divisione.

Nella tabella 2.6 si esegue l'ultima divisione possibile: quella tra D ed E. Ora basta assegnare i *bit* analogamente a quanto fatto in precedenza. A questo punto ad ogni simbolo si trova assegnato un insieme di *bit*, un codice. È sufficiente sostituire ad ogni simbolo il relativo codice per operare la compressione.

In tabella 2.7 si possono osservare i risultati della codifica confrontati con il contenuto di informazione di ogni simbolo del messaggio.

Simbolo	Conteggio	Lunghezza del codice di Shannon-Fano
A	15	2
B	7	2
C	6	2
D	6	3
E	5	3

Tabella 2.7: Lunghezze dei codici di Shannon-Fano ottenuti partendo dalla tabella dei conteggi 2.3.

Se si considera che normalmente i simboli sono memorizzati mediante codici *ascii* a 8 *bit*, si ottiene un notevole risparmio di spazio. Il messaggio sopra è composto da 39 simboli, che avrebbero richiesto 312 *bit* di memoria se utilizzassimo i codici *ascii*, mentre con la codifica di Shannon-Fano ne sono sufficienti 89, con un $RdC \backsimeq 71.5\%$

Codifica di Huffmann

Nel 1952 D. A. Huffmann introdusse un nuovo metodo di codifica [9], che divenne la base per la maggior parte degli algoritmi di compressione da allora in poi, compresi quelli odierni, dominando il panorama di compressione fino ai primi anni '80.

Il metodo di Shannon-Fano costruisce un "albero" a partire dall'alto: divide la tabella in sottotabelle fino a trovare solo sottotabelle contenenti un singolo simbolo. Per conoscere il codice associato ad un dato simbolo è sufficiente risalire l'albero a partire dal simbolo interessato fino in cima annotando il *bit* associato ad ogni "ramo".

Anche il metodo di Huffmann si basa sulla costruzione di un albero ed il processo di codifica è esattamente lo stesso. La differenza sta nella costruzione dell'albero: quello di Huffmann si costruisce dal basso.

Il metodo è il seguente:

- Una volta ottenuta la lista di conteggi (che chiameremo "pesi" associati ai simboli) la si organizza in ordine decrescente di peso, per esempio partendo da sinistra con i simboli più pesi e arrivando a destra con i simboli meno pesi.

- Si prendono i due simboli con il peso più basso e li si collega ad un "nodo".

- Si eliminano i due simboli dalla lista ed al loro posto si mette il nodo, che si comporterà come un simbolo fittizio dal peso pari alla somma dei pesi dei simboli ad esso collegati.

- Si riorganizza di nuovo la lista in ordine di peso in modo che il nodo si trovi al posto

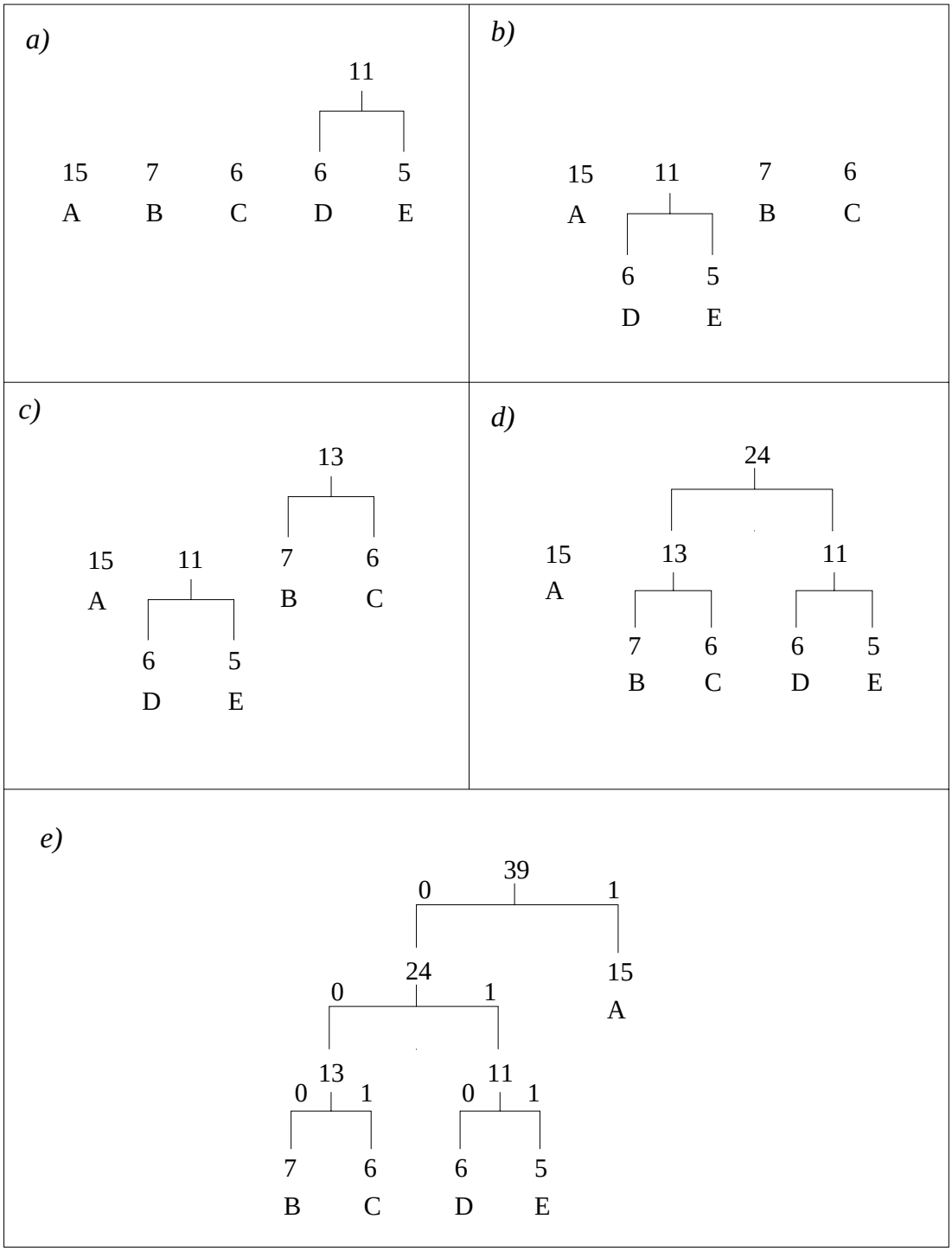


Figura 2.5: Schema di costruzione di un albero di Huffmann (i numeri sopra i simboli ed i nodi sono i pesi). Si collegano i due nodi più leggeri (a). Si sposta il nuovo nodo al posto giusto guardandone il peso (b). Di nuovo si collegano i due nodi più leggeri (c). Nuovo riordino dei nodi in base al peso e collegamento dei nodi più leggeri (d). In (e) si può vedere l'albero di Huffmann definitivo con aggiunti i *bit* (0 o 1) di codifica per ogni nodo.

giusto. Difatti il nodo ha un peso che è la somma dei simboli ad esso collegati, potrebbe dover essere spostato più in alto nella lista.

Si ripetono i passaggi sopra, finché tutti i simboli fanno capo ad un unico nodo.

A questo punto non resta che assegnare ad ogni nodo un *bit*, ad esempio al nodo di sinistra di ogni coppia il *bit* 0 ed all'altro il *bit* 1 (questa è una convenzione ma va fissata e condivisa dal decodificatore).

Simbolo	**Conteggio**	**Lunghezza del codice di Huffmann**	**Codice di Huffmann**
A	15	1	1
B	7	3	000
C	6	3	001
D	6	3	010
E	5	3	011

Tabella 2.8: Codici di Huffmann e loro lunghezza ottenuti partendo dalla tabella di conteggi 2.3

Ripetendo l'esempio già utilizzato con il metodo di Shannon-Fano, si puó ossevare il processo passo per passo nella figura 2.5 e si ottengono i risultati riportati in tabella 2.8.

Confrontando la tabella 2.8 del metodo di Huffmann con la tabella 2.7 del metodo di Shannon-Fano, si vede che usando la codifica di Huffmann il messaggio occupa 87 *bit* ed ha un $RdC \backsimeq 72.1\%$, ottenendo due *bit* in meno rispetto al caso di Shannon-Fano che corrisponde allo 0.6% in RdC. Se pure il risultato è simile, si può dimostrare che il metodo di Huffmann otterrà sempre un risultato migliore o almeno uguale a quello di Shannon-Fano [10].

Se l'albero di Huffmann viene costruito correttamente con il metodo descritto sopra e se numeriamo i nodi progressivamente mentre li assegnamo, questi possiedono una particolare proprietà, detta "di fratellanza" (proprietà di *sibling*). Secondo questa proprietà [9] i nodi saranno in ordine decrescente di peso (dall'alto verso il basso secondo lo schema di figura 2.5e) ed inoltre si può dimostrare che in ogni coppia di nodi "figli" di uno stesso nodo, il nodo di sinistra avrà sempre peso maggiore o uguale di quello di destra. Questo è utile nella codifica con modello adattativo, in cui l'albero va costruito in corsa. In questo caso ci serviremo delle regole per costruire e modificare un albero di Huffmann regolare.

Capitolo 3

Applicazioni allo spettrometro tracciante di PAMELA

3.1 Il processore di segnali digitali

In questo capitolo illustrerò gli algoritmi implemententati sui DSP del rivelatore a microstrisce al silicio di PAMELA.

Il processore di segnali digitali (*digital signal processor*, D.S.P.) è il cuore dell'*hardware* dedicato alla compressione dei dati (eseguita implementando i programmi di compressione in essi) di PAMELA e quindi conviene descriverne la struttura in dettaglio.

In PAMELA si utilizzano *chip* della *Analog Devices*, della famiglia 218X. Ci sono alcune differenze di architettura tra le varie famiglie di DSP (perfino all'interno di una stessa famiglia) e alcuni particolari della programmazione cambiano a seconda del tipo di architettura. I programmi che riguardano questa tesi sono stati sviluppati per la famiglia 218X.

Il cuore del processore è costituito da tre unità di analisi: l'unità logico-aritmetica (*arithmetic-logic unit*, ALU), il moltiplicatore-accumulatore (*multiplier-accumulator*, MAC) ed infine il traslatore (*barrel shifter*).

Il DSP contiene inoltre due generatori di indirizzi ed un *program sequencer*. I primi forniscono gli indirizzi della memoria dati e della memoria programmi anche simultaneamente, e questo permette notevoli risparmi di tempo. Il secondo genera le sequenze di istruzioni e provvede al controllo del flusso dei programmi.

Il processore dispone di registri di memoria innestati nel *chip*. La memoria a disposizione è divisa in memoria dati (*data memory*) e memoria programmi (*program memory*). La *data memory* ha parole di 16 *bit*, mentre la *program memory* di 24 *bit*. La memoria a disposizione varia da modello a modello. Il lavoro di tesi si è incentrato sull'uso dei processori della famiglia 218X, in particolare sui 2185 e 2187. Nel modello 2185 sia la

memoria dati che la memoria programmi sono composte da 16383 parole. Nel modello 2187 le dimensioni delle memorie sono doppie rispetto al 2185.

Inoltre il DSP dispone di scambiatori interni di indirizzi che permettono trasferimenti molto efficienti da e per le memorie interne. Questi trasferimenti sono effettuati in maniera completamente asincrona e possono avvenire anche quando il DSP sta lavorando a piena potenza.

Il dialogo con l'esterno del processore è effettuato mediante due porte seriali (*sport0* e *sport1*) che possono essere pilotate da un'ampia gamma di dispositivi in circolazione. Ognuna di esse può essere usata con un *clock* interno programmabile oppure con un *clock* esterno. Sono presenti poi altre due porte non seriali, IDMA (*internal data memory address*) e BDMA (*byte data memory address*), che possono essere utilizzate in maniera completamente asincrona rispetto alle operazioni che vengono compiute dal DSP sotto il controllo del programma.

L'accoppiamento ed il trasporto dati avviene attraverso una serie di bus interni: a 14 *bit* per gli indirizzi, a 16 *bit* per la memoria dati e l'interscambio dei registri delle unità di calcolo, a 24 *bit* per la memoria programmi.

L'architettura del *chip* è riportata in figura 3.1.

Programmazione

La programmazione del DSP può essere eseguita sia in C che in *assembler*.

Il primo è un linguaggio evoluto che permette di utilizzare alcune caratteristiche complesse, come le strutture, utili soprattutto nella codifica di Huffmann.

Il secondo è un linguaggio di programmazione a basso livello che permette di programmare il DSP in modo diretto. I comandi più usati in questo linguaggio sono operazioni di lettura/scrittura sulle memorie e istruzioni per la ALU (somme fra addendi, sottrazioni, operazioni logiche ecc.)*.

A differenza dei linguaggi *assembler* di alcuni anni fa, molto complicati, l'*assembler* della famiglia 218X è accessibile con relativa facilità. Gli attuali compilatori permettono una programmazione a livello molto basso (ovvero con un accesso particolareggiato all'architettura del processore) ed al contempo abbastanza evoluta da facilitare la creazione di programmi anche complessi.

La compilazione dei programmi in C può essere eseguita anche in modo che il compi-

*Un breve riassunto dei comandi principali può essere trovato in Appendice A.

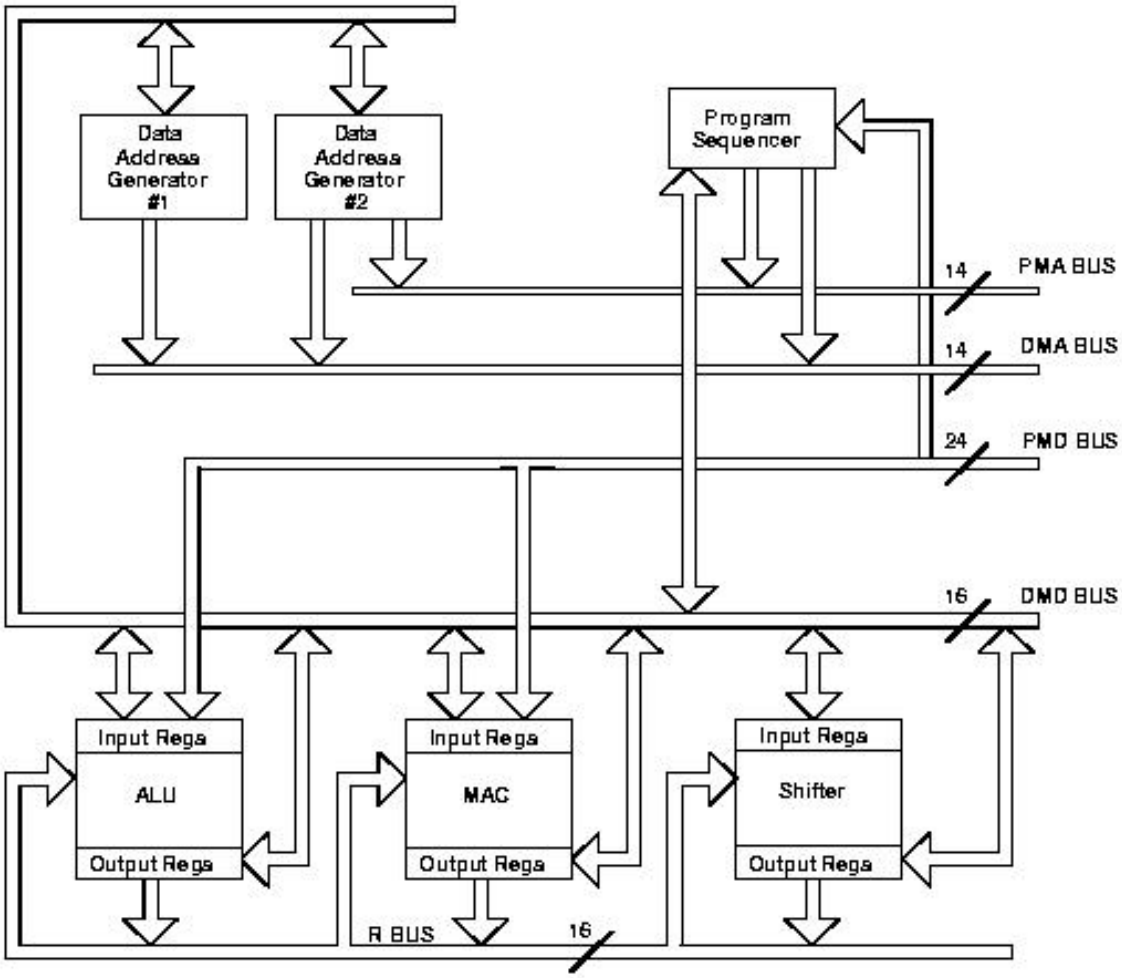


Figura 3.1: Architettura base del DSP (figura tratta da [11]).

latore produca una versione del programma in *assembler*, invece che direttamente l'eseguibile.

Velocità e consumi

Il DSP è in grado di eseguire operazioni ad alta velocità. Naturalmente è necessario giungere ad un compromesso tra velocità e potenza dissipata, in quanto la prima è in stretta relazione con la seconda.

I DSP preposti alla trattazione dei dati del rivelatore a microstrisce di silicio in PAMELA verranno alimentati con una tensione di 3.3 V e fatti lavorare con una frequenza di *clock* di circa 33 MHz. Questo significa che il DSP impiega circa 30 ns per eseguire ogni istruzione.

Il consumo di potenza interno del DSP a questa frequenza di lavoro ed a questa tensione di alimentazione si aggira attorno ai 100 mW durante l'elaborazione, mentre è di circa 20 mW nello stato di quiescenza.

Difatti il DSP è dotato di una funzione (*IDLE*, vedi appendice A), che gli permette di porsi in uno stato quiescente (di risparmio energetico) in attesa di un segnale di interruzione (*interrupt*). Quando un *interrupt* viene mandato al DSP, questo esegue un'istruzione predefinita dalla tabella degli *interrupt* (vedi appendice A). Il DSP dispone di diversi possibili segnali di *interrupt* a cui possono essere collegate diverse istruzioni.

Ad esempio è possibile che l'istruzione da eseguire sia un salto all'inizio di un programma, che saremo così in grado di far partire mediante un segnale esterno.

I registri

All'interno delle unità computazionali (ALU, MAC e traslatore) sono presenti delle celle di memoria dedicate, dette registri. Quando una delle unità deve eseguire una operazione, i dati da processare devono essere caricati in appositi registri di ingresso. Quando poi l'operazione è stata eseguita, il risultato viene caricato in appositi registri di uscita. In realtà l'utilizzo dei registri non è così rigida: molti registri di uscita possono essere utilizzati come registri di ingresso, anche appartenenti a diverse unità computazionali, permettendo una programmazione più flessibile. Esistono poi dei registri adibiti a funzioni speciali. In tabella 3.1 sono riassunti i vari registri.

Tutti i registri di dati (AX, AY, MR ecc.) sono doppi, ovvero ne esiste una copia normalmente inattiva. È possibile passare a questo insieme alternativo di registri cambiando un paricolare *bit* in uno speciale registro, detto registro di stato del processore (*processor*

registro	numero di *bit*	unità computazionale di appartenenza
AX0	16	ALU
AY0	16	ALU
AX1	16	ALU
AY1	16	ALU
AR	16	ALU
AF	16	ALU
MX0	16	MAC
MY0	16	MAC
MX1	16	MAC
MY1	16	MAC
MR0	16	MAC
MR1	16	MAC
MR2	8	MAC
MF	16	MAC
SI	16	Traslatore
SE	8	Traslatore
SB	5	Traslatore
SR0	16	Traslatore
SR1	16	Traslatore

Tabella 3.1: Tabella riassuntiva dei registri presenti nelle varie unità computazionali.

mode status register, MSTAT). Ogni *bit* di questo registro controlla lo stato dei vari sistemi del DSP. Non è possibile utilizzare contemporaneamente i registri ed i loro doppi, ma è possibile interrompere un'elaborazione in corso, passare ai registri alternativi ed iniziare un'elaborazione alternativa (e viceversa). Questo permette di eseguire cambiamenti di contesto velocissimi.

I vari registri sono scrivibili e leggibili direttamente, senza passare dalle unità di calcolo, tranne i registri AF e MF. La lettura e scrittura sono possibili da e verso sia la memoria dati che la memoria programmi. Come si può vedere in figura 3.1 il trasporto dei dati è affidato ad appositi *bus* interni, a 16 *bit* per la memoria dati e 24 *bit* per la memoria programmi. Per permettere lo scambio di dati tra la memoria dati e la memoria programmi sono presenti apposite unità di scambio. Nel caso di passaggio tra memoria programmi e memoria dati vengono passati i 16 *bit* meno significativi.

L'elaborazione dei programmi richiede che sia possibile assegnare indirizzi alle memorie. A questo scopo sono presenti due unità, dette generatori di indirizzi (*data address generator* o DAG, vedi fig. 3.1) che consentono di gestire l'indirizzamento delle variabili.

Ogni DAG (DAG1 e DAG2) può gestire fino a quattro indirizzi. Questo avviene tramite tre tipi di registri, tutti a 14 *bit*.

- I registri **I** che che contengono gli indirizzi veri e propri.

- I registri **M** che permettono di modificare l'indirizzo contenuto nei registri I (nella lettura/scrittura indiretta della memoria, vedi il paragrafo seguente).

- I registri **L**, usati nella definizione dei *buffer* circolari (un valore uguale a zero indica un *buffer* non circolare).

I registri sono numerati da 0 a 3 per il DAG1 (I0, M3, L2 ecc.) e da 4 a 7 per il DAG2 (M7, L5, I4 ecc.). Ognuno dei registri M di un DAG può essere usato per modificare uno qualsiasi dei registri I dello stesso DAG, ma non dell'altro.

Solo il DAG2 può generare indirizzi per la memoria programmi, mentre quelli per la memoria dati possono essere generati da entrambi i DAG. Inoltre il DAG2 è in grado di invertire l'ordine dei *bit* degli indirizzi in esso contenuti prima che vengano mandati alla loro destinazione.

La memoria

La memoria può essere letta o scritta sia direttamente (dichiarando esplicitamente l'indirizzo della locazione), sia indirettamente (scrivendo in un registro apposito (I#) l'indirizzo della locazione che deve essere scritta o letta).

Ad esempio, se si vuole trasferire nel registro AX1 il valore della locazione 0x02a3 della memoria dati, ciò può essere fatto direttamente con l'istruzione *assembler* `AX1=dm(0x02a3);` o indirettamente mediante l'istruzione `AX1=dm(I1,M1);` dove precedentemente nel registro I1 è stato posto il valore 0x02a3. In questo esempio M1 è un registro il cui valore indica l'incremento che subisce il valore di I1 dopo la lettura. Nel nostro esempio, se M1=1, dopo avere letto la locazione 0x02a3 in modo indiretto il valore in I1 verrà incrementato di 1 ed una successiva lettura `AX1=dm(I1,M1);` leggerà la locazione di memoria 0x02a4.

Si noti che poiché I1 appartiene al DAG1 l'esempio sopra potrebbe essere stato fatto anche con M0, M2 e M3, ma non con M4, M5, M6 e M7, che appartengono al DAG2. Inoltre, se la lettura/scrittura avesse riguardato la memoria programmi, avrebbero potuto essere usati soltanto i registri del DAG2.

La memoria programmi può essere letta e scritta solo in modo indiretto.

ALU (*arithmetic-logic unit*)

La ALU permette una serie di operazioni aritmetiche e logiche. Le prime sono somma, sottrazione, incremento, decremento, valore assoluto, negazione. Le seconde sono AND, OR, XOR, NOT. Esiste poi un'operazione (PASS) che fa passare l'operando attraverso la ALU senza nessuna modifica.

I registri che contengono gli operandi della ALU vanno divisi in due classi: i registri X ed i registri Y. Molte delle operazioni della ALU hanno delle limitazioni sull'uso di questi registri. Ad esempio nelle operazioni di somma e sottrazione di due numeri è necessario che uno di essi venga caricato in un registro X e l'altro in un registro Y.

Per una migliore flessibilità il sistema di bus interni permette l'uso anche dei registri delle altre unità di calcolo, ma solo per gli operandi. I registri X sono AX0, AX1, AR, MRO, MR1, SR0 e SR1, i registri Y sono AY0, AY1 e AF.

Gli operandi delle operazioni della ALU vengono inseriti nei registri X e Y ed il risultato viene spedito nel registo AR o AF. Il registro AR è il registro principale per i risultati della ALU. Il registro AF è un registro di *feedback* immediato, che permette di ricaricare il risultato di un'operazione di nuovo nella ALU. Il registro AF ha un uso leggermente più limitato rispetto agli altri registri in quanto non vi si può spostare direttamente il contenuto di altri registri o di una locazione di memoria (ad esempio `AF=AX1;` è un'istruzione illegale). Per caricare un valore in AF, esso deve passare necessariamente dalla ALU (ad esempio `AF=PASS AX1;`).

La ALU può inoltre operare anche la divisione mediante un circuito apposito.

Il moltiplicatore-accumulatore (MAC)

Come nella ALU, anche nel MAC i registri in cui caricare gli operandi si possono dividere in due classi X e Y. I registri X ammessi sono MX0, MX1, MR0, MR1, MR2, AR, SR0 e SR1. I registri Y ammessi sono MY0, MY1 e MF.

Il risultato dell'operazione del MAC viene mandato nel registro MR. Questo è un registro particolare che può essere visto come un unico registro a 40 *bit*, oppure come tre registri (MR0, MR1 e MR2 rispettivamente a 16, 16 e 8 *bit*). Quando viene eseguita un'operazione del MAC, che essendo un moltiplicatore può dare risultati molto grandi, il risultato viene mandato in MR come se fosse un singolo registro: a questo punto il registro MR0 contiene i 16 *bit* meno significativi del risultato, MR1 i successivi 16 *bit* e MR2 gli 8 *bit* più significativi. In questo modo si può manipolare il risultato agevolmente, essendo

diviso in parole di 16 e 8 *bit*. Inoltre la divisione in tre registri rende l'utilizzo di MR molto più flessibile che se fosse un unico registro a 40 *bit*.

Il risultato può essere mandato anche nel registro MF, ma esso, essendo ampio solo 16 *bit*, contiene solo l'equivalente di MR1 (i 16 *bit* successivi ai 16 *bit* meno significativi).

Il traslatore (barrel shifter)

Il traslatore permette di traslare un numero in formato binario verso il *bit* più significativo o meno significativo di un certo numero di posti: ad esempio il numero "1" potrebbe essere spostato di un posto verso "sinistra" (verso il *bit* più significativo) diventando il numero "2" in binario.

Per fare ció l'operando viene caricato nel registro SI (che è un registro a 16 *bit*) ma può essere caricato anche nei registri di altre unità, quali AR, MR0, MR1, MR2, SR0 e SR1.

Il traslatore è ampio 32 *bit*. Il valore da traslare può essere caricato nei 16 *bit* più significativi o nei 16 *bit* meno significativi. Il valore viene poi traslato di un certo numero di *bit*, che può essere contenuto nel registro SE (volendo è possibile negare il numero memorizzato nel registro SE prima di essere usato), oppure direttamente nell'istruzione *assembler* che ha chiamato in causa il traslatore. A questo punto il risultato viene passato al registro SR. Come il registro MR nel MAC, il registro SR è particolare in quanto può essere visto come unico registro ampio 32 *bit*, oppure come due registri: SR0 e SR1, ognuno dei quali è ampio 16 *bit*. In SR0 vengono posti i 16 *bit* meno significativi del risultato ed in SR1 i 16 *bit* più significativi.

Le operazioni più importanti sono la traslazione aritmetica e la traslazione logica. La differenza tra le due risiede nel fatto che nella traslazione logica, quando un numero viene traslato verso destra i *bit* che compaiono a sinistra vengono riempiti con degli 0, mentre nella traslazione aritmetica vengono riempiti con degli 1. Negli spostamenti verso sinistra i *bit* che compaiono a destra vengono sempre riempiti con degli 0.

Ad esempio si inserisca il numero 3 (0000000000000011) nei 16 *bit* più significativi del traslatore e spostiamo il numero di un *bit* verso destra. Nel caso della traslazione logica SR1 contiene ora il numero 1 (0000000000000001) e SR0 il numero 32768 (1000000000000000). Nel caso della traslazione aritmetica si ottiene lo stesso risultato in SR0 ma in SR1 si avrebbe il numero 32769 (1000000000000001).

3.2 Implementazione degli algoritmi

Un singolo DSP del tracciatore di PAMELA gestisce 3 unità di rivelazione per un totale di $1024 \times 3 = 3072$ strisce.

Quello che ci aspettiamo, una volta che i dati provenienti dai rivelatori sono passati dai convertitori analogico-digitali al DSP, è qualcosa come in figura 3.2.

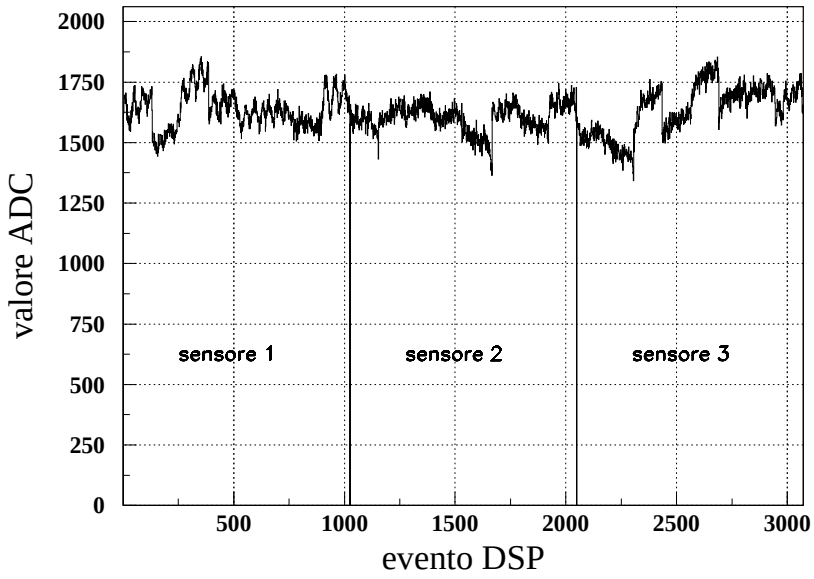


Figura 3.2: Tipico risultato della lettura di tre unità di rivelazione (1024 strisce) del rivelatore a microstrisce di silicio.

In pratica, come descritto in dettaglio nel paragrafo 3.3, ogni canale fluttua evento per evento attorno ad un valore di equilibrio detto "piedistallo". Inoltre ci sono fluttuazioni dei segnali dette "di modo comune", cioè ci sono gruppi di strisce i cui segnali fluttuano in maniera solidale. In figura 3.3 si può vedere un dettaglio (128 strisce) in cui sono riportati diversi eventi sovrapposti. È evidente la fluttuazione solidale delle 128 strisce per i diversi eventi. Queste fluttuazioni si sovrappongono alle fluttuazioni di ogni singolo canale. Le fluttuazioni di modo comune vengono determinate separatamente per ogni VA1, pioché i 128 segnali relativi ad uno stesso VA1 fluttuano solidarmente. Se una particella è passata attraverso il rivelatore depositando energia nel silicio, in corrispondenza delle strisce colpite si ha un ulteriore segnale che si sovrappone ai normali segnali di rumore.

Se dal segnale di ogni striscia sottraiamo il relativo piedistallo, dovremmo quindi ottenere un profilo quasi piatto spostato rispetto allo zero di un valore casuale per ogni evento dovuto alle fluttuazioni di modo comune, con in più un eventuale picco dovuto al

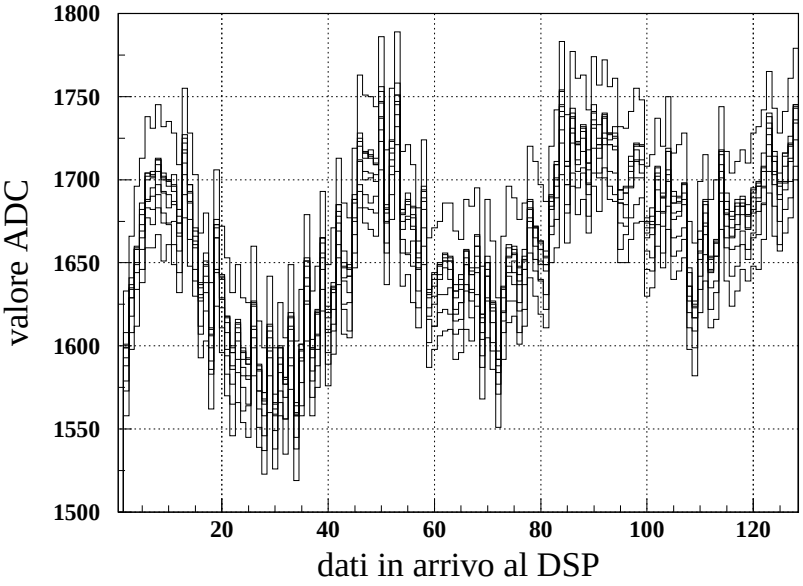


Figura 3.3: Sovrapposizione di un campione di eventi per 128 strisce.

passaggio di una particella. La posizione di questo picco è l'informazione che ci interessa maggiormente. In figura 3.4 si possono vedere 128 strisce per un campione di più eventi di sovrapposti, dopo aver effettuato la sottrazione dei piedistalli. Si noti la diversa scala delle ordinate rispetto alla figura 3.3. Si vede che le strisce assumono un profilo quasi piatto con valori che fluttuano attorno ad un valore comune per i diversi eventi.

Dopo la sottrazione del piedistallo, data la natura dei dati da analizzare (3072 valori per DSP fluttuanti attorno ad un valore costante, eventualmente con un picco), si vede che non conviene utilizzare algoritmi di compressione particolarmente elaborati.

Per quanto riguarda gli algoritmi di compressione reversibili, è evidente che i modelli statistici di ordine superiore allo zero sono inutili, in quanto è improbabile che si abbiano gruppi di simboli ripetuti in sequenza, senza contare la proibitiva richiesta di memoria. Allo stesso modo i modelli basati su di un dizionario fanno affidamento sulla presenza nel flusso dati di ripetizioni di gruppi di simboli, quindi sono di scarsa utilità in questo specifico caso.

Anche gli algoritmi non reversibili basati sul passagio al dominio delle frequenze, come la trasformata di Fourier, sono di limitata applicabilità: abbiamo una distribuzione quasi piatta con un eventuale picco largo 3-4 strisce che assomiglia ad una *delta* di Dirac. È noto che la *delta* di Dirac contiene tutte le frequenze, quindi applicare una maschera in frequenza ai dati rischia di degradare l'informazione che ci serve di più.

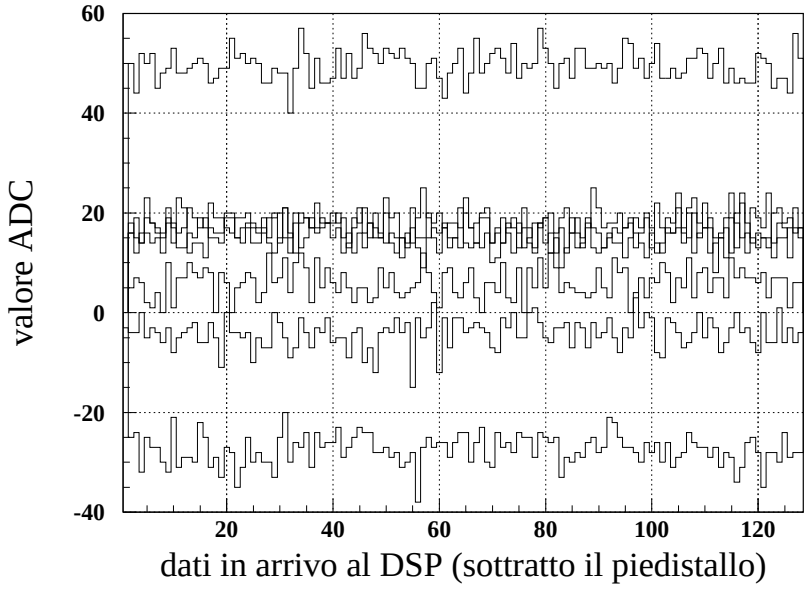


Figura 3.4: Sovrapposizione per 128 strisce di un campione di eventi dopo aver effettuato la sottrazione dei piedistalli.

Si è deciso quindi di usare un semplice ZOP (*zero order predictor*) per quanto riguarda gli algoritmi di compressione non reversibili ed una codifica di Huffmann applicata ad un modello statistico adattativo di ordine zero tra gli algoritmi reversibili (vedere capitolo 2).

Il programma implementato sul DSP è costituito da un programma di analisi, comune a tutti gli algoritmi di compressione che svolgerà alcune funzioni prima e dopo aver passato il comando all'algoritmo di compressione che opererà la compressione dati vera e propria.

Nei prossimi paragrafi descriverò il programma di analisi del DSP e successivamente l'implementazione degli algoritmi di compressione (nell'appendice B viene riportato integralmente il codice *assembler* del programma di analisi, del predittore di ordine zero e dell'algoritmo reversibile di Huffmann).

Il programma di analisi

Riassumiamo quanto già detto nel capitolo 1 a proposito dell'acquisizione dati dello spettrometro al silicio. Quando una particella attraversa il telescopio la seguente catena di eventi prende inizio:

- Il sistema di *time of flight* (TOF) genera un segnale di *trigger* che segnala l'inizio del-

l'evento. Il segnale viene inviato ai DSP preposti alla trattazione dati del rivelatore ed all'elettronica di *front-end*.

- Dopo circa $1\mu s$ dall'arrivo del segnale di *trigger*, il circuito di *sample&hold* dei VA1 si apre e la carica integrata viene bloccata in attesa di venire convertita in segnali digitali da trasmettere ai DSP.

- Il programma di acquisizione dei DSP parte attraverso l'utilizzo di un segnale di *interrupt*. Prima di iniziare ad analizzare i dati esso attende che i dati siano stati tutti scritti nelle memorie interne.

- Dopo $2\mu s$ dall'arrivo del segnale di *trigger* gli ADC iniziano a leggere e convertire in forma digitale a 12 *bit* la tensione ai capi dei condensatori di *hold* dei VA1, dopodiché i dati digitali vengono scritti nelle memorie dei DSP attraverso la porta IDMA.

- Dopo $1.1ms$ dall'arrivo del segnale di *trigger* il processo di scrittura è completato ed il DSP inizia l'analisi dei dati vera e propria.

Il programma di analisi coordina il trattamento dei dati nel DSP. Lo scopo del programma è quello di fornire in uscita (o, come verrà detto nel seguito, "trasmettere") i dati in formato compresso che saranno memorizzati in un supporto di memoria di massa. La trasmissione dal DSP avviene tramite la porta *sport1*.

Al momento del caricamento del programma all'interno del DSP una routine iniziale viene eseguita automaticamente. Essa inizializza le variabili e chiama alcune subroutine che azzerano le memorie. Alla fine di questa routine si trova un ciclo infinito contenente un'istruzione IDLE (vedi sezione 3.1). A questo punto il DSP rimane in attesa in uno stato di quiescenza a basso consumo fino a che gli viene mandato un segnale di *interrupt* che indica l'inizio di un evento, in questo caso il segnale di *interrupt* viene fornito dal *trigger*.

All'inizio il programma trasmette delle parole dette *header*. Queste consistono in una serie di informazioni tra cui il tipo di compressione utilizzato, il numero progressivo dell'evento, l'indirizzo attuale della porta IDMA ed altre, per un totale di dieci parole. Queste sono:

- Inizio dell'*header* 0xDAF0 se la trasmissione dati non è compressa, 0xDAC0 se lo è.

- Il numero dell'evento (viene incrementato di uno ad ogni evento).

- Il contenuto della *Mailbox* (una variabile che contiene 1 se la trasmissione dei dati è non compressa e 2 se è compressa).

- Contenuto dell'indirizzo attuale della porta IDMA (0x3FE0) spezzato in due parole a 8 *bit*.

- Il contenuto delle variabili *Contr1, Contr2, Contr3, Contr4* (semplici variabili di controllo, non ancora utilizzate nel programma).

- Fine dell'*header*: 0x4AF0 se la trasmissione dati non è compressa, 0x4AC0 se lo è.

La variabile "MailBox" viene caricata nel DSP insieme al programma. Dipendendo dal valore memorizzato, il programma passa il comando ad una subroutine che trasmette i dati senza comprimerli, oppure al programma di compressione qualunque esso sia.

Una volta che la trasmissione dei dati vera e propria ha esaurito il suo compito, il comando ritorna al programma di analisi che trasmette delle parole dette *trailer*: altre quindici parole di controllo analoghe all'*header*.

- Inizio del *trailer* 0xDAF1 se la trasmissione dati non è compressa, 0xDAC1 se lo è.

- Il numero dell'evento (viene incrementato di uno ad ogni evento).

- Il contenuto della *Mailbox* (una variabile che contiene 1 se la trasmissione dei dati è non compressa e 2 se è compressa).

- Contenuto dell'indirizzo attuale della porta IDMA (0x3FE0) spezzato in due parole a 8 *bit*.

- Il contenuto delle variabili *Contr1, Contr2, Contr3, Contr4*.

- Il mumero di parole trasmesse (contenuto nella variabile *WORDTRAS*).

- Una parola di controllo che può essere utilizzata per vedere se i dati si sono degenerati durante la trasmissione, contenuto nella variabile *CHECKSUM* (non ancora implementata).

- Tre parole di controllo *InfoCompr1, InfoCompr2, InfoCompr3* contenenti informazioni sulla compressione (non ancora implementate).

- Fine del *trailer*: 0x4AF1 se la trasmissione dati non è compressa, 0x4AC1 se lo è.

Fatto questo il programma di analisi torna al ciclo infinito della routine iniziale e, una volta incontrata di nuovo l'istruzione IDLE, rimanda il DSP in uno stato quiescente, in attesa del prossimo evento.

3.3 Piedistalli e rumore

Allo scopo di comprimere i dati i vari algoritmi di compressione hanno bisogno di disporre di alcune informazioni riguardo ai segnali che normalmente provengono dalle varie strisce. Ogni striscia manda un segnale fluttuante attorno un valore medio (piedistallo). È necessario calcolare questi valori medi e la varianza delle rispettive fluttuazioni per implementare degli algoritmi efficienti.

A questo scopo periodicamente vengono acquisiti ed analizzati una serie di eventi di calibrazione senza particelle per stimare i piedistalli delle strisce e la relativa varianza delle fluttuazioni. Questi eventi vengono acquisiti senza compressione.

Come descritto precedentemente le fluttuazioni, o rumore, possono essere divise in due componenti principali: il rumore di modo comune e le fluttuazioni di ogni striscia attorno al piedistallo.

Il rumore di modo comune è una fluttuazione casuale comune a tutte le strisce collegate ad uno stesso *chip* VA1. Per stimarlo è sufficiente, per un evento i-esimo, operare una media sui segnali provenienti dalle 128 strisce di ogni VA1 secondo la formula:

$$RMC^i = \sum_{k=1}^{128} \frac{SEGN_k^i - PED_k}{128} \tag{3.1}$$

dove PED_k è il piedistallo della k-esima striscia (definito nel seguito), $SEGN_k^i$ è il segnale relativo alla k-esima striscia ed all'i-esimo evento. In questo caso RMC^i è il rumore di modo comune relativo ad un certo *chip* VA1. In pratica quindi si tratta di un livello uguale per tutte le strisce di uno stesso *chip* VA1 ma diverso da evento a evento.

Una volta acquisito il campione di eventi per la calibrazione i piedistalli vengono stimati con la seguente formula:

$$PED_k = \sum_{i=1}^{N_{ev}} \frac{SEGN_k^i - RMC^i}{N_{ev}} \tag{3.2}$$

dove RMC^i è il rumore di modo comune dell'i-esimo evento, mentre N_{ev} è il numero di eventi utilizzati per operare la calibrazione.

Inoltre viene calcolata anche la varianza delle fluttuazioni di ogni singola striscia mediante la formula:

$$SIG_k = \sqrt{\frac{\sum_{i=1}^{N_{ev}} (SEGN_k^i - PED_k - RMC^i)^2}{N_{ev} - 1}} \tag{3.3}$$

L'algoritmo per la determinazione dei piedistalli PED_k e di SIG_k procede in due fasi. Inizialmente RMC^i viene considerato nullo, si usa la (3.2) per fare una prima stima dei piedistalli, successivamente si usa la (3.1) per calcolare i nuovi RMC^i. Il procedimento viene poi ripetuto per calcolare PED_k.

In realtà l'algoritmo per la stima di PED_k e SIG_k è più complesso a causa della presenza di strisce difettose nei rivelatori. I difetti sono principalmente di due tipi: capacità di disaccoppiamento cortocircuitate e cortocirciti tra strisce adiacenti (nel caso del lato ohmico inoltre è possibile che le strisce metalliche della doppia metallizzazione siano cortocircuitate tra di loro). In più è possibile che alcune strisce non siano connesse all'elettronica di lettura, oppure che più strisce siano connesse allo stesso canale di lettura. Le strisce sconnesse a causa di cortocircuiti sulle microsaldature, sono caratterizzate da un rumore molto più basso rispetto alle altre in quanto il rumore dipende dalla capacità in ingresso al sistema di amplificazione, quindi se sono disconnesse questa capacità è piccola. Al contrario, se due strisce sono cortocicuitate tra di loro, l'elettronica di lettura vede una capacità doppia, e quindi un rumore più grande. Di conseguenza è possibile distinguere le strisce difettose mediante la valutazione del rumore. Queste strisce difettose non verranno quindi considerate nel calcolo di RMC^i.

Dopo l'acquisizione degli eventi di calibrazione le valutazioni di PED_k e SIG_k vengono effettuate da una CPU esterna al DSP e poi mandate ai relativi DSP che le memorizzano in appositi *buffer*. Poiché ad ogni DSP sono assegnate tre viste dei piani di rivelazione, si tratta di 3072 parole per i piedistalli ed altre 3072 parole per le deviazioni standard del rumore. Finita questa operazione di calibrazione e caricamento il DSP può processare e comprimere i dati degli eventi successivi.

Il formato dei dati in uscita dal DSP

I dati in uscita dai DSP passano attraverso una scheda *taxi* [12] che opera trasmettendo serialmente parole a 10 *bit*, di conseguenza è necessario spezzare le parole in due parti prima che possano essere trasmesse. Poiché spezzando parole a 16 *bit* in due si ottengono parole a 8 *bit*, nella trasmissione attraverso i *taxi* a 10 *bit* due *bit* rimangono inutilizzati. Si è scelto quindi di assegnare a questi due *bit* la funzione di caratteri di controllo. Alla prima metà di una parola (gli 8 *bit* più significativi) è assegnata la configurazione "10" ai

bit di controllo, mentre alla seconda metà (gli 8 *bit* meno significativi) viene assegnata la configurazione "01". Ad esempio, se dovessimo trasmettere la parola 0x0634, essa verrebbe divisa in due parole: 0x0206 e 0x0134. Come si vede la prima parola contiene gli 8 *bit* più significativi ed è caratterizzata dalla cifra esadecimale "02" mentre la seconda parola contiene gli 8 *bit* meno significativi ed è caratterizzata dalla cifra esadecimale "01".

3.4 Predittore di ordine zero (ZOP)

Il predittore di ordine zero (vedere capitolo 2) si basa sul confronto tra il segnale eventualmente da trasmettere e l'ultimo segnale trasmesso.

Nel nostro caso ogni DSP riceve i segnali di tre viste del rivelatore (1024 strisce per vista per un totale di 3072 strisce). Le strisce di ogni vista vengono analizzate separatamente. La prima striscia di ogni vista viene sempre trasmessa. La seconda striscia viene poi confrontata con la prima, dopo che da ognuna sono stati sottratti i rispettivi piedistalli. Se la differenza in valore assoluto tra i segnali è maggiore di una determinata quantità C_i, il dato della seconda striscia viene trasmesso, altrimenti no. La procedura descritta sopra viene ripetuta 1023 volte confrontando il valore delle strisce da trasmettere con l'ultimo valore trasmesso. Un esempio di tale algoritmo è mostrato in figura 3.5. Il processo viene ripetuto in modo analogo per le altre due viste.

Originariamente era stata fatta una simulazione della compressione in cui il DSP eseguiva anche un calcolo del rumore comune evento per evento per eliminare l'oscillazione attorno ai piedistalli che esso comporta.

Visto che questa procedura comportava un carico di lavoro eccessivo ed il rischio che un errore nel calcolo potesse deteriorare l'informazione trasmessa, si è deciso di eliminare il calcolo del rumore di modo comune a bordo del DSP.

Il valore di taglio C_i per l'i-esima striscia è definita come $CUTC \times SIG_i$, dove SIG_i è la stima del rumore dell'i-esima striscia e $CUTC$ una costante fissata. In questo modo le strisce più rumorose hanno meno probabilità di venire trasmesse: esse, fluttuando in maniera più accentuata delle altre, superebbero spesso un semplice taglio costante. Moltiplicando invece il taglio $CUTC$ per la deviazione standard si ottiene un peso più corretto per ogni striscia.

Variando la costante $CUTC$ si ottengono fattori di compressione più o meno elevati. Naturalmente il prezzo di una maggiore compressione è una più grande perdita di informazione. Valori tipici per $CUTC$ sono 3 o 4. Si noti che ponendo $CUTC$ uguale a zero tutte le strisce verrebbero trasmesse.

Con il sistema descritto sopra le strisce vengono trasmesse "a salti", ovvero non necessariamente due strisce trasmesse di seguito sono adiacenti. Di conseguenza il decodificatore non potrebbe dire a quale striscia si riferisce un dato trasmesso. Per questo è necessario, ogni volta che viene trasmessa una striscia dopo un "vuoto" di trasmissione, trasmettere la sua posizione sotto forma di "indirizzo" (un numero tra 2 e 1023 in un formato che ci consenta di distinguerlo dagli altri dati).

Il flusso dati dovrebbe avere un aspetto del tipo in figura 3.6.

Il formato che distingue un dato da un indirizzo sfrutta il fatto che i dati digitalizzati dagi ADC sono a 12 *bit*. Si sfrutta quindi il tredicesimo *bit* come *bit* di controllo: quando il tredicesimo *bit* è "1" si ha un indirizzo, se è "0" si ha un dato.

Il decodificatore si limita a immagazzinare dati fino a quando incontra un indirizzo. Anche il decodificatore ha accesso ai piedistalli utilizzati dal DSP per la compressione. Una volta trovato un indirizzo esso "riempie" l'intervallo tra l'ultima striscia immagazzinata e la striscia a cui punta l'indirizzo per mezzo dei valori immagazzinati nei *file* dei piedistalli, dopodiché continua ad immagazzinare i dati del *file* compresso fino al prossimo indirizzo.

Ad esempio, ammettiamo che, decomprimendo un *file* compresso, il decodificatore incontri solo dati (il tredicesimo *bit* è sempre "0") fino alla striscia 34, dopodiché incontra la parola 0x102a (0001000000101010). Poiché il tredicesimo *bit* è "1" si tratta di un indirizzo, che punta alla striscia 0x002a (= 42). A questo punto il decodificatore accede al *file* dei piedistalli e ricostruisce i dati relativi alle strisce da 35 a 41. Dopo aver fatto questo il decodificatore ricomincia la normale analisi dei dati, a partire dalla striscia 42.

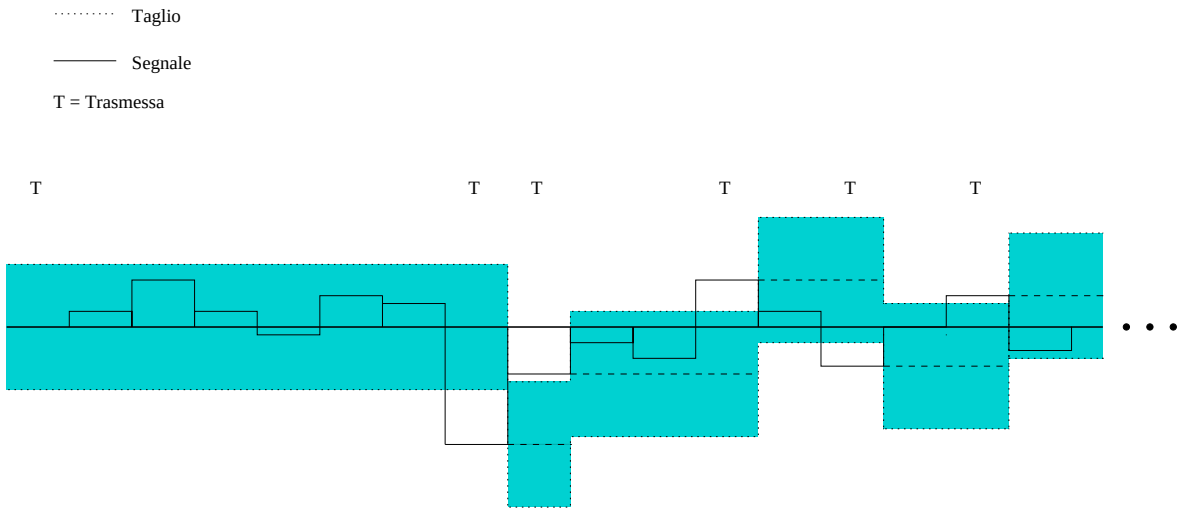


Figura 3.5: Esempio di funzionamento dello ZOP implementato in PAMELA (assumendo, per semplicità un valore di taglio T_i uguale per tutte le strisce). Le strisce trasmesse sono quelle indicate con T.

Figura 3.6: Tipico flusso dati da uno ZOP.

Il cercapicchi

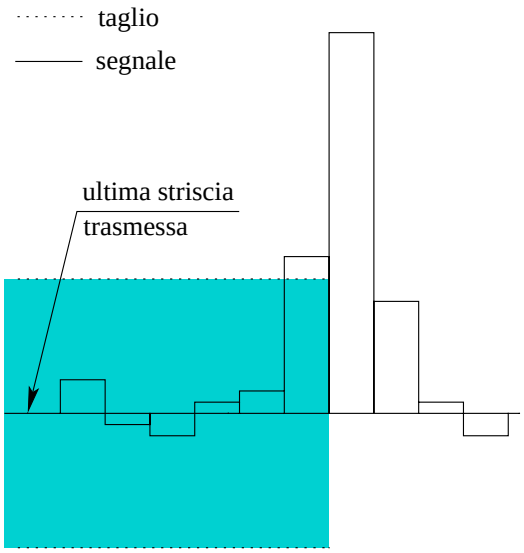
Poiché la misura di maggiore importanza nel rivelatore a microstrisce di silicio è la misura della posizione dei picchi di segnale causati dal passaggio delle particelle, è importante che tutte le strisce in cui viene rilasciata energia vengano trasmesse.

Parte del lavoro di tesi è stato lo sviluppo di un algoritmo per la localizzazione dei picchi (cercapicchi). Il funzionamento è il seguente.

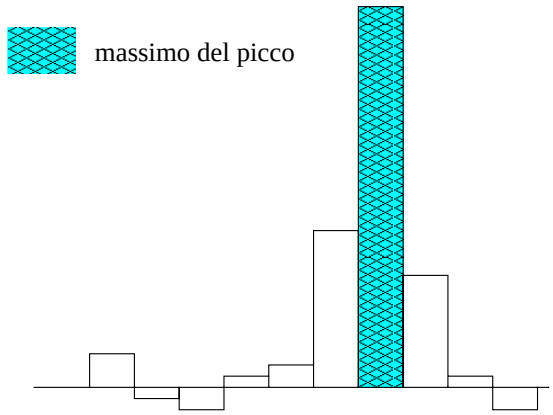
Il programma di compressione (lo ZOP) opera come di consueto, ma prima di fare il confronto tra la striscia da trasmettere e l'ultima striscia trasmessa basato sul taglio CUTC, ne fa un altro, del tutto analogo ma basato su di un altro parametro chiamato CUTCL. CUTCL è impostato più grande di CUTC (valori tipici sono 6 o 7). In realtà c'e una differenza tra il confronto con CUTC e CUTCL in quanto i picchi nella vista X sono di segno invertito rispetto a quelli nella vista Y, quindi nel confronto con CUTCL non si usa il valore assoluto delle differerenze come nel confronto con CUTC. È necessario quindi disporre di due versioni dello ZOP leggermente diverse (si tratta in pratica di invertire il segno del confronto), una da caricare nei DSP che analizzano le viste X, e l'altra da caricare in quelli che analizzano le viste Y.

Quando il confronto supera il taglio basato su CUTCL il programma assume di trovarsi in presenza di un picco dovuto al passaggio di una particella e chiama il programma cercapicchi. Questo cercherà il massimo del picco (ovvero la striscia con il segnale più grande o più piccolo, tra le strisce seguenti). La ricerca viene eseguita continuando a confrontare i segnali delle strisce successive (dopo aver sottratto i piedistalli).

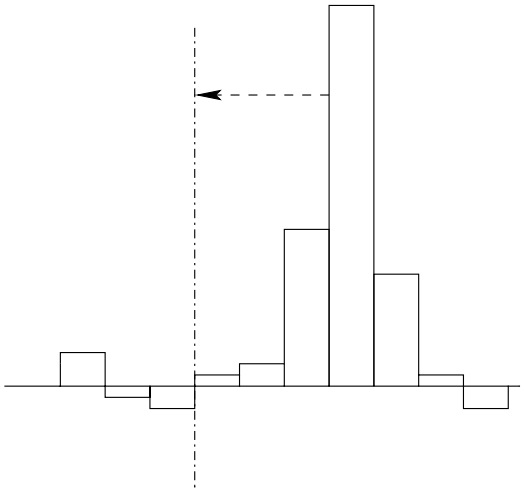
Una volta trovato il massimo del picco (inteso in senso assoluto), il programma torna indietro di un certo numero di strisce fissato da una costante $NCLUST$. Comincia poi a trasmettere integralmente tutte le strisce una dopo l'altra a partire da questo nuovo riferimento, a meno non che siano giá state trasmesse. In questo caso si inizia a trasmettere dall'ultima striscia trasmessa più uno (questo per evitare di trasmettere più volte la stessa striscia), fino al massimo del picco più $NCLUST$. In questo modo per ogni picco vengono trasmesse integralmente almeno $NCLUST$ strisce a destra ed a sinistra del massimo per un totale di $2 \times NCLUST + 1$ strisce, evidenziando il picco. Naturalmente il programma



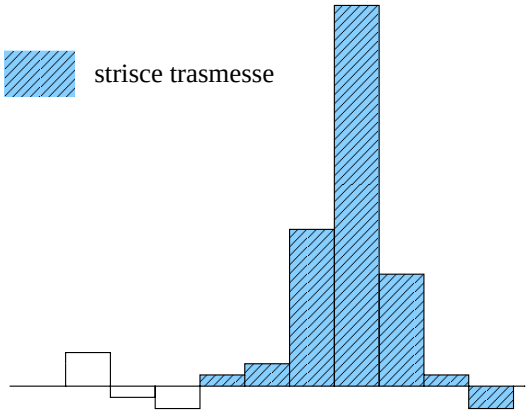
a) Viene trovato un possibile picco



b) Viene trovato il massimo del picco



c) Si torna indietro di NCLUST strisce



d) Vengono trasmesse NCLUST strisce
a destra ed a sinistra del massimo

Figura 3.7: Schematizzazione del funzionamento del cercapicchi con NCLUST pari a 3 (assumendo, per semplicità un taglio T_i uguale per tutte le strisce).

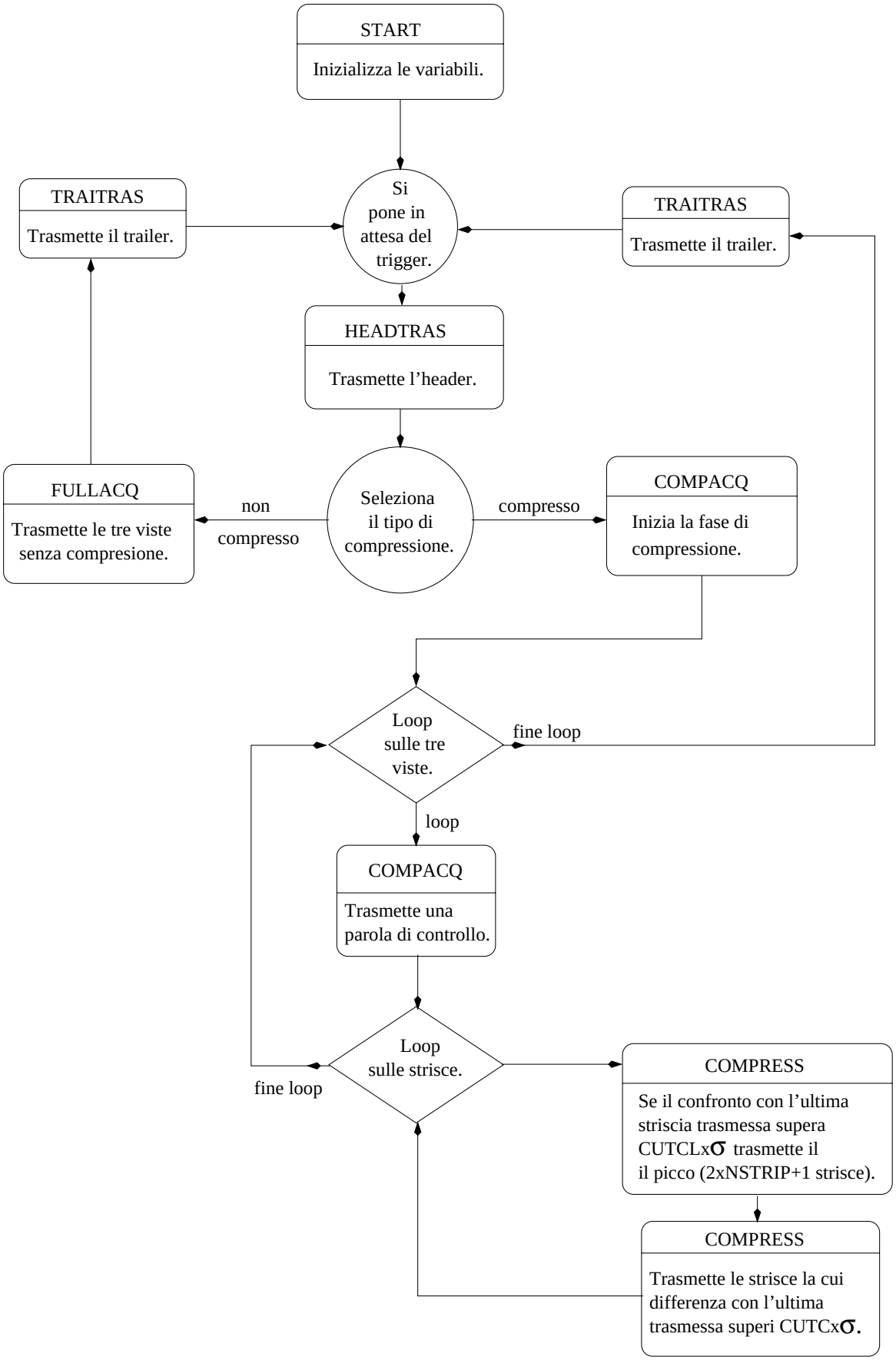


Figura 3.8: Uno schema del funzionamento dello ZOP inplementato sul DSP.

di compressione non va mai più indietro della prima striscia e non trasmette più della 1024 ma striscia. Lo schema del funzionamento del programma cercapicchi è mostrato in figura 3.7.

Il programma ripassa poi all'analisi usuale della vista con lo ZOP, richiamando il cercapicchi tutte le volte che serve. Il parametro $NCLUST$ può essere variato in modo da evidenziare zone più o meno larghe. Valori tipici sono 2 o 3, per ottenere zone evidenziate di 5 o 7 strisce rispettivamente.

L'algoritmo dello ZOP del *tracker* è schematizzato in figura 3.8 ed in appendice B è riportato il codice in *assembler*.

3.5 Algoritmo di Huffmann

La mia scelta per un algoritmo di compressione reversibile è caduta su di un modello statistico adattativo con codifica di Huffmann. La codifica di Huffmann per la sua relativa semplicità, il modello adattativo perché lascia aperta l'opzione di comprimere più eventi senza ripartire ogni volta da zero.

Un primo problema sta nel formato dei dati da comprimere. I segnali delle strisce vengono tramessi ai DSP sotto forma di parole a 12 *bit*, mentre tipicamente gli algoritmi di Huffmann sono pensati per testi a codici *ascii* a 8 *bit*. È stato necessario quindi fare delle modifiche per poter operare con più di 8 *bit*. Questo però ha comportato un serio aumento di dimensioni dell'albero di Huffmann, come si vedrà nel seguito.

Un albero di Huffmann (vedi capitolo 2) è dotato di tante foglie quanti sono i simboli possibili, più due simboli speciali che servono nella codifica (per un totale di $256+2 = 258$ foglie nel caso di simboli a 8 *bit* e $4096 + 2 = 4098$ foglie nel caso di simboli a 12 *bit*). L'albero consiste poi di nodi che collegano foglie ed altri nodi fino ad un nodo centrale a cui sono tutti, direttamente o indirettamente, collegati. Il numero massimo di nodi è pari a $2 \times NF - 1$ (515 nel caso a 8 *bit* e 8195 nel caso a 12 *bit*) dove NF è il numero di foglie. La cella di memoria del DSP corrispondente ad una foglia contiene il numero del nodo a cui essa è collegata; ad esempio, se la foglia N.22, corrispondente ad un determinato simbolo, contiene il numero 2, questo significa che essa è collegata al nodo numero 2. Ogni nodo è costituito da quattro variabili, custodite in altrettante celle di memoria nel DSP. La prima contiene il numero del nodo a cui esso è collegato (nodo genitore). La seconda contiene un carattere di controllo che stabilisce se ad esso è collegata una foglia o altri nodi (assume il valore "1" nel caso di una foglia, "0" nel caso di nodi). La terza contiene il numero dell'elemento collegato ad esso, della foglia nel caso di una foglia, del nodo figlio

nel caso di altri nodi. Infine nella quarta cella troviamo il peso in conteggi del nodo, che è dato dal numero di volte in cui il simbolo è stato incontrato nel caso di una foglia, o nella somma dei pesi dei due nodi figli nel caso di nodi. In figura 3.9 è mostrato un esempio di albero ed in tabella 3.2 il relativo contenuto di foglie e nodi.

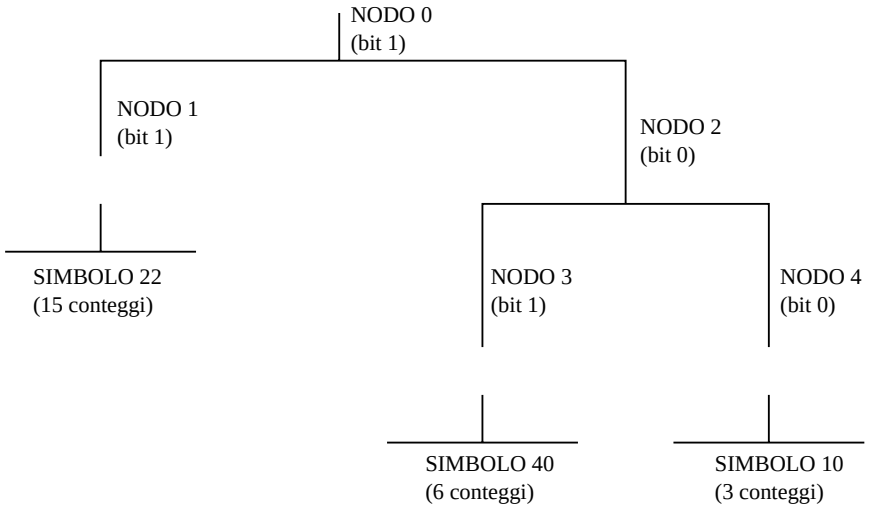


Figura 3.9: Esempio di albero per l'algoritmo di compressione di Huffmann.

Foglia	Contenuto della cella di memoria
10	4
22	1
40	3

Nodo	Genitore	Tipo di figlio	Figlio	Peso
0	-	0	1	24
1	0	1	22	15
2	0	0	3	9
3	2	1	40	6
4	2	1	10	3

Tabella 3.2: Contenuto delle celle di memoria del DSP associate alle foglie (a sinistra) ed ai nodi (a destra) corrispondenti all'albero in figura 3.9.

La codifica di Huffmann richiede che ad ogni nodo sia associato un *bit*. Risalendo l'albero fino alla cima a partire dal simbolo interessato si trova il codice compresso. Come

spiegato nel capitolo 2 è sufficiente fissare una convenzione per l'assegnazione dei *bit*. Nel nostro caso ad ogni nodo (che è numerato) è associato il *bit* meno significativo del numero del nodo, di conseguenza ai nodi di numero dispari è associato il *bit* 1, mentre ai nodi con il numero pari il *bit* 0.

Vediamo la quantità di memoria occupata da un albero. Ogni foglia occupa una locazione di memoria. È necessario allocare memoria sufficiente per il numero massimo di nodi dell'albero (anche se non è detto che vengano utilizzati tutti). Ogni nodo occupa 4 locazioni di memoria, che contengono il nodo genitore, il nodo figlio, il peso e il tipo del figlio (nodo o foglia). Il numero massimo di nodi di un albero a NF foglie è $2 \times NF - 1$, come ci si può rendere conto immaginando un albero del tipo mostrato in figura 3.10. Queste considerazioni sono riassunte nella formula (3.4).

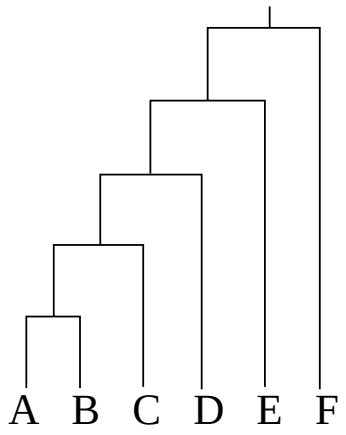


Figura 3.10: Albero con il massimo numero di nodi possibile. Come si vede, se il numero totale di simboli è NF=6, sono necessari $2 \times NF - 1$=11 nodi per costruire l'albero.

$$\begin{aligned} DIMENSIONI &= NF + 4 \times (2 \times NF - 1) \\ &= 9 \times NF - 4 \end{aligned} \tag{3.4}$$

Nel caso di 8 *bit* ($NF = 258$) le dimensioni dell'albero sono pari a 2318 parole, mentre nel caso a 12 *bit* ($NF = 4058$) sono pari a 36518 parole. In quest'ultimo caso l'utilizzo dell'albero di Huffmann è impraticabile data la quantità di memoria dati a disposizione (circa 16000 parole nel caso dei DSP di modello 2181 e 2185, il doppio nel caso del modello 2187).

Un modo di aggirare il problema consiste nell'utilizzare parte della memoria programmi, che è in gran parte inutilizzata, per memorizzare parte dell'albero. Il fatto che la

memoria programmi sia a 24 *bit* non è un problema: nelle lettura e scrittura della memoria programmi in e da registri (tipicamente a 16 *bit*), solo i 16 *bit* meno significativi vengono passati.

Inoltre il punto centrale nella codifica di Huffmann è che, per poter operare una buona compressione, devono essere usati meno simboli diversi possibile i quali devono essere ripetuti il più possibile. Conviene quindi sottrarre dal segnale di ogni striscia il relativo piedistallo prima di effettuare la codifica di Huffmann. In questo modo la maggior parte dei valori da comprimere saranno raccolti attorno ad un valore costante (che sarà intorno allo zero), lasciando inutilizzati buona parte dei simboli a disposizione. È possibile quindi anche usare alberi con un numero inferiore di *bit*. In questo ultimo caso sorge però un altro problema: una volta operata la sottrazione del piedistallo si possono avere anche valori negativi. Poiché utilizziamo meno di 16 *bit* non è possibile distinguere i numeri positivi dai negativi. Ad esempio 0xffff ($= -1$ in decimale) e 0x0fff ($= 4095$ in decimale) sono lo stesso numero se ne osserviamo soltanto i 12 *bit* meno significativi. Di conseguenza è necessario sommare una costante ai valori da comprimere in modo che i dati non fluttuino più attorno allo zero ma attorno a quella costante. In seguito il programma di decompressione provvederà a sottrarla automaticamente dai dati. Questa costante viene presa pari a metà del numero di simboli analizzabile dall'albero. Ad esempio nel caso di simboli a 8 *bit* l'intervallo di valori analizzabili è pari a 256 (2^8) e la costante viene fissata a 128, mentre nel caso di simboli a 12 *bit* il numero di valori è pari a 4096 (2^{12}) che significa una costante pari a 2048.

Naturalmente può accadere che il valore da comprimere non rientri nell'intervallo di valori analizzabile dal programma di compressione di Huffmann. In questo caso il dato viene trasmesso in forma non compressa, causando però una perdita di compressione. Più le dimensioni dell'albero saranno grandi, tanto più sarà improbabile che questo avvenga, ottenendo così una miglior compressione.

Utilizzando la memoria programmi è possibile memorizzare alberi a 10 *bit* anche nei DSP di tipo 2181 e 2185, mentre nel modello 2187 è possibile memorizzare perfino alberi a 12 *bit*.

Vediamo adesso in dettaglio il funzionamento di questo algoritmo di compressione implementato nel corso del lavoro di tesi. Questo algoritmo è basato su di un albero a 10 *bit*. Si è scelto questa soluzione in modo da poter utilizzare l'algoritmo di Huffmann anche nei DSP di tipo 2181 e di tipo 2185. Di conseguenza l'intervallo di valori a disposizione per l'algoritmo di Huffmann è di 1024 simboli (numerati da 0 a 1023). Vengono definiti due

simboli di controllo chiamati ESCAPE ed END_OF_STREAM, di cui vedremo l'utilizzo in seguito.

Inizialmente il programma alloca memoria sufficiente a contenere l'albero. Questo avviene definendo appositi *buffer* per le foglie e per i nodi. Per quanto riguarda il *buffer* "foglie" è necessario allocare una cella di memoria per ognuno dei simboli, compresi i simboli di controllo, per un totale di 1026 locazioni di memoria, che conterranno i numeri dei nodi a cui le foglie sono agganciate. Ai simboli END_OF_STREAM ed ESCAPE vengono assegnati le foglie 1024 e 1025, rispettivamente. Ogni nodo è caratterizzato da quattro celle di memoria (numero del nodo o della foglia figlia, numero del nodo genitore, tipo di figlio e peso del nodo); poiché un albero a 10 *bit* è composto al massimo da $2 \times 1026 - 1 = 2051$ nodi, è necessario allocare $4 \times 2051 = 8204$ locazioni di memoria ai *buffer* dei nodi.

L'algoritmo poi esegue una fase di inizializzazione in cui viene costruito un albero a cui vengono collegate due sole foglie, relative ai simboli ESCAPE ed END_OF_STREAM. In figura 3.11 si può vedere l'albero iniziale. Tutte le altre foglie vengono poi inizializzate con il valore "-1".

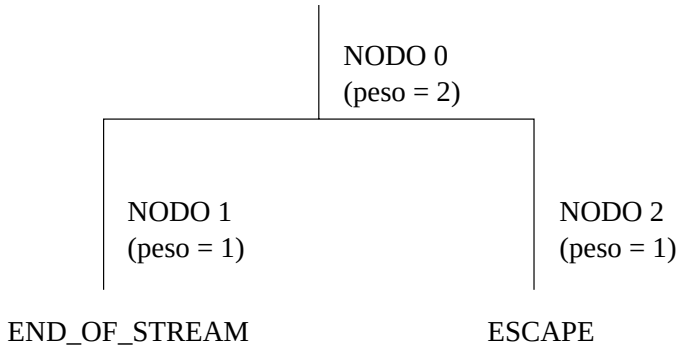


Figura 3.11: Albero iniziale.

Successivamente l'algoritmo provvede a sottrarre dai dati in memoria i relativi piedistalli ed ad aggiungere una costante pari a 512, in modo che i dati fluttuino attorno a questa costante. Dopodiché inizia il processo di codifica vero e proprio, che consiste nell'aggiungere all'albero le foglie relative ai simboli incontrati, e nel ricavare i codici compressi.

I valori in memoria vengono codificati uno ad uno. Quando un valore viene analizzato, il programma controlla la foglia relativa al valore. Ad esempio, se il dato da codificare è il valore 422 verrà controllata la foglia 422. Poiché nelle celle di memoria sono contenuti i numeri dei nodi (quindi non negativi) a cui esse sono collegate, il numero negativo "-1"

segnala che la foglia non è collegata ad alcun nodo. Quando il programma, leggendo il contenuto della foglia, trova il valore "-1", assume che il valore in esame non sia mai stato incontrato prima (quindi inizialmente il programma "conoscerà" soltanto i due simboli controllo ESCAPE ed END_OF_STREAM). In questo caso verrà trasmesso il codice attuale del simbolo "ESCAPE" e di seguito il nuovo simbolo in maniera non compressa (a 12 *bit*). Questo processo nel modello adattativo fa le veci della trasmissione della tabella dei simboli dei modelli statici (vedi sezione 2.2), ma avviene "in corsa" durante la codifica. Il simbolo "ESCAPE" servirà al decodificatore per indicare che ad esso seguirà un nuovo simbolo non compresso. Dopo che il nuovo simbolo è stato trasmesso in modo non compresso, la relativa foglia viene aggiunta all'albero, con una procedura chiarita nel seguito.

Se, al contrario, il numero inserito nella foglia relativa al simbolo in esame è diverso da "-1", il simbolo è già stato incontrato in precedenza (la sua foglia è già stata aggiunta all'albero) e non resta altro da fare che codificarlo (la codifica è riportata nel seguito). Se un valore da analizzare (valore anomalo) non rientra nell'intervallo gestibile dal programma di compressione (< 0 o > 1023), viene trasmesso il codice del simbolo "END_OF_STREAM", seguito dal valore anomalo in maniera non compressa (a 16 *bit*). Quando il programma di decompressione incontrerà il simbolo END_OF_STREAM, interromperà la normale decompressione ed estrarrà il valore anomalo dal flusso dati per poi ricominciare la decompressione.

Il processo di codifica avviene risalendo l'albero, a partire dalla foglia in esame, fino al nodo centrale (nodo 0). L'albero viene risalito leggendo le celle di memoria dei nodi che contengono i numeri dei nodi genitori: una volta trovato il genitore di un nodo, se ne trova il genitore a sua volta, ripetendo il procedimento fino al nodo 0. Per determinare il codice compresso si accumulano *bit* che dipendono dai nodi incontrati risalendo l'albero: viene accumulato un *bit* 1 se il nodo incontrato ha un numero dispari, o un *bit* 0 se il nodo incontrato ha un numero pari. Ad esempio, se i nodi incontrati risalendo l'albero sono $14, 10$ e 1, il codice ottenuto sarà "001".

Una nuova foglia viene agganciata all'albero "dividendo" in tre il nodo dell'albero con il numero più alto (che chiamiamo "nodo k"). Il procedimento è il seguente:

- Viene staccato il nodo o la foglia attaccato al nodo k.
- Vengono agganciati al nodo k due nuovi nodi (k+1 e k+2) che diventano i nodi figli del nodo k.

- La foglia o il nodo a suo tempo staccata dal nodo k viene agganciata al nodo k+1. Al nodo k+1 viene assegnato lo stesso peso del nodo k.

- la nuova foglia viene agganciata al nodo k+2, al quale viene temporaneamente attribuito un peso pari a zero.

In figura 3.12 è mostrato l'effetto sull'albero iniziale di figura 3.11 dell'aggiunta di un nuovo simbolo chiamato NUOVO_SIMBOLO.

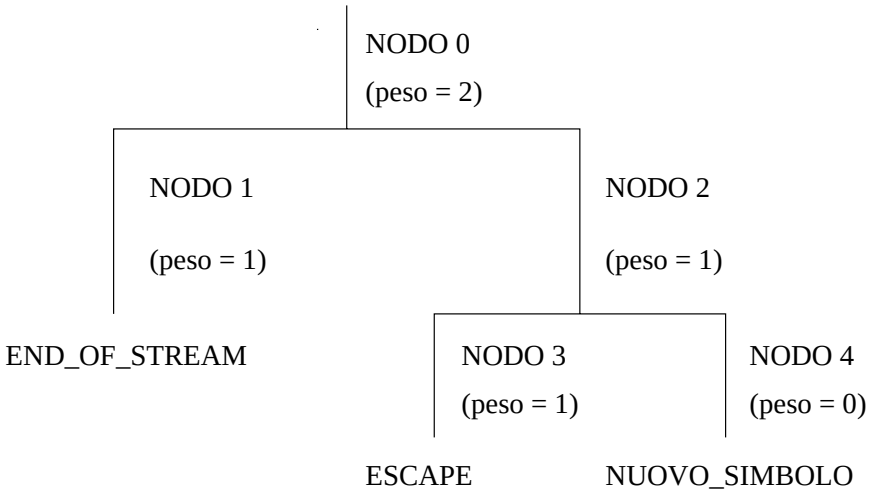


Figura 3.12: Aggiunta di un nuovo simbolo all'albero iniziale.

Dopo la codifica il peso del nodo genitore della nuova foglia viene incrementato di uno (nel caso di un nuovo simbolo passa da 0 a 1). La stessa cosa avviene ai nodi genitori risalendo l'albero, fino al nodo centrale (nodo 0) (vedere figura 3.13).

A questo punto l'albero viene riaggiustato, se necessario, in modo che corrisponda alle regole sulla costruzione di un albero di Huffmann (vedere sezione 2.2). In pratica quello che si fa è scambiare tra di loro i nodi in modo che siano al posto giusto sulla base dei loro pesi (secondo la regola di fratellanza) e lo si fa ricorsivamente risalendo l'albero. Se la regola della fratellanza è rispettata l'albero sarà "legale" secondo Huffmann. Questa operazione è mostrata in figura 3.14.

Tutto questo viene ripetuto per le tre viste del *tracker* gestite da un DSP ed eventualmente per più eventi. Il processo di inizializzazione avrà luogo soltanto ogni NUM_EV_COM eventi, dove NUM_EV_COM è una costante che definisce il numero di eventi da comprimere di seguito senza reinizializzare l'albero. Ad esempio se NUM_EV_COM viene fissato pari a 10 l'albero verrà inizializzato soltanto all'evento 1, 11, 21, 31 ecc.

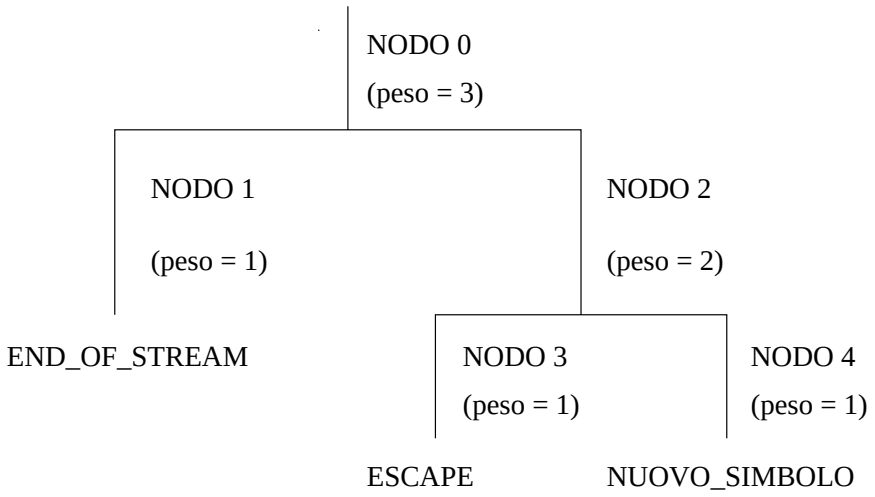


Figura 3.13: Albero precedente con i pesi corretti.

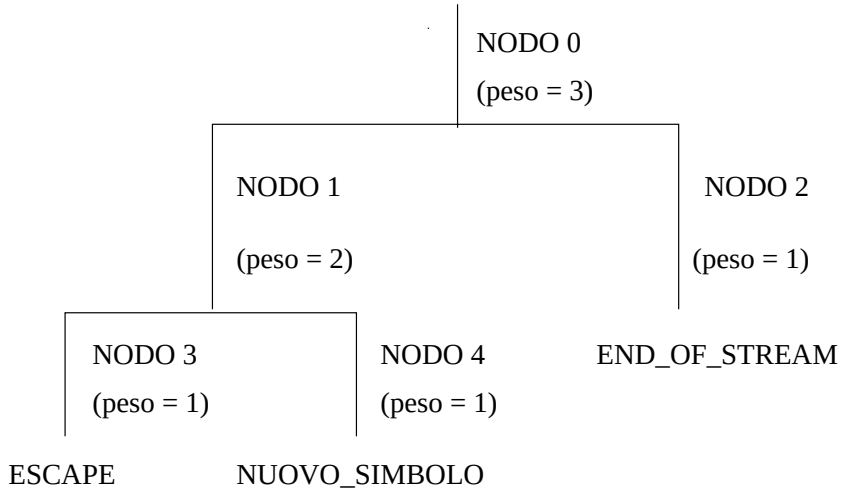


Figura 3.14: Riordino dell'albero di Huffmann di figura 3.13 attraverso lo scambio di due nodi.

I codici in realtà non vengono trasmessi direttamente: la trasmissione dei codici compressi avviene a blocchi di 16 *bit*, di conseguenza i vari codici compressi vengono accumulati in una variabile temporanea a 16 *bit*. L'accumulo dei codici nella variabile temporanea avviene *bit* per *bit*. Ogni volta che la variabile temporanea si riempie, il suo contenuto viene trasmesso ed essa viene azzerata. Questo implica che il codice relativo ad un simbolo può venire spezzato in trasmissioni separate (ma è anche possibile che più codici vengano trasmessi insieme).

L'algoritmo del programma di compressione di Huffmann del *tracker* è schematizzato in figura 3.15 e in appendice B è riportato il codice in *assembler*.

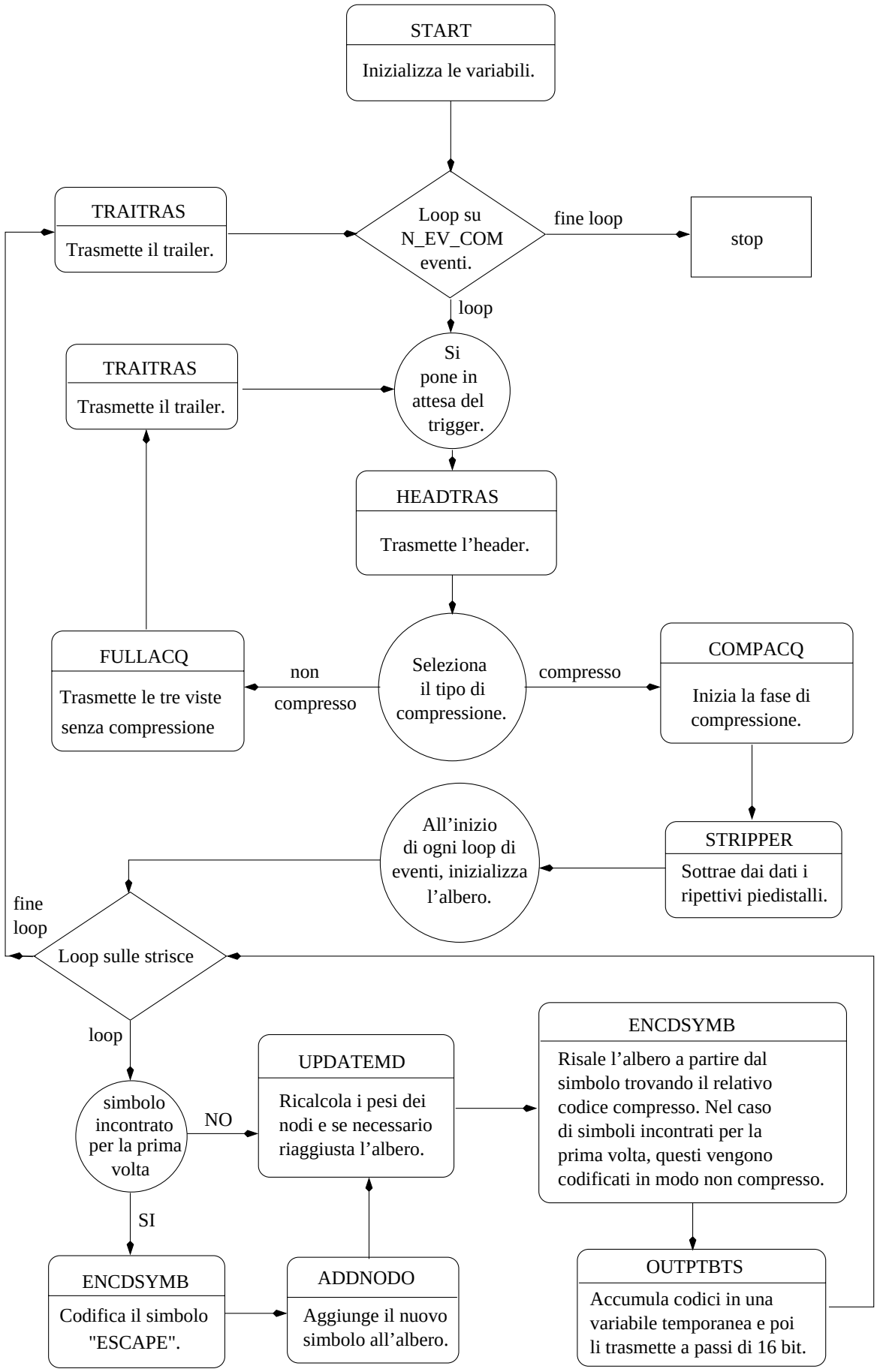


Figura 3.15: Schema del funzionamento dell'algoritmo di Huffmann implemantato sul DSP.

Capitolo 4

Risultati utilizzando i rivelatori di PAMELA

Nel Luglio del 2000 è stato eseguito un *test* su fascio all'SPS del CERN a Ginevra. Lo scopo del *test* era la prova dei rivelatori di PAMELA sollecitati da particelle ad alta energia.

Durante questo *test* parte dei dati è stata acquisita in formato compresso utilizzando l'algoritmo ZOP modificato con il cercapicchi con lo scopo di stimarne il rapporto di compressione. I dati compressi sono stati confrontati con i dati non compressi per determinare quanto i dati vengano degradati dallo ZOP modificato.

Successivamente è stato svolto un *test* in laboratorio a Firenze in cui si è utilizzato un campione di dati non compressi, acquisiti durante il *test* al CERN, per controllare l'algoritmo di Huffmann, in particolare per studiare i rapporti di compressione ed il tempo necessario per comprimere un evento per mezzo del DSP.

4.1 *Test* su fascio

Nel luglio del 2000 si è svolto, alla *west area* del CERN a Ginevra, un *test* su fascio. Il fascio utilizzato era quello proveniente dall'SPS. Si trattava di un fascio di particelle selezionabili con energie fino a 100 GeV. Posizionando degli assorbitori e delle targhette era possibile selezionare anche il tipo di particelle presenti nel fascio. In questo modo è stato possibile osservare le prestazioni dei vari rivelatori in risposta ai vari tipi di particelle ed a diverse energie.

Per questo *test* sono stati usati dei prototipi dei rivelatori che comporranno il telescopio PAMELA. Questi rivelatori sono stati posti in una geometria simile a quella che avranno una volta assemblati nel satellite (fig. 4.1), per verificare la bontà delle soluzioni tecniche scelte.

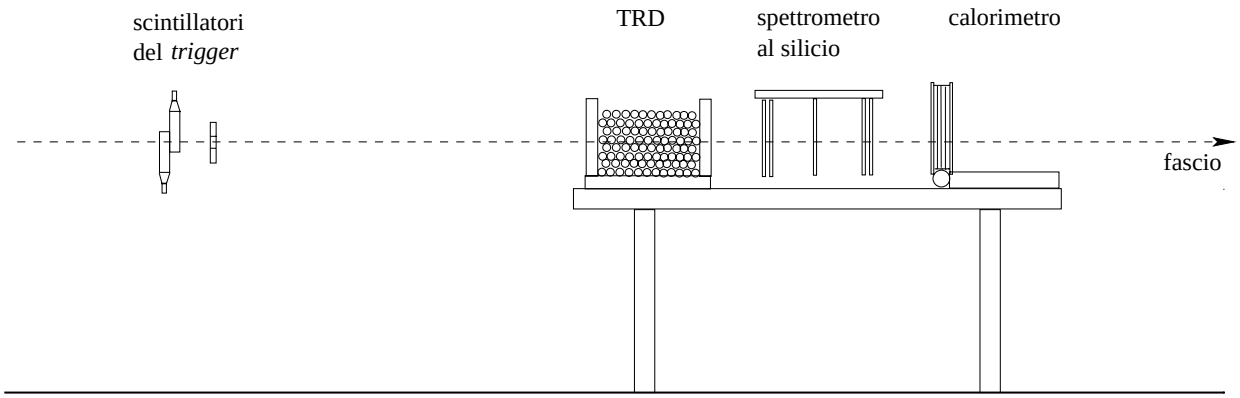


Figura 4.1: Schematizzazione della posizione dei vari rivelatori durante il test di Luglio 2000.

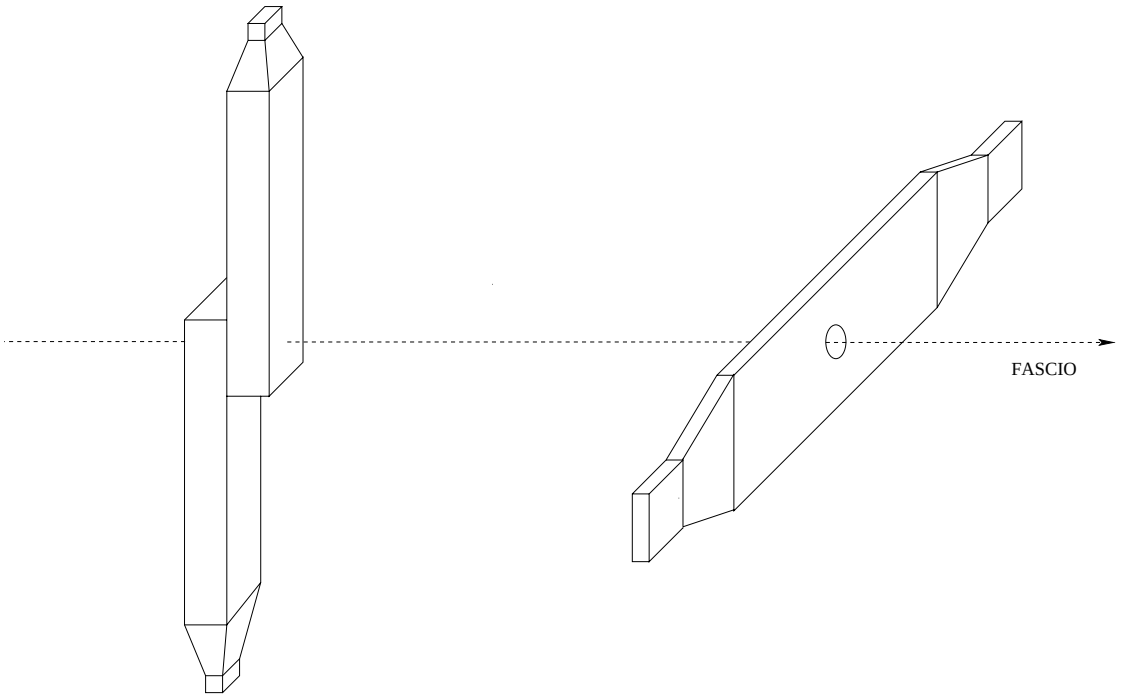


Figura 4.2: Gli scintillatori utilizzati per la generazione del segnale di *trigger* durante il test di Luglio 2000.

Non essendo presente il TOF, i *trigger* che segnalavano il passaggio di una particella (e quindi l'inizio di un evento) provenivano da un sistema di scintillatori posti a monte dei rivelatori. Questi erano costituiti da due scintillatori "a dito" disposti lungo la traiettoria del fascio seguiti da una piastra di scintillatore con un foro al centro attraverso il quale passava il fascio (fig.4.2). Il segnale di *trigger* era costituito dall'AND dei segnali provenienti dai due scintillatori a dito che a sua volta era in anticoincidenza con la piastra scintillatrice bucata che serviva da collimatore. Un segnale di particella veniva generato solo se questa produceva un segnale nei due scintillatori a dito ma non nella piastra bucata. I rivelatori erano posti in fila in modo che il fascio li attraversasse tutti. Il primo rivelatore incontrato dal fascio era il TRD, successivamente il *tracker* ed infine il calorimetro. Inoltre lo scintilatore S4 era appoggiato al calorimetro (a valle) perpendicolarmente al fascio.

Appositi fasci di cavi mettevano in comunicazione i rivelatori con l'elettronica di acquisizione posta in una cabina a distanza di sicurezza dal fascio.

Il *tracker* era costituito da cinque piani di rivelatori a microstrisce di silicio costituiti ognuno da una singola unità di rivelazione. I piani erano montati su di un supporto a "pettine", come in figura 4.3, per una migliore maneggevolezza.

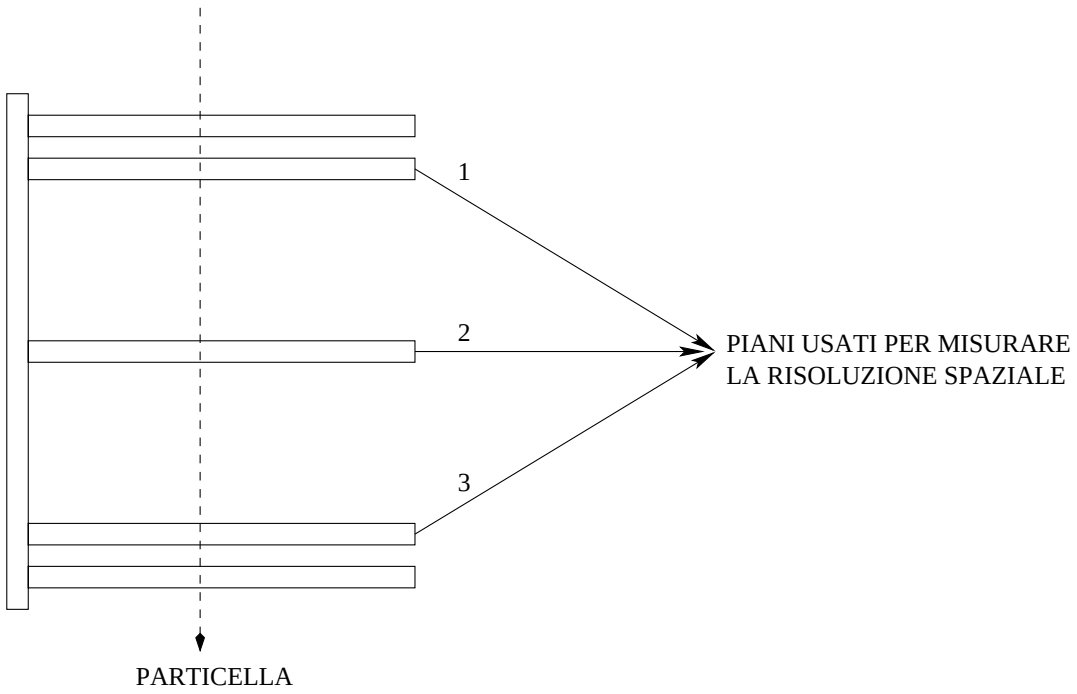


Figura 4.3: Per misurare la risoluzione di un piano di rivelazione sono necessari almeno tre piani.

Uno degli scopi principali del *test* era la prova dell'algoritmo di compressione non reversibile ZOP modificato con il cercapicchi, descritto nella sezione 3.4. A tale scopo la

maggior parte dei dati è stata acquisita in formato compresso. Periodicamente venivano acquisiti dati in formato non compresso al fine di calcolare i piedistalli e le sigma, necessari per la compressione. L'operazione è stata eseguita più volte durante il *test* per tenere conto di un'eventuale variazione di questi parametri nel tempo. Gli stessi dati sono stati utilizzati in una fase successiva per valutare la degradazione subita dall'informazione a seguito dell'algoritmo di compressione.

4.2 I risultati della compressione con lo ZOP

Come precedentemente spiegato (vedere sezione 3.4), i parametri che caratterizzano lo *zero order predictor* sono CUTC (che definisce il taglio dello ZOP), CUTCL (che definisce il taglio per la chiamata del cercapicchi) e NCLUST (che indica quante strisce il cercapicchi deve trasmettere a destra ed a sinistra del massimo del picco). Per questa prova sono stati fissati i parametri CUTC = 4, CUTCL = 7 e NCLUST = 2.

Nel seguito verranno discussi i rapporti di compressione e la degradazione dei dati provocata dallo ZOP.

Rapporti di compressione

	parole (13 *bit*)	***bit***	**RdC**
dati con particelle	554517	7208721	94.1%
dati senza particelle	489321	6361173	94.8%

Tabella 4.1: Rapporti di compressione ottenuti con due *file* di dati (1000 eventi), uno con particelle (sopra) ed uno senza (sotto). Entrambi i *file*, una volta decompressi, sono composti da 10240000 parole a 12 *bit*, equivalenti a 122880000 *bit*.

Un evento non compresso è composto da 10240 ($= 1024 \times 5 \times 2$) parole a 12 *bit*, per un totale di 122880 *bit*. Applicando l'algoritmo di compressione il numero di *bit* necessario per codificare un evento si riduce notevolmente; in tabella 4.1 è riportato il numero di parole totale trasmesse, per un campione di 100 eventi (si ricordi che il formato dei dati compressi è a 13 *bit*). Per stimare i rapporti di compressione sono stati considerati due diversi *file* di dati, entrambi di 1000 eventi, uno con particelle ed uno senza. I rapporti di compressione ottenuti sono del 94.1% per il *file* con le particelle e del 94.8% per il *file* senza particelle. Il rapporto di compressione per il *file* con particelle è leggermente inferiore in quanto in presenza di segnali indotti dalle particelle il numero di informazioni che devono

essere trasmesse è superiore. In figura 4.4 è mostrato il numero di parole trasmesso dallo ZOP per un campione di 100 eventi.

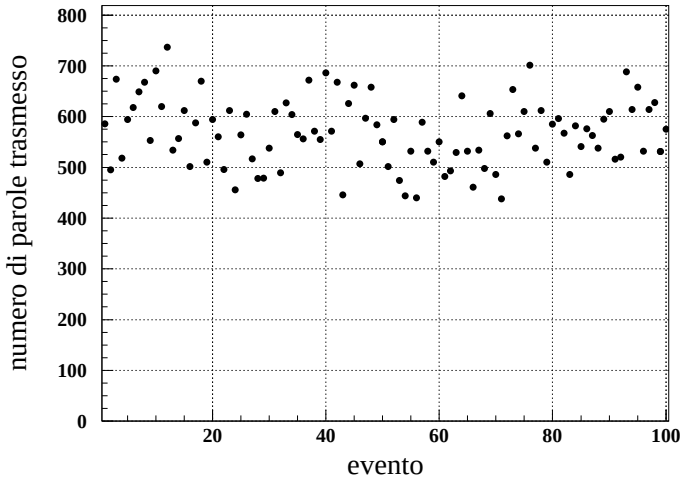


Figura 4.4: Numero di parole compresse trasmesse dallo ZOP per un campione di 100 eventi.

Questi risultati consentono di stimare il livello di compressione ottenibile per il *tracker* di PAMELA. A tal proposito ricordiamo che un piano del *tracker* è costituito da tre unità di rivelazione di cui possiamo supporre che in media solo una conterrà un segnale di particella. Prendiamo quindi come stima del rapporto di compressione di un piano la media pesata dei rapporti di compressione riportati in tabella 4.1; il valore che si ottiene è 94.6%.

Tenendo conto del fatto che il *tracker* di PAMELA è composto da 6 piani di rivelazione e che ciascuno di essi trasmetterebbe 6144 parole a 12 *bit* senza compressione, si otterrebbe un volume di dati pari a ≈ 55 Kbyte per evento. Con il rapporto di compressione stimato per l'algoritmo ZOP questo valore si riduce a ≈ 3 Kbyte. Assumendo un flusso dati giornaliero di $\approx 3\ 10^5$ eventi, si ottiene una quantità di dati da trasmettere a terra di ≈ 1 Gbyte al giorno per il solo *tracker*.

Degradazione dei dati

Poiché lo ZOP è un algoritmo di compressione non reversibile, i dati decompressi subiscono una degradazione. Si dovrà quindi verificare che le quantità rilevanti ai fini dell'esperimento non subiscano deterioramenti eccessivi.

Il pericolo della compressione non reversibile consiste nella possibile perdita di informazioni nel flusso dati e in una conseguente non corretta ricostruzione del picco, che si

riflette direttamente sulla risoluzione spaziale. Questa quantità è di primaria importanza, in quanto le prestazioni dello spettromentro magnetico sono ad essa strettamente legate. È quindi necessario che la procedura di compressione non ne alteri il valore.

Per valutare il livello di deterioramento dell'informazione a seguito della compressione parte dei dati compressi è stata decompressa e analizzata. La stessa procedura di analisi è stata applicata ad un campione di dati non compressi e i risultati sono stati messi a confronto. Nel seguito viene descritta la procedura di analisi e vengono commentati i risultati ottenuti.

Analisi dei dati

La procedura di analisi utilizzata è stata sviluppata e applicata piú volte in precedenza. In questo paragrafo ne viene data una breve descrizione; ulteriori dettagli sono riportati in [14].

Il primo passo consiste nell'individuazione e nella ricostruzione dei picchi. Per ogni evento, dai segnali di ogni striscia vengono sottratti il relativo piedistallo ed il rumore di modo comune (vedere sezione 3.3). La ricerca dei picchi viene successivamente effettuata richiedendo che il segnale risultante dalla sottrazione superi un taglio di $7 \times \sigma_i$ (in questo paragrafo σ_i corrisponde alla varianza SIG_i della formula 3.3). Tale striscia viene chiamata seme del picco. Nel picco vengono inoltre incluse tutte le strisce contigue al seme il cui segnale superi un taglio pari a $2 \times \sigma_i$.

Una volta individuato un picco è possibile definirne la molteplicità e il segnale complessivo. La molteplicità N_{picco} è definita come il numero totale di strisce incluse nel picco e il segnale complessivo è dato da

$$S_{tot} = \sum_{i=1}^{N_{picco}} S_i \,, \tag{4.1}$$

dove S_i è il segnale della striscia i-esima.

Una quantità molto importante per valutare la qualità dei dati è il rapporto segnale/rumore. La definizione usata è quella di Turchetta *et al.* [13] definita come

$$\left(\frac{S}{R}\right)_{picco} = \sum_{i=1}^{N_{picco}} \frac{S_i}{\sigma_i} \,. \tag{4.2}$$

Lo scopo principale del rivelatore è misurare la posizione di passaggio delle particelle. L'algoritmo usato per ricostruire il punto di incidenza a partire dalle informazione contenute in un picco è l'algoritmo non lineare η. Esso è un'estensione della tecnica del baricentro ed è risultato essere il miglior algoritmo fra quelli considerati (referenze [7, 13, 14]).

Per la misura della risoluzione spaziale è necessario utilizzare le informazioni di almeno tre piani. I piani utilizzati per questa analisi sono i tre centrali (figura 4.3).

Consideriamo un evento in cui i tre piani centrali siano stati colpiti. Si definisce residuo posizionale la distanza tra la posizione del picco nel piano intermedio e l'intersezione dello stesso piano con la retta passante per i punti di incidenza misurati sui due piani esterni, ovvero, nel caso del lato giunzione o vista x,

$$\Delta = \frac{x_1 + x_3}{2} - x_2 \, . \tag{4.3}$$

Assumendo che la risoluzione spaziale (σ_x) sia la stessa per i tre piani, la varianza dei residui è data dalla relazione

$$\sigma_\Delta = \sqrt{\frac{3}{2}}\, \sigma_x. \tag{4.4}$$

Di conseguenza la distribuzione dei residui dà una diretta informazione sulla risoluzione spaziale. È utile introdurre un residuo posizionale ridotto, definito dalla relazione $\delta = \sqrt{2/3}\, \Delta$; dalle relazioni precedenti si ottiene che la varianza di tale grandezza coincide con la risoluzione spaziale. Analoghi ragionamenti valgono per il lato ohmico o vista y.

È necessario osservare che per ottenere la corretta distribuzione dei residui deve prima di tutto essere effettuata un'operazione di allineamento dei piani. La procedura ed il *software* usati sono gli stessi applicati nei precedenti test dei rivelatori. Una descrizione dettagliata della procedura di allineamento è descritta in [14].

Confronto tra dati compressi e dati non compressi

Per valutare l'effetto della compressione sulla qualità dei dati la procedura di analisi descritta nel precedente paragrafo è stata applicata ad un campione di dati compressi. Gli eventi del campione sono stati decompressi secondo i criteri descritti nella sezione 3.4 ed analizzati; la stessa procedura di analisi è stata applicata ad un campione di dati non compressi e i risultati messi a confronto. I dati riportati sono relativi ad elettroni di 100 GeV.

Per capire come il procedimento di compressione possa causare la perdita di parte dell'informazione contenuta in un picco facciamo un esempio. Ogni volta che un picco viene identificato, il cercapicchi trasmette un numero minimo fissato di strisce (2× NCLUST +1). Supponiamo che non tutta la carica rilasciata venga raccolta dalle strisce incluse in questo intervallo minimo, ma parte di essa venga raccolta da strisce esterne. Supponiamo inoltre che queste strisce non vengano identificate dallo ZOP in quanto il loro segnale non supera il taglio CUTC. Come spiegato nella sezione 3.4 il segnale di queste strisce

non viene trasmesso e, in fase di decompressione, viene ad esse assegnato il livello dell'ultima striscia trasmessa. Supponiamo che tali strisce siano quelle a sinistra del picco; il livello dell'ultima striscia trasmessa ha in questo caso un valore compatibile con il livello comune per l'evento considerato, valore che sottostima il segnale effettivo. Nel caso in cui le strisce non trasmesse siano invece quelle a destra del picco, il livello assegnato è quello dell'ultima striscia trasmessa dal cercapicchi, valore che ne sovrastima il segnale effettivo.

Un'ulteriore fonte di deterioramento dei dati risiede nel calcolo del rumore di modo comune. Nell'analisi dei dati non compressi il livello di modo comune viene stimato utilizzando il segnale di tutte le strisce. Nel caso dei dati compressi manca parte dell'informazione in quanto non tutte le strisce vengono trasmesse. Questo introduce un'ulteriore indeterminazione nella stima del livello di modo comune che si ripercuote sulla ricostruzione del picco e la successiva analisi.

Per valutare l'entità di questi effetti sono state studiate due grandezze indicative della distribuzione del segnale all'interno del picco: la molteplicità e il rapporto segnale/rumore, definite nel paragrafo precedente.

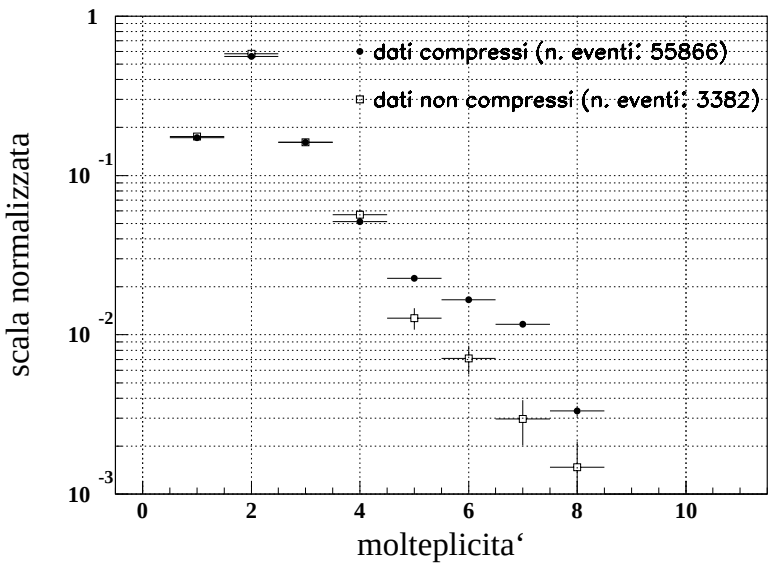


Figura 4.5: Distribuzione della molteplicità dei picchi relativi ai dati compressi e non compressi.

In figura 4.5 sono confrontate le distribuzioni normalizzate della molteplicità, ottenute per i due campioni di dati. Si osserva che in entrambi i casi circa il 90 % dei picchi ha molteplicità compresa tra 1 e 3 con un massimo in corrispodenza di molteplicità 2. Si osserva inoltre che la compressione porta ad un leggero eccesso di picchi con molteplicità elevata.

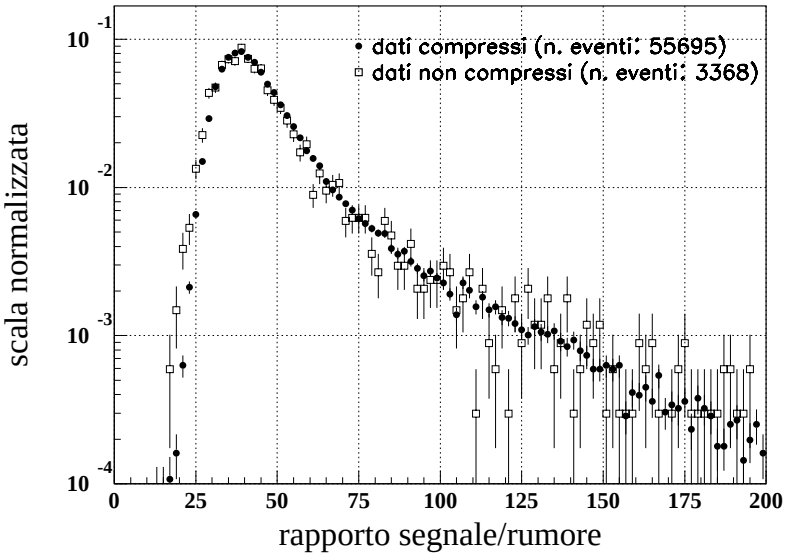


Figura 4.6: Rapporto segnale-rumore del rivelatore a microstrisce di silicio ottenuta con e senza compressione non reversibile.

La figura 4.6 mostra invece il confronto fra le distribuzioni normalizzate del rapporto segnale/rumore, ottenute per i due campioni di dati. Come si può osservare, le due distribuzioni coincidono, entro gli errori, per valori del rapporto segnale/rumore superiori a 25, raggiungendo un massimo in corrispondenza di un valore pari a ≈ 40. Le due distribuzioni si discostano per valori del rapporto segnale/rumore inferiori a 25; in questa regione la distribuzione ottenuta dai dati compressi presenta una deficienza di eventi.

Le leggere discrepanze osservate in figura 4.5 ed in figura 4.6 tra dati compressi e non, sono entrambe compatibili con gli effetti discussi precedentemente.

Il termine ultimo di paragone è comunque la risoluzione spaziale. Lo scopo principale di questo tipo di rivelatori è infatti la misura delle coordinate del punto di incidenza delle particelle. La risoluzione spaziale è di conseguenza il parametro indicativo della qualità di questi rivelatori. La figura 4.7 mostra il confronto tra la distribuzione normalizzata dei residui ridotti, ottenuta dai due campioni di dati. Come spiegato nel precedente paragrafo, la risoluzione spaziale è data dalla larghezza di tale distribuzione ($3.0 \pm 0.1 \mu$m in queso caso). Come si può vedere, nonostante le discrepanze osservate nelle figure precedenti, le distribuzioni dei residui ridotti sono in accordo, entro gli errori statistici. Questo ci assicura che la risoluzione spaziale non è deteriorata in maniera significativa dal processo di compressione. Il fatto che la compressione influisca sulla molteplicità e sul rapporto segnale/rumore ma non alteri la risoluzione spaziale è dovuto all'algoritmo utilizzato per

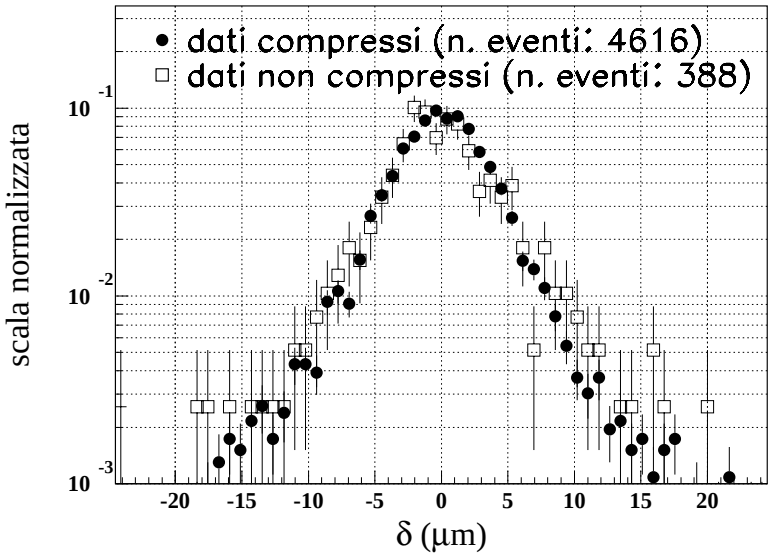


Figura 4.7: Distribuzione normalizzata dei residui ridotti ottenuta dai dati compressi e dai dati non compressi per il lato giunzione dei sensori. Dalla larghezza della distribuzione si ottiene direttamente la risoluzione spaziale, $3.0 \pm 0.1 \mu$m in queso caso.

ricostruire la posizione del picco, che utilizza le sole due strisce con il segnale maggiore (si veda [14]); questo riduce gli effetti di un eventuale errore nella ricostruzione del picco.

4.3 Sperimentazione in laboratorio

Nei laboratori a Firenze è stata preparata una scheda elettronica contenente un DSP per la sperimentazione dell'algoritmo di Huffmann. È stato inoltre preparato un *file* di 100 eventi non compressi (scelti tra quelli acquisiti durante il *test* del Luglio 2000 al CERN e relativi ad un DSP) da trasferire nel DSP. Il trasferimento dei dati è avvenuto mediante la porta seriale di un PC collegata alla porta IDMA del DSP. Una scheda di elettronica opportuna consentiva poi la lettura dei dati compressi dal DSP. Le *routine* di trasmissione del programma sono state modificate opportunamente per la trasmissione dei dati: ogni parola da trasmettere (a 16 *bit*) veniva spezzata in due parole a 8 *bit* ad ognuna delle quali veniva anteposta una parola (a 8 *bit*) che permetteva di distinguere le due metà. Questo è stato necessario in quanto il programma di compressione di Huffmann utilizza tutti i 16 *bit* a sua disposizione per trasmettere i dati in formato compresso e quindi non era possibile distinguere i dati dai comandi. Altre due parole di controllo venivano trasmesse, allo scopo di segnalare l'inizio e la fine di un evento compresso. Il programma sul PC provvedeva a riassemblare le due parti di ogni parola a 16 *bit* fino a che riceveva

la parole che segnalava la fine della trasmissione. Gli eventi potevano essere trasmessi sia uno per uno che tutti sequenzialmente.

I risultati della compressione con l'algoritmo di Huffmann

Al contrario dello ZOP l'algoritmo di Huffmann non provoca una degradazione dei dati. Non è necessario quindi fare confronti con i dati non compressi.

simboli	conteggio	codice compresso
< 459 >:	1	111011101001
< 460 >:	1	111000111111
< 466 >:	1	111000110011
< 468 >:	1	111000111110
< 471 >:	1	00100011110
< 474 >:	1	111011100100
< 475 >:	2	0010001000
< 478 >:	2	11101110110
< 479 >:	1	111000110001
< 481 >:	3	1011101010
< 482 >:	1	111000110110
< 483 >:	4	1110001110
< 484 >:	1	111011101110
< 485 >:	5	001000110
< 486 >:	4	1110111000
< 487 >:	6	101110110
< 488 >:	14	11100001
< 489 >:	17	11101111
< 490 >:	24	1011100
< 491 >:	49	101010
< 492 >:	50	101011
< 493 >:	75	00010
< 494 >:	89	10000
< 495 >:	85	01001
< 496 >:	99	10110
< 497 >:	92	10100
< 498 >:	116	11010
< 499 >:	84	00101
< 500 >:	78	00011
< 501 >:	63	111010
< 502 >:	45	100010
< 503 >:	50	101111
< 504 >:	47	100011
< 505 >:	63	111001
< 506 >:	85	01000
< 507 >:	114	11001
< 508 >:	131	11110
< 509 >:	146	0000
< 510 >:	167	0011
< 511 >:	184	1001
< 512 >:	177	0101
< 513 >:	178	0110
< 514 >:	181	0111
< 515 >:	119	11011
< 516 >:	104	11000
< 517 >:	74	111111
< 518 >:	72	111110
< 519 >:	44	001001
< 520 >:	33	1110110
< 521 >:	18	0010000
< 522 >:	15	11100010
< 523 >:	14	11100000
< 524 >:	8	101110111
< 525 >:	2	00100011111
< 526 >:	2	11100011010
< 527 >:	1	00100010010
< 528 >:	3	1011101001
< 529 >:	3	1011101011
< 530 >:	1	111011101011
< 531 >:	3	0010001110
< 532 >:	5	001000101
< 535 >:	3	1011101000
< 536 >:	2	11100011110
< 539 >:	1	111000110000
< 544 >:	1	111011101010
< 546 >:	1	111011100111
< 551 >:	1	111011100101
< 556 >:	1	00100010011
< 559 >:	1	111011101111
< 652 >:	1	111000110111
< 699 >:	1	111000110010
< 1024 >:	1	111011101000
< 1025 >:	1	111011100110

Tabella 4.2: Esempio di una tabella dei codici ottenuta alla fine della compressione di un evento.

Sono state compiute due prove: una comprimendo i 100 eventi uno per uno (compressione singola), e l'altro comprimendo 10 eventi consecutivamente per dieci volte, per un totale di 100 eventi (compressione multipla).

In tabella 4.2 è mostrata la tabella dei codici prodotta alla fine di un evento (si ricordi che durante la compressione adattativa i codici relativi ai simboli cambiano). Nella prima colonna sono elencati i simboli utilizzati in un evento. Si noti che ne vengono usati solo 71 su 1024 (i numeri 1024 e 1025 corrispondono ai simboli speciali ESCAPE ed END_OF_STREAM, vedi sezione 3.5). Nella seconda colonna sono elencati i conteggi dei simboli e nella terza i codici compressi. Si noti che i simboli con i conteggi più elevati sono codificati con codici più corti. Inoltre i conteggi più alti sono distribuiti attorno al simbolo 512, che corrisponde all'*offset* aggiunto per evitare di avere numeri negativi. È presente un leggero aumento dei conteggi attorno al simbolo 498, imputabile alla fluttuazione del rumore di modo comune, diverso per i diversi VA1.

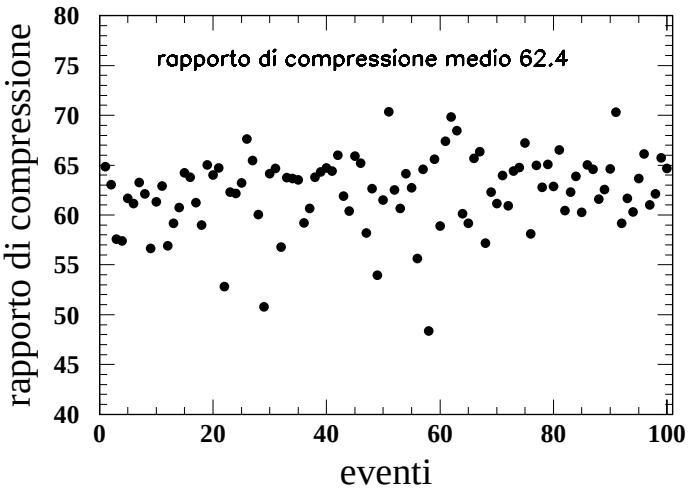


Figura 4.8: Rapporti di compressione relativi ad un singolo DSP per cento eventi.

In figura 4.8 si possono vedere riportati in grafico i rapporti di compressione, evento per evento, risultanti dalla compressione multipla. La maggior parte dei rapporti di compressione si trovano tra il 60% ed il 65%, con massimi del 72% e minimi del 48%.

	RdC
Compressione singola	65.7%
Compressione multipla	62.4%

Tabella 4.3: Rapporti di compressione ottenuti per la compresione singola e per la compressione multipla.

Ci si aspetta che comprimendo serie di 10 eventi consecutivi si ottengano rapporti di compressione migliori di quelli ottenuti comprimendo gli eventi singolarmente. Invece si

ottengono rapporti di compressione medi (calcolati rispetto a parole non compresse a 16 *bit*) del 62.4% per la compressione multipla e del 65.7% per quella singola, come si può vedere in tabella 4.3. Il motivo può essere imputato al rumore di modo comune che fa fluttuare troppo i piedistalli. Il guadagno maggiore della compressione multipla è dovuto al non trametterе ogni volta i simboli più comuni. Ma, come visto in tabella 4.2, questi sono pochi (≈ 70) ed il vantaggio viene soffocato dalla fluttuazione dei simboli causata dal rumore di modo comune, che costringe l'algoritmo ad usare codici poco favorevoli. Poiché l'algoritmo è adattativo, ci dovrebbe essere un riaggiustamento della lunghezza dei codici, ma evidentemente le fluttuazioni sono troppo forti, da evento ad evento. Un modo per ovviare a questo potrebbe essere calcolare e sottrarre il rumure di modo comune dai dati, ma questo aggiungerebbe un carico di calcolo non indifferente ad un programma già di per sé molto lento, come riportato nel paragrafo successivo.

4.4 Tempi di compressione

Ci si aspetta che il tempo di compressione dello ZOP sia sensibilmente minore rispetto a quello ottenuto mediante l'algoritmo di Huffmann in quanto il primo opera con una procedura molto più diretta (la compressione consiste in pratica solo nello scegliere quali strisce trasmettere). Al contrario l'algoritmo di Huffmann deve operare una complessa conversione di dati in codici ed in seguito trasmetterli tutti.

Per calcolare i tempi di lavoro dello ZOP e dell'algoritmo di Huffmann si è utilizzato il simulatore-emulatore *Debugger* della *Analog Devices*. Questo programma, gestibile da un normale PC, è in grado di simulare le funzioni di un DSP e di monitorarne i processi. Il *Debugger* fornisce anche il conteggio dei cicli eseguiti dal DSP mentre esegue un programma. Sapendo che i DSP di PAMELA lavorano a 33 ns/ciclo, si risale facilmente ai tempi di lavoro degli algoritmi.

A tale scopo è stato preparato un campione di 10 eventi (in cui ogni evento contiene i 3072 valori relativi ad un DSP) ed il *file* di piedistalli e sigma associato a quegli eventi. Questi sono stati processati tramite il simulatore annotando, per ognuno, i tempi di lavoro, sia per lo ZOP che per l'algoritmo di Huffmann.

Il risultato è stato una media 53 ± 5 ms/ev per l'algoritmo di Huffmann, mentre lo ZOP ha ottenuto una media di 5.9 ± 0.1 ms/ev. Come previsto lo ZOP è più veloce di un fattore 10.

Tempo morto

Il tempo morto è definito come il minimo intervallo di tempo che deve intercorrere tra due eventi perché questi possano venir registrati. Nel caso di PAMELA siamo nel caso di un sistema non paralizzabile, ovvero un evento che occorra durante il tempo morto viene ignorato.

Calcoliamo ora la probabilità che un evento occorra mentre i processori stanno ancora elaborando un evento precedente. Se indichiamo il tempo morto con τ e con n il rate medio di eventi, il numero di eventi aspettato durante il tempo morto è τn. Quindi, se al tempo T= 0 arriva un evento, la probabilità che ne arrivi almeno un altro entro il tempo morto è pari a $P = 1 - P_0(\tau)$, dove $P_0(\tau)$ è la probabilità che avvengano zero eventi nell'intervallo di tempo τ. Supponendo che gli eventi siano distribuiti secondo una distribuzione di Poisson con media τn, la probabilità che k eventi avvengano durante l'intevallo di tempo τ è pari a:

$$P_k(\tau) = \frac{e^{-\tau n}(\tau n)^k}{k!} \tag{4.5}$$

Dalla 1.5 si ottiene quindi $P = 1 - e^{-\tau n}$. Per quanto riguarda il tempo morto, esso è costituito principalmente dal tempo di operazione dei DSP, a cui va aggiunto 1.1 ms per il caricamento dei dati nei DSP. Quindi il tempo morto è pari a $\approx$ 54 ms per l'algoritmo di Huffmann ed a $\approx$ 7 ms per lo ZOP. Il flusso di particelle previsto per PAMELA oscilla tra 0.2 e 8 Hz (vedi sezione 1.5). Con questi valori si trova che la probabilità, che un evento occorra durante il tempo morto (e quindi l'inefficienza del rivelatore) oscilla nell'intervallo $1 - 35\%$ per l'algoritmo di Huffmann e nell'intervallo $0.1 - 5\%$ per lo ZOP.

Bibliografia

[1] R. L. Golden et al., *WIZARD: a proposal to measure the Cosmic Rays including antiprotons, positrons and nuclei and to conduct a search for Primordial Antimatter*, 1988

[2] O. Adriani et al., *The PAMELA Experiment*, Proc. of the 26th ICRC, Salt Lake City, 17-25 August, 1999, OG.4.2.04.

[3] Autori vari, *PAMELA TECHNICAL REPORT, 1996*

[4] `http://wizard.roma2.infn.it/pamela/fram_des.htm`

[5] J. D. Sullivan, *Geometrical factor and directional response of single and multi-element particle telescopes*, NIM 95(1971): 5.

[6] Oscar Adriani et al., *The silicon tracker of the PAMELA experiment.* NIM sez.A, 409(1998): 447-450.

[7] The PAMELA collaboration, *The PAMELA Experiment on Satellite and its Capability in Cosmic Rays Measurements*, Proceedings of IX Vienna Conference on Instrumentation 2001, to be published on NIM.

[8] Claude Shannon, *A mathematical theory of communication. The Bell system technical journal*, 1948: 379-636.

[9] David A. Huffmann,*A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, Volume 40, Numero 9 (1952).

[10] Mark Nelson, *The data compression book.* Prentice Hall, M&T Books.

[11] *ADSP-2100 family users manual, third edition.* Analog Devices.

[12] O. Adriani, G. Castellini, C. Civinini, R. D'Alessandro, M. Meschini and M. Pieri, *A data acquisition system for silicon microstrip detectors.*, IEEE Trans. on Nucl. Sci. 45(1998): 864.

[13] R. Turchetta, *Spatial resolution of microstrip detectors.* NIM sez.A, 335(1993) pag.44-58.

[14] E. Vannuccini, *Prototipo del sistema tracciante per l'esperimento PAMELA su satellite: ottimizzazione del sistema e sue prestazioni.* Tesi di laurea in Fisica, Università degli studi di Firenze, A.A. 1998-99.

Appendice A

Breve prontuario di *assembler*

In questa appendice si trovano alcune istruzioni utili per l'*assembler* dei DSP della *Analog Devices*. Queste sono solo una piccola parte di quelle esistenti e sono molto semplificate; inoltre alcune di esse dipendono dal modello di DSP. Quelle presentate qui sono corrette per la famiglia 218X. Per maggiori chiarimenti consultare [3].

La scrittura di un programma avviene sotto forma di moduli (*routine*), ognuno dotato di un nome, che può essere chiamato da altri moduli. Ogni modulo inizia con:

```
>
> .module/ram <nome del modulo>;
> .entry <etichetta>;
>
...
> <etichetta>:
>
...
```

Segue il programma vero e proprio. Tra le dichiarazioni iniziali e l'etichetta possono essere dichiarate le variabili. L'etichetta è una semplice parola che serve a contrassegnare un punto del programma. In questo caso segnala il punto di entrata del modulo: quando esso viene chiamato il programma inizia ad eseguire istruzioni a partire dall'etichetta indicata da `.entry` fino alla fine del modulo, indicata da:

```
.endmod;
```

Esempio:

```
>
> .module/ram main;
> .entry ;
>
...
> <etichetta>:
>
...
> .endmod;
```

Seguono alcune fra le istruzioni *assembler* più comuni. Da notare che ogni istruzione di programmazione deve finire con un punto e virgola.

- **Dichiarazione delle variabili**:

  ```
  .var <nome della variabile>;
  ```

Dopo `.var` si può dichiarare se porre la variabile in memoria dati (`.var/dm;`) oppure in memoria programmi (`.var/pm;`). Inoltre si può specificare la locazione di memoria in cui porre la variabile aggiungendo `/abs=<indirizzo>`. La variabile può essere anche resa globale (ovvero utilizzabile da tutti i moduli del programma) con il comando:

```
.global <nome della variabile>;
```

Esempio:

```
>
> .global temporaneo;
> .var/dm/abs=0x0ac3 temporaneo;
>
```

In questo esempio si dichiara la variabile `temporaneo` come globale e fissata alla locazione di memoria `0x0ac3` della memoria dati.

- **Dichiarazione delle costanti**:

```
.const <nome della costante>=<valore della costante>;
```

Esempio:

```
>
> .const NUMERO=1024;
>
```

- **Lettura/scrittura**:

Sia la memoria programmi che la memoria dati possono essere lette o scritte. Per la prima si utilizza il comando `PM` e per la seconda `DM`. Negli esempi successivi si userà `DM` ma, salvo diversamente specificato, queste istruzioni valgono anche per `PM`.

  - Lettura/scrittura diretta: l'operazione agisce su di una particolare locazione di memoria.

    Lettura:

    ```
    <registro> = dm(variabile);
    ```

    Dove `<registro>` è un qualsiasi registro (`ar`, `ax1`, `sr0`, `i5`, ecc.) eccetto `af` e `mf`, mentre `variabile` è una variabile definita nella memoria dati. Dopo aver eseguito questa istruzione nel registro c'è il contenuto della cella di memoria della variabile.

    Scrittura:

    ```
    dm(variabile)=<registro>;
    ```

    Analogamente alla lettura, ora nella cella di memoria della variabile si trova il contenuto del registro.

  - Lettura/scrittura indiretta: l'operazione può essere eseguita ricorsivamente.

    Lettura:

```
<registro> = dm(I#,M#);
```

Dove `<registro>` ha significato analogo alla lettura diretta, `I#` sono registri di indirizzo (`I0 – I7`) e `M#` sono registri d'incremento (`M0 – M7`). In questo modo si legge la locazione di memoria `I#` e si incrementa di `M#` (può essere zero).

Esempio:

```
> I1=0x0c01;
> M3=2;
> ar=dm(I1,M3);
```

Dopo questa operazione in ar si trova il contenuto della locazione di memoria `0x0c01` della memoria dati ed in I1 si trova l'indirizzo `0x0c03`.

Scrittura:

```
dm(I#,M#) = <registro>;
```

Con significati analoghi alla lettura.

NOTA: per la lettura/scrittura indiretta della memoria programmi (`PM`) possono essere utilizzati soltanto i registri `I4 – I7` e `M4 – M7`.

- **Operazioni della ALU:**
  - Somma e sottrazione:

    ```
    AR o AF = <operatore X(Y)> +/- <operatore Y(X)>;
    ```

    Esempio:

    ```
    > ar=2;
    > ay1=6;
    > ar=ar-ay1;
    ```

  - Operazioni logiche (AND, OR ,XOR):

    ```
    AR o AF = <operatore X> AND/OR/XOR <operatore Y>;
    ```

    Esempio:

    ```
    > ar=0XFFFF;
    > ay1=0X00F0;
    > ar=ar AND ay1;
    ```

    Dopo questa operazione in `ar` avremo `0x00F0`.
  - Negazione:

    ```
    AR o AF = - <operatore X(Y)>;
    ```

  - Negazione logica:

    ```
    AR o AF = NOT <operatore X(Y)>;
    ```

  - Valore assoluto:

    ```
    AR o AF = ABS <operatore X>;
    ```

Dove gli operatori `X` sono `AR, AX0, AX1, MR0, MR1, MR2, SR0` o `SR1` e gli operatori `Y` sono `AY0, AY1` o `AF`.

- **Operazioni MAC:**

  – Moltiplicazione:

  ```
  MR o MF = <operatore X> * <operatore X(Y)> (SS/SU/US/UU);
  ```

  I simboli tra parentesi specificano il tipo di operandi (`S`: con segno, *signed*; `U`: senza segno, *unsigned*).
  Esempio:

  ```
  > mx0=-4;
  > my1=3;
  > MR = mx0 * my1 (SU);
  ```

  Si noti che in `mx0` c'è un numero con segno (`S`), mentre in `my1` c'è un numero non segnato (`U`).

  – Moltiplicazione con accumulo:

  ```
  MR o MF =MR + <operatore X> * <operatore X(Y)> (SS/SU/US/UU);
  ```

  Utile da utilizzare in *loop* per fare somme ricorsive.
  Gli operatori `X` sono `MR0, MR1, MR2, MX0, MX1, SR0, SR1, AR` e gli operatori `Y` sono `MY0, MY1, MF`.

- **Operazioni del traslatore:**

  – Traslazione immediata:

  ```
  SR = lshift/ashift <operatore X> by # (HI/LO);
  ```

  Dove `lshift` opera una traslazione logica e `ashift` una traslazione aritmetica del numero `<operatore X>` di # *bit* (# positivo = spostamento verso sinistra, # negativo, verso destra). I simboli tra parentesi indicano se `<operatore X>` va caricato nei 16 *bit* di sinistra del traslatore (`HI`) o in quelli di destra (`LO`).
  Esempio:

  ```
  > si=0x0FFF;
  > sr=lshift si by -8 (HI);
  ```

  Dopo questa operazione in `sr0` abbiamo `0xF000` e in `sr1 0x00FF`.

  – Traslazione indiretta:

  ```
  se=#;
  SR = lshift/ashift <operatore X> (HI/LO);
  ```

  con significati dei simboli analoghi. Gli operatori `X` sono `SI, SR0, SR1, AR, MR0, MR1, MR2`.

- **Tabella degli *interrupt*:**

  Qualora il programma abbia bisogno di segnali esterni per poter lavorare (ad esempio segnali di avvio o blocco per far partire o fermare l'elaborazione) il DSP è predisposto per ricevere e trasmettere segnali di *interrupt*. Esistono dodici *interrupt* che possono essere ricevuti dal DSP. Affinché il programma li possa gestire è necessario creare la tabella degli *interrupt*. Questa fornisce al programma le istruzioni da eseguire in caso di segnale di *interrupt*. È possibile fornire fino a quattro istruzioni (che verranno eseguite una dopo l'altra). Tutti gli *interrupt* possono essere "mascherati" (disabilitati) tranne quello di *power down*. I diversi *interrupt* sono dotati di un ordine di priorità, in questo modo in caso di più *interrupt* questi vengono serviti in ordine di priorità.

```
>.module/ram/abs=0 INTERRUPTTABLE;
>
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {reset}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {power down}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {IRQ2}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {IRQL1}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {IRQL0}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {SPORT0 tx}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {SPORT0 rx}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {IRQE}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {BDMA}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {SPORT1 tx/IRQ1}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {SPORT1 rx/IRQ0}
> <1a istruz>; <2a istruz>; <3a istruz>; <4a istruz>; {timer}
>
>.endmod;
```

  Nello schema sopra gli *interrupt* sono elencati in ordine di priorità (`reset` ha la priorità più alta e `power down` quella più bassa). L'istruzione `RTI` fa proseguire il programma all'istruzione successiva.

  Esempio:

```
>.module/ram/abs=0 INTERRUPTTABLE;
>
>  jump inizio;  rti;            rti;    rti;     {reset}
>  jump copydat; jump shutdown;  rti;    rti;     {power down}
>  rti;          rti;            rti;    rti;     {IRQ2}
>  rti;          rti;            rti;    rti;     {IRQL1}
>  rti;          rti;            rti;    rti;     {IRQL0}
>  rti;          rti;            rti;    rti;     {SPORT0 tx}
>  rti;          rti;            rti;    rti;     {SPORT0 rx}
>  jump princip; rti;            rti;    rti;     {IRQE}
>  rti;          rti;            rti;    rti;     {BDMA}
>  rti;          rti;            rti;    rti;     {SPORT1 tx or IRQ1}
>  rti;          rti;            rti;    rti;     {SPORT1 rx or IRQ0}
>  rti;          rti;            rti;    rti;     {timer}
>
>.endmod;
```

  In questo esempio, se arriva un segnale di *reset* il proramma riparte dalla *routine* `inizio`, se arriva un segnale di *power down* chiama la routine `copydat` e poi `shutdown`, infine, se arriva un segnale sul *pin* `IRQE`, salta al programma principale, in questo ordine di priorità.

- **Istruzioni multiple:**

È possibile eseguire più istruzioni in un singolo ciclo del DSP. Le operazioni di lettura delle memorie e dei registri vengono eseguite all'inizio di ogni ciclo mentre le operazioni di scrittura alla fine, quindi è possibile combinarle.

In particolare è possibile eseguire (operazione = operazione della ALU o del MAC o del traslatore):

- Operazione + lettura memoria
- Operazione + trasferimento da registro a registro
- Scrittura + operazione
- Lettura memoria programmi + lettura memoria dati
- Operazione ALU/MAC + lettura memoria programmi + lettura memoria dati

Per eseguirle è sufficiente separare le istruzioni con una virgola. Le operazioni vengono eseguite nell'ordine indicato sopra: prima ciò che sta a sinistra del "+", poi ciò che si trova a destra, tranne la lettura memoria programmi + lettura memoria dati, che può essere eseguita in qualsiasi ordine.

Esempio:

```
>
> ar=ax0+ay0 , ax0=dm(I2,M1) , mr1=pm(i7,M4);
>
```

in questa istruzione si somma il contenuto di `AX0` e `AY0` e lo si scrive in `AR`, poi il contenuto della locazione di memoria di indirizzo contenuto in `I2` viene letta e posta in `ax0`; infine la locazione di memoria `I7` della memoria programmi viene letta e scritta in `MR1`.

Possono essere anche usati dei registri in comune tra le due operazioni a patto che il registro di destinazione delle due operazioni sia diverso.

Per esempio l'istruzione

```
>
> ar=ax0+ay0 , ar=dm(I2,M1);
>
```

è illegale.

- **Varie:**
  - `NOP`:

    Istruzione nulla: il DSP non esegue alcuna istruzione per la durata di un ciclo.
  - `IDLE`:

    Il DSP si pone in una modalità a basso consumo di energia finché arriva un segnale di *interrupt*. A questo punto esegue le istruzioni presenti nella tabella degli *interrupt*.
  - Caricamento di un registro:

```
<registro>=<registro>;
<registro>=<costante>;
<registro>=^<buffer>;
```

In cui `<registro>` è un qualsiasi registro eccetto `AF` e `MF`, mentre `^ <buffer>` indica l'inizio di un *buffer* precedentemente definito e `<costante>` è un numero.

Esempio:

```
> ar=sr0;
> ax0=16;
> i0=^esempio;
```

Dopo questo esempio in `ar` si trova il contenuto si `sr0`, in `ax0` il numero 16 ed in `i0` l'indirizzo dell'inizio del *buffer* `esempio`, che deve essere stato definito in precedenza.

– `JUMP`:

```
jump <etichetta>;
```

Il programma salta ad un altro punto della *routine* corrente, indicato da un'etichetta (cioè una parola che non sia un'istruzione, seguita dai due punti).

Esempio:

```
>main:
>
> jump prova;
>
 ...
>
>prova:
>
> nop;
> ...
```

Una volta incontrata l'istruzione `jump prova`, il programma salta all'istruzione `nop` dopo l'etichetta `prova`.

È anche possibile usare, al posto dell'etichetta, l'indirizzo contenuto nel `DAG2` (`I4 – I7`).

Esempio:

```
>main:
>
> i5=0x0ac2
> jump (i5);
>
 ...
```

– `CALL`:

```
call <subroutine>;
```

Il programma salta ad un'altra *subroutine*.

– `RTS`:

Il programma torna dalla *subroutine* in cui si trova l'istruzione `rts` alla *routine* che l'ha chiamata (mediante un'istruzione `call`), all'istruzione immediatamente successiva a quella di chiamata.

Esempio:

```
main:
 call stripper;
 nop;
```

Incontrata questa istruzione il programma salta alla *subroutine* `stripper` e la esegue fino a che incontra all'interno di essa l'istruzione `rts`. A questo punto il programma torna alla *routine* `main` ed esegue l'itruzione `nop`.

– `DO`:

```
DO <etichetta> UNTIL <condizione>;
```

Esegue le istruzioni che seguono il `DO` fino a quella successiva all'etichetta e continua a ripetere questo ciclo fino a che `<condizione>` diventa vera. Alcune possibili condizioni sono: `CE` (fino a che il contatore non raggiunge lo zero, dove il contatore è il registro `CNTR` che viene decrementato alla fine di ogni ciclo), `EQ, LT, LE, NE, GT, GE` (l'ultima operazione eseguita ha avuto risultato rispettivamente uguale, minore, minore o uguale, non uguale, maggiore, maggiore o uguale a zero).

Esempi:

```
>main:
>
> i2=0x0;
> m0=1;
>
> CNTR=10;
> do lettura until CE;
>
>lettura:
>
>  ar=dm(i2,m0);
>
 ...
```

Esegue `ar=dm(i2,m0)` dieci volte.

```
>main:
>
> i2=0x0;
> m0=1;
> ar=10;
>
> do lettura until LE;
>
>  ar=ar-1;
>
>lettura:
>
>  ax0=dm(i2,m0);
>
 ...
```

Esegue le istruzioni `ar=ar-1;` e `ar=dm(i2,m0);` dieci volte.

– IF:

```
 IF <condizione> <azione> ;
```

Se la condizione `<condizione>` è vera esegue l'operazione `<azione>`. Alcune condizioni possibili sono `EQ, LT, LE, NE, GT, GE` (vedi sopra) e sono riferite all'ultima operazione eseguita dalle unità di calcolo del DSP.

Esempio:

```
>main:
>
> ax0=dm(esempio);
> ar=ax0-3;
> if EQ call stripper;
>
```

Se nella locazione di memoria della variabile `esempio` si trova il numero 3, il programma chiama la *subroutine* `stripper`.

NOTA: le operazioni di lettura e scrittura o di caricamento di registri **non** sono operazioni delle unità computazionali del DSP e quindi non valgono per azionare le funzioni `IF` e `DO`.

```
>main:
>
> ax0=dm(esempio);
> ar=ax0-3;
> ar=ax0;
> ax0=dm(esempio2);
> if EQ call stripper;
```

Questo esempio è perfettamente analogo a quello precedente e porta agli stessi risultati: le istruzioni `ar=ax0-3;` e `ar=ax0;` non influenzano `IF`.

In sintesi

Il DSP può effettuare contemporaneamente in un singolo ciclo:

- Generare il prossimo indirizzo del programma.
- Prendere la prossima istruzione.
- Eseguire fino a due trasferimenti di dati.
- Aggiornare fino a due puntatori alla memoria dati.
- Effettuare un'operazione.
- Ricevere e/o trasmettere dati tramite le porte seriali.
- Ricevere e/o trasmettere dati tramite gli scambiatori interni di indirizzi.

I punti di forza della famiglia 2100 sono:

- **Aritmetica veloce**: in un singolo ciclo il DSP può eseguire una moltiplicazione o una moltiplicazione con accumulo, una traslazione, una operazione logico-aritmetica.
- **Protezione dagli *overflow***: i risultati delle somme accumulate dal MAC sono protetti da un accumulatore a 40 *bit* lasciando così 8 *bit* di sicurezza. Dovrebbero avvenire 256 *overflow* prima di perdere dei dati.
- **Riciclo degli operandi**: Il DSP è in grado di prendere i due operandi di una somma in un singolo ciclo.
- ***Buffer* circolari**: Il DSP possiede la possibilità di manipolare *buffer* circolari implementata nell'*hardware*.

- **Tempi morti minimi**: il DSP effettua istruzioni cicliche e condizionali con zero tempi morti.
- **Porta IDMA**: essa permette la lettura/scrittura del DSP in modo completamente asincrono dalle operazioni che il DSP sta eseguendo.

Appendice B

In questa appendice sono riportati i listati dei programmi di acquisizione e di compressione (ZOP e Huffmann). Le parti tra parentesi graffe e tra i due simboli `/*` e `*/` sono commenti. Tutte le *routine* includono il *file* `constant.k` che contiene le costanti utilizzate. Nella prima sezione sono riportati i testi del programma di acquisizione generale seguito dalla sezione relativa allo ZOP e da quella relativa all'algoritmo di Huffmann.

Programma di acquisizione

Tavola degli *interrupt*. Questa tavola contiene le istruzioni da eseguire all'arrivo dei vari tipi di *interrupt*. In questo caso all'arrivo del segnale di *reset* il programma ricomincia dall'inizio (*routine* `start`, che esegue le inizializzazioni), mentre all'arrivo del segnale `IRQE` salta alla *routine* `main`, che inizia la compressione.

```
.module/ram/abs=0 INTERRUPTTABLE;
.include <constant.k>;
        jump start;          rti; rti; rti;     { 00: reset }
        rti;                 rti; rti; rti;     { 04: IRQ2 }
        rti;                 rti; rti; rti;     { 08: IRQL1 }
        rti;                 rti; rti; rti;     { 0c: IRQL0 }
        rti;                 rti; rti; rti;     { 10: SPORT0 tx }
        rti;                 rti; rti; rti;     { 14: SPORT0 rx }
        jump main;           rti; rti; rti;     { 18: IRQE }
        rti;                 rti; rti; rti;     { 1c: BDMA }
        rti;                 rti; rti; rti;     { 20: SPORT1 tx or IRQ1 }
        rti;                 rti; rti; rti;     { 24: SPORT1 rx or IRQ0 }
        rti;                 rti; rti; rti;     { 28: timer }
        rti;                 rti; rti; rti;     { 2c: power down }
.endmod;
```

constant.k. Contiene tutte le costanti, gli indirizzi dei registri di memoria e la definizione delle *routine*.

```
{
   Assignement of a symbolic name to each memory mapped register
}
.const  IDMA=                   0x3fe0;
.const  BDMA_BIAD=              0x3fe1;
.const  BDMA_BEAD=              0x3fe2;
.const  BDMA_BDMA_Ctrl=         0x3fe3;
.const  BDMA_BWCOUNT=           0x3fe4;
.const  PFDATA=                 0x3fe5;
.const  PFTYPE=                 0x3fe6;
.const  SPORT1_Autobuf=         0x3fef;
.const  SPORT1_RFSDIV=          0x3ff0;
.const  SPORT1_SCLKDIV=         0x3ff1;
.const  SPORT1_Control_Reg=     0x3ff2;
.const  SPORT0_Autobuf=         0x3ff3;
.const  SPORT0_RFSDIV=          0x3ff4;
.const  SPORT0_SCLKDIV=         0x3ff5;
.const  SPORT0_Control_Reg=     0x3ff6;
.const  SPORT0_TX_Channels0=    0x3ff7;
.const  SPORT0_TX_Channels1=    0x3ff8;
.const  SPORT0_RX_Channels0=    0x3ff9;
```

```
.const  SPORT0_RX_Channels1=    0x3ffa;
.const  TSCALE=                 0x3ffb;
.const  TCOUNT=                 0x3ffc;
.const  TPERIOD=                0x3ffd;
.const  DM_Wait_Reg=            0x3ffe;
.const  System_Control_Reg=     0x3fff;

{*************************************************************
 *
 *              Constant and parameter definition
 *
 ************************************************************* }
.const ADDRESS0DM=b#1000000000000000; { Starting address for data in DM}
.const NSTRIP=1024;         { Number of strips for each sensor }
.const NSTRIP2=2048;        { 2*NSTRIP }
.const TOTSTRIP=3072;       { Total number of strips  }
.const NLADD=3;             { Number of ladders }
.const ADCMIN=500;          { min ADC value for transmission }
.const ADCMAX=3500;         { max ADC value for transmission }

{
Constants needed for adapt. Huffman compression
}
{*************************************************************************
*   NBITS   ENDSTREA   ESCAPE  SYMCOUNT  TOTNODES  size of the tree(words)
*
*   8       256        257     258       515             2317
*
*   9       512        513     514       1027            4623
*
*  *10      1024       1025    1026      2051            9231
*
*   11      2048       2049    2050      4099           18447
*
*   12      4096       4097    4098      8195           36879
*
*************************************************************************}
.const NBITS=10;
.const VALMAX=1023;        { 2EXP(NBITS) -1  }
.const HALFVALMAX=512;     { (VALMAX +1)/2 }
.const ENDSTREA=1024;
.const ESCAPE=1025;
.const SYMCOUNT=1026;
.const TOTNODES=2051;
.const ROOTNODE=0;
.const MAXWEIGH=0x8000;
.const TRUE=1;
.const FALSE=0;
.const N_EV_COM=20;

{ ************************************************************
 *
 *              External symbols definition
 *
 * ************************************************************ }
{
 Definition of the entries for each module or subroutine of the system
}
.external start;         { module start: init. system at reset }
.external main;          { module main: main section, data acquisition handling}
.external clearbuf;      { clean-up of data buffer }
.external fullacq;       { full mode acquisition}
.external compacq;       { compressed mode acquisition}
.external doubacq;       { full + compressed data with 1 header and trailer}
.external waitlong;      { 1.1 msec waiting loop}
.external headtras;      { transmit transmission header }
.external traitras;      { transmit transmission trailer }
.external wait;          { waiting loop }
.external clustfind;
.external transmit;
.external stripper;
.external encdsymb;
.external inittree;
.external outptbts;
.external swapnodo;
.external updatemd;
.external rebutree;
```

```
.external addnodo;
.external compress;
.external clobtfil;
.external compfile;

{
 Definition of global variables
}
   { vector variables }
.external ped1, sig1;   { Ped & sigma buffers }
.external ped2, sig2;   { Ped & sigma buffers }
.external ped3, sig3;   { Ped & sigma buffers }
.external buffer;       { Data buffers }
   { variables }
.external MailBox;      { MailBox to define the procedure}
.external Contr1;       { Control word before ped table}
.external Contr2;       { Control word after ped table}
.external Contr3;       { Control word before sig table}
.external Contr4;       { Control word after sig table}
.external EventNumTrig; { Progressive number of the event triggered}
.external EventNumTras; { Progressive number of the event transmitted}
.external TrasmDato;    { Datum to be transmitted}
.external Counter;      { Counter }
.external TRASMESSO;    { Flag for transmitted strip }
.external Checksum;     { Checksum of the transmitted numbers}
.external WordTras;     { Number of words transmitted}
.external Temp;         { Temporary variable}
.external Temp2;        { Temporary variable}
.external Temp3;        { Temporary variable}
.external InfoCompr1;   { Information 1 on compression}
.external InfoCompr2;   { Information 2 on compression}
.external InfoCompr3;   { Information 3 on compression}
.external newdp;        { New data-ped in compression}
.external olddp;        { Old data-ped in compression}
   { definition of BIT_FILE }
.external BIT_FILE_rack;
.external BIT_FILE_mask;
   { definition of tree }
.external foglia;
.external next_free_nodo;
.external nodoparent;
.external nodochild_is_foglia;
.external nodochild;
.external nodoweight;
.external numero_di_eventi_compressi;
```

Start.dsp. *Routine* di partenza che inizializza tutte le variabili e pone il DSP in stato di quiescenza in attesa dei segnali di *interrupt*.

```
.module/ram START;
.include <constant.k>;
.entry start;
{*****************************************************************
 *
 *              Global directives for defined variables
 *
 *****************************************************************}
{
   vectors variables
}
.global ped1, sig1;            { ped & sigma buffers ladder 1}
.global ped2, sig2;            { ped & sigma buffers }
.global ped3, sig3;            { ped & sigma buffers }
.global buffer;                { data buffers }
   { definition of BIT_FILE}
.global BIT_FILE_rack;
.global BIT_FILE_mask;
.global next_free_nodo;
.global foglia;
.global nodoparent;
.global nodochild_is_foglia;
```

```
.global nodochild;
.global nodoweight;
{
  single location variables
}
.global MailBox;        { Used to write control value for DSP:
                          1 Fulltrasm, 2 Compressed}
.global Contr1;         { Control word before ped table}
.global Contr2;         { Control word after ped table}
.global Contr3;         { Control word before sig table}
.global Contr4;         { Control word after sig table}
.global EventNumTrig;   { Progressive number of the event triggered}
.global EventNumTras;   { Progressive number of the event transmitted}
.global Counter;        { Counter }
.global IS;             { Current strip (on compression) }
.global IT;             { Last transmitted strip }
.global TRASMESSO;      { Flag for transmitted strip }
.global Checksum;       { Checksum of the transmitted numbers}
.global WordTras;       { Number of words transmitted}
.global InfoCompr1;     { Information 1 on compression}
.global InfoCompr2;     { Information 2 on compression}
.global InfoCompr3;     { Information 3 on compression}
.global TrasmDato;      { Datum to be transmitted}
.global Temp;           { Temporary variable}
.global Temp2;          { Temporary variable}
.global Temp3;          { Temporary variable}
.global newdp;          { New dat-ped on compression }
.global olddp;          { Old dat-ped on compression }
.global Mail_XY;        { Used to set the ladder's view:
                          1 = Y view, 0 = X view }
.global numero_di_eventi_compressi;
.global stack;

{*****************************************************************
 *
 *              Buffer and variables definitions
 *
 *****************************************************************}
.VAR/DM/ABS=0x0000 buffer[TOTSTRIP];     { strip data buffer }
.VAR/DM/ABS=0x0C01 ped1[NSTRIP];         { pedestal value table }
.VAR/DM/ABS=0x1803 sig1[NSTRIP];         { sigma value table }
.VAR/DM/ABS=0x1001 ped2[NSTRIP];         { pedestal value table }
.VAR/DM/ABS=0x1c03 sig2[NSTRIP];         { sigma value table }
.VAR/DM/ABS=0x1401 ped3[NSTRIP];         { pedestal value table }
.VAR/DM/ABS=0x2003 sig3[NSTRIP];         { sigma value table }
{*****************************************************************
 *
 *              Variables location definitions
 *
 *****************************************************************}
.var/dm/abs=0x0c00 Contr1;       { Control word before ped table}
.var/dm/abs=0x1801 Contr2;       { Control word after ped table}
.var/dm/abs=0x1802 Contr3;       { Control word before sig table}
.var/dm/abs=0x2403 Contr4;       { Control word after sig table}
.var/dm/abs=0x2404 MailBox;      { Used to write control value for DSP}
.var/dm/abs=0x2405 EventNumTrig; { Progr. number of the triggered ev.}
.var/dm/abs=0x2406 EventNumTras; { Progr. number of the transmitted ev.}
.var/dm/abs=0x2407 Checksum;     { Checksum of the transmitted numbers}
.var/dm/abs=0x2408 WordTras;     { Number of words transmitted}
.var/dm/abs=0x2409 InfoCompr1;   { Information 1 on compression}
.var/dm/abs=0x240a InfoCompr2;   { Information 2 on compression}
.var/dm/abs=0x240b InfoCompr3;   { Information 3 on compression}
.var/dm/abs=0x240c Counter;      { Counter }
.var/dm/abs=0x240d IS;           { Current strip (on compression) }
.var/dm/abs=0x240e IT;           { Last transmitted strip }
.var/dm/abs=0x240f TRASMESSO;    { Flag for transmitted strip }
.var/dm/abs=0x2410 TrasmDato;    { Datum to be transmitted }
.var/dm/abs=0x2411 Temp;         { Temporary variable }
.var/dm/abs=0x2412 newdp;        { New data-ped in compression }
.var/dm/abs=0x2413 olddp;        { Old data-ped in compression }
.var/dm/abs=0x2414 Mail_XY;      { Side X or side Y  }
```

```
.var/dm/abs=0x2415 Temp2;        { Temporary variable }
.var/dm/abs=0x2416 Temp3;        { Temporary variable }
{declaration of BIT_FILE}
.var/dm/abs=0x2417 BIT_FILE_rack;
.var/dm/abs=0x2418 BIT_FILE_mask;
.var/dm/abs=0x2419 numero_di_eventi_compressi;
{declaration of TREE }
.var/dm/abs=0x3fac stack[50];                    {parte di albero }
.var/dm/abs=0x2420 next_free_nodo;               {in DM           }
.var/dm/abs=0x2421 nodochild_is_foglia[TOTNODES]; {_______________}
.var/pm/abs=0x500 foglia[SYMCOUNT];              {parte di albero }
.var/pm/abs=0x902 nodoparent[TOTNODES];          {in PM           }
.var/pm/abs=0x1105 nodochild[TOTNODES];          {                }
.var/pm/abs=0x1908 nodoweight[TOTNODES];         {                }

{*****************************************************************
 *
 *     System initialization routine
 *     Jump here at reset
 *     Setting interrupt mask, control register and init memory
 *
 *****************************************************************}
{
  Interrupt settings, mask and control registers
 }
start:
{
   Writing of IMASK interrupt mask register (see below):
   Only IRQE interrupt is allowed from acquisition
}
      imask = b#0000010000;     { 1=enable, 0=disable }
          {     |||||||||+ | timer
                ||||||||+- | SPORT1 rec or IRQ0
                |||||||+-- | SPORT1 trx or IRQ1
                ||||||+---- | BDMA
                |||||+----- | IRQE               (only this is enabled)
                ||||+------ | SPORT0 rec
                |||+------- | SPORT0 trx
                ||+-------- | IRQL0
                |+--------- | IRQL1
               +---------- | IRQ2
           }
{
   Writing IFC (Interrupt Force Control) register, write only
}
      {     Force->|<-Clear       }
      IFC=b#0000000000000000;  { 0 means normal interrupt, no force }
{           ||||||||||||||||
IRQ2        +|||||||||||||||+ | timer
SPORT0 rec   +||||||||||||||+- | SPORT1 rx or IRQ0
SPORT0 trx    +|||||||||||||+-- | SPORT1 tx or IRQ1
IRQE (neg)     +||||||||||||+--- | BDMA
BDMA            +|||||||||||+---- | IRQE (neg)
SPORT1 tx or IRQ1+||||||||||+----- | SPORT0 rec
SPORT1 rx or IRQ0 +|||||||||+------ | SPORT0 trx
Timer              ++++++++++------- | IRQ2
}
{
   Writing ICNTL (Interrupt CoNTrol Register, see below
   No nesting allowed, level on IRQ, but not used
}
      ICNTL=b#00000;            { interrupt edging and nesting setting }
        {     ||||+ | IRQ0 sensitivity |
              |||+- | IRQ1 sensitivity |> 1=edge, 0=level
              ||+-- | IRQ2 sensitivity |
              |+--- | unused
              +---- | Interrupt nesting (1=enable, 0=disable)
        }
{
   Writing MSTAT, see below
   Forcing MAC to integer
}
        MSTAT = b#0010000;  { =0x0010, =16 }
          {       ||||||+- | Data register bank select
                  |||||+-- | FFT bit reverse mode (DAG1)
```

```
                |||+--- | ALU overflow latch mode, 1=sticky
                ||+---- | AR saturation mode, 1=saturate, 0=wrap
                |+----- | MAC result, 0=fractional, 1=integer
                +------ | timer enable
               +------- | GO MODE
          }
{*****************************************
        Setting Memory Mapped Registers:
        - they range from 0x3FFF to 0x3FE0
        - these values set some general registers of
          the DSP (like timer, SPORTs, etc,)
 *****************************************}
{
  Setting System control register (System_Control_Reg)
  0x3fff=b#0001110000000000 (0x1C00)
             ||+ bit 10 SPORT1 Configure: 1 SPORT, 0=FI,FO,IRQ,SCLK
             |+  bit 11 SPORT1 Enable: 1 enable, 2 disable
             +   bit 12 SPORT0 Enable
 }
        AX0=0x1C00;                  { SPORT0 enabled, SPORT1 en. & config.}
        DM(System_Control_Reg)=AX0;  { System Control Register }
{
   Setting waitstate and timer registers
 }
        AX0=0;               { Set register to zero }
        DM(DM_Wait_Reg)=AX0; { Waitstate Control Register 0=disable }
        DM(TPERIOD)=AX0;     { Timer registers: TPERIOD }
        DM(TCOUNT)=AX0;      { Timer registers: TCOUNT }
        DM(TSCALE)=AX0;      { Timer registers: TSCALE (lower byte) }
{
        Serial Port Registers:
        SPORT1 and SPORT0 settings.
        MCWE = MultiChannel Word Enable
        SCLKDIV = Serial ClocK DIVide
        ACR = Autobuffer Control Register
 }
        DM(SPORT0_RX_Channels1)=AX0;    { Clean-up of SPORT0 MCWE RX 31-16 }
        DM(SPORT0_RX_Channels0)=AX0;    { Clean-up of SPORT0 MCWE RX 15-0 }
        DM(SPORT0_TX_Channels1)=AX0;    { Clean-up of SPORT0 MCWE TX 31-16 }
        DM(SPORT0_TX_Channels0)=AX0;    { Clean-up of SPORT0 MCWE TX 15-0 }
        DM(SPORT0_SCLKDIV)=AX0;         { Clean-up of SPORT0 Modulus }
        DM(SPORT0_Autobuf)=AX0;         { Clean-up of SPORT0 ACR }
{
  SPORT1 register setting
 }
        DM(SPORT1_SCLKDIV)=AX0;         { Clean-up of SPORT0 Modulus }
        DM(SPORT1_Autobuf)=AX0;         { Clean-up of SPORT0 ACR }
{
        Set SPORT0 Control Register 0x3FF2 (SPORT1_Control_Reg)
        0x3FF2=b#0111111100001001     (0x7F09)
        ||||||||||||++++  0-3 Serial Word Lenght (set to 9+1=10)
        ||||||||||++      4-5 DTYP, 0=right justify, zero fill unused MSB
        |||||||||+        6 INVRFS Invert Receve Frame (0=no)
        ||||||||+         7 INVTFS Invert Transmit Frame (0=no)
        |||||||+          8 IRFS External Frame Sync for - rec (1=internal)
        ||||||+           9 ITFS Internal Frame Sync - tras (0=external)
        |||||+            10 RFSW Normal=0 or Alternate=1 tras Framing
        ||||+             11 TSFR Transmit Frame Sync on all word (0=on 1st)
        |||+              12 TFSW Normal=0 or Alternate=1 rec.  Framing
        ||+               13 RSFR Receive Frame Sync on all word (0=on 1st)
        |+                14 ISCLK=0 External clock set (1=internal)
        +                 15 Flag-Out (read only)
}
        AX0=0x7F09;                     { serial 0 }
        DM(SPORT1_Control_Reg)=AX0;     { SPORT0 Control Register }

        AX0=255;
        DM(DM_Wait_Reg)=AX0;
{
        Set SPORT0 Control Register 0x3FF6 (SPORT0_Control_Reg)
        0x3FF6=b#0010111000001111    (0x4E0F)
        ||||||||||||++++  Serial Word Lenght -1 (set to 16)
        ||||||||||++      DTYP, 0=right justify, zero fill unused MSB
```

```
        |||||||||+        Invert Receve Frame (0=no)
        ||||||||+         Invert Transmit Frame (0=no)
        |||||||+          External Frame Sync for Receive (1=internal)
        ||||||+           Internal Frame Sync for Transmit (0=external)
        |||||+
        ||||+             Transmit Frame Sync on all word (0=only on 1st)
        |||+
        ||+               Receive Frame Sync on all word (0=only on 1st)
        |+                ISCLK=0 External clock set (1=internal)
        +
}
      AX0=0x4E0F;                       { serial 0 }
      DM(SPORT0_Control_Reg)=AX0;       { SPORT0 Control Register }

 {*****************************************************************
 *      Initialization and buffer erasing section:
 *      sigma values are set for cut and transmission
 *****************************************************************}

        AR=0;                           { clean-up of AR }
{
        1) Zeroing of variables;
        2) Initialization of address registers:
           L0-L7 are for circular buffer
           M0-M7 are for automatic increase
 }
     L0=AR; L1=AR; L2=AR; L3=AR;  { Circolar buffer lenght registers }
     L4=AR; L5=AR; L6=AR; L7=AR;  { 0 to avoid circular buffers }
     M0=0; M1=1; M2=1024; M3=3;   { Increments for address registers }
     M4=0; M5=1; M6=1024; M7=3;   { All MX are init. to needed values }
{
        Zeroing of ALU, MAC and shifter registers
}
     AX0=AR; AX1=AR; AY0=AR; AY1=AR;         { ALU registers }
     MR=0; MY0=AR; MY1=AR; MX0=AR; MX1=AR;   { MAC registers }
     SR0=AR; SR1=AR; SE=AR; SI=AR; SB=AR;    { Shifter registers }
{
        Variables initialization
}
      DM(EventNumTrig)=AR;      { Event number clean-up }
      DM(EventNumTras)=AR;      { Event number clean-up }
      DM(WordTras)=AR;          { Number of words transm. clean-up}
      DM(Counter)=AR;
      DM(IS)=AR;
      DM(IT)=AR;
      DM(TrasmDato)=AR;
      DM(TRASMESSO)=AR;
      DM(Checksum)=AR;
      DM(WordTras)=AR;
      DM(InfoCompr1)=AR;
      DM(InfoCompr2)=AR;
      DM(InfoCompr3)=AR;
      DM(numero_di_eventi_compressi)=AR;
      AR=1;
      DM(Mail_XY)=AR;           { default mail_XY value 1=Y view }
      AR=2;                     { default mailbox value for compr. trans.}
      DM(MailBox)=AR;           { set mailbox }
{
        Buffers erasing (and number of transmitted words)
}
   call clearbuf;               { clean-up of data buffer, but not ped and sig }
{*************************************************************
 *
 *      End of initialization:
 *      the program goes to idle cycle and waits for trigger
 *
 *************************************************************}
wait:   NOP;
        idle;
        jump wait;
.endmod;
```

Main.dsp. *Routine* principale. Qui si legge la variabile `Mailbox` e di conseguenza si passa alla trasmissione compressa (`compacq.dsp`) o non compressa (`fullacq.dsp`). Una

volta eseguita una di queste *routine* si incontra il comando `RTI`, che pone di nuovo il DSP in stato di quiescenza in attesa di segnali di *interrupt*.

```
.module/ram MAIN;
.include <constant.k>;
.entry  main;

{*****************************************************
 *      Program beginning: the code jumps here
 *      at each interrupt generated by the trigger system.
 *      This routine simply checks
 *      the MailBox communication location and calls the needed
 *      routine to perform the desired task
 *****************************************************}
main:
        cntr=10;
        do wt11 until ce;
wt11:   nop;
        ar=ADDRESS0DM;
        dm(IDMA)=ar;
        ar=dm(EventNumTrig);    { number of trigger from start }
    ar=ar+1;
    dm(EventNumTrig)=ar;
{*********************************************************}
    call waitlong;  {wait for 1.1 msec, until all the data are in memory}
{*********************************************************}
    ar=dm(MailBox);
    ar=pass ar;         { pass Mailbox through ALU }
    af=ar-1;
    if ne jump other;   { value 1 means full acquisition }
    call headtras;      { transmit header}
    call fullacq;       { transmit event }
    call traitras;      { transmit trailer}
    rti;                { end }
other:
    af=ar-2;
    if ne jump never;   { value 2 means compressed format }
    call headtras;      { transmit header}
    call compacq;       { do compression }
    call traitras;      { transmit trailer}
    rti;
never:
    RTI;
.ENDMOD;
```

Clearbuf.dsp. Azzera i *buffer* dei dati, dei piedistalli e delle sigma e la variabile `WordTras`.

```
.module/ram CLEARBUF;
.include <constant.k>;
.entry  clearbuf;
{*****************************************************************
 *      Erasing data buffer but not ped and sig
 *      Erasing number of words transmitted
 *****************************************************************}
clearbuf:
{
        Initial clean-up of data, pedestal and sigma buffers
}
        CNTR=%buffer;  { take lenght of buffer }
        I0=^buffer;    { address for buffer}
        AR=0;          { clean-up of AR }
{
        Clean-up of data, pedestal and sigma buffers
}
        DO CLEAR UNTIL CE;              { erasing loop }
CLEAR:     DM(I0,M1)=AR;                { erase buffer }
{
        Clean-up number of transmitted words
}
        dm(WordTras)=ar;
        ar=ADDRESS0DM;
        dm(IDMA)=ar;
```

```
{
   End of erasing loop
}
        RTS;
.endmod;
```

Wait.dsp. Questa *routine* fa 14 cicli a vuoto. In questo modo il DSP rimane in attesa per 17×33 ns $= 561$ ns.

```
.module/ram WAIT;
.include <constant.k>;
.entry wait;
wait:
     cntr=14;
     do lkh until ce;
lkh: nop;
     rts;
.endmod;
```

Waitlong.dsp. Questa *routine* fa 44950 cicli a vuoto. In questo modo il DSP rimane in attesa per 1.1 ms.

```
.module/ram WAITLONG;
.include <constant.k>;
.entry waitlong;
waitlong:
     cntr=4095;
     do lkh1 until ce;
lkh1: nop;
     cntr=4095;
     do lkh2 until ce;
lkh2: nop;
     cntr=4095;
     do lkh3 until ce;
lkh3: nop;
     cntr=4095;
     do lkh4 until ce;
lkh4: nop;
     cntr=4095;
     do lkh5 until ce;
lkh5: nop;
     cntr=2000;
     do lkh6 until ce;
lkh6: nop;
     rts;
.endmod;
```

Fullacq.dsp. Trasmette senza compressione tutte le 3072 strisce contenute in *buffer*.

```
.module/ram FULLACQ;
.include <constant.k>;
.entry fullacq;
fullacq:
{
        Transmission :
        information is 3072 * 12 bit. We transmit
        3072 * 2 * 10 bit (6144 taxi transmissions)
        First word:     01+i7/i0
        Second word:    10+i15/i8
}
        i0=^buffer;
        cntr=TOTSTRIP;
        do tras until ce;
                ar=dm(i0,m1);
                ay1=0xfff;
                ar=ar and ay1;
                dm(TrasmDato)=ar;
                call transmit;
tras:       nop;
        rts;
.endmod;
```

Headtras.dsp. Trasmette le 10 parole dello *header*.

```
.module/ram HEADTRAS;
.include <constant.k>;
.entry headtras;
headtras:
{
        Transmission with identifier

11      Init Header:    DAF0 for full, DAC0 for compressed
10+     EventNumTrig (14 bits)
10+     MailBox
10+     IDMA address
10+     Contr1, Contr2, Contr3, Contr4
10+     WordTras
01      End Header:    4AF0 for full, 4AC0 for compressed
{10 words, it means 20 transmissions}
{
        Init of header
}
        ar=0xDAE0;
        ax0=ar;             { if init is 0xDAE0 --> error in MailBox }
        ar=dm(MailBox);
        ar=pass ar;         { pass Mailbox through ALU }
        af=ar-1;
        if eq jump a1;
        af=ar-2;
        if eq jump a2;
        jump tx;
a1:     ax0=0xDAF0;         { value 1 is full acquisition }
        jump tx;
a2:     ax0=0xDAC0;         { value 2 is compressed acquisition }
tx:     ar=ax0;
        dm(TrasmDato)=ar;
        call transmit;
{
        Number of triggers
}
        ax0=DM(EventNumTrig);
        ay1=0x3FFF;                     { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        MailBox
}
        ax0=DM(MailBox);
        ay1=0x3FFF;                     { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        IDMA Address splitted in two 8-bits words
}
        i1=0x3fe0;
        ax0=DM(I1,M1);
        ay1=0xFF;                       { mask for 8 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
        i1=0x3fe0;
        ax0=DM(I1,M1);
        si=ax0;                         { take 1st word }
        sr=lshift si by -8 (HI);        { and shift it 8 bit left  }
        ay1=0xFF;                       { mask to have 8 bits }
        ar=sr1 AND ay1;                 { take it for 2nd word }
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        Contr1
}
        ax0=DM(Contr1);
        ay1=0x3FFF;                     { mask for 14 bits }
```

```
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        Contr2
}
        ax0=DM(Contr2);
        ay1=0x3FFF;                     { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        Contr3
}
        ax0=DM(Contr3);
        ay1=0x3FFF;                     { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        Contr4
}
        ax0=DM(Contr4);
        ay1=0x3FFF;                     { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;             { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{
        End of header
}
        ar=0x4AE0;
        ax0=ar;                 { if end is 0x4AE0 --> error in MailBox }
        ar=dm(MailBox);
        ar=pass ar;             { pass Mailbox through ALU }
        af=ar-1;
        if eq jump b1;
        af=ar-2;
        if eq jump b2;
        jump btx;
b1:     ax0=0x4AF0;             { value 1 is full acquisition }
        jump btx;
b2:     ax0=0x4AC0;             { value 2 is compressed acquisition }
btx:    ar=ax0;
        dm(TrasmDato)=ar;
        call transmit;
        rts;
.endmod;
```

Traitras.dsp. Trasmette le 15 parole del *trailer*.

```
.module/ram TRAITRAS;
.include <constant.k>;
.entry traitras;

traitras:
{
        End of transmission with identifier
11      Init Trailer:   DAF1 for full, DAC1 for compressed
10+     EventNumTrig (14 bits)
10+     MailBox
10+     IDMA address
10+     Contr1, Contr2, Contr3, Contr4
10+     WordTras
10+     Checksum
10+     InfoCompr1, InfoCompr2, InfoCompr3
01      End Trailer: 4AF1 for full, 4AC1 for compressed
        15 words, means 30 transmissions
}
{****** Init of trailer ******}
        ar=0xDAE1;
        ax0=ar;                 { if init is 0xDAE1 --> error in MailBox}
        ar=dm(MailBox);
        ar=pass ar;             { pass Mailbox through ALU }
```

```
        af=ar-1;
        if eq jump a1;
        af=ar-2;
        if eq jump a2;
        jump tx;
a1:     ax0=0xDAF1;             { value 1 is full acquisition }
        jump tx;
a2:     ax0=0xDAC1;             { value 2 is compressed acquisition }
tx:     ar=ax0;
        dm(TrasmDato)=ar;
        call transmit;
{****** Number of triggers *******}
        ax0=DM(EventNumTrig);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { set bit 15 for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** MailBox ********}
        ax0=DM(MailBox);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { set bit 15 for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** IDMA Address Split in 2 8-bits words *****}
        i1=0x3fe0;
        ax0=DM(I1,M1);
        ay1=0xFF;               { mask for 8 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit set 15 for control }
        dm(TrasmDato)=ar;
        call transmit;
        i1=0x3fe0;
        ax0=DM(I1,M1);
        si=ax0;                 { take 1st word }
        sr=lshift si by -8 (HI);{ and shift it 8 bit left }
        ay1=0xFF;               { mask to have 8 bits }
        ar=sr1 AND ay1;         { take it for 2nd word }
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** Contr1 ******}
        ax0=DM(Contr1);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** Contr2 ******}
        ax0=DM(Contr2);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** Contr3 *******}
        ax0=DM(Contr3);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** Contr4 ******}
        ax0=DM(Contr4);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** InfoCompr1 *******}
        ax0=DM(InfoCompr1);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
```

```
{****** InfoCompr2 *******}
        ax0=DM(InfoCompr2);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** InfoCompr3 *******}
        ax0=DM(InfoCompr3);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** Checksum *******}
        ax0=DM(Checksum);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { bit 15 set for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** Number of transmitted words (excluding the last 4 transmissions)}
        ax0=DM(WordTras);
        ay1=0x3FFF;             { mask for 14 bits }
        ar=ax0 AND ay1;
        ar=setbit 15 of ar;     { set bit 15 for control }
        dm(TrasmDato)=ar;
        call transmit;
{****** End of trailer ******}
        ar=0x4AE1;
        ax0=ar;                 { if end is 0x4AE1 --> error in MailBox }
        ar=dm(MailBox);
        ar=pass ar;             { pass Mailbox through ALU }
        af=ar-1;
        if eq jump b1;
        af=ar-2;
        if eq jump b2;
        jump btx;
b1:     ax0=0x4AF1;             { value 1 is full acquisition }
        jump btx;
b2:     ax0=0x4AC1;             { value 2 is compressed acquisition }
btx:    ar=ax0;
        dm(TrasmDato)=ar;
        call transmit;
{****** Zeroing of Buffer and WordTras *******}
        call clearbuf;
        rts;
.endmod;
```

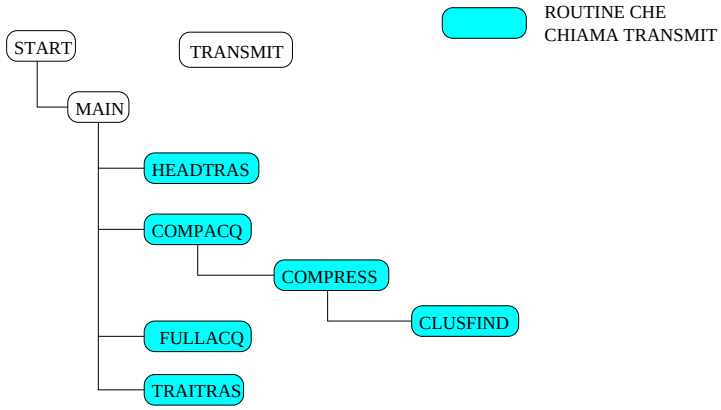
Transmit.dsp. Trasmette la parola contenuta nella variabile `TrasmDato`; essa viene spezzata in due parole a 8 *bit* alle quali vengono anteposti i *bit* di controllo “01” e “10”.

```
.module/ram TRANSMIT;
.include <constant.k>;
.entry transmit;
transmit:
{
        Init transmission:
        information is 3072 * 16 bit. We transmit
        3072 * 2 * 10 bit (6144 taxi transmissions)
        First word:     01+i7/i0
        Second word:    10+i15/i8
}
            ax0=dm(TrasmDato);          { 1st word (16 bit) }
            si=ax0;                     { take 1st word }
            sr=lshift si by -8 (HI);    { and shift it 8 bit left  }
            ay1=0xFF;                   { mask to have 8 bits }
            ar=sr1 AND ay1;             { take it for 2nd word }
            ar=setbit 9 of ar;          { set bit 9 for control }
            TX1=ar;                     { transmit }
            ar=dm(WordTras);            { number of transm. words in ev.}
            ar=ar+1;
            dm(WordTras)=ar;
         call wait;
{
```

```
   1st word most significant 8 bits transmitted
}
           ay1=0xFF;                    { mask for low 8 bits }
           ar=ax0 AND ay1;              { take it for 1sr word }
           ar=setbit 8 of ar;           { set bit 8 for control }
           TX1=ar;                      { transmit 1sr word low 8 bits }
           ar=dm(WordTras);             { number of transm. words in ev.}
           ar=ar+1;
           dm(WordTras)=ar;
         call wait;
{
   1st word least significant 8 bits transmitted
}
        rts;
.endmod;
```

Zero Order Predictor (ZOP)

Di seguito sono riportate le *routine* dell'algoritmo del predittore di ordine zero.



Compacq.dsp. Comprime consecutivamente i dati delle tre viste gestite dal DSP. Prima di ogni compressione viene trasmessa una parola di controllo.

```
.module/ram COMPACQ;
.include <constant.k>;
.entry  compacq;
compacq:
{ IS current strip counter }
{ IT last transmitted strip }
        i0=^buffer;
        i4=^ped1;
        i2=^sig1;
        call compress;
        ar=0x1801;
        dm(TrasmDato)=ar;
        call transmit;
        i0=^buffer+1;
        i4=^ped2;
        i2=^sig2;
        call compress;
        ar=0x1802;
        dm(TrasmDato)=ar;
        call transmit;
        i0=^buffer+2;
        i4=^ped3;
        i2=^sig3;
        call compress;
        ar=0x1803;
        dm(TrasmDato)=ar;
        call transmit;
    RTS;
.ENDMOD;
```

Compress.dsp. Opera la compressione delle 1024 strisce di una unità di rivelazione. Trasmette sempre la prima striscia, poi opera il confronto dei dati successivi con un taglio (`CUTCL`$\times\sigma_i$) e poi con un secondo taglio (`CUTC`$\times\sigma_i$), una volta sottratti i piedistalli. Se il dato supera il primo taglio, viene chiamata la *routine* `clusfind`; se supera il secondo taglio, la striscia viene trasmessa. Se, infine, nessuno dei tagli viene superato, la striscia non viene trasmessa.

```
.module/ram COMPRESS;
.include <constant.k>;
.entry  compress;
compress:
/* IS current strip counter  */
/* IT last transmitted strip */
       AR=0;
       DM(IS)=AR;
       DM(IT)=AR;            /* 1st strip always transmitted */
       AR=1;
       DM(TRASMESSO)=AR;     /* transmission flag */
       AX0=DM(I0,M3);        /* take datum */
       AY0=DM(I4,M5);        /* take pedestal */
       MY1=DM(I2,M1);        /* take sigma */
         AY1=0xFFF;          /* mask for low 12 bits */
         AR=AX0 AND AY1;
         AX0=AR;
       AR=AX0-AY0;           /* take difference d-p */
       sr0=AR;               /* store it in sr0=OLDVAL */
        dm(olddp)=AR;
        dm(TrasmDato)=ax0;
        call transmit;
next:
         AR=dm(IS);
       AR=AR+1;
       dm(IS)=AR;
       AX0=DM(I0,M3);         /* take next datum */
       AY0=DM(I4,M5);         /* take next pedestal */
       MY1=DM(I2,M1);         /* take next sigma */
         AY1=0xFFF;           /* mask for low 12 bits */
         AR=AX0 AND AY1;
         AX0=AR;
       dm(Temp)=ax0;
         sr0=dm(olddp);
/*
   Control range of values: if they are out of range
   strip is not transmitted
*/
      ay1=ADCMIN;
      ar=ax0-ay1;
      if le jump notrasmit;
      ay1=ADCMAX;
      ar=ax0-ay1;
      if gt jump notrasmit;
/*
   Now look at values
*/
       AF=AX0-AY0;           /* make difference*/
       mx1=CUTCL;            /* take cut */
         ar=pass AF;         /* put d-p in ar */
       mr=mx1*my1 (UU);      /* compute cut */
       sr1=ar;               /* store new d-p */
       dm(newdp)=AR;

         ar=dm(Mail_XY);     /* select view using Mail_XY:  */
         ar=PASS ar;         /* 0=X view                    */
         if eq jump Xview;   /* anything else=Y view        */
/*
  Look for cluster: only on Y view.
*/
       af=af-sr0;            /* take difference of differences */
         jump cluslook;
/*
  Look for cluster: only on X view
*/
```

```
Xview:
       af=sr0-af;            /* take difference of differences */
cluslook:
         ar=mr0-af;
         if gt jump normal;
        call clusfind;
        jump fine;
        normal:
/*
  Normal strip search
*/
       mx1=CUTC;             /* take cut */
       mr=mx1*my1 (UU);      /* compute cut */
       AR=dm(newdp);
       AY0=dm(olddp);
       ar=ar-ay0;            /* take difference of differences */
       af=abs ar;            /* in absolute value */
       ar=mr0-af;            /* subctract to cut */
       if le jump trasmit;
notrasmit:
       ar=0;
       dm(TRASMESSO)=AR;
       jump fine;
/*
  Transmission
*/
trasmit:
/*       SR0=SR1;*/           /* set previous d-p */
       AR=dm(newdp);
       dm(olddp)=AR;
       AR=DM(TRASMESSO);     /* look transmission flag*/
       AR=PASS AR;
       AR=DM(IS);            /* take current strip address */
       dm(IT)=ar;            /* set last strip transmitted */
       dm(TRASMESSO)=AR;
       IF NE JUMP NOINDIRIZ;
               AR=setbit 12 of ar;
               ay1=0x13FF;
               AR=ar AND ay1;
               dm(TrasmDato)=ar;
               call transmit;
NOINDIRIZ:  ax0=dm(Temp);
               dm(TrasmDato)=ax0;
               call transmit; /* transmit strip datum */
/*
 end loop control
*/
fine:
        ar=dm(IS);
        ar=ar+1;
        ar=ar-1024;           /* compare with NSTRIP */
        if lt jump next;      /* if not last repeat */
RTS;
.ENDMOD;
```

Clusfind.dsp. Questa *routine* viene chiamata quando il taglio `CUTCL`$\times\sigma_i$ viene superato (la versione qui riportata è per picchi positivi). La *routine* cerca il massimo del picco, poi trasmette `NSTRIP` a destra e a sinistra del massimo del picco, oppure a partire dall'ultima striscia trasmessa, se questa viene oltre il massimo del picco meno `NSTRIP`.

Dopo aver fatto questo il comando torna alla *routine* `compress`, che riprende la normale analisi a partire dalla striscia successiva all'ultima striscia trasmessa.

```
.module/ram CLUSFIND;
.include<constant.k>;
.entry clusfind;

/*********************************************************************
*       tabella degli indirizzi di memoria iniziali:                 *
*                                                                    *
*       i0=prossima strip    i4=piedistallo prossima strip           *
```

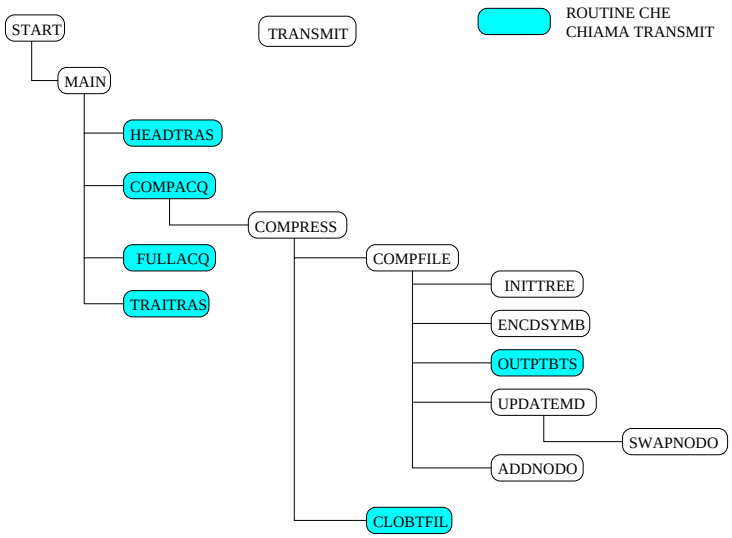
```
*                                                                     *
*       m0=m4=0                                                       *
*                                                                     *
*       m3=m7=3    (il DSP legge tre viste per volta, quindi le       *
*                   strip di uno stesso ladder sono poste in          *
*                   memoria a salti di tre)                           *
*                                                                     *
*       m1=m5=1    (i piedistalli delle strip sono posti in           *
*                   indirizzi di memoria contigui)                    *
*                                                                     *
*       m2=m6=1024 (numero di strip in un ladder)                     *
***********************************************************************/
clusfind:
         ay1=dm(newdp);  /*dato strip attuale pulita*/
           ar=dm(IS);
           dm(ISCL)=ar;
        my1=-1;
        ar=dm(Mail_XY);
        ar=PASS ar;
        if eq jump maxloop;
        my1=1;
maxloop:
        ax0=dm(i0,m3);  /*dato prossima strip*/
        ay0=dm(i4,m5);  /*piedistallo prossima strip*/
        modify(i2,m1);  /*sigma prossima strip*/
        af=PASS 0xfff;
        ar=ax0 AND af;
        ax0=ar;
        ar=dm(ISCL);
        ar=ar+1;        /*aggiorna IS(indice della strip)  */
        dm(ISCL)=ar;
        af=PASS ar;
        ar=NSTRIP;
        ar=af-ar;
        if ge jump continue;
        ar=ADCMIN;
        ar=ax0-af;
        if le jump continue;
                                /*bloccaggio overflow */
        ar=ADCMAX;
        af=PASS ar;
        ar=ax0-af;
        if gt jump continue;
      ar=ax0-ay0;               /*prossima strip pulita*/
                        /*differenza tra le due strip */
      af=ar-ay1,        /*(solo per segnali positivi)*/
      ay1=ar;           /*imposta la prossima strip come attuale*/
                        /*per il prossimo loop */
        ar=PASS af;
        mx1=ar;
        mr=mx1*my1 (SS);
        ar=PASS mr0;
      if ge jump maxloop;       /* se la distribuz.  crescente il loop */
                                /* continua; se non lo , i4 contiene   */
                                /* l'indirizzo della strip corrisp.     */
                                /* al massimo del cluster +1 ( stata   */
                                /* incrementata di m5)                  */
continue:

        ay1=dm(ISCL);           /* ora ay1=l'indice della strip del   */
        ar=ay1-1;               /* massimo del cluster                */
        ay1=ar;                 /*                                    */
        dm(ISCL)=ar;
        ar=dm(IT);              /* indice dell'ultima strip trasmessa */
        af=ay1-ar;              /* ar=distanza tra l'ultima riga      */
        ar=af-1;                /* trasmessa+1 e il max del cluster   */
        ay1=NCLUST;             /* af=distanza tra  margine inferiore del */
        af=ar-ay1;              /* cluster ed ultima strip trasmessa+1    */
        if lt jump readdress;   /* se  minore di zero non c' bisogno di */
                                /* riaggiustare ar                        */
        ax1=dm(IS);
        ay1=dm(ISCL);
        ar=ay1-ax1;
        ay1=NCLUST;
        af=ar-ay1;
```

```
        if lt ar=PASS ay1;
readdress:
        af=ar+2;
        ax1=i4;          /* Ora ax1 contiene l'indirizzo del margine */
        ar=ax1-af;       /* inferiore del cluster (piedistallo)      */
        i4=ar;           /* i4 viene riposizionato a bordo cluster   */
                         /* oppure all'ultima strip trsmessa+1       */

        ax1=i2;          /* Riposiziona l'indirizzo delle sigma      */
        ar=ax1-af;       /* al bordo inferiore del cluster           */
        i2=ar;           /* oppure all'ultima strip trasmessa+1      */
        ar=PASS af;
        mx0=3;           /* Poich i0 contiene i dati delle strip    */
        my0=ar;          /* a salti di tre, af va moltiplicata per   */
        mr=mx0*my0(UU);  /* tre prima di essere sommato ad i0        */

        ay1=mr0;         /*                                          */
        ax1=i0;          /* Ora i0 contiene l'indirizzo del bordo    */
        ar=ax1-ay1;      /* inferiore del cluster oppure dell'ultima */
        i0=ar;           /* strip trasmessa+1                        */

        ar=dm(ISCL);
        ar=ar-af;        /* Aggiorna IS (indice della strip)         */
        ar=ar+2;
        dm(ISCL)=ar;
        ax1=ar;

        ay1=dm(IT);             /* controllo posizioni di IS e IT:   */
        ar=ay1+1;               /* se sono attigui non c' bisogno   */
        ay1=ar;                 /* di mandare un nuovo indirizzo     */
        ar=ax1-ay1;
        if eq jump nonindiriz;
        ar=setbit 12 of ax1;
        ay1=0x13FF;
        ar=ar AND ay1;          /* nuovo indirizzo per la nuova serie*/
        dm(TrasmDato)=ar;       /* di dati                           */
        call transmit;
nonindiriz:
        ar=NCLUST;              /* loop di trasmissione strip        */
        ar=ar-1;
        ar=ar+af;
        cntr=ar;
        do tras until ce;
                modify(i2,m1);  /* aggiorna l'indirizzo della sigma  */
                ax0=dm(i0,m3);  /* presa dati da trasmettere         */
                ay0=dm(i4,m5);
                ar=dm(ISCL);
                ar=ar+1;        /* aggiorna IS(indice della strip)   */
                dm(ISCL)=ar;
                ay1=ADCMIN;
                ar=ax0-ay1;
                if le jump notras;
                                /* bloccaggio overflow          */
                ay1=ADCMAX;
                ar=ax0-ay1;
                if gt jump notras;
                AY1=0xFFF;
                ar=ax0 and ay1;      /* pulizia del dato da trasm.   */
                ax0=ar;
                ax1=dm(ISCL);
                ar=ax1-NSTRIP;       /* controllo                    */
                if lt jump nostop;
                OWRCNTR=1;
        nostop:
                dm(TrasmDato)=ax0;   /* trasmissione del dato        */
                call transmit;
                ar=ax0-ay0;
                dm(olddp)=ar;
                ar=dm(ISCL);
                ar=ar-1;             /* aggiornamento di IT          */
                dm(IT)=ar;
                ar=1;
                dm(TRASMESSO)=ar;    /* aggiornamento della flag     */
                jump tras;
        notras:
                OWRCNTR=1;
```

```
                ar=i0;
                ar=ar-3;
                i0=ar;
                ar=i2;
                ar=ar-1;
                i2=ar;
                ar=i4;
                ar=ar-1;
                i4=ar;
                ar=dm(ISCL);
                ar=ar-1;
                dm(ISCL)=ar;
        tras: NOP;
fine:
     ar=ax0-ay0;
     dm(olddp)=ar;
     ar=dm(IT);
     dm(IS)=ar;
     RTS;
.endmod;
```

Algoritmo di Huffmann

Di seguito sono riportate le *routine* dell'algoritmo di Huffmann.



Compacq.dsp. Dopo aver chiamato la *routine* `stripper`, trasmette una parola di controllo, chiama la *routine* `compress`, che opera la compressione vera e propria, quindi trasmette altre due parole di controllo.

```
.module/ram COMPACQ;
.include <constant.k>;
.entry  compacq;
compacq:
        call stripper;
       ar=0x1801;
       dm(TrasmDato)=ar;
         call transmit;
         call compress;
       ar=0x1802;
       dm(TrasmDato)=ar;
         call transmit;
       ar=0x1803;
       dm(TrasmDato)=ar;
         call transmit;
RTS;
.ENDMOD;
```

Stripper.dsp. Sottrae da ogni dato contenuto nella variabile `buffer` il relativo piedistallo, aggiunge `HALFVALMAX` e sostituisce il risultato al dato.

```
.module/ram STRIPPER;
.include <constant.k>;
.entry  stripper;
stripper:
  i2=^buffer;           /* init i2 to the beginning of data buffer */
  i6=^ped1;             /* init i6 to the beginning of ped buffer */
  cntr=3;               /* loop on 3 ladders */
  do view until ce;
  i0=i2;                /* set i0 at ^buffer,^buffer+1,^buffer+2 */
  i4=i6;                /* set i4 at ped1,ped2,ped3 */
  cntr=NSTRIP;          /* loop on single ladder strips */
  do strip until ce;
  ax0=dm(i0,m0);        /* datum,modified by m0=0 */
  ay0=dm(i4,m5);        /* pedestal relative to datum modified by m5=1 */
  af=ax0-ay0;           /* datum minus pedestal */
  ar=HALFVALMAX;        /* plus HALFVALMAX */
  ar=ar+af;
  dm(i0,m3)=ar;         /* we store in the previous location of datum */
                        /* datum minus pedestal plus HALFVALMAX       */
strip:  NOP;
  modify(i2,m1);        /* set i2 for the next loop */
  m6=1024;
  modify(i6,m6);        /* set i6 for the next loop */
view:   NOP;
        RTS;
.endmod;
```

Compress.dsp. Inizializza le variabili `BIT_FILE_rack` e `BIT_FILE_mask`, quindi chiama la *routine* `compfile`, che comprime i dati; infine chiama la *routine* `clobtfil`, che chiude il *file* se l'evento è l'ultimo della serie di eventi consecutivi da comprimere.

```
.module/ram COMPRESS;
.include <constant.k>;
.entry compress;
compress:
  ar=0;
  dm(BIT_FILE_rack)=ar;
  ar=0x8000;
  dm(BIT_FILE_mask)=ar;
  call compfile;
  ar=dm(numero_di_eventi_compressi);
  af=PASS ar;
  ar=N_EV_COM;
  ar=ar-af;
  if gt jump noclose;
  call clobtfil;
noclose:
    rts;
.endmod;
```

Compfile.dsp. Chiama la *routine* `inittree`, che inizializza l'albero all'inizio di ogni serie di eventi da comprimere consecutivamente. Poi, se il dato non rientra tra quelli comprimibili con Huffmann, codifica il simbolo speciale `ENDSTREA` e quindi codifica il simbolo in maniera non compressa. Se il simbolo è codificabile con Huffmann viene chiamata la *routine* `encdsymb`, che codifica il simbolo e quindi la *routine* `updatemd`, che aggiorna l'albero.

```
.module/ram COMPFILE;
.include <constant.k>;
.entry compfile;
compfile:
  ar=dm(numero_di_eventi_compressi);
  af=PASS ar;
  if ne jump noinitt1;
```

```
  ar=0;
  dm(BIT_FILE_rack)=ar;
  ar=0x8000;
  dm(BIT_FILE_mask)=ar;
  call inittree;
noinitt1:
  ar=dm(numero_di_eventi_compressi);
  ar=ar+1;
  dm(numero_di_eventi_compressi)=ar;
  af=PASS ar;
  ar=N_EV_COM;
  ar=ar-af;
  if ge jump noinit2;
  ar=0;
  dm(BIT_FILE_rack)=ar;
  ar=0x8000;
  dm(BIT_FILE_mask)=ar;
  call inittree;
noinit2:
  i0=^buffer;
  i1=i0;
  cntr=TOTSTRIP;
  do encode until ce;
  ax0=dm(i1,m1);        /* ax0=datum to be encoded */
  af= PASS ax0;
  if lt jump reroute;
  ar=VALMAX;
  ar=af-ar;
  if gt jump reroute;
  jump noreroute;
reroute:
  dm(Temp3)=ax0;        /* in Temp3 there is the anomalous datum */
  ax0=ENDSTREA;         /* encoding of ENDSTREA */
  call encdsymb;
  ay0=dm(Temp3);        /* transission of the anomalous datum */
  ay1=16;               /* signed datum is always 16 bit long */
  call outptbts;
  jump encode;
noreroute:
  call encdsymb;
  call updatemd;
encode:
  nop;
    rts;
.endmod;
```

Inittree.dsp. Inizializza l'albero di Huffmann in modo che contenga soltanto i due simboli `ENDSTREA` ed `ESCAPE`. Inoltre scrive in tutte le foglie il valore "-1".

```
.MODULE/ram INITTREE;
.include <constant.k>;
.entry inittree;
inittree:
    i7=^nodochild;
    ar=ROOTNODE+1;
    pm(i7,m5)=ar;     /* assuming ROOTNODE=0, if we don't need any change  */
    ar=ENDSTREA;
    pm(i7,m5)=ar;
    ar=ESCAPE;
    pm(i7,m5)=ar;
    i7=^nodochild_is_foglia;
    ar=FALSE;
    dm(i7,m5)=ar;     /* assuming ROOTNODE=0, if we don't need any change */
    ar=TRUE;
    dm(i7,m5)=ar;
    dm(i7,m5)=ar;
    i7=^nodoweight;
    ar=2;
    pm(i7,m5)=ar;     /* assuming ROOTNODE=0, if we don't need any change */
    ar=1;
    pm(i7,m5)=ar;
    pm(i7,m5)=ar;
    i7=^nodoparent;
    ar=-1;
    pm(i7,m5)=ar;     /* assuming ROOTNODE=0, if we don't need any change */
    ar=ROOTNODE;
    pm(i7,m5)=ar;
```

```
    pm(i7,m5)=ar;
    ar=ROOTNODE+3;
    dm(next_free_nodo)=ar;
    i7=^foglia;
    ar=ENDSTREA;
    m6=ar;
    ar=ROOTNODE + 1;
    modify(i7,m6);
    pm(i7,m4)=ar;
    i7=^foglia;
    ar=ESCAPE;
    m6=ar;
    ar=ROOTNODE + 2;
    modify(i7,m6);
    pm(i7,m4)=ar;
    i7=^foglia;
    ar=ENDSTREA;
    cntr=ar;
    ar=-1;
    do initialize until ce;
    pm(i7,m5)=ar;
initialize:
    nop;
  RTS;

.endmod;
```

Encdsymb.dsp. Codifica il simbolo contenuto in `ax0`. Se il simbolo è incontrato per la prima volta viene codificato il simbolo `ESCAPE` e poi il simbolo stesso in maniera non compressa, infine viene chiamata la *routine* `addnodo`, che aggiunge la foglia relativa al nuovo simbolo all'albero. Se il simbolo non è incontrato per la prima volta viene codificato in maniera compressa.

```
.module/ram ENCDSYMB;
.include <constant.k>;
.entry encdsymb;

encdsymb:
  ay0=0;                    /* ay0=code */
  ay1=0;                    /* ay1=code_size */
  sr0=1;                    /* sr0=current_bit */

  i7=^foglia;
  m6=ax0;
  modify(i7,m6);
  ax1=pm(i7,m4);
  ar=ax1+1;
  if ne jump noescape;      /* if leaf(c)=-1 send ESCAPE */
  i7=^foglia;
  ar=ESCAPE;
  m6=ar;
  modify(i7,m6);
  ax1=pm(i7,m4);            /* now ax1=ESCAPE */
noescape:
  ar=ROOTNODE;
  af=PASS ar;
  ar=ax1-af;
  if eq jump out_of_find_root_loop;
find_root:
  ar=1;
  af = PASS ar;
  ar=ax1 AND af;
  if ne jump bit_EQ_0;
  ar= sr0 OR ay0;
  ay0=ar;
bit_EQ_0:
  sr=lshift sr0 by 1 (lo);
  ar=ay1+1;
  ay1=ar;
  i7=^nodoparent;
  m6=ax1;
  modify(i7,m6);
  ax1=pm(i7,m4);
/* end of while cycle to find code */
```

```
  ar=ROOTNODE;
  af=PASS ar;
  ar=ax1-af;
  if ne jump find_root;
out_of_find_root_loop:
  call outptbts;
  i7=^foglia;
  m6=ax0;
  modify(i7,m6);
  ar=-1;
  af=PASS ar;
  ar=pm(i7,m4);
  ar=ar-af;
  if ne jump fine;
  ay0=ax0;                  /* code=symbol */
  ay1=NBITS;                /* code_size=size of symbol */
  call outptbts;
  call addnodo;
fine:
  nop;
    rts;
  .endmod;
```

Outptbts.dsp. Accumula i codici nella variabile `BIT_FILE_rack` e, quando questa è piena, la trasmette.

```
.module/ram OUTPTBTS;
.include <constant.k>;
.entry outptbts;
outptbts:
    ar=ay1-1;               /* ar=code_size-1 */
    si=1;
    sr0=0;
    sr1=0;
    se=ar;
    sr=lshift si (LO);      /* sr=mask */
whileloop:
    ar=PASS sr0;
    if eq jump no_loop;
    ar=sr0 AND ay0;
    if eq jump no_do;
    ar=dm(BIT_FILE_mask);
    af=PASS ar;
    ar=dm(BIT_FILE_rack);
    ar=ar OR af;            /* ar=BIT_FILE_rack;  af=BIT_FILE_mask */
    dm(BIT_FILE_rack)=ar;
no_do:
    dm(Temp)=sr0;
    sr0=0;
    sr1=0;
    si=dm(BIT_FILE_mask);
    sr=lshift si by -1 (LO);
    dm(BIT_FILE_mask)=sr0;
    ar=PASS sr0;
    if ne jump notransmit; /* if rack is not full do not transmit */
    dm(Temp2)=ax0;
    ar=dm(BIT_FILE_rack);
    dm(TrasmDato)=ar;
    call transmit;
    ax0=dm(Temp2);
    ar=0;
    dm(BIT_FILE_rack)=ar;
    ar=0x8000;
    dm(BIT_FILE_mask)=ar;
notransmit:
    nop;
    sr0=0;
    sr1=0;
    sr0=dm(Temp);
    sr=lshift sr0 by -1 (LO);
    jump whileloop;         /* end of while loop */
no_loop:
    nop;
    rts;
.endmod;
```

Updatemd.dsp. Aggiunge 1 al peso del nodo relativo al simbolo codificato e fa la stessa cosa ai nodi che incontra risalendo l'albero fino al nodo `ROOT`. Poi controlla che l'albero rispetti la regola della fratellanza e, se questo non avviene, aggiusta l'albero scambiando i nodi mediante la *routine* `swapnodo`.

```
.module/ram UPDATEMD;
.include <constant.k>;
.entry updatemd;
updatemd:
    i7=^foglia;
    m6=ax0;
    modify(i7,m6);
    ay0=pm(i7,m4);
    ar=ay0+1;
    if eq jump out_out_of_while_loop;
loop1:
    i7=^nodoweight;
    m6=ay0;
    modify(i7,m6);
    ar=pm(i7,m4);
    ar=ar+1;
    pm(i7,m4)=ar;
    ay1=ay0;                    /* new_node = current_node */
    ar=ROOTNODE;
    ar=ar-ay1;
    if ge jump out_of_for_loop1;
loopbis:
    ar=ay1-1;
    i7=^nodoweight;
    m6=ar;
    modify(i7,m6);
    ar=pm(i7,m4);
    af= PASS ar;
    i7=^nodoweight;
    m6=ay0;
    modify(i7,m6);
    ar=pm(i7,m4);
    ar=af-ar;
    if ge jump out_of_for_loop1;   /* break */
/*end of for loopbis */
    af=PASS ay1;
    ar=ay1-1;
    ay1=ar;
    ar=ROOTNODE;
    ar=ar-af;
    if lt jump loopbis;        /* if(new_node > ROOT_NODE) jump loopbis */
out_of_for_loop1:
    nop;
    ar=ay1;
    ar=ar-ay0;
    if eq jump noswap;
    call swapnodo;
    ay0=ay1;
noswap:
    nop;
    i7=^nodoparent;
    m6=ay0;
    modify(i7,m6);
    ay0=pm(i7,m4);
/* end of while loop */
    ar=-1;
    ar=ay0-ar;
    if ne jump loop1;
out_out_of_while_loop:
    rts;
.endmod;
```

Swapnodo.dsp. Scambia il nodo, il cui numero si trova nel registro `ay0`, con un altro nodo, il cui numero si trova nel registro `ay1`.

```
.Module/ram SWAPNODO;
.include <constant.k>;
```

```
.entry swapnodo;
swapnodo:
  i7=^nodochild_is_foglia;
  m6=ay0;
  modify(i7,m6);
  ar=dm(i7,m4);       /* ar=current_nodo_child_is_foglia  */
  af=PASS ar;
  i7=^nodochild;
  m6=ay0;
  modify(i7,m6);
  ar=PASS af;
  ar=pm(i7,m4);       /* ar=current_nodo_child */
  if eq jump else_1;  /* if referred  current_nodo_child_is_foglia */
  i7=^foglia;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ay1;      /* (current_nodo_child)_foglia=new_nodo */
  jump if_2;
else_1:
  i7=^nodoparent;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ay1;      /* (current_nodo_child)_nodo_parent=new_nodo */
  ar=ar+1;
  i7=^nodoparent;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ay1;      /* (current_nodo_child+1)_nodo_parent=new_nodo */
if_2:
  i7=^nodochild_is_foglia;
  m6=ay1;
  modify(i7,m6);
  ar=dm(i7,m4);       /* ar=new_nodo_child_is_foglia  */
  i7=^nodochild;
  af=PASS ar;
  m6=ay1;
  modify(i7,m6);
  ar=PASS af;
  ar=pm(i7,m4);       /* ar=new_nodo_child */
  if eq jump else_2;
  i7=^foglia;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ay0;      /* (new_nodo_child)_foglia=current_nodo */
  jump swap;
else_2:
  i7=^nodoparent;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ay0;      /* (new_nodo_child)_nodo_parent=current_nodo */
  ar=ar+1;
  i7=^nodoparent;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ay0;      /* (new_nodo_child+1)_nodo_parent=current_nodo */
swap:
/*
         swaps current node with new node
         notice that parents of the two nodes are not swapped
*/
  i7=^nodochild_is_foglia;
  m6=ay0;
  i5=i7;
  modify(i7,m6);
  ar=dm(i7,m4);
  dm(Temp)=ar;        /* current_nodo_child_is_foglia in temp */
  m6=ay1;
  modify(i5,m6);
  ar=dm(i5,m4);
  dm(i7,m4)=ar;       /* current_nodo_child_is_foglia is put in
                                 new_nodo_child_is_foglia */
  ar=dm(Temp);
  dm(i5,m4)=ar;       /* new_nodo_child_is_foglia is put in
                          current_nodo_child_is_foglia */
  i7=^nodochild;
  m6=ay0;
```

```
  i5=i7;
  modify(i7,m6);
  ar=pm(i7,m4);
  dm(Temp)=ar;        /* current_nodo_child in temp */
  m6=ay1;
  modify(i5,m6);
  ar=pm(i5,m4);
  pm(i7,m4)=ar;       /* current_nodo_child is put in new_nodo_child */
  ar=dm(Temp);
  pm(i5,m4)=ar;       /* new_nodo_child is put in current_nodo_child */
  i7=^nodoweight;
  m6=ay0;
  i5=i7;
  modify(i7,m6);
  ar=pm(i7,m4);
  dm(Temp)=ar;        /* current_nodo_weight in temp */
  m6=ay1;
  modify(i5,m6);
  ar=pm(i5,m4);
  pm(i7,m4)=ar;       /* current_nodo_weight is put in new_nodo_weight */
  ar=dm(Temp);
  pm(i5,m4)=ar;       /* new_nodo_weight is put in current_nodo_weight */
    rts;
  .endmod;
```

Addnodo.dsp. Aggiunge un nuovo nodo all'albero, al quale è collegata la foglia relativa al nuovo simbolo (contenuto nel registro `ax0`). Questo viene fatto dividendo in due il nodo più leggero dell'albero (ovvero quello con il numero più alto). Il nuovo nodo è ora il fratello di quello più leggero. Il peso del nuovo nodo è zero: è compito della *routine* `updatemd` aggiornare i pesi e sistemare l'albero.

```
.Module/ram ADDNODO;
.include <constant.k>;
.entry addnodo;
addnodo:
  ax1=dm(next_free_nodo);      /* ax1=new nodo */
  ar=ax1+2;
  dm(next_free_nodo)=ar;
  ar=ax1-1;                    /* ar=lightest_nodo */
  i7=^nodochild_is_foglia;
  i5=i7;
  m6=ar;
  modify(i7,m6);
  m6=ax1;
  modify(i5,m6);
  ar=dm(i7,m4);
  dm(i5,m4)=ar;
  ar=ax1-1;                    /* ar=lightest_nodo */
  i7=^nodochild;
  i5=i7;
  m6=ar;
  modify(i7,m6);
  m6=ax1;
  modify(i5,m6);
  ar=pm(i7,m4);
  pm(i5,m4)=ar;
/* tree->foglia[ tree->nodos[ new_nodo ].child ] = new_nodo */
  i7=^foglia;
  m6=ar;
  modify(i7,m6);
  pm(i7,m6)=ax1;
  ar=ax1-1;                    /* ar=lightest_nodo */
  i7=^nodoweight;
  i5=i7;
  m6=ar;
  modify(i7,m6);
  m6=ax1;
  modify(i5,m6);
  ar=pm(i7,m4);
  pm(i5,m4)=ar;
/*
```

```
    tree->nodes[ new_node ].parent = lightest_node;
*/
  i7=^nodoparent;
  m6=ax1;
  modify(i7,m6);
  ar=ax1-1;                     /* ar=lightest_nodo */
  pm(i7,m4)=ar;
 /* tree->nodos[ lightest_nodo ].child = new_nodo; */
  ar=ax1-1;                     /* ar=lightest_nodo */
  i7=^nodochild;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ax1;
 /* tree->nodos[ lightest_nodo ].child_is_foglia = FALSE; */
  ar=ax1-1;                     /* ar=lightest_nodo */
  i7=^nodochild_is_foglia;
  m6=ar;
  modify(i7,m6);
  ar=FALSE;
  dm(i7,m4)=ar;
/*
    tree->nodes[ zero_weight_node ].child = c;
*/
  ar=ax1+1;                     /* ar=zero_weight_nodo */
  i7=^nodochild;
  m6=ar;
  modify(i7,m6);
  pm(i7,m4)=ax0;
/*
    tree->nodes[ zero_weight_node ].child_is_leaf = TRUE;
*/
  ar=ax1+1;                     /* ar=zero_weight_nodo */
  i7=^nodochild_is_foglia;
  m6=ar;
  modify(i7,m6);
  ar=TRUE;
  dm(i7,m4)=ar;
/*
    tree->nodes[ zero_weight_node ].parent = lightest_node;
*/
  ar=ax1+1;                     /* ar=zero_weight_nodo */
  i7=^nodoparent;
  m6=ar;
  modify(i7,m6);
  ar=ax1-1;                     /* ar=lightest_nodo */
  pm(i7,m4)=ar;
/*
    tree->nodes[ zero_weight_node ].weight = 0;
*/
  ar=ax1+1;                     /* ar=zero_weight_nodo */
  i7=^nodoweight;
  m6=ar;
  modify(i7,m6);
  ar=0;
  pm(i7,m4)=ar;
/*
    tree->leaf[ c ] = zero_weight_node;
*/
  i7=^foglia;
  m6=ax0;
  modify(i7,m6);
  ar=ax1+1;                     /* ar=zero_weight_nodo */
  pm(i7,m4)=ar;
  rts;
  .endmod;
```

Clobtfil.dsp. Poiché la routine `outptbts` trasmette il contenuto della variabile `BIT_FILE_rack` solo quando questa è piena, è possibile che alla fine di una serie di compressioni la variabile sia riempita soltanto in parte. In questo caso la *routine* `outptbts` provvede a trametterla.

```
.module/ram CLOBTFIL;
.include <constant.k>;
.entry clobtfil;
```

```
clobtfil:
    ar=dm(BIT_FILE_mask);
    ar=ar-0x8000;
    if eq jump nofill;
    ar=dm(BIT_FILE_rack);
    dm(TrasmDato)=ar;
    call transmit;
    ar=0;
    dm(numero_di_eventi_compressi)=ar;
nofill:
    nop;
    rts;
.endmod;
```