Sistemi programmabili per telecomunicazioni

Alberto Tibaldi

16 novembre 2010

Indice

Capitolo 1

Metodi formali per il mapping da algorimi ad architetture

L'obiettivo di questo capitolo è quello di introdurre una rappresentazione formale iniziale in grado di, dato un algoritmo, rappresentarlo e poter applicare sulla rappresentazione così ottenuta una metodologia in grado di tradurla in un'architettura digitale.

Al fine di fare ciò, considereremo ora un caso pratico, che verrà dunque in seguito spiegato meglio (in quanto argomento della trattazione); nel mentre nelle varie sottosezioni saranno introdotti alcuni parametri fondamentali per la caratterizzazione delle architetture digitali.

1.1 Analisi di un filtro FIR - latenza e throughput

Si consideri per esempio, al fine di introdurre tutte le grandezze e le metodologie per determinarle, un filtro FIR, ossia un dispositivo in grado, dato in ingresso un certo insieme di campioni $x[n]$, di produrne in uscita altri, $y[n]$, filtrati. Un possibile filtro FIR è un dispositivo che realizza la somma pesata in una certa finestra temporale; supponendo per semplicità che questa finestra nel nostro caso sia ampia 3 (nella pratica si fanno almeno ampie 16 o 32), si avrà una funzione di questo tipo:

$$y[n] = ax[n] + bx[n-1] + cx[n-2]$$

Si hanno tre campioni successivi, ossia il campione n e i due precedenti a esso, si combinano linearmente mediante i tre pesi a, b, c, e il risultato di tutto ciò è $y[n]$.

Per realizzare un sistema in grado di fare ciò servono tre tipi di oggetti:

- sommatori;
- moltiplicatori;
- elementi in grado di mantenere in memoria i due campioni precedenti; supponiamo che siano dei registri.

Un modo di realizzare ciò è il seguente:
seguendo l'operazione passo passo, si vede che si può fare un qualcosa del genere. Una prima analisi della struttura ci permette di vedere che essa è **scalabile**: questo è un filtro con finestra temporale larga 3, ma aggiungendo altri rami analogamente all'ultimo si può ottenere un numero indefinito di elementi.

Questa è l'unica struttura possibile? È la migliore? Sotto quale punto di vista? Beh, questa struttura di sicuro è molto parallelizzata: con un solo colpo di clock si calcola tutto, istantaneamente (si tornerà su ciò); si ha molta velocità, ma anche molte risorse impiegate, molto hardware: per avere molta velocità di solito si deve avere a che fare con soluzioni anche molto dispendiose come questa, che ha molti elementi uguali tra loro; questo filtro sarà veloce ma molto impegnativo da realizzare. La stessa cosa si può fare in altre maniere?

Per ora preoccupiamoci di quantificare le prestazioni di questo filtro, e ciò si può fare mediante due parametri: la latenza L e il throughput T_h.

- La latenza si definisce come il ritardo complessivo che si ha tra il calcolo del risultato e l'istante in cui è disponibile l'ingresso; in altre parole, dato l'ingresso, la latenza è il tempo che si impiega per avere disponibile l'uscita (il ritardo input/output).
- Il throughput è il numero di campioni che si riesce a produrre in una certa unità di tempo.

Si cerchi di capire un concetto molto importante: questi due concetti sono ben distinti, e spesso pure uno scorrelato dall'altro. La latenza è un ritardo, dunque un tempo; il throughput è il numero di campioni che si produce rapportato all'unità di tempo, dunque è un t^{-1}. Cerchiamo di capire meglio cosa siano, con un paio di esempi semplificati rispetto al filtro FIR. Si consideri di dover realizzare qualcosa del genere:

$$y = x_1 + x_2 + x_3 + x_4$$

Ciò si può semplicemente fare mediante la seguente struttura:

A questo punto, se conosco il ritardo del sommatore T_a (in buona approssimazione di solito possono essere $T_\mathrm{a} \sim 2$ ns), il ritardo complessivo è dato dal tempo impiegato dal segnale per arrivare dall'input all'output: introdotto l'ingresso, quanto ci vuole per averlo in uscita. In questo caso, questo tempo è $3T_\mathrm{a}$:

$$L = 3T_\mathrm{a}$$

Il throughput T_h è, **in questo caso**, $T_\mathrm{h} = \frac{1}{3T_\mathrm{a}}$: da $t = 0$ introduco i quattro ingressi, dunque solo dopo $3T_\mathrm{a}$ ho l'uscita e posso iniziare un'altra operazione di somma.

Si può usare un'architettura alternativa:

Cosa è cambiato? Beh, in questa maniera al primo colpo di clock si fa in modo da registrare la somma $x_1 + x_2$, al secondo somma x_3 a ciò, al terzo x_4. In altre parole:

$$L = 3T_\mathrm{CK}$$

dove T_CK è un periodo di clock, da noi impostato (a vedremo quali condizioni). Di solito si parla di frequenza di clock:

$$f_\mathrm{CK} = T_\mathrm{CK}^{-1}$$

Il periodo di clock si valuta nel caso peggiore; vedremo che, da registro a registro, il tempo può essere qualcosa del tipo:

$$T_\mathrm{CK} \sim T_\mathrm{a}$$

Giustificheremo il $\sim$. Questo ci dice che il throughput è:

$$T_\mathrm{h} = \frac{1}{T_\mathrm{a}}$$

Come mai? Beh, si supponga di considerare una condizione *a regime*, ossia supponendo che il sistema sia stato acceso già da un tempo sufficiente da aver caricato almeno una volta tutti i registri in maniera corretta, e che sia correttamente funzionante; il fatto di aver introdotto questi registri porta ad un aumento del throughput, nel senso che se da un lato la latenza rimane la stessa (quando si introduce un certo ingresso il risultato relativo a quell'ingresso arriva comunque dopo una latenza di $3T_\mathrm{a}$ circa), si hanno molte più uscite: a ogni colpo di clock escono i risultati relativi a un'altra operazione. Nel precedente non era così (quello senza registri), dal momento

che per iniziare a calcolare ciascun risultato è necessario che quello prima sia stato presentato.

Questa seconda implementazione ha un throughput maggiore: se la devo fare continuamente, allora è più utile; se invece l'operazione si fa ogni tanto, può andar bene anche quella standard: il fatto di avere throughput aumentato migliora semplicemente il caso in cui il sommatore funzioni continuamente.

Una volta definiti questi due parametri, possiamo tornare al filtro di prima: si hanno due oggetti hardware (registri a parte, e se ne parlerà dopo), dunque due ritardi, T_a (per l'adder, sommatore), T_m (per il moltiplicatore, multiplier). Per quanto riguarda la latenza, come già detto è il tempo che passa dal momento in cui applico un nuovo ingresso a quello in cui ho una nuova uscita.

Ancora alcune note sulla latenza: nel caso della struttura che stiamo studiando, la latenza è uguale al periodo di clock; questo fatto è dovuto al fatto che vi è un percorso che collega direttamente ingresso e uscita, con solo logica combinatoria in mezzo; nel caso vi siano solo percorsi con registri in mezzo, si identifichi il percorso con meno registri in mezzo, supponendo che il numero di registri nel suddetto sia N; si avrà

$$L = (N+1)T_\mathrm{CK}$$

si ricorda che non è stato ancora detto niente su come calcolare T_CK.

Dato il sistema in studio, si possono identificare tre percorsi:

A questo punto una nota: i percorsi interessanti, per il calcolo di T_CK, sono sostanzialmente quelli di questo tipo:

- quelli da ingresso a uscita;
- quelli dall'ingresso a un registro;
- quelli da un registro a un altro registro;
- quelli da un registro all'uscita.

Possiamo dunque identificare tre percorsi:

$$P_3 = T_\mathrm{m} + P_\mathrm{a}$$

$$P_2 = P_1 = T_\mathrm{m} + 2T_\mathrm{a}$$

questi ultimi due sono i *critical path*:

$$L = T_{\mathrm{m}} + 2T_{\mathrm{a}}$$

Per quanto riguarda il throughput, ovviamente io posso acquisire un nuovo campione solo dopo che tutti i segnali si sono propagati. T_{CK} non può sicuramente essere più corto di L: se usassimo un clock più lento di questo, i segnali non farebbero in tempo a propagarsi completamente, dunque le uscite non sarebbero stabili. Si ha:

$$T_{\mathrm{h}} = \frac{1}{T_{\mathrm{CK}}} = \frac{1}{T_{\mathrm{m}} + 2T_{\mathrm{a}}}$$

Una soluzione alternativa potrebbe essere la seguente:

Come prima cosa si procurano i tre prodotti parziali di $x[n]$ per i vari coefficienti, poi si fanno "invecchiare"; spieghiamolo per bene: al primo colpo di clock si fa la somma tra il primo prodotto parziale e quello invecchiato mediante il registro R_1; usciti da R_1 si ha un altro registro, R_2, che fa invecchiare ulteriormente il valore di uno. In altre parole, il passaggio nel registro "invecchia", nel senso che mantiene il valore, per un colpo di clock, e lo ripresenta al colpo successivo. Vediamo i passi (si supponga sempre di essere in *regime*:

1. al primo colpo di clock si fa la somma tra il primo prodotto parziale (quello per b), e quello del colpo precedente (quello per c); fuori dal R_1 avrò dunque $cx[n-1]$, che passa per il sommatore insieme a $bx[n]$, ottenendo fuori da esso e in ingresso a R_2 il termine $bx[n] + cx[n-1]$;

2. al secondo colpo, il termine $ax[n]$ viene sommato al termine in uscita da R_2, che sarà il termine del punto 1, ritardato di 1:

$$y[n] = ax[n] + bx[n-1] + cx[n-2]$$

Il risultato è uguale a prima, dunque l'algoritmo implementato è il medesimo. Dal punto di vista del costo implementativo non vi sono differenze: le risorse hardware impiegate sono le stesse. L'unica differenza, che però si andrà a rispecchiare anche nelle prestazioni, è quella architetturale. Vediamo che percorsi ci sono ora:

- da ingresso a uscita;
- da ingresso a R_2;
- da ingresso a R_1;

- da R_1 a R_2;
- da R_2 a uscita.

Di questi percorsi, due hanno moltiplicatore e sommatore: questi (nell'ipotesi che tutti i moltiplicatori e i sommatori abbiano gli stessi ritardi) saranno dunque il caso peggiore: $T_\mathrm{m} + T_\mathrm{a}$. Il throughput sarà:

$$T_\mathrm{h} = \frac{1}{T_\mathrm{m} + T_\mathrm{a}}$$

Definizione/calcolo di T_CK

Precedentemente è stato scritto che $T_\mathrm{CK} \sim T_\mathrm{a}$, nell'esercizio con due sommatori. In questo caso, quello dell'architettura alternativa, verrebbe da dire:

$$T_\mathrm{CK} = T_\mathrm{m} + T_\mathrm{a}$$

Cosa significa? Cos'è T_CK? Cosa significa il $\sim$?

In un circuito **sincrono**, si tiene conto, in un'analisi dettagliata, anche dei registri: spesso la situazione è la seguente:

Qual è la f_max, ossia la massima frequenza di funzionamento di un sistema di questo tipo? Beh, questa sarà quella associata al minimo T_CK utilizzabile; noi imposteremo la frequenza di clock, e dunque con essa il periodo di clock, in maniera tale da garantire che il sistema funzioni nella maniera corretta. Analizziamo i vari contributi di cui si deve tenere conto:

- di solito, il contributo dominante è quello relativo alla logica combinatoria: $T_\mathrm{combinatorio}$ o T_lc; non è possibile applicare un clock più veloce del tempo impiegato per percorrere il critical path, ossia il percorso più pesante sotto il punto di vista dell'attraversamento di blocchi combinatori; se non si rispetta questa regola, il flip-flop va a campionare un segnale che non ha ancora attraversato tutta la *nuvoletta*; questo è il contributo dominante ma non è l'unico;
- T_CKQ: ciascun flip-flop impiega un certo tempo per memorizzare un valore; questo è come si chiama;
- T_SU, o **tempo di setup**: si tratta di un margine di sicurezza che si va a prendere: il dato deve essere presentabile all'ingresso del flip-flop per un certo tempo, al fine di essere correttamente campionato; al fine di comprendere meglio questo concetto, si immagini di attraversare una

porta automatica: non vogliamo che questa si chiuda nel momento in cui passiamo, per prenderci in mezzo, ma dopo qualche secondo che siamo usciti, in maniera da completare correttamente e senza problemi il passaggio; vale lo stesso discorso qui;

- T_{w}: ritardo dei **wire** (ossia delle interconnessioni); più è lunga l'interconnessione, più lentamente il segnale si propaga su esso.

Nella maggioranza dei casi, il tempo combinatorio predomina; più generalmente:

$$T_{\mathrm{CK}} \geq T_{\mathrm{lc}} + T_{\mathrm{SU}} + T_{\mathrm{CKQ}} + T_{\mathrm{w}}$$

Le rappresentazioni finora utilizzate per passare da un algoritmo a un cenno di architettura, ossia per la rappresentazione dell'algoritmo, si chiama DFG: Data Flow Graph. Si tratta di qualcosa abbastanza simile a un diagramma a blocco, tenendo conto che i componenti hanno una forma "elettronica". Gli archi che vanno da un nodo a un altro sono il flusso dei dati che viaggiano, mentre ciascun nodo rappresenta le operazioni che vengono svolte al suo interno (somma, sottrazione..).

Si consderi il seguente esempio, rappresentante un filtro recursivo:

$$y[n] = ax[n] + by[n-1]$$

questo si può rappresentare mediante il seguente DFG:

Una notazione spesso utilizzata nell'ambito del signal processing è quella del SFG (Signal Flow Graph), che rappresenterebbe qualcosa del genere:

in questo caso gli elementi ritardanti vengono indicati con z^{-1}, ossia come l'elemento di ritardo definito secondo la trasformata $\mathcal{Z}$.

Si provi a questo punto a quantificare le prestazioni di questo circuito, calcolando le varie grandezze:

$$T_{\mathrm{CK}} = T_{\mathrm{a}} + T_{\mathrm{m}}$$

e questo vale per i tre percorsi uguali.

Il sistema in questione è un sistema SISO: dato un singolo ingresso, per un colpo di clock si ottiene in uscita un singolo valore. Questo ci dice che:

$$T_{\mathrm{S}} = T_{\mathrm{CK}}$$

dunque

$$f_{\mathrm{S}} = T_{\mathrm{h}} \leq \frac{1}{T_{\mathrm{S}}}$$

Dal momento che si ha un collegamento diretto tra ingresso e uscita, la latenza sarà uguale a un singolo periodo di clock.

1.2 Loop bound e iteration bound

Al fine di poter quantificare le prestazioni del nostro sistema, è necessario definire alcune altre grandezze: il loop-bound T_{lb} e l'iteration bound T_{∞}; si parla di ciò adesso perchè, come si può vedere, nell'esempio proposto si ha un loop; queste grandezze vanno calcolate solo per DFG nei quali vi siano uno o più loop.

- Per quanto riguarda il loop-bound, esso è dato dal rapporto di due quantità: al numeratore si ha il ritardo combinatorio del loop, come ritardo combinatorio complessivo del percorso d'anello che si sta considerando; si consideri un certo anello, e si rimuovano per un istante tutti gli elementi con memoria, sostituendoli con un semplice pezzo di filo; sommando tutti i ritardi introdotti dagli elementi, si ricava il T_{lb}; questo rappresenta, per il loop considerato, il miglior tempo di clock raggiungibile: $T_{\text{ck}} = T_{\text{lb}}$ è infatti la soluzione ottimale della disposizione dei registri (ribadisco, per un singolo anello): se si riesce a distribuire i registri in maniera tale per cui la distanza temporale tra ciascun registro e un altro è uguale, ho massimizzato le prestazioni del circuito. Al numeratore si introduce il numero di elementi di ritardo (ossia il numero di registri): è un limite inferiore per il tempo di clock relativo al loop.

- In generale, in un circuito, sono presenti diversi loop; T_{∞} rappresenta, dato un circuito con molti loop, il maggiore di tutti i loop-bound: esso sarà il limite inferiore, in un DFG, di tutti i tempi di clock raggiungibili:

$$T_{\infty} \triangleq \max\{T_{\text{lb,j}}\}$$

  Si vuole ribadire l'idea secondo cui questo è un limite inferiore: non è detto che esso sia raggiungibile mediante modifiche del circuito, ma è sicuro che il tempo di clock non potrà essere inferiore a esso.

Si consideri a questo punto un altro esempio:

$$y[n] = ax[n]by[n-2]$$

In questo caso, si ha:

$$T_{\mathrm{CK}} = T_{\mathrm{m}} + T_{\mathrm{a}}$$

dunque

$$f_{\mathrm{S}} \leq \frac{1}{T_{\mathrm{m}} + T_{\mathrm{a}}}$$

ma, in questo caso, si può vedere che, data la presenza dei due registri nel loop, l'iteration bound si dimezza: stessi elementi combinatori, ma più elementi sequenziali:

$$T_{\infty} = \frac{T_{\mathrm{m}} + T_{\mathrm{a}}}{2}$$

Un modo per migliorare il circuito potrebbe essere il seguente:

Realizzando un loop di questo tipo, il percorso critico non è più $T_{\mathrm{m}}+T_{\mathrm{a}}$: in questa maniera ho "suddiviso i compiti" su due cicli di clock; io avrei dovuto comunque avere due cicli di clock per quel loop, ma in questa maniera non ho più necessità di averlo troppo corto. Il percorso ingresso-uscita rimane uguale, ma poi questo potrà essere messo a posto, come vedremo, con il pipelining.

1.2.1 LPM: Longest Path Matrix

Si consideri ora la spiegazione di questo metodo, accoppiata con un esempio pratico. Si definisca in un grafo il numero d come il numero complessivo di elementi di ritardo; come esempio, si consideri il seguente:

Su ciascun elemento sono stati scritti i tempi di ritardo che esso introduce. Per questo grafo, $d = 4$: in esso infatti sono presenti quattro elementi di ritardo.

Cosa si deve fare, per portare a termine questo metodo? Beh, prima di tutto, il primo passo è calcolare gli elementi di una matrice $\underline{L}^{(1)}$ di dimensione $d \times d$ (nel nostro caso 4×4) dove ciascun elemento $l_{ij}^{(1)}$ rappresenta il peggiore ritardo dal registro i-esimo al registro j-esimo, calcolabile nel caso il percorso sia completabile in un singolo colpo di clock (ossia da un registro a un altro senza possibilità di passare da altri registri). Si analizzi, come esempio, il seguente:

$$l_{11}^{(1)} = -1$$

infatti, non è possibile, in un colpo di clock, partire dal registro 1 e tornare al registro 1: è almeno necessario impiegare due colpi di clock, poichè bisogna passare dal registro 2. Visto che in questo caso questo percorso non esiste, si

utilizza un numero negativo, -1, come identificativo: ovviamente -1 non ha senso come ritardo: il ritardo deve essere un numero positivo.

$$l_{12}^{(1)} = 0$$

infatti, vi è una connessione diretta tra i registri 1 e 2. Verso 3 e 4, poi, non sarà possibile la connessione con un colpo di clock solo.

Completando con lo stesso ragionamento tutto, si può vedere che questa matrice vale:

$$\underline{L}^{(1)} = \begin{bmatrix} -1 & 0 & -1 & -1 \\ 4 & -1 & 0 & -1 \\ 5 & -1 & -1 & 0 \\ 5 & -1 & -1 & -1 \end{bmatrix}$$

Fatto ciò, si continua con una seconda matrice: $\underline{L}^{(2)}$: essa è la matrice degli elementi raggiungibili mediante 2 colpi di clock (non 1, non 3); allo stesso modo di prima, ragionando, si può per esempio vedere che in due colpi di clock il registro 1 può raggiungere sè stesso, o il registro 3; 2 è troppo vicino, 4 troppo lontano. Ragionando così, si ricava:

$$\underline{L}^{(2)} = \begin{bmatrix} 4 & -1 & 0 & -1 \\ 5 & 4 & -1 & 0 \\ 5 & 5 & -1 & -1 \\ -1 & 5 & -1 & -1 \end{bmatrix}$$

Si continua così fino a raggiungere la matrice $\underline{L}^{(d)}$.

Si può immaginare che più si va avanti coi conti, più essi diventano "rognosi". Esiste in realtà un metodo alternativo, per il calcolo delle matrici sopra quella di ordine 1 (che va invece calcolata per ispezione): conoscendo le matrici precedenti, e conoscendo le proprietà additive dei ritardi, si può pensare di fare qualcosa di questo genere:

Un percorso percorribile in 3 colpi di clock, per esempio, potrebbe essere diviso in due percorsi, uno che richiede un colpo di clock, uno che ne richiede 2. Questo significa che, sommando opportuni coefficienti delle matrici, è possibile ricavare un coefficiente di un'altra matrice. Si immagini per esempio, per la matrice 3, di avere qualcosa del tipo:

$$l_{ij}^{(3)} = l_{ik}^{(1)} + l_{kj}^{(2)}$$

è possibile che sussistano più percorsi; ovviamente, il peggiore sarà quello da considerare:

$$\Longrightarrow l_{ij}^{(3)} = \max_i \left\{ l_{ik}^{(1)} + l_{kj}^{(2)} \right\}$$

Si consideri per esempio di dover calcolare $l_{11}^{(3)}$; come si può fare, con questo metodo? Vediamo: da $\underline{L}^{(1)}$, dobbiamo partire dal primo registro e tornare al primo registro; dalla matrice $\underline{L}^{(1)}$, dunque, dobbiamo considerare tutti i possibili elementi della prima riga: l'unico accettabile è il secondo elemento, con tempo 0; abbiamo trovato per ora dunque che il primo termine sarà quello con $k = 2$: dal primo registro si va verso il secondo. Giunti al secondo registro, dobbiamo tornare al primo, e ciò in questo caso è possibile: con ritardo 5. Si avrà, nello specifico caso dell'esercizio:

$$l_{11}^{(3)} = l_{12}^{(1)} + l_{21}^{(2)} = 0 + 5 = 5$$

In questo caso si aveva una sola possibilità; nel caso le possibilità sono molteplici, vanno calcolate **tutte**, e scelta la peggiore.

Si può dimostrare che le matrici sono:

$$\underline{L}^{(3)} = \begin{bmatrix} 5 & 4 & -1 & 0 \\ 8 & 5 & 4 & -1 \\ 9 & 5 & 5 & -1 \\ 9 & -1 & 5 & -1 \end{bmatrix}$$

$$\underline{L}^{(4)} = \begin{bmatrix} 8 & 5 & 4 & -1 \\ 9 & -8 & 5 & 4 \\ 10 & 9 & 5 & 5 \\ 10 & 9 & -1 & 5 \end{bmatrix}$$

Per fare le varie operazioni, è consigliabile usare il più possibile le matrici di basso ordine: in questa maniera, posso avere meno percorsi aperti possibile, e i calcoli sono ridotti.

A questo punto, è necessario calcolare T_∞: per far ciò si considerino solo gli anelli chiusi, dunque gli l_{ii} (ossia gli elementi delle diagonali); gli elementi delle diagonali sono i ritardi lungo i percorsi ad anello. Si può immaginare intuitivamente che:

$$T_{\text{lb,i}} = \frac{l_{ii}^{(n)}}{n}$$

e quindi:

$$T_\infty = \max_i \left\{ \frac{l_{ii}^{(n)}}{n} \right\}$$

si può vedere facilmente che:

$$T_\infty = 2$$

Questo procedimento è efficiente con molti archi, ma quando si ha a che fare con pochi registri (in modo da avere poche matrici, e di ordine basso).

Si osservi che, per l'ultima matrice, è sufficiente calcolare gli elementi della diagonale: essi sono l'unica parte interessante, per essa.

1.3 Metodi generali per migliorare il throughput

1.3.1 Pipelining

Cosa si può fare per fare pipelining? Prima di tutto, ricordiamo cosa si intende per pipelining: si intende introdurre registri aggiuntivi rispetto a quello già presenti nel sistema, in maniera tuttavia da preservare il sincronismo dei segnali.

Riproponiamo il seguente esempio:

Come già visto, si ha

$$T_{\mathrm{CK}} = T_{\mathrm{S}} = T_{\mathrm{m}} + 2T_{\mathrm{a}}$$

L'idea del pipelining è collocare registri aggiuntivi in modo da ottenere percorsi più corti; andando a cercare quali sono i percorsi critici, se si spezzano essi diventano "meno critici", e così le prestazioni (in termini di frequenza di clock accettabile) del circuito sono più alte. Si supponga di fare qualcosa del genere:

Questa soluzione è errata: se si prova infatti a scriverne la funzione uscita-ingresso, si ottiene:

$$y[n] = ax[n-1] + bx[n-2] + cx[n-2]$$

La relazione temporale, di sincronismo, tra i campioni, non è più la stessa: il blocco moltiplicato per c non è stato più shiftato come gli altri due, dunque questo circuito non lavorerà più come prima.

Quello che si vuole fare a questo punto è proporre una prima tecnica per effettuare un'ottimizzazione del circuito, basata sul pipelining.

Cut set pipelining

A questo punto si vuole introdurre un metodo formale, sistematico, per la realizzazione del pipelining. Prima di introdurlo, tuttavia, sarà necessario introdurre una nozione di teoria dei grafi: si consideri un grafo generico:

Per cut set si intende un insieme di archi che, nel caso rimossi, dividono il grafo in due sottografi; non si hanno vincoli sulle dimensioni dei sottografi, e, per ora, sull'orientamento dei fili:

A questo punto, un passo successivo: per **feedforward cut set** si intende un insieme di taglio (cut set) tale per cui tutti gli archi che si vanno a rimuovere hanno la stessa direzione, ossia sono tutti diretti da o verso il sottografo.

In questo caso gli archi tagliati sono diretti dal sottografo sinistro al sottografo destro, dunque questo cut set è un feedforward cut set.

Date queste definizioni, è possibile introdurre il metodo di pipelining mediante studio del feedforward cut set: dato il grafo (il DFG) in questione, è possibile introdurre i registri lungo tutti gli archi che appartengono a un feedforward cut set, senza cambiare in nessuna maniera l'algoritmo. Questa cosa poi va fatta in maniera intelligente: conviene andare a cercare i "tagli" in maniera tale che coinvolgano i percorsi critici del sistema, in modo tale da ridurli e rendere migliori le prestazioni del sistema: questo è l'obiettivo del pipelining.

1.3.2 Retiming

Una volta discusso il pipelining, si vuole introdurre un secondo metodo; in realtà si parla più che altro di un insieme di metodi, di procedure di ottimizzazione: riposizionamento dei registri in un grafo, al fine di ridurre i critical path.

Si consideri un generico nodo, numerato (n):

Il retiming del nodo (n) è il numero di elementi di ritardo che noi spostiamo dagli archi entranti agli archi uscenti dal nodo. Si consideri qualcosa del genere:

questo è ciò che capita, se $r(n) = 1$: dire che $r(n)$ (ossia che il retiming del nodo (n)) è uguale a 1 significa che rimuovo uno degli elementi di ritardo da ciascuno degli archi entranti nel nodo, e ne metto uno in ciascuno degli archi uscenti. Dualmente, se avessi scritto $r(n) = -1$, si sarebbe dovuto togliere un elemento di ritardo da ciascuno degli archi di uscita, e metterne uno in ciascuno degli archi di ingresso. Attuando retiming, non si altera l'algoritmo.

Ovviamente, la condizione alla quale il retiming può essere fatto è non lasciare in nessun ramo un "numero negativo" di elementi di ritardo: dire

che c'è un ritardo negativo significa che ci dovrebbe essere una sorta di "anticipatore", ossia un qualcosa in grado di fornire il valore di qualcosa prima ancora di introdurre ingressi; questo sarebbe anticausale, dunque insensato.

Con il retiming, è possibile:

- minimizzare il numero totale di registri: data la proprietà prima detto, volendo, è possibile usarla in modo intelligente per ridurre il numero di registri utilizzati;
- ottimizzare le prestazioni, in casi in cui il pipelining applicato sui cut set feedforward non è applicabile; caso tipico di questa seconda cosa sono i sistemi con dei loop, nei quali non vi sono tagli feedforward (in un loop ci sarà sempre un percorso in direzione "opposta").

A questo punto, abbiamo introdotto la possibilità di fare retiming; il problema è: come applicarlo? Come si può trovare la soluzione "giusta", in grado di ottimizzare quindi le prestazioni del sistema? Esiste una metodologia generale, che verrà ora introdotta.

Si supponga di aver a che fare con due nodi di un grafo, u e v, collegati da un arco e:

Come peso $w(e)$ dell'arco si definisce il numero di registri presenti nell'arco e.

Adottando il retiming, sarà possibile scegliere un certo retiming per i nodi u e v; questo farà cambiare il peso del nodo e, dal momento che applicando il retiming si toglieranno/metteranno nuovi elementi di ritardo nella rete. Applicato il retiming, si avrà un nuovo $w_{\mathrm{r}}(e)$, che varrà:

$$w_{\mathrm{r}}(e) = w(e) + r(u) - r(v) \geq 0$$

Questo peso deve essere come sempre maggiore o uguale a 0: se così non fosse, si perderebbe la realizzabilità fisica del circuito (dal momento che si avrebbero di nuovo degli anticipatori). Questa è una condizione necessaria, ma non sufficiente, al fine di migliorare le prestazioni.

Questo discorso è stato applicato su un generico percorso e, senza specificare che esso faccia parte di un effettivo percorso critico, ossia di uno di quei percorsi che vincola il tempo di clock minimo del sistema; nel caso questo arco fosse parte del percorso critico, si impone, anzichè la relazione appena proposta, una relazione più stringente:

$$w_{\mathrm{r}}(e) \geq 1$$

In questa maniera, si chiede che lungo l'arco appartenente al percorso critico vi sia almeno un registro, in modo da spezzare il percorso critico e renderlo più breve.

$$\begin{cases} w_{\mathrm{r}}(e) \geq 0, & \text{arco appartenente a percorso non critico} \\ w_{\mathrm{r}}(e) \geq 1, & \text{arco appartenente a percorso critico} \end{cases}$$

Cut set retiming

Si vuole a questo punto proporre un metodo meno formale del precedente per effettuare il retiming, un poco più semplice. Si consideri il metodo applicato sul seguente esempio:

Si consideri, per questo grafo, un certo cut set, senza che esso sia per forza feedforward. A seconda della direzione di questi archi rispetto ai due sottografi che si vengono a formare, si potrà agire come descritto tra breve. Si scelga, come cut set, quello tale per cui si abbia un grafo formato dal nodo 2, e uno dai nodi 1, 3, 4. Gli archi appartenenti a questo cut set sono discordi, dal momento che e_{21} va dal grafo G_1 al grafo G_2, e viceversa gli altri vanno da G_2 a G_1. Si può dimostrare che il comportamento dell'algoritmo non subisce variazioni se si sommano K elementi agli archi direzionati da G_1 a G_2 e se si sottraggono K elementi su quelli da G_2 a G_1. K può essere, al solito, un numero intero positivo o negativo. Se si introducesse $K = -1$, si otterrebbe qualcosa di questo tipo:

Gli archi non appartenenti al cut set non subirebbero variazioni, mentre quelli del cut set sarebbero modificati come in figura.

Il risultato appena applicato permette di introdurre dunque un metodo per l'ottimizzazione del circuito mediante retiming, ma ha un problema: bisogna scegliere il cut set in modo appropriato, ossia in modo che il retiming porti benefici. Sarà necessario fare in modo che il cut set incroci il cammino critico (o i cammini critici, nel caso ve ne sia più di uno).

1.3.3 Unfolding (o "loop unrolling")

Si vuole a questo punto introdurre un'altra tecnica atta a migliorare le prestazioni del circuito. Sebbene il nome della tecnica contiene la parola "loop", questa tecnica di fatto è applicabile a sistemi di qualsiasi tipo. Qual è l'idea di questa tecnica? Sostanzialmente, data un'architettura, aumentarne la parallelizzazione, ossa ottenere un'architettura in grado di realizzare la stessa funzione, elaborando però più campioni alla volta invece di uno solo. Al fine di comprendere questa teoria, la si introduce applicata direttamente a un esempio.

Si consideri un filtro di questo tipo:

$$y[n] = ay[n-9] + x[n]$$

il DFG di questo sistema sarà il seguente:

Le prestazioni di questo circuito, come noto dalla teoria precedentemente introdotta, sono quantificabili mediante i seguenti parametri:

$$T_{\mathrm{cp}} = T_{\mathrm{a}} + T_{\mathrm{m}} = T_{\mathrm{CK}} = T_{\mathrm{S}}$$

Dal momento che si elabora un campione per volta, si può dire che i tempi di clock e di campionamento sono coincidenti. Analizzando le prestazioni assolute raggiungibili:

$$T_{\infty} = \frac{T_{\mathrm{a}} + T_{\mathrm{m}}}{9} \ll T_{\mathrm{CK}}$$

Ossia, esiste la possibilità di andare 9 volte più velocemente rispetto a quanto si ha finora.

Come si può operare? Un modo per migliorare un poco le prestazioni è basato sul retiming: si potrebbe spostare uno dei nove registri in modo da spezzare il critical path. In questa maniera, essendo il tempo della moltiplicazione il maggiore, si avrebbe

$$T_{\mathrm{CK}} = T_{\mathrm{m}}$$

Abbiamo migliorato la cosa ma non radicalmente: si può andare molto più velocemente.

Una seconda idea, sempre collegata al retiming, sarebbe quella di suddividere le operazioni di moltiplicazione e di addizione in operazioni basilari più semplici, in maniera tale che, mettendo i registri tra le varie sotto-operazioni, si potrebbe ridurre ulteriormente il tempo di clock: una sorta di "pipelining a grana fine".

La nostra idea è ancora diversa: invece di effettuare un pipelining, ciò che vogliamo fare è parallelizzare le operazioni, ossia introdurre più campioni per volta, e farli elaborare tutti assieme. Proviamo prima di tutto ad applicare in maniera rudimentale il metodo, per poi trovare una formulazione più sistematica. Si supponga di suddividere il sistema prima rappresentato in due sistemi, dove per esempio si considerano ingressi e uscite "pari" e "dispari". Si avrà a che fare con due espressioni nuove: $y[2k]$ per gli elementi pari, $y[2k+1]$ per quelli dispari. Ciò significa, sostituendo banalmente nell'espressione di partenza del filtro $n \to 2k$ e $n \to 2k+1$, che dovremo ricavare in formato DFG le seguenti espressioni:

$$y[2k] = ay[2k-9] + x[2k]$$

$$y[2k+1] = ay[2k-8] + x[2k+1]$$

A questo punto si deve realizzare un'architettura in grado di soddisfare questo set di equazioni, e non più le due precedenti. Per fare ciò, riscriviamo rapidamente le equazioni nella seguente maniera, più idonea:

$$y[2k] = ay[2k-9] + x[2k] = ay[2(k-5)+1] + x[2k]$$

$$y[2k+1] = ay[2k-8] + x[2k+1] = ay[2(k-4)] + x[2k+1]$$

Come mai questo passaggio? Beh, per evidenziare alcune osservazioni, che permetteranno di capire meglio il problema in questione: quando passa un colpo di clock, a differenza di prima, verranno processati non uno, ma due valori per volta; dovendo dunque introdurre elementi di ritardo nella rete, non dovremo metterne 8 o 9 come la prima coppia di equazioni vuole far vedere, bensì 4 e 5: in 4 colpi di clock infatti tendenzialmente saranno processati cinque valori, con 5 dieci valori. Come mai si è fatta dunque questa modifica "algebrica" ? Per mettere in evidenza il fatto che per costruire l'uscita "pari" sarà necessario prendere l'uscita "dispari" e ritardarla, nella fattispecie di 5 colpi di clock: il fatto di avere, nella parentesi quadra, qualcosa del tipo $2() + 1$ fa capire che questa cosa non si possa costruire se non dall'uscita dispari, con un opportuno ritardo (di 5 unità di tempo). Stesso discorso per l'uscita dispari, ottenuta ritardando di 4 colpi quella pari. Ciò permette di disegnare il seguente DFG:

Questa è un'operazione di unfolding: un singolo loop è stato "srotolato", ottenendo un loop con J "parallelizzazioni".

Prima di passare a una formalizzazione di questo metodo, alcune note.

- Il numero di registri totale è lo stesso di prima, e sarà sempre così, a meno che non si faccia anche del retiming; questo perchè il numero di registri è una proprietà dell'algoritmo: esso è collegato alla quantità di memoria che si deve utilizzare per realizzare l'algoritmo.

- Che prestazioni sono state ottenute? Vediamo, coi soliti metodi:

$$T_{\mathrm{cp}} = T_{\mathrm{a}} + T_{\mathrm{m}} = T_{\mathrm{CK}}$$

Ma, si faccia bene attenzione, ora $T_{\mathrm{CK}} \neq T_{\mathrm{S}}$: ora infatti non si processa più un valore per volta, ma due per volta; si avrà dunque:

$$f_{\mathrm{S}} = \frac{1}{T_{\mathrm{S}}} = \frac{2}{T_{\mathrm{CK}}} = \frac{2}{T_{\mathrm{a}} + T_{\mathrm{m}}}$$

ossia, in questo specifico caso, f_{S} è il rapporto tra il numero di rami paralleli e il periodo di clock. Si noti che questa **non è una regola generale**: in questo caso aumentando le parallelizzazioni abbiamo ottenuto il doppio della frequenza di sampling, ma generalmente non è vero. Un motivo che dimostra ciò è l'esistenza di T_∞: al di sotto di esso non è possibile arrivare.

- I vantaggi derivanti da questa operazione di parallelizzazione non sono gratuiti: per poter applicare il metodo abbiamo di fatto raddoppiato le risorse di calcolo rispetto a quelle che avevamo precedentemente: abbiamo il doppio di sommatori e di moltiplicatori.

Si propone a questo punto una formulazione generale del metodo dell'unfolding.

Dato un DFG non parallelo, si vuole applicare un procedimento tale da aumentare il parallelismo fino a J; per fare ciò è necessario operare in due passi.

1. per ogni nodo u appartenente al DFG, disegnare J nodi di tipo u, ossia con lo stesso nome, numerandoli progressivamente da 0 a $J-1$, nel grafo DFG'.

2. Completare DFG' introducendo archi e ritardi sugli archi. Per ogni arco nel grafo di partenza, identificare i nodi di partenza u e i nodi di arrivo v applicando la seguente regola: l'arco che va dalla replica i-esima del nodo di partenza (u_i) alla j-esima del nodo di arrivo (v_j), con peso dell'arco w_j, devono essere scelte in modo che:

$$\begin{cases} j = (i+w)\%J \\ w_j = \lfloor (i+w)/J \rfloor \end{cases}$$

Per questo metodo, esistono alcune considerazioni aggiuntive, che permettono di ridurre i calcoli a semplici ispezioni, per casi particolari:

- per tutti gli archi tali per cui $w = 0$, si ha che:

$$\begin{cases} j = i \\ w_j = 0 \end{cases}$$

in altre parole, si collegano tra loro i nodi con lo stesso pedice, e il peso dell'arco è 0;

- se $w = J$, si ha che:

$$\begin{cases} j = i \\ w_j = 1 \end{cases}$$

ossia, i pesi si distribuiscono equamente;

- se $w = KJ$, ossia se il peso dell'arco è multiplo del parallelismo J, si ha che:

$$\begin{cases} j = i \\ w_j = \frac{w}{J} \end{cases}$$

1.3.4 Folding (time sharing)

Si vuole a questo punto proporre un'ultima tecnica per l'ottimizzazione del circuito, tecnica duale all'unfolding: anzichè "srotolare" il circuito, lo si "ripiega", in maniera da utilizzare un numero ridotto di risorse per realizzare circuiti dove vi sono molte risorse che fanno, in occasioni diverse, la stessa operazione. Si consideri per esempio un circuito di questo tipo:

In questo circuito si hanno due elementi che fanno la stessa operazione, ossia la somma. Ciò che si potrebbe fare è la stessa cosa, utilizzando meno risorse: utilizzare un unico sommatore che iterativamente serva le due operazioni che si vogliono realizzare. La soluzione sarebbe la seguente:

Si introducono degli interruttori, i quali hanno due posizioni legate a due situazioni:

1. in questa situazione si portano al sommatore $a[n]$ e $b[n]$;
2. in questa situazione si portano al sommatore $c[n]$ e $a[n] + b[n]$, il quale è memorizzato nel registro.

A questo punto, si vuole introdurre la teoria in modo un po' più generale. Si consideri la seguente macchina:

U e V sono due operazioni, e un arco, $w(e)$ il numero di registri nell'arco che collega i due nodi. Si deve associare a U, operazione, un H_U, ossia un

elemento fisico in grado di realizzare l'operazione U e altre operazioni uguali a essa (nel caso di prima, per esempio U sarebbe una delle due somme, e H_U sarebbe un sommatore). Dal momento che si vuole considerare un caso generale, si considera la presenza di una seconda operazione V (potrebbe, in questo caso, essere per esempio una moltiplicazione, giusto per avere qualcosa di diverso rispetto a prima), e il blocco che serve l'operazione V H_V. Si ha a che fare con sostanzialmente due problematiche da risolvere:

- decidere, per ciascuna situazione, quando e come posso collegare H_U e H_V: risolvere dunque in maniera generale il problema di dove mettere lo switch (si tratta dunque sostanzialmente di un problema di scheduling: come assegnare gli ingressi giusti dello switch);
- stabilire quanti elementi di ritardo si devono posizionare su ciascun arco.

A questo punto, esprimiamo il procedimento nel modo più generale possibile: nel DFG che si intende rappresentare, si allocano tanti nodi quanti sono gli strumenti necessari per compiere tutte le operazioni (nel caso che ci interessa, due nodi: H_U e H_V). Fatto ciò, si stabilisce un asse dei tempi, ossia un asse discreto nel quale si fa variare l'indice temporale n. Fare folding significa sostanzialmente dividere ciascun tempo di campionamento in L sottointervalli, dove per saltare da ciascun sottointervallo al successivo è necessario che passi un T_{CK}. Definita dunque la base tempi a partire dal termine di folding L (ossia il termine in cui foldiamo la nostra struttura), si dovran definire, in maniera abbastanza arbitraria, gli istanti per cui il blocco H_U andrà a realizzare l'operazione U, e quelli per cui il blocco H_V realizzeranno V: chiamiamo questi istanti rispettivamente u e v. Decidere per esempio $u = 0$ equivale a dire che al primo colpo di clock H_U eseguirà U; dire che dunque $v = 2$ significa che al terzo colpo di clock H_V eseguirà V. Al termine di ciascun ciclo, lungo L colpi di clock, si avrà lo start da capo, ossia da 0. Solitamente si considerano gli ingressi sincronizzati con 0.

Un suggerimento da tenere presente è quello di numerare sequenzialmente i valori degli istanti con il progredire del flusso dei dati: se non fosse così, bisognerebbe aumentare la latenza fino a fare in modo che il dato si mantenga correttamente nel circuito.

Al fine di completare il metodo, dunque, si vuole proporre la soluzione all'ultimo problema: la determinazione del numero di registri da posizionare su ciascun arco, w_{f}. Qual è l'obiettivo di questi registri? Beh, sostanzialmente, quello di preservare la corretta sincronizzazione dei dati da un blocco a un altro. Ciò che si deve fare è tenere conto sia dei ritardi definiti tra un blocco

e un altro al momento di performare le operazioni richieste, sia dei registri intrinsicamente già presenti, i quali dovranno mantenere il loro ruolo, ossia mantenere il dato per un intero ciclo (e non per un solo colpo di clock!). La formula sarà:

$$w_{\mathrm{f}} = [v + L(n + w) - (u + nL)]$$

Questa è la differenza tra i tempi in cui il dato è disponibile all'ingresso del nodo H_V e quello in cui il dato è disponibile all'uscita del nodo sorgente H_U. Semplificando:

$$w_{\mathrm{f}} = v - uwL$$

1.3.5 Resource sharing / decomposition

Si introdurrà in maniera rapida un'ultima tecnica applicabile su qualsiasi architettura, per certi versi simile al time sharing. L'idea sostanzialmente è: dato un grafo, in cui a un certo punto si ha un certo nodo F, dividerlo in sotto-nodi più semplici, f, g, h:

Ciò potrebbe ricordare un pipelining a grano fine; in pratica, nel caso del resource sharing ciò si fa al fine di allocare un'unica risorsa hardware che deve realizzare tutte le operazioni; questo sarà tendenzialmente più complicato dei blocchi precedenti, ma i tempi come vedremo subiranno variazioni. Un esempio di far ciò potrebbe riguardare la somma: data una somma a più addendi, che normalmente si dovrebbe realizzare introducendo più sommatori, quello che si può fare è realizzare un singolo blocco in grado di sommare autonomamente più addendi tra loro (sommatore a più operandi).

1.3.6 Confronto delle trasformazioni universali

Si vuole a questo punto proporre un confronto dei metodi noti come "trasformazioni universali" introdotti, al fine di capire come muoversi in determinate situazioni.

Consideriamo a questo punto alcune osservazioni:

- applicando il metodo del pipelining, ci si muove idealmente verso sinistra: idealmente infatti non aumenta la complessità hardware del sistema, però il throughput migliora;
- applicando il metodo dell'unfolding, si introduce sostanzialmente una parallelizzazione del sistema: ciò fa aumentare l'hardware presente nella macchina, ma d'altra parte fa anche migliorare il throughput;

- applicando il metodo del folding, di sicuro si riduce l'hardware, però d'altra parte si aumentano i tempi, riducendo il throughput;

- applicando la decomposition, capita qualcosa di duale rispetto al pipelining: di fatto la complessità hardware diminuisce, dal momento che si riducono le risorse atte a realizzare una certa operazione; di fatto si può dimostrare in modo intuitivo che però il throughput rimane circa uguale (idealmente parlando); si ha infatti che, dal momento che da un'unità di calcolo se ne ottengono 3 (per esempio, seguendo la figura di prima) che vengono riunite in una sola, $T'_{\mathrm{CK}} = \frac{1}{3} T_{\mathrm{CK}}$, dunque il clock è linearmente velocizzato; d'altra parte il numero di campioni per colpo di clock si riduce dello stesso fattore di cui si fa la decomposition, dal momento che a ogni colpo di clock si produrrà solo uno dei campioni con l'operazione decomposta, dunque in sostanza:

  $$T''_{\mathrm{h}} = \frac{1}{T_{\mathrm{a}}} \frac{1}{3} = \frac{1}{T_{\mathrm{CK}}}$$

  rimane così.

Si è detto che queste tecniche sono universali; questo dal momento che esse non dipendono dall'applicazione in cui vanno utilizzate: qualsiasi algoritmo può essere di fatto trattato con queste operazioni.

Ciò che si può fare tuttavia è cercare alcune altre tecniche, in grado, per quanto in applicazioni più specifiche, di migliorare ulteriormente le prestazioni.

1.3.7 Osservazioni generali - introduzione di tecniche particolari

Proprietà associativa

Una prima cosa può essere sfruttare una proprietà di tipo associativo. Si consideri qualcosa di questo tipo:

$$y = x + a + b$$

In una situazione come questa, se non siamo in un loop, la soluzione banale per migliorare il throughput sarebbe quella di utilizzare il pipelining; questo ramo, però, potrebbe essere parte di un loop, la cui funzione potrebbe per esempio essere quella di aggiornare un nuovo valore di x. Ciò che si può fare è utilizzare la proprietà associativa: calcolare a parte $a+b$, e dunque solo

con un altro sommatore aggiungere il risultato della somma a x, ottenendo qualcosa del genere:

In questa maniera, abbiamo portato il sommatore fuori dal loop, e dunque all'interno del loop si ha solo una somma.

Proprietà distributiva

Si può fare un discorso simile al precedente per quanto concerne la proprietà distributiva: se ho un'operazione del tipo

$$y = d + c(a + b)$$

come prima, nel caso in cui siamo in un ramo feedforward, tutto tranquillo; nel caso il ramo sia in un loop, bisogna prendere le nozioni con le pinze. Ciò che si può fare è scrivere y come:

$$y = d + ac + bc$$

In questa maniera, si ottiene banalmente una cosa del tipo:

Cosa significa? Beh, la parte fuori dal loop ora è pipelineabile, e dunque abbiamo portato fuori dal loop, con un altro gioco, un percorso critico, riducendo la complicazione di ciò che è all'interno di esso.

Trasformazione da grafo lineare ad albero

Un esempio classico in cui si fa una trasformazione da grafo lineare ad albero è quello del calcolo del minimo di un certo insieme di componenti:

$$y[k] = \min_i \{x_i[k]\}$$

Per fare ciò, senza pensare molto, cosa si deve fare? Sostanzialmente, prendere un valore per volta, e costruire un circuito che, volta per volta, calcoli la differenza dei due valori e trovi quello più piccolo dei due, per ogni combinazione. Il ritardo di una struttura di questo tipo aumenta linearmente con il numero di elementi: è circa $N - 1$ volte il tempo dell'operazione. Un modo più elaborato ma sicuramente interessante per fare ciò è il seguente:

In questo caso, il ritardo è proporzionale al solo $\log_2(N)$.

Look-ahead

Consideriamo a questo punto un'altra tecnica non generale; questa tecnica si presta piuttosto bene ad essere applicata ad un filtro IIR, come per esempio:

$$y[n] = x[n] + ay[n-1]$$

In questo caso, si ha che:

$$T_{\text{cp}} = T_{\text{CK}} = T_{\text{a}} + T_{\text{m}}$$

dunque, il throughput è:

$$T_{\text{h}} = \frac{1}{T_{\text{a}} + T_{\text{m}}}$$

Quanto vale T_∞? Beh, per ispezione:

$$T_\infty = \frac{T_{\text{a}} + T_{\text{m}}}{1}$$

Questo fatto sembrerebbe dirci che non è possibile migliorare le prestazioni, in termini di velocità, del circuito. Ciò non è vero: questo fatto ci dice che non è possibile migliorare **con tecniche universali** le prestazioni del circuito, ma non in assoluto: usando tecniche specifiche, è di fatto possibile far di meglio. Vediamo come.

Scriviamo l'espressione per $n+1$ del nostro filtro, "guardando avanti", come il nome della tecnica suggerisce:

$$y[n+1] = x[n+1] + ay[n]$$

ma cosa abbiamo? $y[n]$ è nota da prima, e dunque, sostituendovi l'espressione precedente:

$$y[n+1] = x[n+1] + ax[n] + ay[n-1]$$

ora, nessuno ci vieta di riscrivere quest'ultima espressione sostituendo $n \to n-1$:

$$y[n] = x[n] + ax[n-1] + ay[n-2]$$

disegnando dunque il DFG di questa cosa, si ha:
ora si ha

$$T_{\text{cp}} = T_{\text{m}} + 2T_{\text{a}}$$

Ma si noti che questo grafo contiene percorsi di tipo feedforward, dunque si possono fare due cuts; per quanto riguarda il loop, invece, si ha, dopo l'applicazione dell'idea look-ahead:

$$T_\infty = \frac{T_\mathrm{a} + T_\mathrm{m}}{2}$$

dunque, T_∞ è diminuito, e ora posso usare altre tecniche, quali il retiming o l'unfolding, per ridurre i critical paths. Dopo tutte queste operazioni, posso ottenere qualcosa di questo tipo:

A questo punto, con pipeline di grano fino, si può tendere al limite asintotico. Il look-ahead si può applicare ovviamente più di una volta, e così migliorare ulteriormente le prestazioni.

Si noti che questa tecnica è finalizzata all'incremento della velocità; ha ovviamente portato all'aumento delle risorse hardware, ma può essere usata, quando possibile, per velocizzare il sistema.

Questa tecnica non è universale dal momento che se l'algoritmo è non lineare, la tecnica non fornisce benefici, e dunque è evitabile.

Interleaving (interlacciamento)

Si consideri a questo punto lo stesso esempio di prima, e vi si applichi questa idea. La presenza del loop ci impedisce di fare, come detto, pipelining; sarebbe tuttavia estremamente comodo fare qualcosa di questo genere:

A questo punto, la nostra idea è la seguente: $x[k]$ è il campione che immettiamo nel sistema con un certo tempo; se il tempo di campionamento del segnale rimane quello di prima, la presenza del nuovo registro cambia il sincronismo dei segnali, facendoci ottenere una funzione di uscita del tipo:

$$y[k] = x[k] + ay[k-2]$$

Questa funzione è formalmente diversa dalla precedente, dal momento che di fatto effettua un'operazione diversa. Come potremmo fare per far tornare a funzionare correttamente le cose? Beh, sostanzialmente, ciò che dovremmo fare è introdurre, campionare i segnali con un tempo più lento, nella fattispecie in questo caso con un tempo pari a 2 colpi di clock:

$$f_\mathrm{S} = \frac{1}{2} f_\mathrm{CK} \iff T_\mathrm{S} = 2T_\mathrm{CK}$$

In questa maniera, ciò che faccio è dilatare il tempo di campionamento, avendo però ridotto (tendenzialmente) il tempo di clock. Il sistema in questo modo torna a funzionare correttamente: è come se avessimo, di fatto, introdotto un delay (un registro) all'ingresso, ri-sincronizzando tutto.

Cosa è successo per ora? Beh, di fatto, il throughput non è migliorato: tutto ciò che abbiamo fatto è stato fare una cosa di questo genere:

Stiamo usando il nostro hardware solo per un certo tempo, lasciandolo a tutti gli effetti "libero" nel tempo successivo. Il throughput dunque non migliora.

Ciò che possiamo a questo punto fare per migliorare le prestazioni è utilizzare il sistema anche in queste "pause", nel quale per ora non fa niente. L'interleaving è una tecnica che permette di fare ciò. Si supponga di fare qualcosa del genere:

di suddividere lo stream di dati singolo in due sotto-stream: uno con indici pari, uno con indici dispari. Ciò che si fa a questo punto è immettere gli stream nel filtro, e in questa maniera ottenere un'operazione di fatto diversa da prima. Si noti però un fatto: quando si ha a che fare con qualcosa di questo genere, la sequenza originale ha una certa lunghezza, finita; una volta che essa termina, riprende da capo una nuova sequenza; si ha per esempio, terminata la prima sequenza $x'[0], x'[1], ..., x'[N-1]$, di N campioni, la sequenza $x''[0], ..., x''[N-1]$. Ciò che si fa dunque è mescolare i due blocchi di dati, facendo una combinazione del tipo:

$$x'[0], x''[0], x'[1], x''[1], ..., x'[N-1], x''[N-1]$$

introducendo prima un campione della prima lista, poi uno della seconda, e così via. Il filtro, in questa maniera, lavora contemporaneamente con gli elementi del primo e del secondo blocco di dati, in modo però ordinato, dal momento che si fa questo interlacciamento e che comunque il blocco SIPO agisce in modo da far dividere in pari e dispari (prima e seconda lista). Introdurre un registro diventa accettabile dal momento che è necessario spezzare i tempi.

Il fatto di fare ciò porta ad avere $2N$ colpi di clock per finire l'elaborazione, ma, dal momento che il filtro finisce per elaborare due blocchi di dati per volta, di fatto la velocità resta uguale, ma il clock può essere velocizzato dal momento che si piazzano i registri nei punti critici.

Come detto, si ha a che fare con due sequenze, con due finestre di campioni; trattarli però non è banale, dal momento che prima arrivano tutti i campioni della prima finestra, poi tutti i campioni della seconda. Al fine di poter effettuare questo interlacciamento, è necessario dunque immagazzinare in una memoria tutti i campioni della prima finestra, dunque non appena iniziano ad arrivare nello stream quelli della seconda, incominciare a processare assieme prima uno della prima finestra poi uno della seconda, e così via.

Uno schema di principio che può realizzare questa operazione è il seguente:

Si aspetta prima di tutto di riempire il buffer con i segnali della lista 1, x_1, dunque dopo si avrà il multiplexer che in una porta avrà il buffer, nell'altra

x_2; scegliendo con un'onda quadra prima un campione di una lista poi uno dell'altra, si può far lavorare il dispositivo in questa maniera.

Si noti che memorizzare i campioni è un'operazione dotata di una latenza: questo fatto va gestito in maniera intelligente, dal momento che bisogna scegliere una frequenza di campionamento nel buffer tendenzialmente più alta della frequenza che si utilizza per l'elaborazione: la frequenza di campionamento più alta compatibile con la velocità dello stream di dati.

1.4 Riduzione del consumo di potenza

Finora sono state fatte considerazioni per quanto concerne le caratteristiche dinamiche del sistema: a questo punto si vuole fare qualche considerazione sul consumo di potenza, a partire dal pipelining e dall'unfolding: utilizzando le tecniche già viste l'obeittivo dunque ora sarà quello di ridurre il consumo di potenza.

Ciò che ci manca per adesso è un metodo per la valutazione de consumo dell'architettura, nonchè un modello per la stima dei ritardi; come vedremo tra breve, purtroppo, prestazioni dinamiche e consumo sono due parametri in competizione, dunque si dovrà fare un trade-off dei due al fine di ottenere buone prestazioni sotto tutti i punti di vista.

1.4.1 Modello del consumo di potenza

Quando si ha a che fare con architetture CMOS, come la stragrande maggioranza dei casi, si può modellare il consumo in maniera piuttosto semplice; la potenza P dissipata da un gate sarà:

$$P_i = V_{\mathrm{DD}}^2 C_{\mathrm{L}} f$$

dove V_{DD} è la tensione di alimentazione alla quale si alimenta il gate, C_{L} è la capacità con la quale lo si carica, f è la frequenza di commutazione della porta logica. Combinando questi tre parametri in questa maniera è dunque possibile calcolare il consumo dinamico (non quello statico, il cui modello è più complicato; a prescindere da ciò, in realtà, non è importante avere un modello per il consumo statico, dal momento che esso non è migliorabile con le tecniche che abbiamo finora studiato).

Questo modello vale per una porta, la porta i come è stata da noi sopranominata. Volendo fare qualcosa di più complicato, ossia caratterizzare un intero sistema, cosa si può fare? Si consideri banalmente ciò:

$$P_{\text{tot}} \sum_i V_{\text{DD},i} C_{\text{L},i} f_i$$

Solitamente, tutti i gate sono alimentati con la stessa tensione di alimentazione, dunque è possibile portarla fuori dalla sommatoria. Si possono fare considerazioni anche per quanto concerne la frequenza f_i: se il sistema è regolato, come capita di solito, dallo stesso segnale di clock, si avrà qualcosa del tipo:

$$f_i \leq \alpha f_{\text{CK}}$$

dove α è un numero minore di 1, che dipende sostanzialmente dai dati che viaggiano nella porta: tanto più la porta è utilizzata, tanto più spesso commuterà, tanto più α sarà grande. Ciò che si fa di solito è definire un C'_{L} come:

$$C'_{\text{L}} \triangleq \alpha C_{\text{L}}$$

in questa maniera, la potenza diviene:

$$P_{\text{tot}} = \sum_i P_i = V_{\text{DD}}^2 f_{\text{CK}} \sum_i C'_{\text{L},i} = V_{\text{DD}}^2 f_{\text{CK}} C_{\text{tot}}$$

dove C_{tot} è una "cumulativa della capacità", ossia una somma pesata delle capacità, con peso α, il quale indica il fattore di attività della i-esima porta.

1.4.2 Modello di ritardo

Il modello in grado di tenere conti dei ritardi introdotti dal gate è di questo tipo:

$$T_{\text{d}} = \frac{C_{\text{d}} V_{\text{DD}}}{K \left(V_{\text{DD}} - V_{\text{t}}\right)^2}$$

dove V_{t} è la tensione di soglia del gate, K un parametro. Dato un circuito con un certo percorso critico, si ha un valore che dipende dunque da V_{DD}, da C_{d} (la quale **non** è la capacità totale), dato dalla sommatoria delle capacità solo lungo il percorso critico.

1.4.3 Metodi di ottimizzazione

Ottimizzazione mediante pipelining

Come è possibile ottimizzare, a questo punto, i suddetti parametri? Beh, si consideri l'introduzione del pipelining:

$$P_{\text{originale}} = V_{\text{DD}}^2 f_{\text{CK}} C$$

Questo, nella versione senza pipelining. Quando si passa alla versione con pipeline, si introducono dei registri nell'architettura; ciò non cambia la capacità totale C, dal momento che non si aumentano le risorse hardware del sistema (se non di un minimo, del tutto trascurabile). Ciò che varia sarà la frequenza di clock: se aumentiamo il numero di registri, si aumenta il throughput, ma dunque f_{CK}:

$$f_{\text{CK}} \to N f_{\text{CK}}$$

dove N è il numero di elementi di ritardo introdotti nel sistema. Questo ci porta a capire che, con il pipelining, il consumo di potenza aumenta linearmente con N. Volendo ridurre il consumo di potenza, posso ridurre la tensione di alimentazione V_{DD}, in maniera da ridurre quadraticamente anche la potenza; questo si fa riscalando con un fattore β la tensione di alimentazione; l'espressione risultante sarà:

$$P_{\text{pipeline}} = \beta^2 V_{\text{DD}}^2 N f_{\text{CK}} C$$

A questo punto, sulla base di cosa devo scegliere i parametri N e β? Beh, sulla base di cosa voglio ottenere, ma anche sulla base delle prestazioni di ritardo; per tenere conto di ciò, si presentano dunque i tempi di ritardo nei casi originale e pipelineato:

$$T_{\text{d,originale}} = \frac{C_{\text{d}} V_{\text{DD}}}{K \left(V_{\text{DD}} - V_{\text{t}}\right)^2}$$

$$T_{\text{d,pipeline}} = \frac{1}{N} \frac{C_{\text{d}} \beta V_{\text{DD}}}{K \left(\beta V_{\text{DD}} - V_{\text{t}}\right)^2}$$

Infatti, C_{d} risultante diminuisce, divisa per un fattore N, dal momento che, suddividendo con i registri i vari percorsi critici, anche le loro capacità equivalenti saranno "spezzettate".

Se ci interessa migliorare solo il ritardo, posso mettere $\beta = 1$, e modificare N a piacimento; volendo però migliorare le prestazioni di potenza, ossia di consumo, è necessario fare qualcosa di un poco più elaborato: sarebbe bello che le prestazioni dinamiche non peggiorino, ottimizzando la potenza; ciò porta a dire che il ritardo originale e quello "pipelineato" debbano essere uguali:

$$\frac{C_{\text{d}} V_{\text{DD}}}{K \left(V_{\text{DD}} - V_{\text{t}}\right)^2} = \frac{1}{N} \frac{C_{\text{d}} \beta V_{\text{DD}}}{K \left(\beta V_{\text{DD}} - V_{\text{t}}\right)^2}$$

semplificando:

$$\frac{1}{(V_{\mathrm{DD}} - V_{\mathrm{t}})^2} = \frac{\frac{1}{N}\beta}{(\beta V_{\mathrm{DD}} - V_{\mathrm{t}})^2}$$

Scelto un certo N, è possibile ottenere da queste equazioni un β. N potrà essere scelto sulla base di quanto vogliamo "pipelineare".

Ottimizzazione mediante parallelizzazione

Dopo il pipelining, proviamo ad applicare il secondo metodo che conosciamo per ottimizzare le prestazioni sotto il punto di vista del consumo: proviamo ad aumentare le prestazioni mediante l'introduzione di una parallelizzazione. Come già detto, vale la formula:

$$P_{\mathrm{originale}} = C_{\mathrm{tot}} V_{\mathrm{DD}}^2 f_{\mathrm{CK}}$$

Raddoppiando le unità di elaborazione, di fatto si raddoppiano i gate presenti nella macchina, dunque raddoppia anche la C_{tot}: aumentando il parallelismo, dunque, linearmente con esso aumenta anche la capacità. Cosa si può dire riguardo la frequenza di lavoro? Essa in teoria non cambia: di fatto, lavorando con la stessa frequenza di clock, si smaltiscono più campioni alla volta, ma la frequenza di clock non varia. Volendo ridurre il consumo, ciò che possiamo fare è introdurre una riduzione del tempo di clock: facendo in modo da riportare il throughput a quello che si avrebbe senza introdurre la parallelizzazione, si potrebbe fare ciò: data la frequenza f'_{CK}, il throughput è

$$T_{\mathrm{h}} = \frac{L}{T'_{\mathrm{CK}}} = L f'_{\mathrm{CK}}$$

f'_{CK} è semplicemente f_{CK}, opportunamente scalata. Al fine di avere il throughput uguale a prima, si scalerà di un divisore L:

$$f'_{\mathrm{CK}} = \frac{1}{L} f_{\mathrm{CK}}$$

in questo modo, il throughput è:

$$T_{\mathrm{h}} = \frac{L}{T'_{\mathrm{CK}}} = L f'_{\mathrm{CK}} = L \frac{1}{L} f_{\mathrm{CK}} = f_{\mathrm{CK}}$$

Applicando ora questa semplice relazione alla potenza, si ha:

$$P_{\mathrm{parallela}} = L C_{\mathrm{tot}} V_{\mathrm{DD}}^2 \frac{1}{L} f_{\mathrm{CK}} = P_{\mathrm{originale}}$$

Cosa abbiamo ottenuto? Per ora assolutamente niente: abbiamo preso una macchina, l'abbiamo parallelizzata, in modo da renderla molto più veloce, e dunque l'abbiamo fatta viaggiare lentamente, riducendo il clock; questa è come una Ferrari usata solo in prima: va piano e consuma molto. Ciò che si può fare è qualcosa di più intelligente, a questo punto: abbiamo solo riscalato la velocità, ma potremmo usare qualcosa che nella nostra metafora funga da "carburante low cost": la nostra necessità, a questo punto, è quella di ridurre il consumo: abbiamo accettato di perdere in velocità, non ha senso avere una macchina più grossa per consumare lo stesso. Possiamo a questo punto giocare ancora su un fattore, ossia la V_{DD}: scalando V_{DD} per il solito fattore α, è possibile migliorare le prestazioni del sistema sotto il punto di vista della potenza consumata. Quanto vale questo α ? Beh, si osservi la seguente cosa:

$$LT_{\mathrm{originale}} = parallela$$

Cosa significa ciò? Possiamo, nel sistema parallelo, accettare un ritardo molto più grande rispetto a quello sequenziale: tanto, di fatto, in un singolo colpo di clock possiamo elaborare molti più elementi. Quello che si fa parallelizzando è aumentare il numero di campioni elaborati, dunque, se permettiamo alla macchina di ridurre effettivamente la sua velocità non rispetto al colpo di clock, ma rispetto ai ritardi accettabili internamente (utilizzando il modello che lega i ritardi dei gate alle tensioni), possiamo ottenere grossi risultati: possiamo chiedere alla macchina di andare più lenta (ma non troppo: come vincolo possiamo usare quello appena presentato, in maniera da non ridurre i tempi di elaborazione rispetto al caso standard), guadagnandoci sotto il punto di vista della potenza consumata. Ricordo che:

$$T_{\mathrm{originale}} = \frac{C_{\mathrm{d}} V_{\mathrm{DD}}}{K(V_{\mathrm{DD}} - V_{\mathrm{t}})^2}$$

e

$$T_{\mathrm{parallela}} \sim \frac{C_{\mathrm{d}} V_{\mathrm{DD}} \alpha}{K(\alpha V_{\mathrm{DD}} - V_{\mathrm{t}})^2}$$

Nota: C_{d} non è stata moltiplicata per L, o per altri fattori: questo fatto dipende dal fatto che C_{d} non è la capacità locale, ma la capacità associata al percorso critico; per ora stiamo assumendo il fatto che questa non cambi con la parallelizzazione; in realtà non è detto, dal momento che si potrebbero formare percorsi a capacità più elevata, ma questo va sostanzialmente visto caso per caso. Dunque:

$$L\frac{C_{\mathrm{d}}V_{\mathrm{DD}}}{K(V_{\mathrm{DD}}-V_{\mathrm{t}})^2}=\frac{C_{\mathrm{d}}V_{\mathrm{DD}}\alpha}{K(\alpha V_{\mathrm{DD}}-V_{\mathrm{t}})^2}$$

Risolvendo questa equazione è possibile trovare il valore α di riscalamento della tensione che si ottiene.

Capitolo 2

Filtri numerici

2.1 Introduzione

I filtri sono sistemi tempo-discreti, LTI: data una sequenza di ingresso $x[n]$, una sequenza di uscita $y[n]$, si ha una trasformazione $T(.)$ dotata sostanzialmente di due proprietà:

1. linearità (ossia diritto di utilizzare il principio della sovrapposizione degli effetti): un segnale si può scrivere dunque come:

$$x[n] = \sum_k x[k]\delta[n-k]$$

se il sistema è lineare, si può scrivere che:

$$y[n] = T\{x[n]\} = \sum_k T\{x[k]\delta[n-k]\} =$$

$$= \sum_k x[k]T\{\delta[n-k]\}$$

ossia, è come sostanzialmente applicare al sistema $T(.)$ una δ, ossia un impulso; ciò che ci importerà del sistema sarà sostanzialmente la "risposta all'impulso":

$$= \sum_k x[k]h_k[n]$$

dove $h_k[n]$ è la risposta all'impulso introdotto all'istante k mostrato all'istante n.

2. invarianza temporale: dato un sistema che risponde a una δ con una risposta ad impulso h, ho:

$$\delta[n] \longrightarrow h[n]$$

ma dunque, se applico uno shift all'ingresso, avrò lo stesso shift anche sull'uscita:

$$\delta[n-k] = h[n-k]$$

posso dunque scrivere:

$$y[n] = \sum_k x[k]h_k[n] = x[n]h[n-k]$$

questa è anche la definizione della convoluzione discreta.

Il sistema ha anche altre proprietà: dal momento che esso deve essere stabile, esso deve anche essere limitato:

$$\left|\sum_k x[k]h[n-k]\right| < \infty \forall \text{ ingresso}$$

da qua, è possibile richiedere una condizione più restrittiva:

$$\left|\sum_k h[n-k]\right| < \infty$$

Consideriamo a questo punto alcune rappresentazioni alternative: anche se il sistema è a tempo discreto, è possibile definire una trasformazione di Fourier, del tipo:

$$X(\omega) = \sum_{n=-\infty}^{+\infty} x[n]\mathrm{e}^{-j\omega n}$$

esiste ovviamente anche la possibilità di antitrasformare questa espressione:

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{+\pi} X(\omega)\mathrm{e}^{j\omega n}\mathrm{d}\omega$$

Nel caso continuo si potrebbe anche parlare della trasformata di Laplace, intesa come generalizzazione di questa trasformazione. A tempo discreto,

esiste qualcosa del genere, ma un poco diversa: la trasformata $\mathcal{Z}$. Essa è definita come:

$$X(z) = \sum_{n=-\infty}^{+\infty} x[n] z^{-n}$$

Il motivo per cui questa rappresentazione è di nostro interesse, sta nel fatto che una classe molto importante di sistemi è quella dei sistemi descritti mediante equazioni alle differenze a coefficienti costanti: si tratta di equazioni con un numero finito di termini e tali per cui si possa usare una descrizione del tipo:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

Questa è l'equazione più generale che possa descrivere questa classe di sistemi: se l'equazione ha questa forma, il sistema è LTI e a coefficienti costanti. Di solito si trattano questi sistemi con una piccola semplificazione: se $a_0 = 1$, si può scrivere che:

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

Nel dominio della trasformata $\mathcal{Z}$, n ritardo si può modellare nella seguente maniera: data variabile temporale n:

$$x[n-k] \xrightarrow{\mathcal{Z}} X(z) z^{-k}$$

Applicando ciò all'equazione scritta poco fa:

$$Y(z) = -\sum_{k=1}^{N} a_k Y(z) z^{-k} + \sum_{k=0}^{M} b_k X(z) z^{-k}$$

è dunque possibile definire una funzione di trasferimento discreta come:

$$H(z) \triangleq \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}$$

La funzione di trasferimento sarà dunque sempre scritta come il rapporto di due polinomi, a coefficienti costanti, nella variabile complessa discreta z. Un'espressione di questo genere, come è evidente vedere, dà luogo a zeri e poli. Solitamente si fa analisi e/o progetto direttamente nel dominio del tempo, tuttavia queste nozioni possono essere senza dubbio (e saranno) utili.

2.2 Filtri FIR

Nel caso dei filtri FIR, ossia Finite Impulse Response, si ha solamente il secondo termine di sommatoria: $N = 0$. Questo significa che ciò che possiamo dire del sistema è la rappresentazione nel dominio del tempo diventa banalmente:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

Questo in altro modo si può dire dicendo che non c'è nessuna memoria interna, e che l'uscita dipende solo adgli ingressi; gli h sono semplicemente i b_k del filtro. Per "memoria interna" si intende il fatto che le uscite precedenti siano ricordate dal sistema; sicuramente si avrà una memoria all'interno del filtro, dal momento che esso ha bisogno di avere i campioni per un certo insieme di tempi, tuttavia si tratta di un buffer dei soli dati in ingresso. Le proprietà dunque sono:

- $h[n] = b$
- la sequenza dei coefficienti è finita, ma dunque lo sarà anche la risposta all'impulso; per questo motivo, il filtro FIR si chiama cos'.

Esistono diversi modi per progettare filtri; un'idea spesso utilizzata è quella di partire dal filtro analogico corrispondente, dunque ricavare un filtro analogo. Ciò nei filtri FIR in realtà non si può fare, dal momento che essi non hanno poli; solo con gli IIR si può tentare una strategia di questo tipo.

Si considerino a questo punto alcune soluzioni hardware.

2.2.1 Forma diretta

La cosiddetta "forma diretta' del filtro FIR è quella ricavabile mediante la semplice descrizione, in DFG, dell'equazione:

Si hanno $M + 1$ moltiplicatori, M sommatori, e le prestazioni sono già note dal capitolo precedente:

$$T_{\mathrm{CP}} = T_{\mathrm{m}} + MT_{\mathrm{a}}$$

Ossia, all'aumentare dell'ordine del filtro miglioro la sua selettività, ma d'altra parte ho bisogno di un maggiore tempo per fare l'elaborazione. Questa implementazione di sicuro funziona, ma non è il massimo.

2.2.2 Forma trasposta

Si usa il principio di trasposizione dei grafi, portando a ottenere un DFG" a partire dal DFG' che abbiamo a disposizione, applicando le seguenti regole:

1. invertire ingresso e uscita;
2. invertire la direzione di ogni arco;
3. tutte le label che indicano ritardi o prodotti devono rimanere invariate;
4. si scambiano di ruolo i nodi "somma" e "biforcazione.

Applicando queste regole, si ottiene qualcosa di questo genere:

In questo modo si fanno tutte le moltiplicazioni in parallelo, con un solo colpo di clock; le somme però sono disposte in modo tale da spezzare i cammini critici, se ne fa una per volta e così si aumenta il throughput:

$$T_{\mathrm{CP}} = T_{\mathrm{m}} + T_{\mathrm{a}}$$

Ciò inoltre non aumenta i componenti da utilizzare.

2.2.3 Forma in cascata

Un'idea alternativa potrebbe essere quella di considerare il filtro come cascata di più filtri "elementari"; nel dominio della trasformata $\mathcal{Z}$, si ha che:

$$H(z) = \sum_{k=0}^{M} h[k] z^{-k}$$

Questo, come già visto, è un polinomio in z^{-k}. Ci è consentito di scrivere come produttoria questa H:

$$H(z) = \prod_{k=1}^{M'} \left(b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2} \right)$$

dove $M = 2M'$, e M è dunque pari. Ciò che si fa, con questa "sostituzione", è dividere le radici del polinomio in tanti termini, ciascuno di ordine 2, ossia tanti filtri con ciascuno 2 radici. Fatto ciò, il filtro può essere pensato come una cascata:

dove

$$H(z) = \prod_{i=0}^{M'} H_i(z)$$

Sostanzialmente il risultato ottenuto da n filtro di questo tipo non cambia, ma la realizzazione sì: si può usare sempre la stessa implementazione di un filtro semplice, cambiando in ciascuna solo i coefficienti; concatenando dunque le varie implementazioni con i coefficienti corretti, si ottiene tutto ciò che si vuole. Il segnale finisce per subire tanti filtraggi, ottenendo lo stesso risultato che si otterrebbe con un unico filtro complicato. Questo principio ovviamente può essere applicato in più forme: grado 1, grado 3, grado 2, a scelta.

Questa struttura non presenta particolari vantaggi, se non il fatto che il problema "difficile" con questa tecnica è scomposto in molti problemi "semplici".

2.2.4 Richiesta di fase lineare

Ciò che ci piacerebbe idealmente avere sarebbe un filtro a fase costante: qualcosa che, con modulo unitario in banda passante, non tocchi la fase. Un sistema del genere sarebbe bello, ma d'altra parte sarebbe anche non causale; per questo motivo chiediamo qualcosa in meno, in modo da ottenere comunque risultati interessanti: ciò che chiediamo al nostro sistema è una variazione di fase lineare rispetto alla pulsazione ω:

$$\angle H(\omega) = -\alpha\omega$$

dove α è una certa pendenza; il modulo, come desiderato, sarà unitario in banda passante. Questa cosa è interessante, dal momento che dire che la fase ha un andamento lineare significa sostanzialmente che tutte le componenti di frequenza avranno lo stesso ritardo; infatti:

$$H(z) = z^{-alpha}$$

ossia, per ogni frequenza avrò un ritardo pari a α. Questa è una condizione molto utilizzata per esempio anche nelle applicazioni audio, dal momento che è una condizione in grado di non introdurre distorsione di fase.

Nel caso di filtri FIR, chiedere ciò sostanzialmente significa chiedere di avere qualcosa di particolare, che cercheremo di capire ora. Come noto, per un FIR si ha:

$$H(z) = \sum_{n=0}^{M} h[n]z^{-n}$$

passando alla trasformata di Fourier $H(\omega)$, si avrebbe:

$$H(\omega) = \sum_{n=0}^{M} h[n] \mathrm{e}^{-j\omega n}$$

La trasformata di Fourier del nostro segnale è composta da una combinazione lineare di esponenziali complessi; conoscendo la formula di Eulero, si ha che:

$$\mathrm{e}^{j\omega n} = \cos(\omega n) - j \sin(\omega n)$$

Ossia, si ha un termine reale che va come un coseno (il quale è una funzione notoriamente pari), e un termine immaginario che va come un seno (il quale è una funzione notoriamente dispari).

Calcolando la fase, si sa che:

$$\angle H(\omega) = \tan^{-1} \left(\frac{\mathbb{I}m \left\{ H(\omega) \right\}}{\mathbb{R}e \left\{ H(\omega) \right\}} \right) = \alpha\omega$$

Ossia, chiediamo che tutta questa espressione sia uguale a una certa costante, per la variabile ω. Come si può fare a ottenere ciò? Beh, si deve chiedere che o solo la parte immaginaria o solo la parte reale siano presenti. Ciò che si deve sostanzialmente fare è imporre una certa condizione su $H(\omega)$ e, di conseguenza, su $h[n]$, al fine di avere in risultante un numero puramente reale o puramente complesso.

La condizione che ci serve è:

$$h[n] = -h[-n]$$

oppure

$$h[n] = h[-n]$$

Dimostriamo questo fatto: si consideri per un secondo una funzione non causale (questa cosa torna molto utile al fine di semplificare i conti e renderli intuitivi):

$$H(z) = \sum_{n=-\frac{M}{2}}^{\frac{M}{2}} h[n] z^{-n}$$

Prendendo il primo e l'ultimo termine della sommatoria, si ha:

$$h\left[-\frac{M}{2}\right] \mathrm{e}^{j\omega\frac{M}{2}} + h\left[\frac{M}{2}\right] \mathrm{e}^{-j\omega\frac{M}{2}}$$

trasformandoli secondo la formula di Eulero, si ottiene:

$$h\left[-\frac{M}{2}\right]\cos\left(\omega\frac{M}{2}\right)+jh\left[-\frac{M}{2}\right]\sin\left(\omega\frac{M}{2}\right)h\left[\frac{M}{2}\right]\cos\left(\omega\frac{M}{2}\right)+jh\left[\frac{M}{2}\right]\sin\left(-\omega\frac{M}{2}\right)$$

In pratica si vede che i termini col coseno rimangono, mentre quelli immaginari si annullano. Se ciò (il fatto che i coefficienti siano uguali) è vero anche per le coppie intermedie, allora la funzione complessiva è puramente reale, e questa è una condizione sufficiente per avere la fase lineare.

Si può fare un discorso analogo per cercare di avere solo termini complessi: $h[k] = -h[-k]$ (condizione di antisimmetria).

Vi può essere un eventuale elemento centrale a fare da elemento di simmetria.

Il fatto di considerare la fase lineare è rilevante anche dal punto di vista implementativo; se devo infatti fare una cosa del tipo:

$$y[n] = \sum_{k=0}^{M} h[k]x[n-k]$$

Posso raggruppare le operazioni in maniera astuta: dividere la sommatoria in 2, andando con una da 0 a $\frac{M}{2} - 1$, trattare a parte il caso a metà (il punto di simmetria), dunque avere una gemella:

$$y[n] = \sum_{k=0}^{\frac{M}{2}-1} h[k]x[n-k] + h\left[\frac{M}{2}\right]x\left[n-\frac{M}{2}\right] + \sum_{k=\frac{M}{2}-1}^{M} h[k]x[n-k]$$

Data però la proprietà di simmetria, si ha:

$$h[k] = h[M-k]$$

Dunque, questo ci può portare a capire che $y[n]$ si può ottenere da una sola delle sommatorie: data la presenza della simmetria, posso elaborare meno campioni:

$$y[n] = h\left[\frac{M}{2}\right] + x\left[n-\frac{M}{2}\right] + \sum_{k=0}^{\frac{M}{2}-1} h[n]\left(x[n-k]+x[n-M+k]\right)$$

Dunque posso usare il sistema per prendere i campioni in ingresso, passarli a una linea di ritardo, facendo sostanzialmente qualcosa di questo genere:

Si ha una sequenza con M registri, e si moltiplica con i vari moltiplicatori per gli h_i. Il termine $\frac{M}{2}$ viene un po' "trattato a parte", nel senso che è gestito dall'ultimo ramo; nel caso in cui M sia dispari, l'ultimo ramo non va inserito. Questa soluzione architetturale ha $\frac{M}{2} + \frac{M}{2}$ sommatori, e $\frac{M}{2}$ moltiplicatori, risultando dunque vantaggiosa sotto il punto di vista dell'hardware da utilizzare.

2.3 Filtri IIR

L'espressione generale per i filtri IIR è la seguente:

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

A questo punto, anche per questo filtro, è possibile presentare un certo numero di forme.

2.3.1 Forma diretta I

Per "forma diretta 1" si intende una forma nella quale si incomincia disegnando la parte non ricorsiva, aggiungendo solo in un secondo momento la parte ricorsiva.

Per comodità si è rappresentata la figura in SFG invece che in DF: ciascun arco orizzontale rappresenta un moltiplicatore per una certa costante (indicato dalla label), ciascun ramo verticale con label z^{-1} è un elemento di ritardo. I nodi in cui convergono oggetti da sinistra e dal basso sono sommatori. Ciò che si può fare dunque per la seconda parte del grafo è ritardare l'uscita di diversi cicli, usando dunque ancora ritardi, moltiplicatori e sommatori.

Sappiamo che esiste la rappresentazione a catena; questa ci porta a pensare che il filtro sia rappresentabile come:

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-1}} = H_1(z)\, H_2(z)$$

dove $H_1(z)$ è la parte "puramente FIR" del filtro, e la seconda la parte contenente dei poli. Il filtro complessivo può essere pensato come la cascata di queste due parti.

2.3.2 Forma diretta II

Come noto, la moltiplicazione (che nel nostro caso assume significato di "mettere in cascata") è un'operazione commutativa; ciò potrebbe portarci a implementare in una forma diretta molto simile alla precedente, con però una peculiarità:

Questo schema è molto simile al precedente, ma in realtà, come si vede, in questo caso gli z^{-1} partono tutti dallo stesso punto, dunque ritardano tutti lo stesso segnale; questo ci porta a capire che è necessaria una sola catena di registri, a partire dalla quale si può prendere il segnale per entrambe le parti del filtro.

2.3.3 Forma cascata

Come nel filtro FIR, esistono le "forme cascata" e le "forme trasposte"; non mostriamo queste ultime, e mostriamo solo molto rapidamente un esempio visivo di forma a cascata:

$$H(z) = \prod_j H_j(z)$$

dove ciascuna sotto-funzione di trasferimento $H_j(z)$ si può semplicemente scrivere come:

$$H_j(z) = \frac{b_{0j} + b_{1j}z^{-1} + b_{2j}z^{-2}}{1 - a_{1j}z^{-1} - a_{2j}z^{-2}}$$

mettendo in cascata, si ottiene qualcosa di questo genere:

2.3.4 Forma parallela

Un'ultima forma nella quale è possibile implementare dei filtri è la cosiddetta "forma parallela". Essa sostanzialmente è basata sulla scrittura di $H(z)$ in termini additivi del tipo:

$$H(z) = \sum_k c_k z^{-k} + \sum_k \frac{A_k}{1 - B_k z^{-1}} + \sum_k \frac{F_k + G_k z^{-1} + K_k z^{-2}}{1 - D_k z^{-1-E_k z^{-2}}}$$

Una funzione di trasferimento può essere decomposta, mediante trucchi algebrici, in somme di funzioni di trasferimento più semplici; un trucco è quello della semplice divisione tra polinomi, un altro è l'uso del procedimento dei fratti semplici.

Il risultato nella forma più generale è qualcosa di questo tipo:

Si possono avere (caso dei soli coefficienti C_k) moltiplicatori con eventuali elementi di ritardo introdotti nel sistema; alternativa potrebbe essere la presenza di loop semplici, oppure forme complesse (le ultime viste) al fine di realizzare interi filtri IIR che a loro volta diventeranno parte del filtro finale.

2.4 Errori di quantizzazione

Abbiamo a che fare con rappresentazioni di tipo numerico; dato un valore numerico X, numero rappresentato (senza quantizzazione) in complemento a 2, esso è scrivibile come:

$$X = X_{\mathrm{m}} \left(-b_0 + \sum_{i=1}^{+\infty} b_i 2^{-i} \right)$$

Si ha a che fare con infinite cifre; si ha che il contenuto della parentesi tonda deve essere minore o uguale di 1, dunque pure b_0 dovrà essere minore di 1; la parentesi rappresenta il numero normalizzato, e X_{m} rappresenta un semplice fattore di scala: è come se nella parentesi tonda vi fossero semplicemente tutte le cifre dopo la virgola, e X_{m}, tipicamente una potenza di 2, rappresentasse il numero in grado di spostare la posizione della virgola.

Ciò che si fa in pratica, sia nel caso dei filtri sia nel caso di elaborazione generica di segnali, è considerare un numero finito di bit, pari a $B + 1$, dove B è un numero intero. Nella pratica si avrà a che fare con qualcosa di questo tipo:

$$\hat{x} = Q_B[x] = X_{\mathrm{m}} \left(-b_0 + \sum_{i=1}^{B} b_i 2^{-i} \right)$$

Il valore nella parentesi è ancora una volta limitato superiormente a 1, ma questa volta esso è composto da un numero finito di bit: non sarà più possibile rappresentare un numero con precisione infinita. In altre parole:

$$\hat{x} \longrightarrow b_0, b_1, b_2, ..., b_B$$

dove b_B è il LSB: Less Significant Bit, ossia il bit meno significativo del numero. X_{m} ha lo stesso significato di prima: è tipicamente una potenza di 2, in grado di riscalare la dinamica del numero: semplicemente, esso sposta la posizione della virgola, dunque cambia il numero di bit nella parte intera del numero.

Esistono due modi di effettuare l'operazione di quantizzazione: per troncamento, o per arrotondamento.

- quando si quantizza per troncamento, si fa qualcosa di questo tipo:

  si troncano le cifre, in modo che si riconduce un certo intervallo di numeri a un singolo numero; si perdono tutte le informazioni da una certa cifra in poi. Dato $\Delta = 2^{-B} X_{\mathrm{m}}$, questo è l'ampiezza massima dell'errore che si possa commettere.

- quando si quantizza per arrotondamento, si fa qualcosa di simile a prima, ma "traslato":

  In questo caso l'ampiezza massima dell'errore è la stessa, ma il fatto di aver introdotto questo "sfasamento" comporta di avere una posizione variabile da $-\frac{\Delta}{2}$ a $+\frac{\Delta}{2}$; l'errore, come si può vedere, ne trae giovamento, dal momento che l'andamento e l'ampiezza massima sono gli stessi, con però una traslazione verso il basso. Questa rappresentazione ha dunque dei vantaggi teorici rispetto alla precedente, ma è più complicata al momento di effettuare una realizzazione hardware.

Ci possiamo a questo punto porre una domanda: in quali occasioni è necessario quantizzare? Si consideri il seguente elenco applicato al successivo DFG:

1. all'ingresso: potremmo avere nello stream un numero di bit superiore a quello che si possa effettivamente gestire, dunque quantizzare con una funzione di quantizzazione Q potrebbe essere fondamentale;

2. al momento di moltiplicare per un certo coefficiente a: a è fornito da altre parti dell'elaboratore, dunque potrebbe essere dotato di troppe cifre; quantizzarlo può essere necessario per effettuare correttamente le operazioni successive;

3. dopo l'operazione di prodotto: il prodotto fa crescere il numero di bit, dunque una quantizzazione è importante;

4. in uscita: può non essere necessario, ma quantizzare non fa sbagliare.

Il fatto di dover quantizzare porta un effetto piuttosto negativo: ciò che implementiamo ora non sarà più la vera funzione di trasferimento $H(z)$, bensì una funzione modificata rispetto a essa, $\hat{H}(z)$

$$\hat{H}(z) = \frac{\sum_{k=0}^{M} \hat{b}_k z^{-k}}{1 - \sum_{k=1}^{N} \hat{a}_k z^{-k}}$$

Il fatto di avere questi coefficienti "modificati" e non quelli reali comporta uno spostamento di poli e zeri del filtro rispetto alla loro normale posizione; spostare un solo coefficiente a numeratore o a denominatore potrebbe far spostare tutti gli zeri o i poli (rispettivamente), e ciò potrebbe dunque modificare il comportamento del filtro.

Ciò che si fa (o che si fa fare a dei tool automatici) è calcolare la sensitivity $\mathrm{S}_{a_j}^{p_i}$ del polo rispetto al j-esimo coefficiente del filtro:

$$\mathrm{S}_{a_j}^{p_i} = \frac{\partial p_i}{\partial a_j}$$

Purtroppo però le forme realizzative non sono tutte equivalenti: a denominatore si avrà sempre una produttoria di termini, del tipo:

$$\frac{1}{\prod_{k \neq i} (p_i - p_k)}$$

dove dunque si moltiplicano, dato un certo polo, tutte le radici date dai vari $p_i - p_k$, i fisso, k variabile. Qua vengono fuori i problemi: un filtro molto selettivo avrà dei valori di sensitivity molto elevati, dal momento che i poli saranno molto vicini tra loro; questo è un problema che colpisce soprattutto i filtri IIR: i filtri FIR da un lato non hanno poli, dall'altro hanno gli zeri solitamente molto ben distanziati. Questo ci porta a intuire che per i FIR la quantizzazione è abbastanza semplice, e dunque anche utilizzare le forme dirette o trasposta può andare bene. Per gli IIR, a causa della vicinanza relativa dei poli, le sensitivity sono elevate, e dunque la forma diretta non si utilizza quasi mai; si punta di solito a utilizzare la forma cascata o la forma parallela.

Analizziamo a questo punto le motivazioni nascoste dietro la quantizzazione nel caso delle varie operazioni. Partiamo dai prodotti: nel caso vi siano dei prodotti:

serve avere a che fare con $2B + 1$ bit, dunque l'operazione di moltiplicazione incrementa di molto il numero di bit, rendendo la quantizzazione necessaria. In questo contesto si parla di "round-off noise": questo è il nome comunemente attribuito a questo tipo di quantizzazione. Esiste un modello in grado di rappresentare il sistema in questa situazione:

Al posto del quantizzatore si potrebbe introdurre un nodo sommatore, che come secondo ingresso ha del rumore, la cui statistica (sostanzialmente varianza) è il più possibile simile a quella del rumore che introduciamo. Quantizzare significa avere meno cifre significative, dunque se si introduce una quantità di rumore in grado di ridurre il numero di cifre significative, sostanzialmente si fa qualcosa di analogo a prima. Questa analisi, a differenza dell'analisi che si dovrebbe fare con l'operazione di quantizzazione,

è un'analisi di tipo lineare, dunque molto interessante e molto più semplice, dal momento che permette di calcolare le uscite mediante una banale applicazione del principio di sovrapposizione degli effetti:

$$y = y' + y''$$

dove uno dei due contributi è di rumore. Dopo un'analisi di questo tipo, è possibile mettere ciò in termini di SNR, e trarre le conclusioni del caso.

Nel caso della somma, il caso peggiore è quello in cui l'operazione di somma è tale da aumentare la dinamica di un bit. Si avrà dunque bisogno, per mantenere tutta l'informazione, di lavorare con più bit di quelli normalmente utilizzati. L'operazione di quantizzazione comporterà ovviamente una perdita di informazione, dunque bisognerà prestarvi attenzione.

Un'altra possibilità è quella di scartare un bit della parte intera, nel caso si abbiano condizioni particolari: nel caso i due bit della parte intera siano uguali, di fatto essi non portano informazione, dunque è possibile scartarne uno e, trattando opportunamente la virgola, mantenere tutta l'informazione del numero.

Una possibilità è quella di "operare per saturazione": si può fare una statistica di quello che è il comportamento del numero risultante dall'operazione e, nel caso l'aumento di bit sia poco frequente, invece che rappresentare il numero corretto introdurre un errore. Si consideri il seguente esempio: $x_1 = 010$, $x_2 = 011$, $y = 0101$: volendo ricondurci a un $\hat{y}$ quantizzato su solo 2 bit ma cercando di perdere meno informazione possibile, ciò che si può fare è spostare la virgola fin dove si deve, e mettere tutti i bit che rimangono al valore più alto, in modo da far saturare il numero al massimo valore a disposizione. Se l'evento è statisticamente poco rilevante, è possibile accettare il fatto di avere degli errori.

2.4.1 Meccanismo di scalamento

Si consideri a questo punto un generico DFG:

Il problema che si vuole risolvere a questo punto è quello di cercare di non avere dell'overflow in nessun punto del grafo: per ogni nodo k appartenente al grafo, tutta l'informazione deve essere contenuta dentro B bit correttamente. In altre parole, ciascun campione elaborato in ciascun nodo k, $w_k[n]$, deve essere tale da poter essere rappresentato correttamente in B bit. Ciò che si deve fare a questo punto è cercare una condizione in grado di garantirci il fatto che ogni campione rispetti la condizione che ci siamo preposti. Si consideri la generica rappresentazione di un campione $w_k[n]$, come risposta ad impulso moltiplicata per l'ingresso:

$$w_k[n] = \sum_n x[n-m]h_k[n]$$

devo essere sicuro che $|w_k[n]|$ sia limitato al massimo numero rappresentabile con i bit a disposizione, al fine di non avere errori interni al circuito. Ciò si può dire dicendo:

$$|w_k[n]| = \left| \sum_n x[n-m]h_k[n] \right|$$

valutare questa espressione non è banale, dal momento che ci servirebbe conoscere molti dati. Ciò che si fa di solito dunque è utilizzare un approccio di tipo worst-case, e sostituire a $x[n-m]$ il massimo valore possibile, $x_{\max}$; applicando dunque il principio della convoluzione discreta:

$$\implies \left| \sum_m x_{\max} h_k[m] \right| = x_{\max} \left| \sum_m h_k[m] \right|$$

posso a questo punto effettuare una maggiorazione classica, derivante dalla generalizzazione della diseguaglianza triangolare, ottenendo che ciò è:

$$\implies \leq x_{\max} \sum_m |h_k[m]|$$

ossia, si chiede che il modulo della somma sia inferiore della somma dei moduli dei termini costituenti la risposta all'impulso. Dopo tutte queste operazioni, la nostra condizione diventa:

$$x_{\max} \sum_m |h_k[m]| \leq 1 \implies |w_k[n]| \leq 1$$

la prima implica la seconda, dal momento che introduce una serie di worst case, rendendola una scelta più conservatrice.

Esempio: filtro FIR

Si consideri a questo punto un esempio pratico di ciò che abbiamo detto, a partire dal quale introdurremo alcuni spunti: un filtro FIR, del tipo

$$y[n] = \sum_{k=0}^{3} b_k x[n-k]$$

Conoscendo l'insieme dei valori b_k, come possiamo essere certi che in ogni punto del filtro non si superi mai il valore assoluto? Beh, prima di tutto,

una buona idea potrebbe essere quella di determinare il punto critico del grafo. Senza ombra di dubbio, questo sarà il punto finale, quello nel quale potrebbero essere sommati i valori più elevati in modulo. Supponendo di avere dunque $B + 1$ bit in complemento a 2, vogliamo che ci siano sempre $B + 1$ bit al massimo su ciascun ramo.

Sommando in valore assoluto i quattro coefficienti, ricordando che per un filtro FIR si ha che $h[k] = b_k$, si ottiene:

$$b_{\text{sum}} = \sum_{k=0}^{3} |b_k|$$

Se prendiamo i campioni di ingresso e li riscaliamo per b_{sum}, otteniamo dei $\tilde{x}[n]$ e questi, applicati al posto di $x[n]$, permetteranno di essere certi che non vi sia mai overflow nel filtro; dovremo, dopo questa cosa, ri-moltiplicare il risultato per b_{sum}.

Ciò che si può fare è approssimare dunque b_{sum} con una potenza di due, e dunque sia il prodotto sia la divisione diventano shift, ottenendo costo hardware trascurabile.

Una scelta come quella appena fatta è estremamente conservativa dunque, per quanto funzionante, ha di sicuro degli svantaggi: esso comporta la perdita di molta informazione, guadagnando però un'assoluta sicurezza contro l'overflow.

Un a tecnica meno conservativa potrebbe essere quella basata sul calcolo della varianza dei campioni:

$$\sqrt{\mathbb{E}\left\{w_k^2[n]\right\}} = \sqrt{\sum_{n=0}^{N} h_k^2[n]}$$

Ciò fornisce sostanzialmente un'idea della situazione media del fattore di guadagno: è un approccio più statistico, ma pure più realistico. Si può aumentare questa deviazione standard (radice della varianza) moltiplicandola per un certo fattore δ di sicurezza, e utilizzare dunque il risultato al posto di b_{sum} per ricavare il fattore di scalamento. Questa cosa non esclude che, per una coda della gaussiana, si possano avere errori, tuttavia si avrà di sicuro una minore perdita di informazione rispetto al caso precedente, dal momento che si deve "tagliare di meno".

Esempio: scalamento

Si supponga a questo punto di avere un filtro FIR con i seguenti valori:

$$h[n] = \{-1,0;\ 3,75;\ 3,75;\ -1,0\}$$

Essendo il filtro a coefficienti simmetrici, si avrà sicuramente un andamento lineare della fase. Si avrà inoltre un solo moltiplicatore, tanto un valore vale 1, ed essendo gli altri due coefficienti uguali si potrà prima sommare la coppia di campioni, poi moltiplicarli entrambi per 3,75. Date le h, è quindi possibile calcolare il guadagno complessivo di caso peggiore cui si va incontro con il filtro:

$$b_{\text{sum}} = \sum_n |h[n]| = 9,5$$

Questo sarebbe il guadagno nel caso peggiore che si possa avere.
Si possono a questo punto seguire diverse strade.

- Se siamo vincolati all'ingresso come numero di bit, nel senso che siamo costretti ad accettare in ingresso un certo numero di bit, allora possiamo dire di quanto aumenteranno, al peggio, i bit sull'uscita:

  in altre parole, nel caso peggiore, si può avere una crescita di 4 bit della dinamica, ossia rispetto al numero di bit iniziali.

  Per capire a un certo punto quanti bit servono in un FIR non alla fine, ma in un generico punto, è sufficiente prendere il filtro, composto da diverse prese in cascata, solo fino alla presa interessata; trattando il filtro come se finisse lì e facendo la stessa cosa fatta, si può capire, per qualsiasi punto, di quanto crescerà la dinamica.

- Una seconda opzione è quella "statistica": possiamo cautelarci nel caso in cui ci sia un overflow, ma non allocare comunque tutti i bit necessari; un modo per fare ciò è quello di utilizzare, come filtro di scalamento, la grandezza legata alla varianza, ossia la già vista radice della somma dei moduli quadri dei coefficienti, moltiplicati per il δ. Si noti che, in questo caso, sarà necessario introdurre nel circuito un meccanismo di saturazione, in modo tale da ridurre gli errori nel caso in cui si vada a finire in stato di overflow.

- Una terza via, sempre per cautelarsi, è quella di riscalare i valori in ingresso: se si fissa il numero di bit in uscita anzichè in ingresso (supponiamo, nel nostro caso, per esempio, di fissare a 10 bit), e riscalare al numero di bit sufficiente per evitare l'overflow l'ingresso, tagliando dunque una certa porzione di informazione. Questa cosa potrebbe aumentare gli errori, dal momento che, invece di 8 bit, in ingresso si

avranno al più 6 bit, ma d'altra parte questa cosa permette di gestire un numero ridotto di bit, senza dover introdurre meccanismi di saturazione nel circuito.

2.5 Tecniche per l'eliminazione dell'operazione di moltiplicazione

Aldilà degli aspetti finora analizzati, esistono tecniche in grado di eliminare le operazioni di moltiplicazione, le quali di fatto sono molto complicate e dunque pesanti da realizzare, in termini di hardware da utilizzare. Buona parte di queste tecniche si basa su una rappresentazione detta "CSD": Canonical Signed Digit. Si parla di digit e non più di bit, dal momento che si può avere l'assunzione di un valore ternario.

Si consideri a questo punto un esempio numerico, atto a chiarire le cose. Si consideri il numero "93", che in forma binaria è:

$$93 = 1011101$$

A questo punto si aprono diverse strade: o allochiamo un moltiplicatore, oppure riconosciamo il fatto che la cifra in binario è la somma di tante potenze di 2:

$$1011101 = 2^6 + 2^4 + 2^3 + 2^2 + 2^0$$

Le operazioni di potenza di due sono "gratuite", in termini di hardware: esse sono sostanzialmente degli shift, delle traslazioni, e se gli shift sono fatti per coefficienti non variabili, allora si tratta semplicemente di spostare dei fili e creare dei collegamenti diversi, senza nessun tipo di elaborazione. Si può per esempio, per questo caso, utilizzare una rappresentazione di questo genere:

Si ha una misura della complessità di questo genere di operazioni: essa è definita come il numero di sommatori presenti per fare l'operazione. In questo caso, come si può vedere, la complessità è pari a 4.

Questo fatto ha introdotto una semplice rappresentazione binaria a partire dalla quale è possibile sostituire un moltiplicatore con un certo numero di sommatori; a partire da questa idea, dunque, si può cercare di far di meglio al fine da ridurre ulteriormente la complessità.

Una prima idea è: spesso, nei numeri convertiti in base 2, si hanno delle sequenze di "1", che possono essere semplificate in maniera intelligente. Si consideri ancora una volta l'esempio precedente, 93:

$$1011101$$

questo numero, come si può vedere, ha una sequenza di tre uni. Come si può migliorare la cosa? Beh, con i seguenti passi:

1. prima di tutto, tutti gli uni isolati vengono lasciati come tali; per ora, gli zeri vengono lasciati zeri;

2. si trovi la sequenza da modificare; in questo caso, tutti gli uni tranne quello meno significativo della sequenza vengono trasformati in zeri; quello meno significativo viene segnato, in maniera da capire che esso non deve essere più contato come "1", ma come "-1"; ciò significa far ciò:

$$1000\bar{1}01$$

3. a questo punto, il bit più significativo del primo bit cancellato diventa un 1:

$$1100\bar{1}01$$

Cosa significa tutto ciò in soldoni? Beh, semplice: invece che considerare una sequenza di 1, la quale porterebbe ad avere altri shift e dunque altri termini da sommare, si considera il numero appena superiore (si ricordi che, nei numeri binari, dopo una sequenza di soli uni ci sarà una sequenza con il MSB a 1 e tutti gli altri a 0), si considera il numero ad essa superiore, e vi si sottrae 1. In questo modo, l'hardware necessario per fare ciò sarà:

Si hanno tre sommatori invece che 4, e questo semplicemente cambiando la rappresentazione del numero in questione.

Questa cosa ha ovviamente senso quando si hanno sequenze di uni più lunghe di 2; in caso contrario non si ha nessun vantaggio; da sequenze di tre uni (comprese) in poi, usare questa procedura (rappresentazione) è dunque una cosa sicuramente intelligente.

Questa cosa ci può avviare verso nuovi orizzonti: possiamo a questo punto fare altre cose, come delle decomposizioni. Si consideri ancora una volta il numero 93 (un numero assolutamente casuale, che non ha particolari proprietà):

$$93 = 3 \times 31$$

si può notare, per questo numero, una cosa piuttosto simpatica:

$$3 \times 31 = (2+1) \times (2^5 - 1)$$

questa cosa si può dunque implementare semplicemente così:

Questo vale per praticamente qualsiasi numero: è possibile ricercare somme o sottrazioni o prodotti per potenze di due o numeri a esse vicini. Si considerino dunque come casi particolari, i coefficienti scrivibili in questo modo:

- hanno costo pari a 1 i coefficienti scrivibili in forma:

$$2^{k_0}\left(2^{k_1} \pm 2^{k_2}\right)$$

- hanno costo pari a 2 i coefficienti scrivibili in forma:

$$2^{k_0}\left(2^{k_1} \pm 2^{k_2} \pm 2^{k_3}\right)$$

- hanno costo pari a 2 i coefficienti scrivibili in forma:

$$2^{k_0}\left(2^{k_1} \pm 2^{k_2}\right)\left(2^{k_3} \pm 2^{k_4}\right)$$

Tutte le strutture appena descritte sono dette "MAG": Multiplier Adder Graph.

Prima di passare a un nuovo argomento, si vuole proporre un metodo atto a dare una strada utile da seguire per i calcoli di questo tipo.

1. Isolare il segno dei coefficienti (tanto esso viene semplicemente gestito con un segnale + o - interno al sommatore).
2. Rimuovere dall'insieme dei coefficienti da processare quelli che sono già potenze di 2, o in essi i fattori che sono potenze di 2, nel caso in cui si possano decomporre alcuni dei coefficienti in prodotti per potenze di 2.
3. Se a questo punto vi sono coefficienti con costo realizzativo 1, si eliminino.
4. Usare i coefficienti tratti ai passi precedenti come fattori dei coefficienti residui, se possibili (riutilizzare).
5. Eliminare i coefficienti di costo 2.
6. Proseguire in questo modo.

Al termine di queste operazioni, disegnando passo-passo tutti gli elementi "eliminati" dall'insieme secondo le regole precedentemente proposte, si può disegnare in maniera ottimizzata il grafo.

2.5.1 Aritmetica distribuita

Data un'operazione di calcolo del tipo:

$$y[n] = \sum_{n=0}^{N-1} c[n]x[n]$$

ossia, una somma di prodotti, le grandezze x e c sono rappresentabili in forma binaria e in complemento a 2. Si consideri per ora la forma binaria unsigned, per poi estendere in seguito le considerazioni che faremo. Il fatto che si possa rappresentare $x[n]$ in binario, significa sostanzialmente che:

$$x[n] = \sum_{b=0}^{B-1} x_b[n]2^b$$

Dove, dunque, $x_0[n]$ è il LSB (Least Significant Bit), x_{B-1} il MSB, $x_b[n]$ può valere 0 o 1. Sostituiamo l'espressione in y:

$$y[n] = \sum_{n=0}^{N-1} c[n] \sum_{b=0}^{B-1} x_b[n]2^b$$

Dal momento che abbiamo delle sommatorie finite, possiamo cambiare l'ordine delle sommatorie:

$$y[n] = \sum_{b=0}^{B-1} 2^b \left(\sum_{n=0}^{N-1} c[n]x_b[n] \right)$$

Si noti l'implicazione enorme causata da questa formulazione: ora, dentro la parentesi, $x_b[n]$ è un **singolo bit** di x: all'interno della parentesi tonda, b è costante, dal momento che varierà solo fuori dalla sommatoria (avendo portato la sommatoria in b fuori). Ciò che si può fare dunque è ricavare in parallelo tutti i bit $x_b[n]$ per tutti i campioni, dunque combinarli con i vari coefficienti $c[n]$. Ciò permette di ottenere una funzione del tipo:

$$f\left(c[n], x_b[n]\right)$$

Come mai questo ragionamento apparentemente strano? Beh, semplice: l'idea dietro l'aritmetica distribuita è quella di calcolare **a priori** tutti gli elementi della funzione f, combinante tutte le possibilità esistenti di x_b e dei vari c, in modo che, quando si ha un certo ingresso x_b, semplicemente lo si possa utilizzare come un indirizzo al quale andare a pescare, in una look-up table (ossia in una memoria) il coefficiente, come già detto, precalcolato, e

produrlo in uscita. Questi andranno infine sommati e moltiplicati per i 2^b, i quali però non saranno moltiplicatori, bens shift.

Per realizzare questa cosa è necessaria una struttura hardware sostanzialmente basata su una LUT, con dentro questi risultati, indirizzati dagli x_b. Una soluzione potrebbe essere la seguente:

Il numero di bit in ingresso alla LUT è N, dunque avrò 2^N locazioni di memoria all'interno di essa. Ciascuna locazione di memoria deve essere tale da poter mantenere al proprio interno il valore massimo della funzione f calcolato, dunque dipende sostanzialmente dai dati. Nella configurazione presentata, si ha la realizzazione diretta di ciò che è stato proposto: si ha un *barrel shifter*, ossia uno shifter variabile, elemento non banale da realizzare. Questa cosa si può vedere in questo modo, o in quest'altro:

Questa cosa evita di usare il barrel shifter: invece che mettere un ritardo verso "sinistra" sul nodo d'ingresso, se ne mette uno verso "destra", introducendo un anello: quando il primo elemento entra, questo andrà poi nell'anello, e tornerà indietro; il primo elemento continuerà a girare nell'anello, e dunque a essere ritardato; introducendo man mano campioni, essi staranno di meno nell'anello, e mantenendo il sincronismo si otterrà lo stesso risultato, "tenendo fermo il filo in alto" e "traslando a destra quello in basso". La vera differenza rispetto alla prima realizzazione sta nel fatto che ora non si ha più il barrel shifter, ma solo un singolo shifter a valore fisso, cosa molto semplice da realizzare in hardware.

L'altra estensione che si può fare, come già citato, è quella che riguarda la gestione dei valori con segno, ossia $x[n]$ espresso in complemento a 2. Ciò che si fa è prendere la parte di prima, aggiungendo una piccola estensione:

$$x[n] = -2^B x_B[n] + \sum_{B=0}^{B-1} x_b[n]$$

dove, al solito, $x_B[n] = \{0, 1\}$; nel caso in cui valga 0, si ha solo il secondo termine, dunque non si complementa; nel caso in cui valga 1, si ha il bit a 1, e dunque si fa la somma, e questa farà la complementazione.

L'aritmetica distribuita è sicuramente semplificativa in termini hardware ma, con B piccoli, è utile anche per puntare a implementazioni **veloci**: in un numero ridotto di cicli si riesce infatti a risolvere rapidamente i problemi.

Capitolo 3

Trasformata discreta di Fourier

3.1 Introduzione

Esistono diverse possibili implementazioni per la trasformata discreta di Fourier. In questo capitolo sostanzialmente ne analizzeremo due, una delle quali particolarmente applicata.

Partiamo, al fine di avere delle basi su cui fondare una trattazione, dalla definizione della DFT; come noto, la trasformata di Fourier dice che:

$$X(f) = \int_{-\infty}^{+\infty} x(t)\mathrm{e}^{-j2\pi ft}dt$$

Questa cosa va discretizzata su t e su f, ottenendo la trasformata discreta:

$$x[n]\mathrm{e}^{-j2\pi\frac{1}{N}kn}$$

dove N è il numero di campioni nella finestra di cui intendiamo calcolare la trasformata discreta. Ovviamente, al posto dell'integrale si avrà una sommatoria. Dunque:

$$X[k] = \sum_{n=0}^{N-1} x[n]\mathrm{e}^{-j2\pi\frac{1}{N}kn}\forall k$$

dunque, k può assumere i valori più disparati: $0, 1, ..., N-1$.

L'operazione in questione, come si può vedere, è già nota: sostanzialmente riconducibile alla ormai solita somma di prodotti, in cui però i coefficienti sono dei numeri complessi. Nella trattazione utilizzeremo una rappresentazione più compatta della cosa, basata sulla definizione di un coefficiente w_N:

$$w_N \triangleq \mathrm{e}^{-j\frac{2\pi}{N}}$$

Dunque:

$$X[k] = \sum_{n=0}^{N-1} x[n] w_N^{kn}$$

L'implementazione di questa equazione comporta avere $N-1$ somme complesse, le quali sono a loro volta $2(N-1)$ somme reali (bisogna fare la somma sia per le parti reali sia per le parti immaginarie), e N prodotti complessi, i quali sono più complicati:

$$(a+jb)(c+jd) = ac - bd + j(bc+ad)$$

In totale, sia avranno $4N$ prodotti reali, $4N-2$ somme reali, e questo per calcolare ciascun coefficiente della trasformata. Dal momento che la finestra richiede il calcolo di N coefficienti, si avranno $4N^2$ prodotti reali, $4N^2 - 2N$ somme reali. La complessità del problema, come si può immaginare, è estremamente elevata, dunque è fondamentale ridurre questa complessità computazionale.

Si può partire dalla DFT nuda e cruda, senza introdurre aspetti implementativi: la DFT infatti ha delle proprietà matematiche che permettono di ridurre N, ma non la dipendenza dal suo quadrato. Analizziamo le principali proprietà:

- proprietà di simmetria:

$$w_N^{k(N-n)} = w_N^{-kn}$$

- proprietà di periodicità:

$$w_N^{kn} = w_N^{k(N+n)} = w_N^{k+N)n}$$

Queste proprietà possono essere usate in maniera astuta per ridurre le operazioni di prodotto: quando infatti si ha a che fare con il campione $x[n]$ e con quello $x[N-n]$, ossia i campioni simmetrici rispetto al campione centrale della finestra, i due coefficienti sono uguali in parte reale, e opposti in parte immaginaria; in altre parole, sono due numeri complessi coniugati:

$$w_N^{-kn} = \left(w_N^{kn}\right)^*$$

Si supponga per esempio di dover calcolare, a un certo punto, qualcosa tipo

$$\mathbb{R}e\left\{x[n]\right\}\mathbb{R}e\left\{w_N^{kn}\right\}$$

Dopo un po', ci si troverà a calcolare:

$$\mathbb{R}e\left\{x[N-n]\right\}\mathbb{R}e\left\{w_N^{k(N-n)}\right\}$$

La proprietà di simmetria permette di raggruppare nella seguente maniera l'operazione, riducendo le cose da calcolare:

$$\left(\mathbb{R}e\left\{x[n]\right\}+\mathbb{R}e\left\{x[N-n]\right\}\right)\mathbb{R}e\left\{w_N^{kn}\right\}$$

Nelle parti immaginarie, si avrà la stessa cosa, con però la sottrazione. Questo permette di capire che un certo numero di operazioni non andrà effettuata, ma ciò non elimina la dipendenza dal quadrato di N.

Per eliminarla o quantomeno attenuarla, esistono diversi approcci, che verranno ora presentati.

3.2 Algoritmo di Goertzel

Questo algoritmo funziona, ma sostanzialmente solo nel caso in cui si debba calcolare solo una parte dei coefficienti. Si provi ad esplicitare la struttura del calcolo nella seguente maniera:

$$X[k]=x[0]+w_N^k\left(x[1]+w_N^k\left(x[2]+...\right.\right.$$

Questa cosa permette, ricorsivamente, di organizzare in questo modo la computazione, annidando ricorsivamente termini di questo tipo. Questa cosa si può descrivere con qualcosa di simile a quanto visto precedentemente sui filtri IIR, mediante una reazione:

Questa cosa è sostanzialmente analoga a un IIR in cui i campioni intermedi però non servono: di tutti i campioni da prendere, l'unico interessante è l'ultimo. Per ciascun valore di k, dunque, prenderemo solo l'ultimo valore del ciclo.

Al fine di comprendere meglio questa tecnica e realizzarne una struttura nota, è possibile trattare questa funzione come un filtro, applicando il formalismo della trasformata $\mathcal{Z}$:

$$H(z)=\frac{1}{1-w_N^k z^{-1}}=\frac{1}{1-w_N^k z^{-1}}\left(\frac{1-w_N^{-k}z^{-1}}{1-w_N^{-k}z^{-1}}\right)$$

sviluppando il prodotto, si può ottenere:

$$= \frac{1 - w_N^k z^{-1}}{1 + z^{-2} - z^{-1}\left(w_N^k + w_N^{-k}\right)}$$

questa cosa è particolarmente interessante: si sta prendendo, dentro la parentesi tonda, la somma di due termini, uno il complesso coniugato dell'altro; ciò produrrà l'eliminazione della parte immaginaria, ottenendo dunque un termine puramente reale, che noi rappresenteremo con il coseno (in riferimento alla nota formula di Eulero di trasformazione dell'esponenziale complesso):

$$= \frac{1 - w_N^k z^{-1}}{1 + z^{-2} - z^{-1} 2\cos\left(\frac{2\pi}{N}k\right)}$$

Questa, a questo punto, è sostanzialmente quella che prima chiamavamo "funzione di trasferimento del filtro"; dal momento che si ha solo una parte reale, questa cosa sarà sicuramente più facile da implementare rispetto alla precedente. Usando dunque la forma diretta 2, il DFG diventerà semplicemente:

Domanda finale: quanti prodotti dobbiamo fare, per una struttura del genere? Beh, un prodotto reale nella parte ricorsiva, che dovrà iterare N volte; $2N$ somme (2, per N volte), sulla parte IIR, somme complesse, dunque $4N$ somme reali, dal momento che, di solito, $x[n]$ è una quantità complessa. La parte FIR avrà un prodotto complesso, una somma complessa, e ancora due somme complesse. La cosa buona però è il fatto che le operazioni sulla parte FIR non devono essere reiterate, dal momento che si deve solamente prendere $y[n]$.

Volendo calcolare tutti i valori dell'uscita, si avrà ancora dipendenza da N^2, ma il numero di coefficienti da calcolare è abbattuto: se si devono solo calcolare dunque alcuni campioni, questa tecnica è ottima.

3.3 FFT: DIT radix-2

A questo punto si vuole presentare una tecnica appartenente alla classe delle FFT, ossia delle trasformate veloci di Fourier: la FFT radix-2. Questa tecnica, come molte delle tecniche FFT, può essere utilizzata solo per valori di N che siano una potenza di 2; ciò in realtà non è un grosso limite dal momento che, con trucchi come lo zero padding, è possibile riempire facilmente la finestra e ricondursi alla situazione desiderata. Questa tecnica è una DIT, ossia una decimazione di tempo: dati i campioni $x[n]$, si dividono in campioni pari e campioni dispari; questo significa sostanzialmente separare la sommatoria in due sommatorie distinte:

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] w_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] w_N^{(2r+1)k}$$

Questa cosa può essere riscritta nella seguente maniera:

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] (w_N^2)^{rk} + w_N^k \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] (w_N^2)^{rk}$$

Questa cosa è molto utile dal momento che ci permette di interpretare il w_N nel seguente modo:

$$w_N^2 = \left(\mathrm{e}^{-j2\pi\frac{1}{N}}\right)^2 = \mathrm{e}^{-j2\pi\frac{1}{\frac{N}{2}}} = w_{\frac{N}{2}}$$

Cosa significa ciò? Questo semplicemente esplicita il fatto che le sommatorie riguardano una popolazione di campioni pari alla metà della popolazione normale: si divide la trasformata in due trasformate, ciascuna delle quali ha la metà degli elementi di prima:

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] w_{\frac{N}{2}}^{rk} + w_N^k \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] w_{\frac{N}{2}}^{rk}$$

Apparentemente, potremmo dire che questa cosa ci faccia fare del lavoro in più, ma si ricordi che la complessità va come il quadrato: se abbiamo diviso le trasformate da N campioni in trasformate da $\frac{N}{2}$ campioni, non tenendo conto delle operazioni che si devono fare per ricombinare gli elementi delle due sommatorie, le sommatorie per essere calcolate richiederanno $\left(\frac{N}{2}\right)^2 + \left(\frac{N}{2}\right)^2 = 2 \times \frac{N^2}{4} = \frac{N^2}{2}$ operazioni.

Questa è l'idea di base, ma i giochi non sono finiti: ciò che abbiamo fatto per ora è stato prendere una operazione di DFT, e scomporla in due operazioni distinte; supponendo per esempio di dover calcolare la DFT di 8 elementi, ciò che abbiamo fatto finora è semplicemente scomporla in due DFT da 4 elementi ciascuna: nella prima si introdurrebbero gli elementi pari ($n = 0, 2, 4, 6$), nella seconda i dispari ($n = 1, 3, 5, 7$). In uscita da ciascun blocco DFT da 4 si avrebbero coefficienti G (per il primo blocco) e H (per il secondo). Come si può ricombinare la cosa? Vediamo direttamente la risposta:

I coefficienti G sarebbero le uscite, come già detto, del blocco 4-DFT che gestisce gli ingressi pari, mentre gli H sono quelli che vengono gestiti dal blocco dei dispari. Come si può intuire osservando l'equazione della trasformazione, si devono combinare, mediante una somma pesata, $G[0]$ e $H[0]$ per

ottenere $X[0]$, ossia il primo coefficiente della trasformata. Ciascun nodo nel disegno rappresenta un sommatore. Come si vede, si devono semplicemente sommare i contributi che vengono dai rami con lo stesso indice, e, nel caso del blocco dispari, moltiplicare il valore in uscita dal blocco per w_N^k, dove k è l'indice dell'elemento in uscita che vogliamo ottenere. Con questo procedimento si è capito come ottenere i primi 4 coefficienti (per questo caso), ma che fare per gli altri 4? Beh, si ricordi che la DFT è una trasformata con proprietà di periodicità, dunque ciò che si ha è che $G[0] = G[4]$, $H[0] = H[4]$, $G[1] = G[5]$, e così via. L'unica cosa che cambierà, dunque, saranno i pesi della somma (si ricordi bene che essi vengono applicati, come moltiplicatori, solo ai rami dispari; i rami pari hanno sempre peso 1, come d'altra parte si evince guardando l'equazione di trasformazione); per esempio:

$$X[4] = G[0] + w_N^4 H[0]$$

Questo ragionamento può essere reiterato: ciascuna DFT da quattro punti può essere decomposta, con il medesimo metodo, in DFT da due punti: per ciascuna 4-DFT si avranno dunque due 2-DFT, una per pari e una per dispari. Ovviamente, anche in questo caso sarà necessario prendere i campioni "pari" e i campioni "dispari": per "pari" intenderemo $0, 4$, per "dispari" $2, 6$ (se ne prende uno sì e uno no).

Questo tipo di struttura è detta "struttura a butterfly": a ogni giro il w_N^k diventa con N dimezzato (passando da 4 a 2 per esempio si ha $w_{N/2}^k$.

La struttura per la 2-DFT è:

Questo, al solito, si ottiene applicando direttamente la decomposizione. Il primo ramo calcolerà l'elemento k-esimo, $k = 0$, mentre il secondo quello con $k = 1$. Usando la definizione, si vede che, per il primo ramo, si avrà la somma dei due coefficienti in ingresso: $a + b$. Qualcosa di apparentemente più complesso per il secondo ramo:

$$a + w_2^1 b$$

cos'è w_2? Beh:

$$w_2 = \mathrm{e}^{-j2\pi\frac{1}{2}} = \mathrm{e}^{-j\pi} = -1$$

dunque, si avrà semplicemente $a - b$.

Quante operazioni si fanno sviluppando questa operazione? Beh, si hanno N coefficienti, dunque N termini da calcolare; vediamo:

- per lo stadio con le 8-DFT, si avranno N somme complesse, N prodotti complessi (senza tenere conto del fatto che alcuni prodotti possono essere semplificati dal momento che valgono 1 o -1);

- per lo stadio precedente, 4-DFT, stesso discorso: si fanno 2 DFT con $N/2$ punti, dunque si avranno 2 volte $N/2$ somme e $N/2$ prodotti: non cambia niente; questo vale dunque per tutti gli stadi successivi al primo;

- per il primo stadio, quello con le 2-DFT, non si hanno prodotti, dal momento che come visto si hanno solo le somme non pesate dei coefficienti; in questo caso si avranno dunque N somme e 0 prodotti.

Serve a questo punto sapere quanti stadi vi sono all'interno del sistema. Dati N coefficienti, il numero di stadi A è:

$$A = log_2 N$$

dunque, si dovranno avere AN somme complesse, che coincidono con $2AN$ somme reali, $(A-1)N$ prodotti complessi, che diventano $4N(A-1)$ prodotti reali, $2N(A-1)$ somme reali. D'altra parte, la complessità, da N^2, diventa $N \log N$: vantaggio estremamente sensibile.

3.3.1 Note sulle implementazioni

Per implementare questa struttura, sono possibili diverse soluzioni, a seconda di ciò che si vuole fare in termini di trade-off tra velocità e hardware parallelo impiegato. Tutto si basa sull'uso di "processori butterfly", ossia processori che facciano la FFT. Si può usare un sistema con due memorie: una memoria X che al proprio interno abbia i campioni elaborati e quelli da elaborare, dunque il butterfly "reazionato" con la memoria, in modo da avere un meccanismo che riempia la memoria con i campioni già elaborati. Si avrà una seconda memoria, dunque, W, contenente i coefficienti w_N precalcolati, in maniera che dopo le operazioni su di esse siano molto più veloci.

Nella memoria X, come detto, dovranno essere introdotti altri valori, da memorizzare dopo la loro elaborazione; questo serve per emulare la presenza di più stadi all'interno della struttura: gli stessi valori elaborati in un primo momento diventeranno dei nuovi input, da elaborare nei momenti successivi dalla stessa struttura butterfly. Dati N valori in ingresso, questa struttura finirà la trasformazione dopo $\frac{N}{2} \log_2 N$ cicli. In questo caso, dunque, si ha una struttura semplicissima (poco hardware), ma lenta.

Al fine di velocizzare, la soluzione è fondamentalmente quella di aumentare l'hardware, introducendo delle parallelizzazioni: se per esempio si allocano $N/2$ processori, si velocizza di $N/2$ volte l'elaborazione.

Nel caso non ci si accontentasse neanche di questo incremento di velocità, si potrebbe mappare un DFG completamente parallelo, ossia con $\frac{N}{2} \log_2 N$

processori: la velocità di esecuzione sarà tale per cui, in un T_{CK}, si avranno tutti i punti, in un solo colpo di clock.

3.3.2 Recupero delle informazoni

Come detto in precedenza, è necessario prelevare i campioni in un certo ordine, la cosiddetta "suddivisione pari-dispari" o "un campione sì uno no". Nel caso delle 2-DFT, per esempio, la sequenza con cui si prelevano i campioni è $0, 4, 2, 6, 1, 5, 3, 7$. Questa sequenza sembra piuttosto casuale, cosa che renderebbe il recupero dei campioni piuttosto problematico. La cosa in realtà non è per niente casuale, come si potrebbe notare espandendo i numeri nelle loro rappresentazioni complesse:

$$000 - 100 - 010 - 110 - 001 - 101 - 011 - 111$$

ossia, "leggendo al contrario", invertendo le priorità dei bit (scambiare LSB con MSB e così via), è come contare da 0 a 7. Generare dunque un hardware in grado di realizzare questa sequenza è molto semplice: basta un contatore che abbia le uscite ordinate, cablate, in modo inverso.

3.3.3 Note sulla precisione

Un altro aspetto molto rilevante è quello della precisione: le operazioni di prodotto sono operazioni che richiedono un certo numero di bit per avere una rappresentazione corretta. Si può però notare un fatto: w_N, definito come un esponenziale complesso, ha sempre modulo pari a 1. Questo fatto ci dice che un generico processore butterfly dunque esegue un calcolo per cui, se a e b sono i valori di ingresso, c e d quelli di uscita, si avrà:

$$c = a + w_N^i b$$

$$d = a + w_N^j b$$

dove i e j sono dei coefficienti. Indipendentemente da essi, tuttavia, $|w_N| = 1$. Questo fatto ci porta a capire che la dinamica delle uscite della butterfly, nel caso peggiore, può incrementarsi solo di una unità. Si tenga ben presente che questi sono bit di dinamica, non di precisione.

3.3.4 DIT radix-4

Esistono diverse varianti alla DIT radix-2, sempre parlando di FFT: una variante è la DIF radix-2, ossia la decimazione in frequenza: un processo

assolutamente analogo, dove invece di raggruppare in una certa maniera i campioni nel tempo, si raggruppano quelli in frequenza.

Una cosa più interessante del radix-2 è il radix-4: l'idea è sostanzialmente la stessa ma, invece di dividere in 2 sequenze, si divide in 4 sequenze: non più pari o dispari, ma suddivisione in 4 gruppi invece che in 2 gruppi. Da una 8-DFT, per esempio, si salterà direttamente in una 2-DFT ($4 \times 2 = 8$). Dato ancora una volta il nostro esempio a 8 campioni, si avran, come sequenze: $0, 4$; $1, 5$; $2, 6$; $3, 7$.

Come si implementa, in pratica, questa cosa? Beh, si hanno 4 blocchetti 2-DFT, ciascuno con in ingresso una coppia di valori dunque, e ciascuno dà una coppia di coefficienti; per ottenere i coefficienti di 8 punti si dovrà fare una serie di collegamenti diversa dal butterfly, più complicata:

Come prima, anche in questo caso i nodi rappresentano dei sommatori; si hanno, per ciascun ramo, 4 somme; data la forma diversa, invece che di butterfly si parla di dragonfly.

La complessità della struttura, in questa implementazione della FFT, sostanzialmente aumenta, però ciò permette anche di ridurre il numero dei componenti da utilizzare: il fatto di raggruppare in stadi di questo tipo permette di ridurre il numero di stadi necessari per effettuare l'operazione a $\log_4 N$.

Questo genere di strutture a volte è utilizzato, ed esistono, sulla carta, altre varianti, come il radix-8; più gruppi si fanno, tuttavia, meno si apprezzano le migliorie in termini di componenti risparmiati, dunque non si va molto oltre il radix-4.