UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea:

APPROSSIMAZIONE DI FUNZIONI MEDIANTE L'USO DI HARDWARE DEDICATO E PROGETTO DI SISTEMA PER IL CALCOLO DEL RECIPROCO

Relatori:

Prof. Roberto Saletti______________________

Prof. Roberto Roncella___________________

Ing. Luca Fanucci______________________

Candidato:

Nicola Vincenti

____

Anno Accademico 2001/2002

INDICE

Pag. 3 **Introduzione**

Pag. 5 **Capitolo 1: Algoritmi per l'implementazione della divisione**

1.1 Classificazione degli algoritmi

1.2 Algoritmi che fanno uso di una look-up table

Pag. 28 **Capitolo 2: L'algoritmo per il calcolo della funzione** x^p

2.1 Formulazione del problema

2.2 Una premessa sulla rappresentazione dell'operando e del risultato

2.3 Schema di principio del circuito per il calcolo della funzione

2.4 Approssimazione della funzione

2.5 Stima dell'errore dovuto all'uso di "aritmetica finita"

Pag. 45 **Capitolo 3: Progetto del dispositivo divisore**

3.1 Sintesi dell'approssimatore e stima degli errori commessi in seguito al troncamento dei risultati parziali

3.2 Il tool di sviluppo

3.3 Descrizione dell'hardware in AHDL

Pag. 65 **Capitolo 4: Le prove sperimentali**

4.1 La procedura per la stima dell'errore

4.2 La sintesi della "macchina di test"

Pag. 77 **Conclusioni**

Pag. 80 **Appendice: Il software di simulazione**

Pag. 115 **Bibliografia**

INTRODUZIONE

È ormai evidente che l'elettronica digitale sia e sarà il cuore di molti sistemi in qualsiasi ambito, come ad esempio nelle centraline di controllo di apparecchi di uso domestico, o nelle macchine di ogni tipo utilizzate nella produzione industriale di ogni genere di bene di consumo. È infatti grazie alla riduzione delle geometrie dei transistori, siano essi bipolari o CMOS, che si è pervenuti al raggiungimento di notevolissime frequenze "di lavoro" ma anche a grandissime possibilità per il progettista di estendere le funzionalità della macchina, dotandola di hardware dedicato allo svolgimento di compiti che prima venivano demandati ad algoritmi implementati via software, guadagnando in termini di efficienza e funzionalità.

È in questo contesto che andremo quindi a studiare come sia possibile implementare funzionalità computazionali come la divisione o l'estrattore di radice che, sebbene storicamente trascurati, adesso richiamano maggior attenzione sia per l'ottimizzazione delle prestazioni di processori "general purpose", sia per la maggiore flessibilità d'uso nell'ambito dei DSP.

Sebbene il problema dell'implementazione della divisione sia stato affrontato da tempo, è solo recentemente che sono stati studiati alcuni algoritmi per effettuarla in maniera efficiente, economica, e rapida. La divisione è infatti un'operazione aritmetica che capita di dover effettuare molto meno frequentemente dell'addizione e della moltiplicazione, dunque si è spesso data scarsa importanza alla sua implementazione nelle Floating Point Unit di processori general-purpose. È stato altresì dimostrato che questo fatto può degradare le performance del sistema in questione; basti pensare alla notevole mole di calcoli che vengono svolti in applicazioni di tipo scientifico, oppure di rendering di immagini.

Per quanto riguarda poi il settore dei DSP (Digital Signal Processors) la disponibilità di tecnologie sempre più spinte, l'alta integrazione e il basso costo suggeriscono lo studio e l'implementazione di hardware dedicato, oltre che alle tradizionali operazioni di somma e di moltiplicazione, ad altre operazioni elementari, come la divisione, il calcolo della radice quadrata, del reciproco della radice per la normalizzazione di vettori, dell'arcotangente e così via.

Queste operazioni, inserite poi all'interno dell'architettura di un DSP risulteranno disponibili all'utilizzatore finale ed aggiungeranno quindi ad esso notevole flessibilità d'uso ed efficienza.

Ci occuperemo, nella prima parte di questa tesi, dello studio dei metodi di calcolo della divisione, e dello sviluppo di un algoritmo per il calcolo del valore di una funzione generica mediante un approssimazione polinomiale. Per tale funzione faremo l'ipotesi di continuità, derivabilità, e di convessità. In tal modo potremo implementare in maniera semplice un algoritmo basato sulla teoria di Chebycev per la determinazione del polinomio di migliore approssimazione uniforme.

Nella seconda parte proporremo una sintesi del sistema approssimante su un dispositivo programmabile. Questa scelta ci darà la possibilità di ottenere dei risultati sperimentali in tempi estremamente brevi e con costi molto contenuti. Potrà dunque essere oggetto di un successivo studio la trasposizione dello stesso progetto su tecnologie differenti e/o lo sviluppo di una diversa architettura per il calcolo del valore del polinomio.

CAPITOLO 1

Algoritmi per l'implementazione della divisione

1.1 Classificazione degli algoritmi

Numerosi sono gli algoritmi che sono stati sviluppati per la sintesi dell'operazione aritmetica di divisione e possono essere raggruppati in 5 grandi categorie:

- estrazione ricorsiva di una singola cifra per volta
- convergenza iterativa
- uso di "look-up table"
- a latenza variabile
- very high radix

In realtà poi, nel momento in cui si sintetizza un dispositivo per la divisione, si possono anche usare tecniche ibride di ciascuno di questi, per esempio ottenere una prima approssimazione del risultato mediante look-up table, e poi raffinare ancora il risultato ottenuto mediante un algoritmo di convergenza iterativa.

Esaminiamo adesso, brevemente, ciascuna di queste categorie; in seguito ne sceglieremo una tramite la quale implementeremo il nostro dispositivo.

L'algoritmo che storicamente è stato introdotto per primo è la procedura ricorsiva di estrazione di una cifra per volta, ed è la diretta implementazione della divisione come ci viene insegnata nei primi anni di scuola. Descriviamone adesso il principio di funzionamento e mostriamo alcune tecniche di calcolo che usano questo metodo.

Il principio di funzionamento di questo algoritmo può essere ricondotto alla seguente formula iterativa

$$R^{(j+1)} = r \cdot R^{(j)} - q_{j+1} \cdot D$$

dove $R^{(j)}$ è il resto al passo j-esimo

r è il numero di cifre dell'alfabeto

q_j è la cifra j-esima del quoziente

D è il divisore

Per semplificare la trattazione consideriamo soltanto il caso in cui gli operandi siano entrambi positivi. Assumiamo anche, senza perdere di generalità, che gli operandi siano normalizzati (cioè minori di 1 in modulo) e che il dividendo sia minore del divisore. Queste condizioni si possono ottenere traslando opportunamente gli operandi e, alla fine, tenerne conto per esprimere il risultato nella forma originale. Poniamo quindi

$$\begin{aligned} Q &= .q_1 q_2 .....q_n \\ D &= .d_1 d_2 .....d_n \quad , \\ R^{(0)} &= .r_1^{(0)} r_2^{(0)} ......r_n^{(0)} \end{aligned}$$

dove Q, D, $R^{(0)}$ sono rispettivamente il quoziente, il divisore e il dividendo con la condizione $R^{(0)} < D$.

Verifichiamo anzitutto che la formula iterativa produce il quoziente richiesto su un numero n fissato di cifre e il corrispondente resto. Si ha

$$\begin{aligned} j = 0, \quad & R^{(1)} = r \cdot R^{(0)} - q_1 \cdot D \\ j = 1, \quad & R^{(2)} = r \cdot R^{(1)} - q_2 \cdot D = r \cdot \left[r \cdot R^{(0)} - q_1 \cdot D \right] - q_2 \cdot D = \\ & = r^2 \cdot R^{(0)} - \left[r \cdot q_1 + q_2 \right] \cdot D \\ \ldots\ldots\ldots \\ j = n, \quad & R^{(n)} = r^n \cdot R^{(0)} - \left(r^{n-1} \cdot q_1 + r^{n-2} \cdot q_2 + ... + r \cdot q_{n-1} + q_n \right) \cdot D \end{aligned}$$

e dunque

$$\frac{R^{(0)}}{D} = r^{-1} \cdot q_1 + r^{-2} \cdot q_2 + \ldots\ldots + r^{-(n-1)} \cdot q_{n-1} + r^{-n} \cdot q_n + \frac{R^{(n)} \cdot r^{-n}}{D}$$

dove sono facilmente riconoscibili il quoziente Q e il resto R

$$Q = \sum_{j=1}^{n} q_j \cdot r^{-j} \quad , \qquad R = r^{-n} \cdot R^{(n)}$$

Stabilendo il criterio con cui vengono determinati il resto $R^{(j)}$ e la cifra quoziente q_j, il procedimento è determinato. È sulla base di questo criterio che si differenziano gli algoritmi appartenenti a questa classe. Il primo che proviamo a descrivere è il più semplice e intuitivo, e prende il nome di *Conventional Restoring Division*.

Sia $r = 2$ e quindi $S = \{0,1\}$ l'alfabeto per la rappresentazione dei numeri. La formula iterativa diventa

$$R^{(j+1)} = 2 \cdot R^{(j)} - q_{j+1} \cdot D$$

Il criterio di cui si parla consiste nell'assumere la condizione

$$0 \le R^{(j+1)} < D\,.$$

Si ottiene, allora la seguente regola per la scelta della cifra quoziente

$$q_{j+1} = \begin{cases} 0 & \text{se } 2 \cdot R^{(j)} < D \\ 1 & \text{se } 2 \cdot R^{(j)} \ge D \end{cases}.$$

Operativamente si suppone $q_{j+1} = 1$ ad ogni iterazione, e si calcola il resto stimato $\hat{R}^{(j+1)}$ mediante una sottrazione,

$$\hat{R}^{(j+1)} = 2 \cdot R^{(j)} - D$$

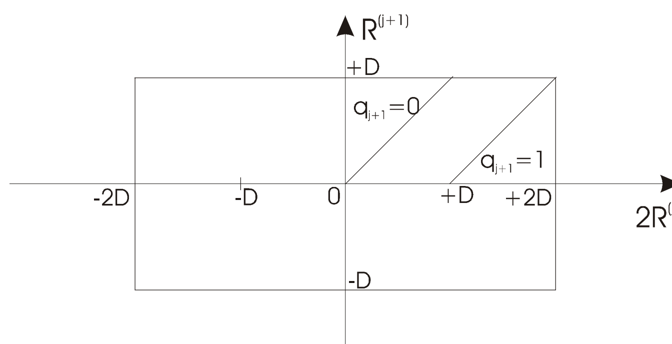
Se il resto stimato è positivo, allora

$$R^{(j+1)} = \hat{R}^{(j+1)}.$$

Se invece è negativo, dobbiamo ripristinare il valore del resto, cioè sommare di nuovo il divisore D a $\hat{R}^{(j+1)}$, ottenendo

$$R^{(j+1)} = \hat{R}^{(j+1)} + D = 2 \cdot R^{(j)}$$

Per questo motivo l'algoritmo prende il nome di Restoring Division. È da notare, comunque, che mediante l'uso di un registro e di un multiplexer l'operazione di somma può essere evitata. Il diagramma di Robertson, riportato nella figura seguente, rappresenta graficamente il criterio di scelta del resto (j+1)-esimo $R^{(j+1)}$ e della cifra quoziente q_{j+1} al variare del dividendo parziale $2 \cdot R^{(j)}$.



Svolgiamo adesso i calcoli dell'operazione di divisione in un caso particolare. Rifaremo poi lo stesso esempio per gli altri algoritmi di questa stessa categoria. Si voglia calcolare il quoziente su 4 cifre e il resto della divisione $\frac{A}{B}$ con

$$A = 0\,.\,0\,1\,0\,1\,0\,1\,1\,0$$

$$B = 0\,.\,1\,1\,0\,0$$

$$R^{(0)} = A = 0\,.\,0\,1\,0\,1\,0\,1\,1\,0$$

$$2 \cdot R^{(0)} = 0\ .\ 1\ 0\ 1\ 0\ 1\ 1\ 0$$
$$D = 0\ .\ 1\ 1\ 0\ 0$$
$$\hat{R}^{(1)} = 2 \cdot R^{(0)} - D = 1\ .\ 1\ 1\ 1\ 0\ 1\ 1\ 0 \qquad \text{Negativo} \Rightarrow R^{(1)} = 2 \cdot R^{(0)},\ q_1 = 0$$

$$2 \cdot R^{(1)} = 1\ .\ 0\ 1\ 0\ 1\ 1\ 0$$
$$D = 0\ .\ 1\ 1\ 0\ 0$$
$$\hat{R}^{(2)} = 2 \cdot R^{(1)} - D = 0\ .\ 1\ 0\ 0\ 1\ 1\ 0 \qquad \text{Positivo} \Rightarrow R^{(2)} = \hat{R}^{(2)},\ q_2 = 1$$

$$2 \cdot R^{(2)} = 1\ .\ 0\ 0\ 1\ 1\ 0$$
$$D = 0\ .\ 1\ 1\ 0\ 0$$
$$\hat{R}^{(3)} = 2 \cdot R^{(2)} - D = 0\ .\ 0\ 1\ 1\ 1\ 0 \qquad \text{Positivo} \Rightarrow R^{(3)} = \hat{R}^{(3)},\ q_3 = 1$$

$$2 \cdot R^{(3)} = 0\ .\ 1\ 1\ 1\ 0$$
$$D = 0\ .\ 1\ 1\ 0\ 0$$
$$\hat{R}^{(4)} = 2 \cdot R^{(3)} - D = 0\ .\ 0\ 0\ 1\ 0 \qquad \text{Positivo} \Rightarrow R^{(4)} = \hat{R}^{(4)},\ q_4 = 1$$

Si ha

$$\frac{A}{B} = 0.0111 + \frac{2^{-4} \cdot 0.0010}{0.1100}\ ,$$

cioè

$$Q = 0.0111 \qquad R = 0.00000010$$

Si nota che per ogni iterazione si deve valutare la differenza $2 \cdot R^{(j)} - D$, e successivamente, mediamente una volta su 2, effettuare anche una somma di ripristino.

Per ottenere un risparmio in termini di tempo sono state fatte alcune modifiche al procedimento. Si è passati da un alfabeto convenzionale a un alfabeto di cifre con segno. Accenniamo alla sua definizione, tralasciando di descriverne tutte le proprietà. Sia S l'insieme delle cifre che lo costituiscono,

$$S = \left\{ \overline{1}, 0, 1 \right\}$$

Vale, analogamente al caso di cifre senza segno, la seguente relazione tra un numero intero N e le cifre della sua rappresentazione:

$$N = \sum_{i=0}^{k} n_i \cdot 2^i$$

dove le cifre n_i possono valere –1, 0, 1 a seconda che siano, rispettivamente, $\overline{1}, 0, 1$. La prima conseguenza di questa scelta è che un numero intero non ha una univoca rappresentazione. Per esempio per N=3 su 3 cifre otteniamo le possibili rappresentazioni

$$\begin{array}{ll} 011 & 1+2=3 \\ 10\overline{1} & 4-1=3 \\ 1\overline{1}1 & 4-2+1=3 \end{array}$$

Vale la pena notare che il segno di N è determinato dal segno della prima cifra diversa da zero. Notiamo anche che l'opposto di N si può ottenere semplicemente cambiando il segno delle cifre di una sua rappresentazione.

Il metodo della "divisione senza ripristino" (Non Restoring Division) fa uso di un alfabeto di cifre con segno per il quoziente, mentre il resto viene rappresentato nel modo convenzionale.

Limitiamo le cifre q_j del quoziente all'insieme $\{\overline{1}, 1\}$ e il criterio che useremo per la scelta di q_{j+1} e $R^{(j+1)}$ consegue alla condizione

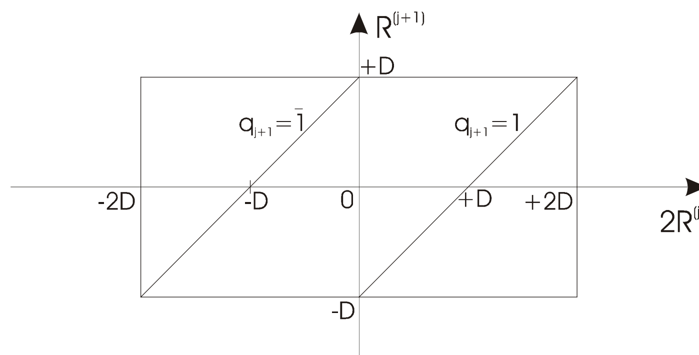
$$\left| R^{(j+1)} \right| < D$$

Il criterio di scelta è allora il seguente

$$R^{(j+1)} = \begin{cases} 2 \cdot R^{(j)} - D & \text{se } 2 \cdot R^{(j)} > 0 \\ 2 \cdot R^{(j)} + D & \text{se } 2 \cdot R^{(j)} < 0 \end{cases}$$

$$q_{j+1} = \begin{cases} 1, & \text{se } 0 < 2 \cdot R^{(j)} < 2 \cdot D \\ \overline{1}, & \text{se } -2 \cdot D < 2 \cdot R^{(j)} < 0 \end{cases}$$

se poi $2 \cdot R^{(j)} = 0$, il resto è nullo e l'operazione viene terminata. Il corrispondente diagramma di Robertson è mostrato in figura.



Eseguiamo ancora la divisione $\frac{A}{B}$ con questa tecnica.

$$
\begin{array}{rl}
A = & 0\ .\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0 \\
B = & 0\ .\ 1\ 1\ 0\ 0 \\
R^{(0)} = A = & 0\ .\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0 \quad \text{Positivo} \Rightarrow q_1 = 1
\end{array}
$$

$$
\begin{array}{rl}
2 \cdot R^{(0)} = & 0\ .\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ - \\
D = & 0\ .\ 1\ 1\ 0\ 0 \\
R^{(1)} = & 1\ .\ 1\ 1\ 1\ 0\ 1\ 1\ 0 \quad \text{Negativo} \Rightarrow q_2 = \overline{1}
\end{array}
$$

$$
\begin{array}{rl}
2 \cdot R^{(1)} = & 1\ .\ 1\ 1\ 0\ 1\ 1\ 0\ + \\
D = & 0\ .\ 1\ 1\ 0\ 0 \\
R^{(2)} = & 0\ .\ 1\ 0\ 0\ 1\ 1\ 0 \quad \text{Positivo} \Rightarrow q_3 = 1
\end{array}
$$

$$
\begin{array}{rl}
2 \cdot R^{(2)} = & 1\ .\ 0\ 0\ 1\ 1\ 0\ - \\
D = & 0\ .\ 1\ 1\ 0\ 0 \\
R^{(3)} = & 0\ .\ 0\ 1\ 1\ 1\ 0 \quad \text{Positivo} \Rightarrow q_4 = 1
\end{array}
$$

$$
\begin{array}{rl}
2 \cdot R^{(3)} = & 0\ .\ 1\ 1\ 1\ 0\ - \\
D = & 0\ .\ 1\ 1\ 0\ 0 \\
R^{(4)} = & 0\ .\ 0\ 0\ 1\ 0
\end{array}
$$

Abbiamo ottenuto

$$\frac{A}{B} = 0.1\bar{1}11 + \frac{0.0010 \cdot 2^{-4}}{0.1100} \quad .$$

Per la conversione del quoziente nella forma convenzionale si può operare come segue. Osservando che il numero rappresentato mediante una sequenza di cifre positive e negative può essere scomposto in due numeri, il primo di sole cifre positive e il secondo di sole cifre negative, si può cambiare di segno quello negativo ed effettuare una sottrazione convenzionale. Si ha

$$Q = 0.1\bar{1}11 = 0.1011 + 0.0\bar{1}00 = 0.1011 - 0.0100 = 0.0111,$$

mentre R è già, in questo caso, nella forma convenzionale. Nel caso in cui l'ultimo resto $R^{(n)}$ sia negativo si deve sommare di nuovo a questo il divisore, e modificare il quoziente di conseguenza (cioè sottrarre a questo 2^{-n}).

Il vantaggio di questa tecnica è che si effettua una somma oppure una sottrazione, ma non si deve mai ripristinare il valore del dividendo parziale $2 \cdot R^{(j)}$ (eccetto il caso in cui l'ultimo resto sia negativo).

Alla fine degli anni '50 Sweeney, Robertson e Tocher hanno proposto, ciascuno indipendentemente, un miglioramento significativo della procedura. Dalle iniziali di coloro che l'hanno proposto, i divisori implementati con questa tecnica prendono il nome di SRT.

Si prende di nuovo un alfabeto di cifre con segno

$$S = \{ \bar{1}, 0, 1 \}$$

ma si ammette questa volta anche lo 0 come possibile cifra del quoziente. Inoltre si normalizzano gli operandi D e $R^{(0)}$ in modo che sia

$$\frac{1}{2} \le \left| D \right| < 1$$

$$\frac{1}{2} \le \left| 2 \cdot R^{(0)} \right| < 1$$

Si può adottare il seguente criterio per $R^{(j+1)}$ e q_{j+1}

$$R^{(j+1)} = \begin{cases} 2 \cdot R^{(j)} + D & \text{se } 2 \cdot R^{(j)} < -D \\ 2 \cdot R^{(j)} & \text{se } -D \le 2 \cdot R^{(j)} \le D \\ 2 \cdot R^{(j)} - D & \text{se } 2 \cdot R^{(j)} > D \end{cases}$$

$$q_{j+1} = \begin{cases} \overline{1} & \text{se } 2 \cdot R^{(j)} < -D \\ 0 & \text{se } -D \le 2 \cdot R^{(j)} \le D \\ 1 & \text{se } 2 \cdot R^{(j)} > D \end{cases}$$

Il miglioramento dei due precedenti metodi può essere ottenuto osservando che non è necessario il confronto del dividendo parziale $2 \cdot R^{(j)}$ con le quantità D o $-D$. Si può cioè modificare il criterio confrontando $2 \cdot R^{(j)}$ con le quantità $\frac{1}{2}$ e $-\frac{1}{2}$: operazione assai più rapida e di facile implementazione in un sistema binario. Si ha dunque per $R^{(j+1)}$ e q_{j+1} il seguente criterio modificato

$$R^{(j+1)} = \begin{cases} 2 \cdot R^{(j)} + D & \text{se } 2 \cdot R^{(j)} < -\frac{1}{2} \\ 2 \cdot R^{(j)} & \text{se } -\frac{1}{2} \le 2 \cdot R^{(j)} < \frac{1}{2} \\ 2 \cdot R^{(j)} - D & \text{se } \frac{1}{2} \le 2 \cdot R^{(j)} \end{cases}$$

$$q_{j+1} = \begin{cases} \overline{1} & \text{se } 2 \cdot R^{(j)} < -\frac{1}{2} \\ 0 & \text{se } -\frac{1}{2} \le 2 \cdot R^{(j)} < \frac{1}{2} \\ 1 & \text{se } \frac{1}{2} \le 2 \cdot R^{(j)} \end{cases}$$

Il vantaggio di questa scelta è che quando $q_{j+1} = 0$ l'operazione di estrazione della cifra consiste solo di un confronto e di una traslazione, ed entrambe le operazioni sono veloci rispetto alla somma/sottrazione. Nel momento in cui si hanno più cifre consecutive del quoziente pari a zero, in una unica iterazione dell'algoritmo si ottengono più cifre del quoziente. Questo porta in definitiva a un risparmio considerevole di tempo. Rieseguiamo l'operazione $\frac{A}{B}$ in quest'ultimo caso.

$$
\begin{aligned}
A &= 0\,.\,0\,1\,0\,1\,0\,1\,1\,0 \\
B &= 0\,.\,1\,1\,0\,0 \\
R^{(0)} = A &= 0\,.\,0\,1\,0\,1\,0\,1\,1\,0
\end{aligned}
$$

$$
\begin{aligned}
2 \cdot R^{(0)} &= 0\,.\,1\,0\,1\,0\,1\,1\,0\ - \qquad 2 \cdot R^{(0)} \geq \frac{1}{2} \Rightarrow q_1 = 1 \\
D &= 0\,.\,1\,1\,0\,0 \\
R^{(1)} &= 1\,.\,1\,1\,1\,0\,1\,1\,0
\end{aligned}
$$

$$2 \cdot R^{(1)} = 1\,.\,1\,1\,0\,1\,1\,0$$

Vengono eliminati i bit più significativi fintanto che non si ha due bit consecutivi diversi. Per ogni eliminazione si ottiene una cifra uguale a 0.

$q_2 = 0, q_3 = 0$

$$
\begin{aligned}
2 \cdot R^{(3)} &= 1\,.\,0\,1\,1\,0\ + \qquad 2 \cdot R^{(3)} < -\frac{1}{2} \Rightarrow q_4 = \overline{1} \\
D &= 0\,.\,1\,1\,0\,0 \\
R^{(4)} &= 0\,.\,0\,0\,1\,0
\end{aligned}
$$

Si ottiene, per la divisione richiesta

$$\frac{A}{B} = 0.100\overline{1} + \frac{0.0010 \cdot 2^{-4}}{0.1100}$$

$$Q = 0.100\overline{1} = 0.1000 + 0.000\overline{1} = 0.1000 - 0.0001 = 0.0111$$

Numerosi altri autori nel corso degli anni hanno perfezionato queste tecniche. Rimane da osservare in ogni modo che il principale vantaggio di questi metodi risiede nella loro economicità, dato che si utilizza ripetitivamente lo stesso hardware per il calcolo delle cifre del quoziente. E, sebbene questo sia il vantaggio principale, ne è allo stesso tempo anche l'inconveniente maggiore poiché questi algoritmi non risultano performanti in termini di throughput e latenza.

Un'altra classe di algoritmi per la divisione è quella che fa uso di metodi iterativi (functional iteration methods). Uno di questi è il noto metodo delle tangenti per il calcolo della soluzione di una equazione in una incognita.

Data l'equazione

$$f(x) = \frac{1}{x} - b = 0\,;$$

applicando il metodo delle tangenti, usiamo la formula iterativa

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}\ ,$$

quindi

$$x_{i+1} = x_i + \frac{\frac{1}{x_i} - b}{\frac{1}{x_i^2}} = x_i \cdot (2 - b \cdot x_i)\,.$$

Partendo da un punto vicino (seme) e reiterando la procedura si ottiene la convergenza verso la soluzione cercata $x = \frac{1}{b}$.

Un vantaggio di questo procedimento è la convergenza quadratica verso la soluzione, infatti mentre prima si estraeva una cifra per volta, e quindi ad ogni iterazione un numero di bit fissato (linearità), adesso si raddoppia ad ogni iterazione il numero di bit esatti del risultato. Questo algoritmo è quindi utile in particolare come successivo stadio per raggiungere elevata precisione nel calcolo del risultato, mentre il fatto che sia itera-

tivo e contemporaneamente utilizzi 2 moltiplicazioni e una sottrazione lo rende non i-doneo per divisori di basso costo e alta velocità.

Un altro algoritmo appartenente a questa stessa categoria è conosciuto con il nome di Godschmidt's Algorithm.

Ricordando che la serie geometrica di ragione $-y$ converge, per $|y| < 1$ alla funzione $g(y) = \frac{1}{1+y}$, si ha

$$g(y) = 1 - y + y^2 - y^3 + y^4 - .....$$

che, nella forma fattorizzata, diventa

$$g(y) = (1 - y) \cdot (1 + y^2) \cdot (1 + y^4) \cdot ....$$

Per avere $g(y) = \frac{1}{b}$ deve essere $y = b - 1$; normalizzando b all'intervallo [0.5,1[si ha $-0.5 \leq y < 0$; in questo modo, quindi, viene assicurata una più rapida convergenza della serie in esame.

Per calcolare la divisione $\frac{a}{b}$ sfruttiamo le proprietà della serie appena enunciata e procediamo nel seguente modo. Iniziamo ponendo

$$\begin{aligned} N_0 &= a \\ D_0 &= b \\ R_0 &= 1 - y = 2 - b \end{aligned}$$

ed eseguiamo la formula iterativa

$$\begin{aligned} N_i &= R_{i-1} \cdot N_{i-1} \\ D_i &= R_{i-1} \cdot D_{i-1} \\ R_i &= 2 - D_i \end{aligned}$$

In tal modo all'i-esimo passo abbiamo che la quantità $q_i = \frac{N_i}{D_i}$ vale

$$q_i = \frac{a \cdot (1-y) \cdot (1+y^2) \cdot (1+y^4) \cdot ..... \cdot (1+y^{2^{(i-1)}})}{1-y^{2^i}};$$

allora, mentre il denominatore di questa frazione converge ad 1, il numeratore N_i converge al valore cercato.

Si può osservare che, dal punto di vista matematico, il metodo prima esposto e questo danno gli stessi risultati, iterazione per iterazione. Esiste tuttavia una differenza nel modo in cui i risultati vengono calcolati. Mentre nel metodo delle tangenti il risultato della prima moltiplicazione viene introdotto come operando nella seconda, nel secondo caso i due moltiplicatori lavorano ciascuno indipendentemente sui propri operandi, e quindi si può ottenere qualche vantaggio in termini di latenza del dispositivo di calcolo. Come svantaggio rispetto a quello precedentemente mostrato si osserva che mentre il primo corregge automaticamente un errore di calcolo nella i-esima iterazione, nel secondo dobbiamo utilizzare moltiplicatori su un numero di bit lievemente maggiore per garantire la stessa precisione, poiché gli errori di troncamento o arrotondamento non vengono recuperati alla successiva iterazione [1].

Per quanto riguarda la classe di algoritmi "very high radix" ci limitiamo a dire che sono una estensione di quelli che estraggono una cifra per volta. Infatti la loro convenienza consiste nello scegliere un alfabeto molto esteso per la rappresentazione dei numeri. Quindi quando viene ricorsivamente estratta una cifra del quoziente si ottengono diversi bit corretti del risultato. Il suo punto a sfavore è la complessità dell'hardware necessario a stabilire il valore della cifra del quoziente (in letteratura Quotient Digit Selection Function). Sono in ogni caso algoritmi competitivi in termini di efficienza e restituiscono inoltre il valore del resto (partial remainder).

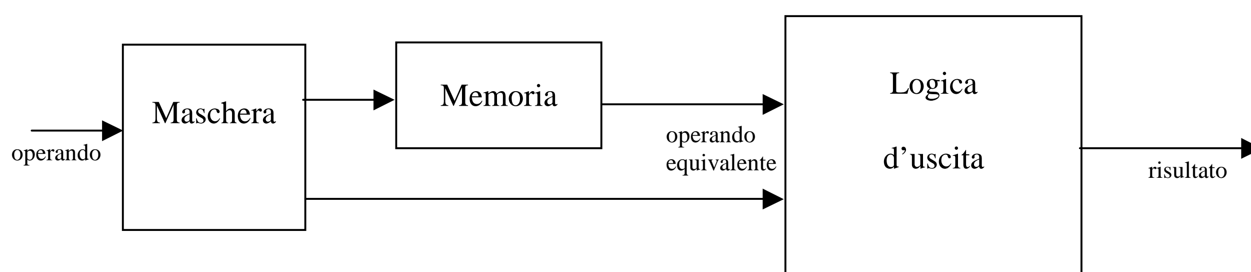
Prima di affrontare la categoria degli algoritmi che fanno uso di look-up table citiamo quelli a latenza variabile. Mentre quelli fin adesso visti calcolano il valore della divisione in un numero di cicli noto a priori, in quelli a latenza variabile, come il nome stesso suggerisce, si cerca di minimizzare il tempo medio di esecuzione dell'operazione aritmetica. Se si prende questo tempo come parametro di riferimento si scopre che que-

sti algoritmi sono assai competitivi con quelli finora enunciati. Dobbiamo però osservare che la maggioranza dei processori non beneficiano di un comportamento di questo tipo. Gli svantaggi di questa tecnica consistono nella complessità della logica per la gestione del clock e per la determinazione dell'evento "risultato pronto" (self-timing device); ed inoltre la progettazione di un dispositivo a latenza variabile ha impatto anche sull'architettura del processore che fa uso di questo stesso dispositivo.

1.2 Algoritmi che fanno uso di una look-up table

Cercheremo, adesso, di descrivere alcuni degli algoritmi che fanno uso di una look-up table, ne sceglieremo poi uno, mediante il quale implementeremo il nostro dispositivo.

L'uso di look-up table per la restituzione del risultato prevede generalmente la mappatura dell'operando di ingresso in un insieme di valori di un operando equivalente in uscita da una memoria. Successivamente su questo operando equivalente vengono effettuate alcune elaborazioni che producono infine il risultato voluto.



Nel caso più semplice la logica d'uscita è un corto circuito, e in memoria verrà quindi registrato, per ogni possibile ingresso, direttamente il risultato da restituire. Sebbene questo approccio permetta di raggiungere la più bassa latenza possibile, risulta, tutto sommato, poco efficiente per il fatto che si portano tutti i bit dell'operando in ingresso al bus degli indirizzi della memoria, e già per un operando su 16 bit si ottengono ben 64K righe di risultati da restituire.

Varie sono le tecniche adottate per manipolare l'informazione in ingresso in modo che con poca memoria si possa attraverso la logica d'uscita estrapolare il risultato corretto. È altresì da osservare che generalmente quanto più si risparmia sulla memoria, tanto più dovremo investire nella logica per il calcolo del risultato, spendendo soprattutto in termini di tempo impiegato, ma anche in termini di area.

Presentiamo ora alcune di queste tecniche, ordinate per valori crescenti del risparmio in bit della look-up table.

Das Sarma e Matula [2] presentano un metodo (Faithful bipartite rom reciprocal tables) per la compressione di tabelle nelle quali è memorizzato il reciproco dell'operando normalizzato $1 \le x < 2$. L'idea che sta alla base del loro lavoro è quella di produrre una interpolazione lineare senza far uso di moltiplicatori e addizionatori. Per far ciò si utilizzano due tabelle, una positiva e l'altra negativa, in cui vengono memorizzati un contributo di ordine 0 (quella positiva) e un contributo del primo ordine (quella negativa). Allora, dividendo l'operando X in tre parti, che vengono chiamate x_h, x_m, x_l, si conviene che x_h, x_m indirizzino la tabella positiva e che x_h, x_l indirizzino la tabella negativa. Se pensiamo a una generica approssimazione lineare della funzione $\frac{1}{x}$ abbiamo

$$\frac{1}{x_0 + \Delta x} \approx c_0(x_0) + c_1(x_0) \cdot \Delta x$$

dove c_0 è positivo e c_1 è negativo. Possiamo perciò memorizzare c_0 in una tabella positiva, e direttamente il prodotto $c_1 \cdot \Delta x$ in una tabella negativa, evitando l'uso di un moltiplicatore. Questo approccio evidentemente utilizza molta memoria, però ha una latenza molto bassa, ed è indirizzato alla costruzione di divisori di floating point unit ad alte prestazioni. Si potrebbe osservare che rimane comunque una addizione (sottrazione) che deve essere implementata.. In realtà ciò non viene fatto perché lo schema di una generica floating point unit prevede che dopo la ROM vengano utilizzati dei metodi iterativi che fanno uso di moltiplicatori, i quali per essere performanti, di norma utilizzano una codifica ridondante in cui un numero viene rappresentato con cifre con segno [3] (Signed-Digit Number Systems) e quindi i risultati in uscita dalle due tabelle vengono portati direttamente al moltiplicatore così come sono.

Detto m il numero di bit sui cui viene rappresentato il reciproco, allora, nell'articolo citato [2] viene mostrato che lo spazio di memoria in bit necessario per l'implementazione di questa tecnica è dato da

$$\text{N. bit} = 2^{\frac{2m}{3}} \cdot m + 2^{\frac{2m}{3}} \cdot \frac{m}{3}$$

Un miglioramento di questa tecnica è l'SBTM (Simmetric Bipartite Table Method), secondo la quale un ulteriore risparmio, senza compromettere la bassa latenza di questo stadio approssimatore, può essere ottenuto tenendo conto di alcune proprietà di simmetria della tabella negativa, grazie alle quali si può dimezzarne la dimensione, ottenendo

$$\text{N. bit} = 2^{\frac{2m}{3}} \cdot m + 2^{\frac{2m}{3}-1} \cdot \frac{m}{3}$$

Dato il notevole spazio di memoria occupato, i due metodi appena enunciati vengono utilizzati qualora le specifiche sulla latenza del divisore siano veramente stringenti, e qualora sia presente a valle un moltiplicatore con operandi rappresentati in codifica ridondante.

Esaminiamo adesso altre tecniche che permettono di risparmiare molto in termini di area della look-up table e possono essere utilizzati per un insieme più esteso di funzioni elementari. Tra queste ci occuperemo in particolare della funzione $X^p, p \in \mathbf{Q}$.

Una di queste tecniche, riportata da Takagi in [4], peraltro assai diffusa, è basata sull'uso dello sviluppo in serie di Taylor arrestato al primo ordine. Se X su m bit è un operando normalizzato all'intervallo [1,2[, prendendo k bit per la parte alta X_1 e m-k bit per la parte bassa X_2 abbiamo

$$\begin{aligned} X_1 &= 1.x_1 x_2 \dots x_k = 1 + \sum_{i=1}^{k} x_i \cdot 2^{-i} \\ X_2 &= .x_{k+1} x_{k+2} \dots x_m = \sum_{i=k+1}^{m} x_i \cdot 2^{-i} \end{aligned} \quad , \quad X = X_1 + X_2$$

Prenderemo i k bit di X_1 per indirizzare la tabella, la distanza da X_1 del punto in cui vogliamo calcolare la funzione sarà rappresentata dai rimanenti m-k bit di X_2. Sviluppando la funzione nel punto di mezzo tra X_1 e $X_1 + 2^{-k}$ si ottiene

$$X^p \approx (X_1 + 2^{-k-1})^p + p \cdot (X_1 + 2^{-k-1})^{p-1} \cdot (X_2 - 2^{-k-1}) .$$

Si può riscrivere l'espressione separando la parte dipendente da X_1 da quella dipendente da X_2 ottenendo

$$X^p \approx \underbrace{(X_1 + 2^{-k-1})^p - p \cdot (X_1 + 2^{-k-1})^{p-1} \cdot 2^{-k-1}}_{c_0} + \underbrace{p \cdot (X_1 + 2^{-k-1})^{p-1}}_{c_1} \cdot X_2$$

Si memorizzano quindi i coefficienti c_0 e c_1 in una look-up table, e una volta estratti, con l'uso di un moltiplicatore/sommatore, si ottiene il risultato in uscita. Per avere una stima dell'errore valutiamo il termine di secondo ordine. Avendosi

$$\varepsilon_{approx} = \frac{p \cdot (p-1) \cdot (X_1 + 2^{-k-1})^{p-2}}{2} \cdot (X_2 - 2^{-k-1})^2 + \sum_{j>2} O(j) ,$$

trascurando i termini di ordine superiore a 2, potremo dire che il numero n di bit esatti del risultato è dato da

$$n = -\log_2 |\varepsilon|_{max} = 1 - \log_2 |p| - \log_2 |p-1| - \max\{0, p-2\} + 2k + 2$$

avendo osservato che

$$(X_1 + 2^{-k-1})^{p-2} < 1 \qquad \text{se } p-2 < 0$$
$$(X_1 + 2^{-k-1})^{p-2} < 2^{p-2} \quad \text{se } p-2 > 0$$

e che

$$\left|X_2 - 2^{-k-1}\right| < 2^{-k-1}$$

Per il caso particolare $p = -1$ si ottiene $n = 2k + 2$, e volendo in uscita dall'approssimatore m bit esatti del risultato, si ha

$$2k + 2 = m + 1 \Leftrightarrow k = \frac{m-1}{2}$$

ottenendo una look-up table di circa $2^{\frac{m}{2}} \cdot \left(m + \frac{m}{2}\right)$ bit, dove si rappresenta c_0 su m bit e c_1 su $\frac{m}{2}$ bit.

In un altro metodo, che fa uso di un solo moltiplicatore, proposto ancora da Takagi in [4], si osserva anzitutto che lo sviluppo in serie di Taylor arrestato al primo ordine può essere riscritto come

$$X^p \approx \underbrace{(X_1 + 2^{-k-1})^{p-1}}_{c} \cdot \left[\underbrace{(X_1 + 2^{-k-1} + p \cdot (X_2 - 2^{-k-1})}_{X'}\right]$$

dove X' prende il nome di operando modificato in quanto si ottiene con semplici operazioni dall'operando di ingresso X. Si suppone, ora, che l'esponente razionale p sia del tipo

$$p = \pm 2^z \text{, z intero, oppure } p = \pm 2^{z_1} \pm 2^{-z_2}\text{, } z_1 \text{ intero, } z_2 \text{ intero non negativo.}$$

Allora, per $p = -2^0 = -1$ si ha

$$\frac{1}{X} \cong (X_1 + 2^{-k-1})^{-2} \cdot \left[X_1 + 2^{-k-1} - X_2 + 2^{-k-1}\right] = (X_1 + 2^{-k-1})^{-2} \cdot \left[X_1 + 2^{-k} - X_2\right];$$

e il fattore $X_1 + 2^{-k} - X_2$ si può ottenere semplicemente complementando la parte bassa X_2. In tal modo l'operando X può essere modificato in X' a basso costo e senza introdurre ritardi rilevanti.

Analoghe semplificazioni sono ottenute in [4] per gli altri valori di p sempre del tipo suddetto.

Ritornando al caso generale per quanto riguarda l'analisi dell'errore, si può osservare che, se il coefficiente c viene leggermente modificato in

$$c' = c + p \cdot (p-1) \cdot 2^{-1} \cdot X_1^{p-3} \cdot 2^{-2k-3} = (X_1 + 2^{-k-1})^{p-1} + p \cdot (p-1) \cdot 2^{-2k-4} \cdot X_1^{p-3}$$

l'errore diminuisce e può essere approssimato mediante la formula

$$\varepsilon = X' p \cdot (p-1) \cdot 2^{-1} \cdot X_1^{p-3} \cdot \left[2^{-2k-3} - (X_2 - 2^{-k-1})^2\right] =$$

$$= \left[X_1 + 2^{-k-1} + p \cdot (X_2 - 2^{-k-1})\right] \cdot p \cdot (p-1) \cdot 2^{-1} \cdot X_1^{p-3} \cdot \left[2^{-2k-3} - (X_2 - 2^{-k-1})^2\right]$$

quindi

$$|\varepsilon|_{max} = |p| \cdot |p-1| \cdot 2^{-2k-3} \text{ quando } p-3<0.$$

Tornando al caso particolare $p = -1$ si ha

$$|\varepsilon|_{max} = 2^{-2k-2}.$$

Assumendo 2k+1 bit esatti nel risultato (rounded to nearest) si trova che per avere m bit esatti dobbiamo prendere $k = \frac{m}{2} - \frac{1}{2} \cong \frac{m}{2}$.

Memorizzando soltanto il coefficiente c' avremo che la dimensione della tabella sarà

$$\text{N.bit} = 2^{\frac{m}{2}} \cdot m$$

cioè il 33% in meno rispetto alla tecnica dell'espansione in serie di Taylor senza modifica dell'operando.

Passiamo ora all'analisi di alcuni metodi che fanno uso di un polinomio di secondo grado per calcolare il valore approssimato di alcune funzioni elementari.

Jain, Wadecar, e Lin [5] presentano un dispositivo multifunzione che calcola il valore di $\sin(x), \frac{1}{x}, \sqrt{x}, \mathrm{arctg}(x)$ in corrispondenza del valore di due bit di selezione in ingresso. Poniamo la nostra attenzione soprattutto sulla scelta che fanno tra i possibili modi di approssimare queste funzioni. Una di queste scelte è quella di interpolare la funzione. Supponiamo che per l'operando X valga la limitazione $R_1 \leq X < R_2$, allora, dividendo X in parte alta X_M e parte bassa X_L, si può pensare che gli M bit di X_M dividano l'intervallo in 2^M parti uguali. Su ciascuna di queste parti $[x_i, x_{i+1}]$ viene costruito il polinomio P(x) individuato dalle condizioni

$$\begin{aligned} P(x_i) &= f(x_i) = f_i \\ P(x_{i+1}) &= f(x_{i+1}) = f_{i+1} \; . \\ P'(x_i) &= f'(x_i) \end{aligned}$$

Posto $y = \frac{x - x_i}{x_{i+1} - x_i}$ e $y_c = \frac{x_{i+1} - x}{x_{i+1} - x_i}$ si può scrivere, per l'intervallo $x_i \leq x < x_{i+1}$,

$$P(x) = f_i + g_i \cdot y + h_i \cdot y \cdot y_c$$

dove

$$\begin{aligned} g_i &= f_{i+1} - f_i \\ h_i &= \Delta \cdot (f_i' - \hat{f}_i') \end{aligned} \quad , \quad \begin{aligned} \hat{f}_i' &= \frac{f_{i+1} - f_i}{\Delta} \\ \Delta &= x_{i+1} - x_i \end{aligned}$$

P(x) può essere così calcolato estraendo dalla memoria i valori $f_i, g_i, h_i, y \cdot y_c$. Osservando poi che il prodotto $h_i \cdot y \cdot y_c$ dà un contributo di ordine superiore ai primi due f_i e $g_i \cdot y$, viene fatta la scelta di precalcolare $y \cdot y_c$ arrotondandolo su un più piccolo nu-

mero di bit di quelli che servono per rappresentarlo e successivamente memorizzarlo in tabella.

Una stima della dimensione della look-up table dà il risultato

$$N.Bit = 2^{\frac{m}{3}} \cdot (m + \frac{2m}{3} + \frac{m}{3} + \frac{m}{3}) = 2^{\frac{m}{3}} \cdot \frac{7m}{3}$$

Osserviamo che, con questa tecnica, vengono memorizzati sia i valori della funzione, sia le differenze tra elementi consecutivi (g_i) e sia le quantità h_i e $y \cdot y_c$; c'è in qualche modo una ridondanza di informazione nella look-up table.

Nell'intento di migliorare la procedura di cui sopra, Cao e Wei [6] razionalizzano il problema della scelta di quanta informazione memorizzare e di quale polinomio utilizzare.

Per quanto riguarda il polinomio da usare, la scelta ricade sulla classe dei polinomi (di secondo grado) che effettuano l'interpolazione semplice della funzione assegnata. Fra tutti i polinomi di interpolazione che si ottengono al variare del gruppo dei nodi, essi osservano che quello che dà i migliori risultati in termini di massimo errore assoluto è il polinomio risultante dalla scelta dei nodi di Chebycev nell'intervallo dato.

Per i tre nodi di Chebycev nell'intervallo $[x_i, x_{i+1}]$ si ha

$$z_{-1} = -\cos\left(\frac{\pi}{6}\right) \cdot \frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2} = -\frac{\sqrt{3}}{2} \cdot \frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2}$$
$$z_0 = -\cos\left(\frac{\pi}{2}\right) \cdot \frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2} = \frac{x_{i+1} + x_i}{2}$$
$$z_1 = \cos\left(\frac{\pi}{6}\right) \cdot \frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2} = \frac{\sqrt{3}}{2} \cdot \frac{x_{i+1} - x_i}{2} + \frac{x_{i+1} + x_i}{2}$$

Una volta scelti i nodi di interpolazione, il polinomio è unico, ma rimane da determinare quale sia il modo migliore per calcolarlo. Una prima possibilità sarebbe quella di memorizzare solo i valori della funzione e poi calcolare il valore del polinomio scritto nella forma di Lagrange. Allora, se diciamo g_{-1}, g_0, g_1 i valori che la funzione assume nei nodi z_{-1}, z_0, z_1 si ha

$$P(z) = \frac{(z-z_0)\cdot(z-z_1)}{(z_{-1}-z_0)\cdot(z_{-1}-z_1)}\cdot g_{-1} + \frac{(z-z_{-1})\cdot(z-z_1)}{(z_0-z_{-1})\cdot(z_0-z_1)}\cdot g_0 + \frac{(z-z_{-1})\cdot(z-z_0)}{(z_1-z_{-1})\cdot(z_1-z_0)}\cdot g_1 .$$

Sebbene questo sistema permetta di risparmiare sulle dimensioni della look-up table, esso non è conveniente per via della complessità dell'hardware dedicato al calcolo del valore del polinomio. Lo stesso polinomio può però essere riscritto nella forma (di Newton)

$$P(z) = \frac{s^2}{2}\left(g_1 - 2g_0 + g_{-1}\right) + \frac{s}{2}(g_1 - g_{-1}) + g_0 = \frac{s^2}{2}\cdot b + \frac{s}{2}\cdot a + c = \frac{s}{2}(a + b\cdot s) + c$$

dove
$$s = \frac{z - z_0}{k},\ \ k = z_1 - z_0 = z_0 - z_{-1}$$
$$a = g_1 - g_{-1}$$
$$c = \frac{g_1 + g_{-1} - b}{2}$$

Si può calcolare il valore del polinomio estraendo dalla look-up table il valore di b e durante l'esecuzione della moltiplicazione $s\cdot b$ calcolare in parallelo il valore dei coefficienti a e c. Allo stadio successivo si completa poi il calcolo del valore del polinomio utilizzando ancora un moltiplicatore e due sommatori.

Per quanto riguarda le dimensioni della look-up table si ottiene

$$N.Bit = 2^{\frac{m}{3}}\cdot\left(m + \frac{2m}{3}\right) = 2^{\frac{m}{3}}\cdot\frac{5m}{3}$$

Si riporta in conclusione in una tabella la dimensione stimata in bit della look-up table necessaria per ottenere m bit esatti del risultato per i vari metodi enunciati.

Metodo	Dimensione in bit come funzione di m	Caso m=15
Faithful Rom Bipartite Table	$2^{\frac{2m}{3}} \cdot m + 2^{\frac{2m}{3}} \cdot \frac{m}{3}$	20.4 kbit
Simmetric Bipartite Table Method	$2^{\frac{2m}{3}} \cdot m + 2^{\frac{2m}{3}-1} \cdot \frac{m}{3}$	17.9 kbit
Taylor arrestato al primo ordine	$2^{\frac{m}{2}} \cdot \left(m + \frac{m}{2}\right)$	4.1 kbit
Taylor con operando modificato	$2^{\frac{m}{2}} \cdot m$	2.7 kbit
Polinomio interpolatore 2 estremi e una derivata	$2^{\frac{m}{3}} \cdot (m + \frac{2m}{3} + \frac{m}{3} + \frac{m}{3}) = 2^{\frac{m}{3}} \cdot \frac{7m}{3}$	1.12 kbit
Polinomio interpolatore nodi di Chebychev	$2^{\frac{m}{3}} \cdot (m + \frac{2m}{3}) = 2^{\frac{m}{3}} \cdot \frac{5m}{3}$	800 bit

Tabella 1.1

Osservando la tabella ci si rende conto di come sia possibile, mediante una accurata scelta del metodo, ridurre fortemente le risorse hardware impiegate. Sarà oggetto dei successivi capitoli il progetto di un divisore che abbia in ingresso un operando su 16 bit, e restituisca il risultato ancora su 16 bit.

Viene evidenziato in tabella il caso $m = 15$, poiché il nostro dispositivo sarà in grado di valutare il reciproco dell'operando in ingresso con un massimo errore permesso di 1 LSB, ovvero di 2^{-16}. Potremo considerarlo quindi equivalente a un dispositivo che restituisce un risultato correttamente arrotondato su 15 bit.

CAPITOLO 2

L'algoritmo per il calcolo della funzione x^p

2.1 Formulazione del problema

Nel primo capitolo si è visto come sia possibile calcolare il valore di una funzione riconducendolo allo svolgimento di alcune operazioni elementari.

Riformuliamo il problema già così descritto e introduciamo un nuovo modo di approssimare la funzione con l'obiettivo di costruire un dispositivo che ne calcoli il valore. Sia

$$f(x) = x^p, \quad p \in Q .$$

Fissato un intervallo I per la variabile x, al fine di approssimare nel modo migliore su tutto l'intervallo la funzione assegnata con un'altra funzione il cui valore in ogni punto x si possa ottenere con un numero minimo di operazioni aritmetiche dirette (addizione e moltiplicazione), cercheremo di determinare un polinomio di secondo grado $p(x)$ che minimizzi il massimo errore assoluto (sull'intervallo) che si compie, quando si sostituisce a f(x) tale polinomio; cercheremo, cioè, di determinare i coefficienti di $p(x)$ che minimizzano l'errore

$$E = \underset{x \in I}{\mathrm{Max}} \left| f(x) - p(x) \right|$$

2.2 Una premessa sulla rappresentazione dell'operando e del risultato

Prima di fare le considerazioni che servono per la soluzione di detto problema si ritiene opportuno accennare alla rappresentazione dell'operando x, e alla rappresentazione del risultato.

Faremo ricorso a una rappresentazione binaria in virgola fissa, osservando tra l'altro che anche pensando alla più diffusa rappresentazione in virgola mobile (IEEE-754), ricondurremo la parte sostanziale del problema al computo di x^p per un numero in virgola fissa. Infatti potremo scrivere in virgola mobile (in "single precision")

$$x = (-1)^s \cdot (1+f) \cdot 2^{e-127},$$

dove

s: bit del segno
f: parte frazionaria su 23 bit
e: esponente su 8 bit

e dove la parte frazionaria f è un numero razionale compreso tra 0 e 1 che possiamo scrivere nella forma

$$f = x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + ... + x_{22} \cdot 2^{-22} + x_{23} \cdot 2^{-23},$$

dopo di che la mantissa 1+f del nostro numero sarà rappresentata con

$$1.x_1 x_2 \cdots x_{23}$$

Per quanto riguarda la rappresentazione del risultato x^p abbiamo:

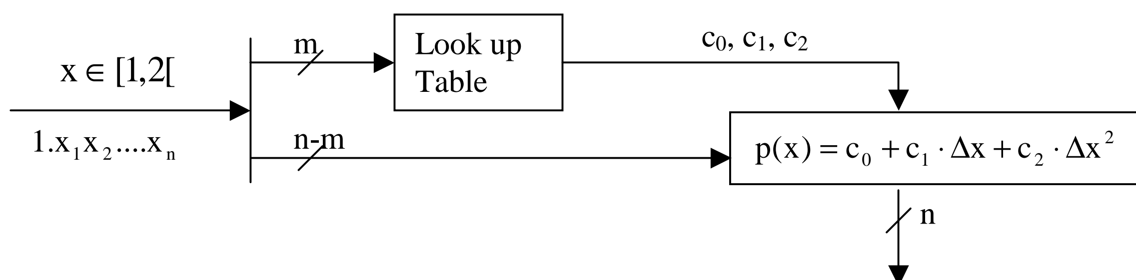
$$x^p = \left[(-1)^s \cdot (1+f) \cdot (2^{e-127})\right]^p = (-1)^{p \cdot s} \cdot (1+f)^p \cdot 2^{p(e-127)}$$

e quindi potremo trattare il calcolo del bit del segno e dell'esponente come problemi separati, e ricondurre poi il nostro problema al calcolo di x^p con x numero in virgola fissa, secondo la già scritta rappresentazione $1.x_1x_2 \cdots x_{23}$, avendo poi cura di "aggiustare" il valore dell'esponente e della mantissa quando l'operazione lo rende necessario.

In base alle dette considerazioni concludiamo che il dominio per la nostra funzione resta fissato all'intervallo [1,2[.

2.3 Schema di principio del circuito per il calcolo della funzione

Sia x su n bit l'operando in ingresso all'approssimatore. Si possono dividere gli n bit dell'operando in due parti, che chiameremo parte alta e parte bassa. Prendendo per la parte alta gli m bit più significativi e per la parte bassa i rimanenti n-m bit si divide l'intervallo [1,2[in 2^m sottointervalli. Cerchiamo poi per ciascuno di questi intervalli di trovare il polinomio di 2° grado di migliore approssimazione uniforme della funzione $f(x)$. Ciò fatto, si memorizzano i coefficienti dei polinomi nella look-up table, di modo che gli m bit della parte alta costituiscano l'indirizzo della memoria. In uscita a quest'ultima abbiamo i coefficienti che in ingresso al dispositivo di calcolo permettono di ottenere il risultato su n bit.



2.4 Approssimazione della funzione

Ricordiamo il teorema di Chebychev, che è un risultato classico della teoria dell'approssimazione secondo il quale, data una funzione f(x) continua sull'intervallo I e un intero positivo n, il polinomio p(x) di grado n è il polinomio di migliore approssimazione uniforme su I per f(x), se e solo se esistono in I n+2 punti $\xi_0 < \xi_1 < ... < \xi_{n+1}$, detti punti di alternanza, tali che

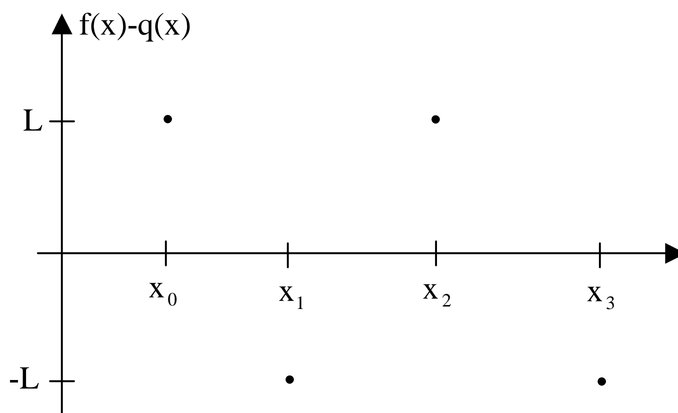
$$f(\xi_i) - p(\xi_i) = \alpha \cdot (-1)^i E\,, \;\; i = 0,...,n+1$$

essendo E il massimo errore assoluto sull'Intervallo I (già definito in 2.1) e $\alpha = 1$ oppure $\alpha = -1$ simultaneamente per tutti gli i.

Posto [a,b]=I, affrontiamo il problema di determinare i coefficienti di p(x) e il numero E mediante l'algoritmo di Remes [7], che è un procedimento ricorsivo a 2 passi, per cui, al primo passo, assegnati comunque n+2 "nodi" $a \le x_0 < x_1 < ...... < x_n < x_{n+1} \le b$, si determinano gli n+1 coefficienti del polinomio q(x) di grado minore o uguale a n e il numero positivo L che soddisfano il seguente sistema di n+2 equazioni.lineari

$$f(x_i) - q(x_i) = (-1)^i \cdot L \qquad i = 0..n+1\ .$$

Nel caso n=2 il polinomio q(x) deve differire dalla funzione f(x) di una quantità L (incognita) a segni alterni, come illustrato nella figura seguente



Determinato così q(x), al secondo passo cerchiamo n+2 nuovi nodi

$$a \leq z_0 < z_1 < ...... < z_n < z_{n+1} \leq b$$

per i quali $\left| f(x) - q(x) \right|$ presenti un massimo locale in ogni z_i, e le quantità $f(x_i)-q(x_i)$ e $f(z_i)-q(z_i)$ abbiano lo stesso segno al variare di i.

Determinati così i nuovi nodi z_i si ritorna al passo 1 e si determina il nuovo q(x). Si reitera poi il procedimento fintanto che non si raggiunge una situazione di stazionarietà e a quel punto i nodi in questione sono, a meno di un errore trascurabile, i punti di alternanza e lo scostamento L è l'errore massimo E.

Nel nostro caso $n = 2$, posto $q(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2$, il sistema da risolvere è il seguente:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & 1 \\ 1 & x_1 & x_1^2 & -1 \\ 1 & x_2 & x_2^2 & 1 \\ 1 & x_3 & x_3^2 & -1 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ L \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \end{pmatrix}$$

il cui determinante,

$$\det = 2 \cdot (x_0 - x_2)(x_1 - x_3)(x_0 - x_1 + x_2 - x_3)$$

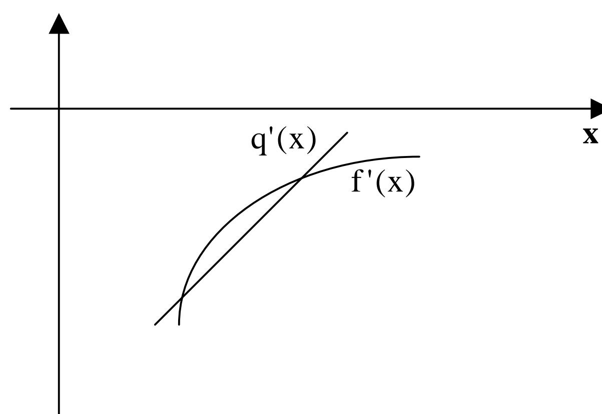
risulta senz'altro diverso da zero data la scelta dei nodi ($x_i < x_{i+1}$ per ogni i); ne consegue che il sistema anzidetto ha un'unica soluzione e quindi il relativo polinomio q(x) risulta univocamente determinato.

Sceglieremo, come nodi di partenza i punti "di Chebychev":

$$x_0 = a\,, \quad x_1 = a + \frac{1}{4}(b-a)\,, \quad x_2 = a + \frac{3}{4}(b-a)\,, \quad x_3 = b\,.$$

Per risolvere il problema dei massimi locali facciamo le seguenti osservazioni. Per il fatto che la funzione f(x)-q(x) si muove a segni alterni due volte tra L e -L, ci si convin-

ce subito che ci sono almeno due massimi locali di $|f(x)-q(x)|$; osservando poi, che la derivata di q(x) è lineare, e che sia la derivata prima, la seconda e la terza di $f(x)=x^p$ sono a segno costante, avremo la situazione per $q'(x)$ e $f'(x)$ mostrata nella figura seguente:



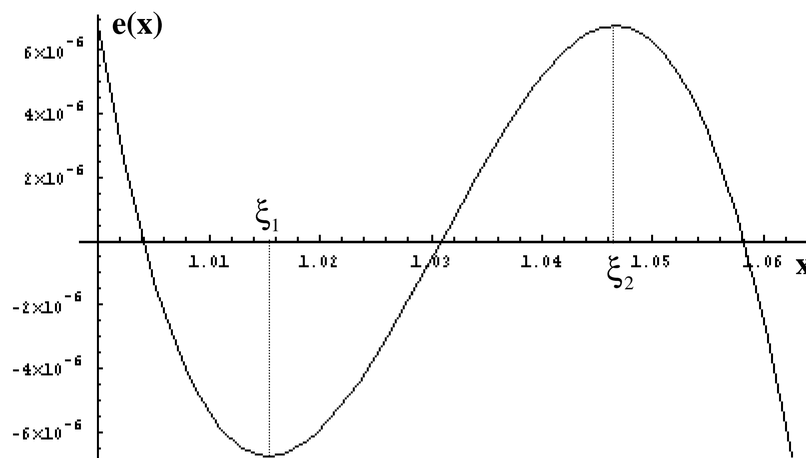
ovvero al più due soluzioni dell'equazione $f'(x)-q'(x)=0$, cioè, in conclusione, esattamente due soluzioni.

A questo punto, con i metodi classici del calcolo numerico possiamo separare le due soluzioni in questione e successivamente approssimarle con opportuna precisione.

Nel caso nostro [a,b]=$\left[1+\frac{i}{2^m},1+\frac{i+1}{2^m}\right]$ e per $i=0$, $m=4$, $p=-1$, otteniamo per i punti di alternanza interni:

$$\xi_1=1{,}01539 \quad , \quad \xi_2=1{,}04664$$

e per l'errore massimo $E\cong 6{,}76\cdot 10^{-6}$. Nella figura che segue si mostra il grafico dell'errore nell'intervallo [1;1,0625]



Per farsi una idea riguardo alla efficienza di tale approssimazione, facciamone un confronto con una soluzione più diretta, come ad esempio il polinomio di Taylor dello stesso grado, con punto iniziale il punto di mezzo del generico sottointervallo.

Sia, ad esempio, come sopra, $f(x)=x^{-1}$ e m=4. Avremo (con evidente significato dei simboli)

$$\tilde{f}_i(x)=f(x_{0i})+(x-x_{0i})\cdot f'(x_{0i})+\frac{(x-x_{0i})^2}{2}f''(x_{0i})=$$

$$=x_{0i}^{-1}-x_{0i}^{-2}\cdot(x-x_{0i})+x_{0i}^{-3}\cdot(x-x_{0i})^2$$

$$x\in\left[x_{0i}-2^{-5},x_{0i}+2^{-5}\right],\ \ x_{0i}=1+i\cdot 2^{-4}+2^{-5}$$

Prendendo per esempio il 1° intervallo si ha

$$i=0,\ \ \tilde{f}_0(x)=\frac{32}{11}+\frac{1024}{363}\cdot x+\frac{32768}{35937}\cdot x^2$$

$$\max_{x\in[1,1+2^{-4}]}\left|\,f_0(x)-\tilde{f}_0(x)\,\right|=2{,}78\times 10^{-5}.$$

Si può osservare quindi che l’errore dovuto all’approssimazione con il metodo del polinomio di migliore approssimazione uniforme è circa 4 volte inferiore a quello ottenuto con il procedimento del polinomio di Taylor.

Il problema di determinare i “punti di alternanza” può essere affrontato, oltre che mediante il metodo ricorsivo di Remes, anche con un metodo diretto, che, data la piccolezza del grado del polinomio di approssimazione, risulta particolarmente facile ed espressivo; a tale scopo osserviamo intanto, che il polinomio di 2° grado q(x) che verifica le condizioni di Chebychev altro non è che il polinomio di grado minore o uguale a 2 che nei punti x_i prende rispettivamente i valori

$$f(x_i)-(-1)^i\cdot L \quad , \quad i=0,1,2\,, \quad \text{mentre} \quad L=q(x_3)-f(x_3)\,.$$

Se diciamo $P_2(x)$ il polinomio di grado ≤ 2 che interpola f(x) nei nodi x_i, e $S_2(x)$ quello che in tali punti prende i valori $(-1)^i$, $i=0,1,2$, risulta

$$L=\frac{P_2(x_3)-f(x_3)}{1+S_2(x_3)}\,, \quad q(x)=P_2(x)-L\cdot S_2(x) \;\; ;$$

È ovvio che, alla fine, lo scostamento $|\,L\,|$ di q(x) da f(x) risulta funzione dei nodi x_i , $i=0,1,2,3$. Fissati allora x_0 e x_3, per ottenere il massimo scostamento E sull'intervallo $[x_0,x_3]$, basta determinare i nodi x_1 e x_2 per i quali $|\,L\,|$ assume il suo valore massimo.

Con $f(x)=\dfrac{1}{x}$, con l'aiuto di un programma per il calcolo simbolico, si trova dopo qualche calcolo,

$$L=-\frac{(x_0-x_1)\cdot(x_1-x_2)\cdot(x_0-x_3)\cdot(x_2-x_3)}{2\cdot x_0\cdot x_1\cdot x_2\cdot(x_0-x_1+x_2-x_3)\cdot x_3}$$

e il valore massimo di $|\,L\,|$ si ottiene per

$$\xi_1=\sqrt{x_0}\cdot\frac{\sqrt{x_0}+\sqrt{x_3}}{2} \quad , \qquad \xi_2=\sqrt{x_3}\cdot\frac{\sqrt{x_0}+\sqrt{x_3}}{2}$$

dove

$$|\,L\,|_{max}=E=\frac{\left(\sqrt{x_3}-\sqrt{x_0}\right)^3}{2\cdot x_0\cdot\left(\sqrt{x_0}+\sqrt{x_3}\right)\cdot x_3}\,.$$

Con $x_0=1$, $x3=1+2^{-4}$ si trovano, per ξ_1, ξ_2, E, gli stessi risultati numerici del metodo di Remes.

Per la funzione $f(x)=\sqrt{x}$ si trova

$$L=\frac{\left(\sqrt{x_0}-\sqrt{x_1}\right)\cdot\left(\sqrt{x_1}-\sqrt{x_2}\right)\cdot\left(\sqrt{x_0}-\sqrt{x_3}\right)\cdot\left(\sqrt{x_2}-\sqrt{x_3}\right)\cdot\left(\sqrt{x_0}+\sqrt{x_1}+\sqrt{x_2}+\sqrt{x_3}\right)}{2\cdot\left(\sqrt{x_0}+\sqrt{x_2}\right)\cdot\left(\sqrt{x_1}+\sqrt{x_3}\right)\cdot\left(x_0-x_1+x_2-x_3\right)}$$

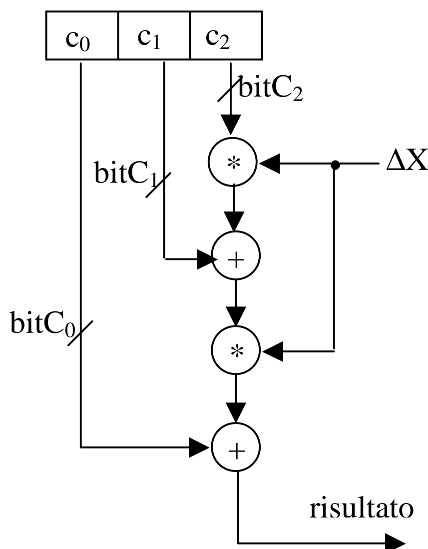
È da notare però che in questo caso i calcoli da effettuare in via simbolica per massimizzare il valore dello scostamento $|L|$ sono molto più complicati ed è quindi da presumere che, per la funzione $f(x)=x^p$, in generale, convenga seguire il metodo numerico di Remes.

Nel caso in esame, estremizzando $|L|$ rispetto a (x_1,x_2) ed assumendo per x_0 e x_3 gli stessi valori di prima, si trova, per via numerica,

$$\begin{array}{l}\xi_1=1{,}015477\\ \xi_2=1{,}046727\end{array}\quad,\quad E=4{,}42\cdot10^{-7}\,.$$

2.5 Stima dell'errore dovuto all'uso di "aritmetica finita"

Abbiamo visto come si possono individuare i coefficienti del polinomio di 2° grado per il quale è minimo il massimo scarto dalla funzione nell'intervallo assegnato. In questa sezione cerchiamo di determinare il migliore arrotondamento di tali coefficienti su un numero fissato di bit. Il risultato sarà un compromesso tra la precisione con cui viene approssimato il valore della funzione e la disponibilità in termini di risorse hardware per la realizzazione del dispositivo che effettua tale approssimazione.



Facendo riferimento alla architettura a lato rappresentata, indichiamo con $bitc_0$, $bitc_1$, $bitc_2$, il numero di bit su cui vengono rappresentati i coefficienti c_0, c_1, c_2 rispettivamente, e con Δx la differenza tra l'argomento della funzione x e l'estremo sinistro dell'intervallo cui x appartiene. Nel caso in cui si scelga di dividere l'intervallo [1,2[in 16 sottointervalli tutti di uguale ampiezza otterremo per la rappresentazione degli estremi sinistri di ciascun intervallo i numeri:

$$1+0\cdot 2^{-4}, 1+1\cdot 2^{-4}, 1+2\cdot 2^{-4} \ldots\ldots 1+i\cdot 2^{-4}, \ldots\ldots 1+15\cdot 2^{-4}$$

Quindi avendo a disposizione la rappresentazione del numero x per esempio su 16 bit, ci basterà prendere i 4 bit più significativi per individuare l'estremo sinistro dell'intervallo in cui cade x, e i restanti 12 bit saranno proprio la rappresentazione della quantità Δx definita come $\Delta x = x - x_{0i}$, x_{0i}: estremo sinistro dell'intervallo i-esimo.

Precisando che d'ora in avanti tratteremo esplicitamente solo il caso $p = -1$, riportiamo per intanto nella seguente tabella i coefficienti del polinomio $p_i(x)$ e l'errore E_i per tutti i 16 sottointervalli in cui abbiamo diviso l'intervallo $[1,2[$.

Indice	x_i	x_{i+1}	c_0	c_1	c_2	E
0	1.0000	1.0625	0.99999324	-0.99801441	0.91286558	6.76E-06
1	1.0625	1.1250	0.94117113	-0.88424605	0.76504879	5.34E-06
2	1.1250	1.1875	0.88888462	-0.78887017	0.64750249	4.27E-06
3	1.1875	1.2500	0.84210180	-0.70812703	0.55285742	3.46E-06
4	1.2500	1.3125	0.79999717	-0.63917043	0.47579667	2.83E-06
5	1.3125	1.3750	0.76190242	-0.57981378	0.41241860	2.34E-06
6	1.3750	1.4375	0.72727077	-0.52835488	0.35981606	1.95E-06
7	1.4375	1.5000	0.69565053	-0.48345267	0.31579276	1.64E-06
8	1.5000	1.5625	0.66666528	-0.44403901	0.27866847	1.39E-06
9	1.5625	1.6250	0.63999882	-0.40925472	0.24714291	1.18E-06
10	1.6250	1.6875	0.61538360	-0.37840235	0.22019914	1.01E-06
11	1.6875	1.7500	0.59259172	-0.35091098	0.19703408	8.75E-07
12	1.7500	1.8125	0.57142781	-0.32630966	0.17700783	7.58E-07
13	1.8125	1.8750	0.55172348	-0.30420713	0.15960624	6.60E-07
14	1.8750	1.9375	0.53333276	-0.28427613	0.14441302	5.78E-07
15	1.9375	2.0000	0.51612852	-0.26624130	0.13108865	5.08E-07

Tabella 2.1

Al fine di valutare l'errore totale che si compie quando al posto di f(x), si sostituisce un polinomio ottenuto da p(x) arrotondando i suoi coefficienti, introduciamo intanto la quantità *errore intrinseco* definito dalla relazione

$$\varepsilon_{\text{intr}} = \underset{i=0,\dots,15}{\text{Max}}\{E_i\} \quad , \quad E_i = \underset{x \in I_i}{\text{Max}}\{\| f(x) - p_i(x) \|\} \quad , \quad I_i = [x_i, x_{i+1}[$$

Conoscendo quindi il massimo errore assoluto E_i relativo al sottointervallo i-esimo, dovuto alla approssimazione della funzione x^{-1} con il polinomio

$$p_i(x) = c_{0i} + c_{1i} \cdot \Delta x + c_{2i} \cdot \Delta x^2 ,$$

chiameremo $\hat{c}_{0i}, \hat{c}_{1i}, \hat{c}_{2i}$ quelli ottenuti dall'arrotondamento dei coefficienti c_{0i}, c_{1i}, c_{2i}.

Abbiamo bisogno perciò, di un criterio per stabilire su quanti bit rappresentare i coefficienti arrotondati $\hat{c}_{0i}, \hat{c}_{1i}, \hat{c}_{2i}$

$$\hat{c}_{0i} = c_{0i} \cdot (1 + \varepsilon_{0i})$$
$$\hat{c}_{1i} = c_{1i} \cdot (1 + \varepsilon_{1i})$$
$$\hat{c}_{2i} = c_{2i} \cdot (1 + \varepsilon_{2i})$$

dove $\varepsilon_{0i}, \varepsilon_{1i}, \varepsilon_{2i}$ sono gli errori relativi commessi nell'arrotondamento dei coefficienti.

Viene definito di conseguenza il polinomio $\hat{p}_i(x)$, che è una approssimazione di $p_i(x)$

$$\hat{p}_i(x) = \hat{c}_{0i} + \hat{c}_{1i} \cdot \Delta x + \hat{c}_{2i} \cdot \Delta x^2 = p_i(x) + \varepsilon_{0i} \cdot c_{0i} + \varepsilon_{1i} \cdot c_{1i} \cdot \Delta x + \varepsilon_{2i} \cdot c_{2i} \cdot \Delta x^2$$

È ovvio che l'arrotondamento dei coefficienti c_{0i}, c_{1i}, c_{2i} in $\hat{c}_0, \hat{c}_1, \hat{c}_2$, produce un errore maggiore rispetto all'errore E_i del polinomio di minimo scarto. Vogliamo stabilire un confine superiore M alla quantità E_{agg_i} così definita

$$E_{agg_i} = E_{arr_i} - E_i; \qquad E_{arr_i} = \underset{x \in I_i}{\mathrm{Max}} \left\{ \left| f(x) - \hat{p}_i(x) \right| \right\}$$

facendo vedere che M dipende solamente dal numero di bit su cui sono rappresentati i coefficienti arrotondati $\hat{c}_{0i}, \hat{c}_{1i}, \hat{c}_{2i}$.

Essendo $|f(x) - p_i(x)| \leq E_i$ avremo

$$-E_i \leq f(x) - p_i(x) \leq E_i$$

$$-E_i + p_i(x) - \hat{p}_i(x) \leq f(x) - p_i(x) + p_i(x) - \hat{p}(x) \leq E_i + p_i(x) - \hat{p}_i(x)$$

$$-E_i - \varepsilon_{0i} \cdot c_{0i} - \varepsilon_{1i} \cdot c_{1i} \cdot \Delta x - \varepsilon_{2i} \cdot c_{2i} \cdot \Delta x^2 \leq f(x) - \hat{p}_i(x) \leq E_i + \varepsilon_{0i} \cdot c_{0i} + \varepsilon_{1i} \cdot c_{1i} \cdot \Delta x + \varepsilon_{2i} \cdot c_{2i} \cdot \Delta x^2$$

$$|f(x) - \hat{p}_i(x)| \leq \left| E_i + \varepsilon_{0i} \cdot c_{0i} + \varepsilon_{1i} \cdot c_{1i} \cdot \Delta x + \varepsilon_{2i} \cdot c_{2i} \cdot \Delta x^2 \right| \leq E_i + |\varepsilon_{0i} \cdot c_{0i}| + |\varepsilon_{1i} \cdot c_{1i} \cdot \Delta x| + \left|\varepsilon_{2i} \cdot c_{2i} \cdot \Delta x^2\right|$$

e in definitiva

$$E_{arr_i} = \underset{x \in I_i}{Max}\left|f(x) - \hat{p}_i(x)\right| \le \underset{x \in I_i}{Max}\left\{\varepsilon_{intr} + \left|\varepsilon_{0i} \cdot c_{0i}\right| + \left|\varepsilon_{1i} \cdot c_{1i} \cdot \Delta x\right| + \left|\varepsilon_{2i} \cdot c_{2i} \cdot \Delta x^2\right|\right\}$$

Osservando che i coefficienti c_{0i}, c_{1i}, c_{2i} sono tutti in modulo minori di 1, qualunque sia i (come riscontrabile dalla tabella 2.1), e che gli errori relativi $\varepsilon_{0i}, \varepsilon_{1i}, \varepsilon_{2i}$ valgono al più $2^{-bitC_0-1}, 2^{-bitC_1-1}, 2^{-bitC_2-1}$ rispettivamente e che Δx vale al più 2^{-4}, si conclude che

$$E_{arr_i} \le E_i + 2^{-bitC_0-1} + 2^{-bitC_2-1-4} + 2^{-bitC_2-1-8}$$

e che quindi il confine superiore M che cercavamo per l'errore aggiunto vale

$$E_{agg_i} = E_{arr_i} - E_i < 2^{-bitC_0-1} + 2^{-bitC_2-1-4} + 2^{-bitC_2-1-8} = M$$

Nel caso in cui si voglia ottenere una rappresentazione del risultato su 16 bit con un massimo errore assoluto di 1 LSB (Least Significant Bit, bit meno significativo) avremo: 1 LSB = $2^{-16} = 1{,}53 \cdot 10^{-5}$. Dai risultati ottenuti per il polinomio di minimo scarto abbiamo un errore intrinseco di $6{,}76 \cdot 10^{-6}$, come si può rilevare dalla tabella 2.1.

Ci rimane quindi un margine d'errore durante l'arrotondamento dei coefficienti e dei risultati intermedi e/o finali di $1{,}53 \cdot 10^{-5} - 6{,}76 \cdot 10^{-6} = 8{,}54 \cdot 10^{-6}$. Tralasciando per il momento il fatto che l'arrotondamento finale su 16 bit comporta un massimo errore di 0,5 LSB, dividiamo il margine d'errore in 3 parti, ottenendo così per ciascuna delle tre cause di errore $\varepsilon_{0i}, \varepsilon_{1i} \cdot \Delta x, \varepsilon_{2i} \cdot \Delta x^2$ un valore di circa $2{,}85 \cdot 10^{-6}$. Dalla disuguaglianza:

$$\varepsilon_{0i} \le 2^{-(bitc_0+1)} \le 2{,}85 \cdot 10^{-6}$$

otteniamo $bitc_0 = 18$, e ragionando in modo analogo per gli altri due coefficienti, $bitc_1 = 14$ e $bitc_2 = 10$. Abbiamo pertanto determinato una stima di quanti bit occorro-

no per la rappresentazione dei coefficienti senza incorrere in errori che superino le specifiche che ci siamo posti. Vedremo poi che questa stima è cautelativa e che in realtà la rappresentazione dei coefficienti con questo numero di bit ci permetterà di tenere conto anche dell'arrotondamento finale su 16 bit senza uscire fuori dal massimo errore permesso sul risultato di 1 LSB.

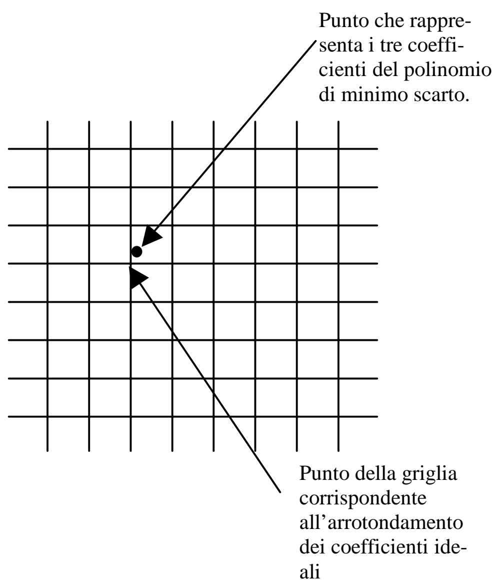
Si riportano nella seguente tabella le rappresentazioni di $\hat{c}_{0i}, \hat{c}_{1i}, \hat{c}_{2i}$ e le quantità E_i, E_{arr_i}, E_{agg_i} nel caso in esame $p = -1,\ m = 4$.

Indice	x_i	x_{i+1}	$\hat{c}_{0R}$	$\hat{c}_{1R}$	$\hat{c}_{2R}$	E	E_{arr}	E_{agg}
0	1.0000	1.0625	262142	-16351	935	6.76E-06	8.53E-06	1.77E-06
1	1.0625	1.1250	246722	-14487	783	5.34E-06	6.73E-06	1.39E-06
2	1.1250	1.1875	233016	-12925	663	4.27E-06	5.00E-06	7.27E-07
3	1.1875	1.2500	220752	-11602	566	3.46E-06	3.64E-06	1.79E-07
4	1.2500	1.3125	209714	-10472	487	2.83E-06	4.58E-06	1.75E-06
5	1.3125	1.3750	199728	-9500	422	2.34E-06	4.57E-06	2.23E-06
6	1.3750	1.4375	190650	-8657	368	1.95E-06	2.99E-06	1.04E-06
7	1.4375	1.5000	182361	-7921	323	1.64E-06	2.93E-06	1.29E-06
8	1.5000	1.5625	174762	-7275	285	1.39E-06	2.94E-06	1.55E-06
9	1.5625	1.6250	167772	-6705	253	1.18E-06	2.35E-06	1.17E-06
10	1.6250	1.6875	161319	-6200	225	1.01E-06	3.37E-06	2.36E-06
11	1.6875	1.7500	155344	-5749	202	8.75E-07	2.26E-06	1.39E-06
12	1.7500	1.8125	149796	-5346	181	7.58E-07	2.18E-06	1.42E-06
13	1.8125	1.8750	144631	-4984	163	6.60E-07	1.29E-06	6.30E-07
14	1.8750	1.9375	139810	-4658	148	5.78E-07	1.46E-06	8.82E-07
15	1.9375	2.0000	135300	-4362	134	5.08E-07	7.35E-07	2.27E-07

Tabella 2.2

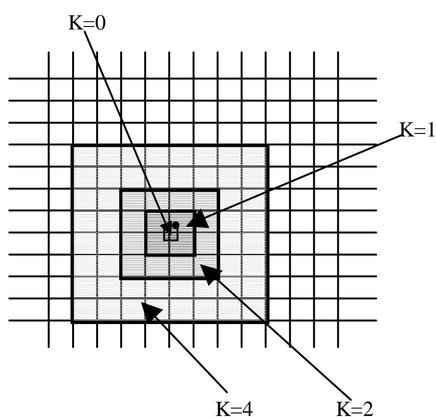
Nota: valgono per i coefficienti arrotondati $\hat{c}_0, \hat{c}_1, \hat{c}_2$ le relazioni con le loro rappresentazioni intere:

$$\hat{c}_{0i} = \hat{c}_{0Ri} \cdot 2^{-bitC_0}$$
$$\hat{c}_{1i} = \hat{c}_{1Ri} \cdot 2^{-bitC_1}$$
$$\hat{c}_{2i} = \hat{c}_{2Ri} \cdot 2^{-bitC_2}$$



Osserviamo ora, che nello spazio tridimensionale "discretizzato" dei possibili coefficienti, aver scelto di arrotondare i coefficienti ideali equivale a prendere il punto della griglia che dista meno di tutti gli altri dal punto che produce il minimo scarto (terna dei coefficienti ideali). Questa scelta può non essere la migliore, basta prendere in considerazione l'ipotesi che ci siano punti della griglia che, nonostante siano locati più lontano dal punto di minimo scarto di quanto non lo sia il punto corrispondente all'arrotondamento dei coefficienti, possano dare un massimo scarto minore rispetto a quello relativo ai coefficienti arrotondati (ma certamente superiore all'errore E_i).

E questo "evento" si pensa che avvenga con probabilità tanto maggiore quanto più l'errore aggiunto introdotto dall'arrotondamento dei coefficienti si avvicina a quella quantità definita confine superiore.



Procediamo quindi nel seguente modo: fissata la terna dei coefficienti arrotondati $\hat{c}_0, \hat{c}_1, \hat{c}_2$, diamo a questi coefficienti degli incrementi multipli interi relativi di una unità base pari a $2^{-bitc0}, 2^{-bitc1}, 2^{-bitc2}$, otteniamo una nuova terna di coefficienti $\tilde{c}_0, \tilde{c}_1, \tilde{c}_2$, e con questa scelta rivalutiamo l'errore massimo. Facendo un numero ragionevole di tentativi e scegliendo quella terna che minimizza l'errore massimo così calcolato, possiamo poi pensare di aver scelto la migliore delle possibili terne di coefficienti rappresentati sul numero di bit che abbiamo fissato a priori.

Rimane da stabilire un criterio riguardo alle dimensioni dell'intorno del punto di partenza $\hat{c}_0, \hat{c}_1, \hat{c}_2$. Uno possibile è fissare, sulla base di alcune sperimentazioni, delle soglie all'errore aggiunto prodotto dall'arrotondamento iniziale.

In base ad alcuni tentativi fatti, si fissa per k_i il seguente criterio

$k_i = 4$ se $0{,}24 \cdot M \le E_{agg_i} \le M$

$k_i = 2$ se $0{,}166 \cdot M \le E_{agg_i} < 0{,}24 \cdot M$

$k_i = 1$ altrimenti

Sebbene potessimo dare a k_i valori anche più grandi (data la potenza di calcolo dei processori odierni) dovremo poi in ogni caso fermarci ad un certo intorno. Si pensi che il numero di disposizioni con ripetizione di 2k+1 elementi 3 a 3 è pari a $(2k+1)^3$, ovvero già per k=4 abbiamo 9^3=729 tentativi da effettuare, e per ciascuno di questi risolvere numericamente un problema di massimo per la funzione $|f(x) - \tilde{p}(x)|$, $x \in I_i$.

Nella seguente tabella riportiamo i risultati ottenuti dopo questo raffinamento dei coefficienti per il caso

$f(x) = x^p, p = -1$

numero sottointervalli: 16

bitc0 = 18

bitc1 = 14

bitc2 = 10

max errore aggiunto	Soglia ‘k=2’	Soglia ‘k=4’
$5{,}72*10^{-6}$	$9{,}50*10^{-7}$	$1{,}37*10^{-6}$

Indice	x_i	x_{i+1}	$\tilde{c}_{0R}$	$\tilde{c}_{1R}$	$\tilde{c}_{2R}$	E	Err. Arr.	Err. fin.	$\tilde{c}_{0R}-\hat{c}_{0R}$	$\tilde{c}_{1R}-\hat{c}_{1R}$	$\tilde{c}_{2R}-\hat{c}_{2R}$	Guadagno
1	1.0000	1.0625	262142	-16350	933	6.76E-06	8.53E-06	7.63E-06	0	1	-2	10.53%
2	1.0625	1.1250	246723	-14489	784	5.34E-06	6.73E-06	6.51E-06	1	-2	1	3.23%
3	1.1250	1.1875	233016	-12926	664	4.27E-06	5.00E-06	4.64E-06	0	-1	1	7.37%
4	1.1875	1.2500	220752	-11602	566	3.46E-06	3.64E-06	3.64E-06	0	0	0	0.00%
5	1.2500	1.3125	209715	-10473	487	2.83E-06	4.58E-06	4.11E-06	1	-1	0	10.11%
6	1.3125	1.3750	199728	-9500	423	2.34E-06	4.57E-06	3.12E-06	0	0	1	31.65%
7	1.3750	1.4375	190650	-8657	368	1.95E-06	2.99E-06	2.99E-06	0	0	0	0.00%
8	1.4375	1.5000	182361	-7922	324	1.64E-06	2.93E-06	2.27E-06	0	-1	1	22.48%
9	1.5000	1.5625	174762	-7274	284	1.39E-06	2.94E-06	2.54E-06	0	1	-1	13.36%
10	1.5625	1.6250	167772	-6705	253	1.18E-06	2.35E-06	2.35E-06	0	0	0	0.00%
11	1.6250	1.6875	161319	-6200	226	1.01E-06	3.37E-06	1.55E-06	0	0	1	53.90%
12	1.6875	1.7500	155345	-5752	204	8.75E-07	2.26E-06	1.81E-06	1	-3	2	19.86%
13	1.7500	1.8125	149797	-5348	182	7.58E-07	2.18E-06	2.04E-06	1	-2	1	6.37%
14	1.8125	1.8750	144631	-4984	163	6.60E-07	1.29E-06	1.29E-06	0	0	0	0.00%
15	1.8750	1.9375	139810	-4657	147	5.78E-07	1.46E-06	9.94E-07	0	1	-1	31.98%
16	1.9375	2.0000	135300	-4362	134	5.08E-07	7.35E-07	7.35E-07	0	0	0	0.00%

Tabella 2.3

Nota: si intende per guadagno la percentuale che bisogna sottrarre all'errore con i coefficienti arrotondati per ottenere l'errore finale.

Come si può osservare dai risultati in tabella, del margine di errore di $8{,}5 \cdot 10^{-6}$ se ne è consumata una piccola frazione $(7{,}63 - 6{,}76) \cdot 10^{-6} = 0{,}87 \cdot 10^{-6}$. Nel peggiore dei casi (primo intervallo), nell'ipotesi di effettuare le 2 moltiplicazioni e le 2 addizioni portandosi dietro tutti i bit, si commette un errore massimo di $7{,}63 \cdot 10^{-6}$.

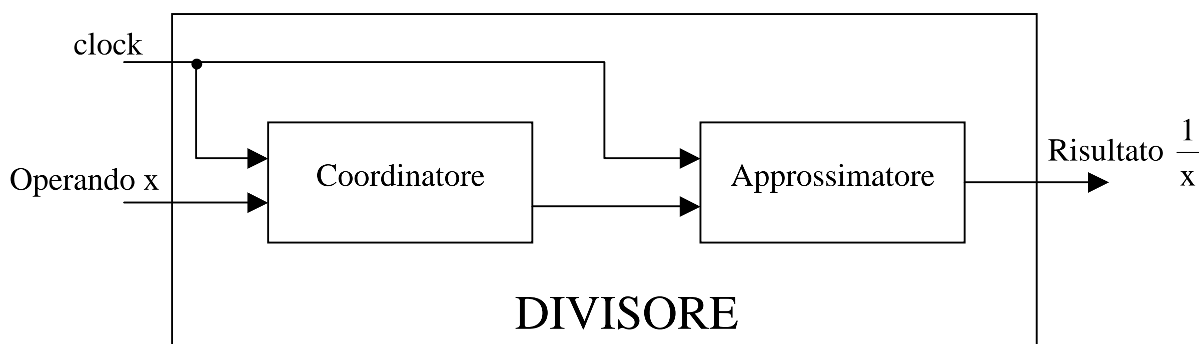
Dovendo poi arrotondare anche i risultati intermedi onde utilizzare moltiplicatori su un numero di bit ragionevole, dobbiamo affrontare il problema delle conseguenze di questi arrotondamenti, e per fare ciò dobbiamo entrare nel dettaglio dell'archittettura dell'hardware che andremo a costruire nella prossima sezione.

CAPITOLO 3

Progetto del dispositivo divisore

Dividiamo anzitutto il progetto in due blocchi funzionali

- L'approssimatore che calcola il valore del polinomio che approssima $\frac{1}{x}$ sull'intervallo [1, 2[.
- Il coordinatore che introduce i coefficienti e la quantità Δx con la giusta temporizzazione.



La look-up table sarà quindi inserita all'interno del blocco coordinatore, mentre i moltiplicatori e i sommatori saranno inseriti all'interno dell'approssimatore.

3.2 Sintesi dell'approssimatore e stima degli errori commessi in seguito al troncamento dei risultati parziali.

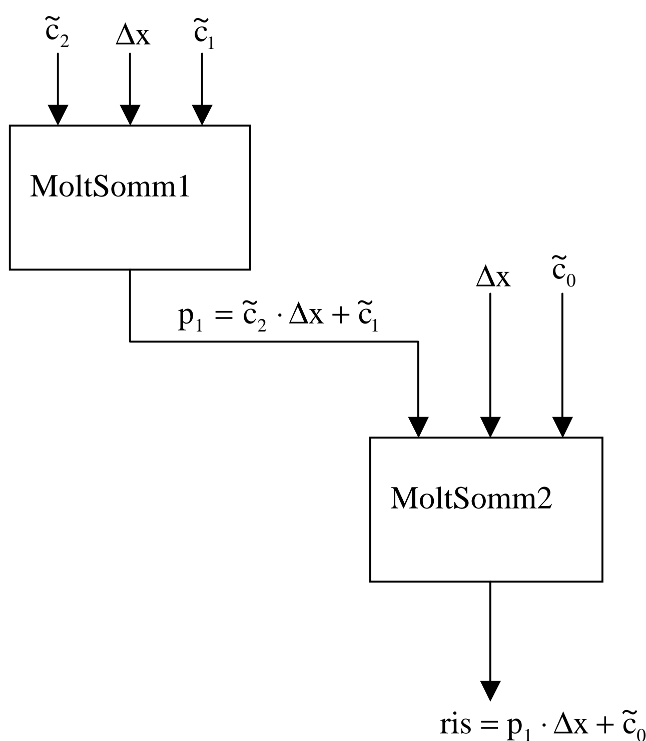
Abbiamo già tenuto conto nel secondo capitolo dell'errore dovuto all'arrotondamento dei coefficienti, e abbiamo stimato che con 18,14,10 bit, rispettivamente per $\tilde{c}_{0R}$, $\tilde{c}_{1R}$, $\tilde{c}_{2R}$, si ha un massimo errore sull'intervallo di $7{,}63 \cdot 10^{-6}$. Aggiun-

gendo poi un errore di 0,5 LSB dovuto all'arrotondamento del risultato finale su 16 bit si ottiene un massimo errore di $7{,}63 \cdot 10^{-6} + 2^{-17} = 1{,}53 \cdot 10^{-5}$, cioè circa 1 LSB.

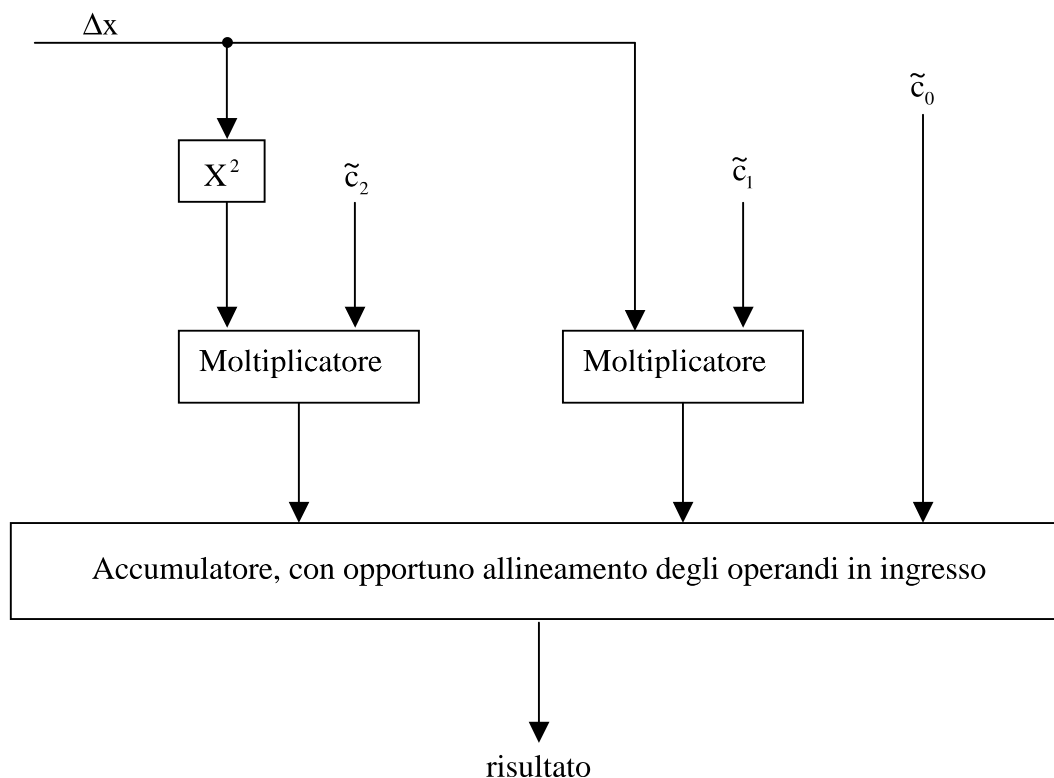
Definiamo ora l'architettura dell'approssimatore e successivamente cerchiamo di ottimizzare ulteriormente la scelta dei coefficienti in base all'architettura proposta. Precisiamo anzitutto che siamo interessati al throughput piuttosto che alla latenza e quindi il dispositivo si inserisce bene nel contesto dei DSP. Se invece avessimo intenzione di ottimizzare la latenza, il circuito troverebbe maggiore applicazione nell'ambito dei processori general-purpose. Precisato l'obiettivo, possiamo pensare di calcolare il valore del polinomio nel seguente modo

$$\tilde{p}(x) = (\tilde{c}_2 \cdot \Delta x + \tilde{c}_1) \cdot \Delta x + \tilde{c}_0$$

In particolare scomponiamo l'unità di calcolo in due unità moltiplicatore/sommatore concatenate



Un’altra architettura possibile sarebbe quella di sviluppare tutto su un piano orizzontale [9].



Possiamo pensare che un’architettura di questo tipo introduca una latenza minore visto che i due moltiplicatori lavorano in parallelo. Si evidenzia però la necessità di sviluppare un quadratore e un accumulatore con tre ingressi. Riteniamo quindi che una scelta di questo tipo permetta di ottenere delle performance migliori, a scapito, però, di una maggiore area di silicio impiegata.

Assumiamo d’ora in avanti che l’architettura che verrà implementata sia la prima che abbiamo proposto e facciamo un certo numero di osservazioni con l’intento di risparmiare quanto più possibile sul numero di bit della look-up table.

a) Ciascuno dei tre coefficienti del polinomio approssimante risulta di segno costante per tutti i sottointervalli in cui si è diviso l’intervallo iniziale. Questo ci permette di sviluppare le moltiplicazioni con interi senza segno, risparmiando un bit della look-up table per ognuno dei coefficienti.

b) Il secondo coefficiente $\tilde{c}_{1i}$ è sempre negativo e in modulo maggiore di $\tilde{c}_{2i} \cdot \Delta x$, qualunque sia l'indice i e qualunque sia Δx. Si utilizza quindi un sottrattore piuttosto che un sommatore e il risultato p_1 in ingresso alla seconda unità risulta quindi cambiato di segno. Ragionando in modo analogo e sapendo a priori che il risultato è sempre positivo, si ottengono le stesse semplificazioni anche per la seconda unità.

c) Una ulteriore importante osservazione che facciamo, sempre con l'intento di diminuire il numero di bit memorizzati nella look-up table, è che i valori 18,14,10 per la rappresentazione dei coefficienti sono stati ottenuti attraverso una maggiorazione ottenuta considerando il caso più sfavorevole. L'idea è di parametrizzare il numero di bit su cui vengono rappresentate le quantità in ingresso alle due unità moltiplicatore/sommatore. Procedendo per tentativi sul valore di questi parametri, si cerca una soluzione che dia ancora un errore massimo di 1 LSB, ma che permetta di risparmiare sulle risorse hardware impiegate. Descrivendo il dispositivo calcolatore in C++ secondo la programmazione strutturata ad oggetti (si rimanda all'appendice per i dettagli), si può automatizzare la procedura.

d) Si può inoltre risparmiare un po' di logica arrotondando il risultato delle moltiplicazioni su un numero di bit inferiore, commettendo un piccolo errore del quale sarà tenuto conto, ma risparmiando sulle dimensioni dei sottrattori.

Definiamo perciò i parametri sui quali svilupperemo il codice che simula il comportamento dell'unità di calcolo. Si rappresentano i numeri $\tilde{c}_0, \tilde{c}_1, \tilde{c}_2$ mediante i numeri interi $\tilde{c}_{0R}, \tilde{c}_{1R}, \tilde{c}_{2R}$ così che

$$\begin{aligned}
\tilde{c}_0 &= \tilde{c}_{0R} \cdot 2^{-\mathrm{bitC}_0} \\
\tilde{c}_1 &= \tilde{c}_{1R} \cdot 2^{-\mathrm{bitC}_1} \\
\tilde{c}_2 &= \tilde{c}_{2R} \cdot 2^{-\mathrm{bitC}_2}
\end{aligned}$$

Si ha poi per Δx l'espressione

$$\Delta x = \Delta x_R \cdot 2^{-\mathrm{bitDif}} \cdot 2^{-4}$$

Oltre ai parametri $bitC_0, bitC_1, bitC_2, bitDif$ si definiscono anche i due parametri $bitMolt_1$ e $bitMolt_2$ che sono rispettivamente il numero di bit su cui è rappresentato il primo risultato parziale p_1 e il numero di bit su cui è rappresentato il risultato finale.

Per quel che concerne il dispositivo che vogliamo sintetizzare, i parametri $bitMolt_2$ e bitDif vengono mantenuti costanti e pari a 16 e 12 rispettivamente. Partendo poi dai valori 18, 14, 10, 15 rispettivamente per $bitC_0, bitC_1, bitC_2, bitMolt_1$ e provando su un ampio numero di configurazioni di questi 4 parametri si valuta l'errore per ciascuna delle configurazioni assegnate. Non resta, poi, che scegliere la configurazione in funzione del massimo errore permesso.

Si riportano nella seguente tabella i risultati più significativi, cioè quelli che, a parità di errore, minimizzano le risorse hardware impiegate.

$bitC_0$	$bitC_1$	$bitC_2$	bitDif	$bitMolt_1$	$bitMolt_2$	Errore	Somma bit LUT	Somma bit Totale	Precisione del risultato in Bit
17	13	7	12	16	16	1.53E-05	**37**	81	**15.00**
17	11	9	12	16	16	1.53E-05	**37**	81	**15.00**
17	12	7	12	16	16	1.56E-05	**36**	80	**14.97**
17	11	8	12	16	16	1.56E-05	**36**	80	**14.97**
17	10	8	12	14	16	1.68E-05	**35**	77	**14.86**
16	11	7	12	15	16	1.78E-05	**34**	77	**14.78**
16	10	7	12	15	16	1.85E-05	**33**	76	**14.72**
16	9	7	12	15	16	2.25E-05	**32**	75	**14.44**
16	9	6	12	14	16	2.64E-05	**31**	73	**14.21**
15	11	5	12	14	16	2.64E-05	**31**	73	**14.21**
15	9	7	12	14	16	2.64E-05	**31**	73	**14.21**
16	8	6	12	14	16	3.06E-05	**30**	72	**14.00**
15	8	6	12	14	16	3.39E-05	**29**	71	**13.85**
15	8	6	12	13	16	3.60E-05	**29**	70	**13.76**
14	10	4	12	15	16	4.03E-05	**28**	71	**13.60**
14	8	6	12	15	16	4.03E-05	**28**	71	**13.60**
15	7	5	12	15	16	4.85E-05	**27**	70	**13.33**
15	7	5	12	14	16	4.86E-05	**27**	69	**13.33**
14	7	5	12	13	16	5.58E-05	**26**	67	**13.13**
13	7	5	12	14	16	6.95E-05	**25**	67	**12.81**
13	7	5	12	13	16	6.95E-05	**25**	66	**12.81**

Tabella 3.1

Abbiamo già detto che vogliamo costruire un divisore che commetta un massimo errore di 1 LSB e quindi possiamo scegliere tra 17,11,9,16 e 17,13,7,16; scegliamo la seconda terna in base all'osservazione che è meglio portare 13 bit in ingresso al sottrattore della prima unità piuttosto che portarne 2 in più al moltiplicatore.

Qualora si voglia, poi, modificare le specifiche sull'errore, si può leggere dalla tabella i nuovi valori per il numero di bit delle quantità $\tilde{c}_{0R}$, $\tilde{c}_{1R}$, $\tilde{c}_{2R}$, p_{1R} e modificare di conseguenza l'architettura dell'approssimatore. È importante osservare che questa riduzione del numero di bit su cui sviluppare i calcoli comporta ovviamente un risparmio in termini delle aree della look-up table, dei moltiplicatori e dei sommatori, ma anche un risparmio in termini di tempo, visto che per ogni bit che si aggiunge aumenta la latenza dell'intero sistema.

Si possono leggere nella tabella che segue i coefficienti $\tilde{c}_{0R}$, $\tilde{c}_{1R}$, $\tilde{c}_{2R}$, le rispettive unità di spostamento rispetto ai coefficienti ottenuti dall'arrotondamento di quelli ideali $\hat{c}_{0R}$, $\hat{c}_{1R}$, $\hat{c}_{2R}$, e l'errore massimo commesso su ciascun sottointervallo, determinati dalla scelta fatta.

Indice	x_i	x_{i+1}	$\tilde{c}_{0R}$	$\tilde{c}_{1R}$	$\tilde{c}_{2R}$	E_{arr}	E_{finale}	$\tilde{c}_{0R} - \hat{c}_{0R}$	$\tilde{c}_{1R} - \hat{c}_{1R}$	$\tilde{c}_{2R} - \hat{c}_{2R}$	Guadagno
0	1.0000	1.0625	131071	-8176	117	1.53E-05	1.53E-05	0	0	0	0.0%
1	1.0625	1.1250	123361	-7244	98	1.51E-05	1.51E-05	0	0	0	0.0%
2	1.1250	1.1875	116508	-6463	83	1.86E-05	1.28E-05	0	-1	0	30.9%
3	1.1875	1.2500	110376	-5802	71	1.75E-05	1.32E-05	0	-1	0	24.9%
4	1.2500	1.3125	104857	-5236	61	1.22E-05	1.22E-05	0	0	0	0.0%
5	1.3125	1.3750	99864	-4750	53	1.34E-05	1.34E-05	0	0	0	0.0%
6	1.3750	1.4375	95325	-4328	46	1.12E-05	1.12E-05	0	0	0	0.0%
7	1.4375	1.5000	91180	-3959	40	1.86E-05	1.16E-05	0	1	0	37.8%
8	1.5000	1.5625	87381	-3635	35	1.39E-05	1.18E-05	0	3	-1	14.9%
9	1.5625	1.6250	83886	-3354	32	1.72E-05	1.03E-05	0	-1	0	40.2%
10	1.6250	1.6875	80660	-3100	28	1.14E-05	1.14E-05	0	0	0	0.0%
11	1.6875	1.7500	77672	-2874	25	1.79E-05	1.08E-05	0	1	0	39.6%
12	1.7500	1.8125	74898	-2674	23	1.77E-05	1.04E-05	0	-1	0	41.1%
13	1.8125	1.8750	72316	-2495	21	2.39E-05	1.19E-05	1	-3	1	50.0%
14	1.8750	1.9375	69905	-2327	18	2.39E-05	1.11E-05	0	2	0	53.5%
15	1.9375	2.0000	67650	-2182	17	1.42E-05	1.05E-05	0	-1	0	26.3%

Tabella 3.2

Siamo in grado a questo punto di confrontare la dimensione della nostra look-up table con le dimensioni di quelle dei metodi visti nel primo capitolo. Volendo il risultato con 15 bit esatti si ha per la nostra look-up table

$$N.Bit = 37 \times 16 = 592$$

Ricordando che, con il metodo del polinomio interpolatore con i nodi di Chebychev il corrispondente numero di bit era pari a 800, possiamo concludere che il miglioramento ottenuto nella dimensione della look-up table è di circa il 26%.

Esaminiamo ora analiticamente come viene ottenuto il risultato. Le relazioni tra le rappresentazioni intere $\tilde{c}_{0R}$, $\tilde{c}_{1R}$, $\tilde{c}_{2R}$ e i coefficienti $\tilde{c}_0$, $\tilde{c}_1$, $\tilde{c}_2$ sono:

$$\begin{aligned}
\tilde{c}_0 &= \tilde{c}_{0R} \cdot 2^{-17} \\
\tilde{c}_1 &= \tilde{c}_{1R} \cdot 2^{-13} \\
\tilde{c}_2 &= \tilde{c}_{2R} \cdot 2^{-7}
\end{aligned}$$

e $\Delta x = \Delta x_R \cdot 2^{-16}$, Δx_R numero intero su 12 bit.

Per il valore del polinomio si ha

$$\tilde{p}(x) = (\tilde{c}_2 \cdot \Delta x + \tilde{c}_1) \cdot \Delta x + \tilde{c}_0 = (\tilde{c}_{2R} \cdot 2^{-7} \cdot \Delta x_R \cdot 2^{-16} + \tilde{c}_{1R} \cdot 2^{-13}) \cdot \Delta x_R \cdot 2^{-16} + \tilde{c}_{0R} \cdot 2^{-17}.$$

È evidente che i bit meno significativi del prodotto $\tilde{c}_{2R} \cdot \Delta x_R$ sono trascurabili. Sappiamo dalla configurazione scelta che il risultato parziale p_1 in uscita dalla prima unità è su 16 bit. Posto $p_{intermedio} = \tilde{c}_{2R} \cdot \Delta x_R \cdot 2^{-23} + \tilde{c}_{1R} \cdot 2^{-13}$, rappresentando $p_{intermedio}$ su un bit in più per la gestione dell'arrotondamento, prendiamo i 17 bit più significativi del prodotto $\tilde{c}_{2R} \cdot \Delta x_R$; si ha allora

$$p_{intermedio} = 2^{-17} \cdot (\tilde{c}_{2R} \cdot \Delta x_R \cdot 2^{-6} + \tilde{c}_{1R} \cdot 2^{+4}) = p_{int\,R} \cdot 2^{-17}$$

ovvero i sei bit meno significativi del prodotto vengono eliminati. Per ottenere un corretto arrotondamento, il bit meno significativo dell'operando $\tilde{c}_{1R} \cdot 2^{+4}$ in ingresso al sottrattore viene collegato a Vcc in modo che si possa poi troncare il risultato su 16 bit anziché 17 e ottenere l'arrotondamento che volevamo (di questo fatto si è tenuto conto nel software di simulazione). Si ottiene in definitiva in uscita dalla prima unità

$$p_1 = 2^{-16} \cdot p_{1R}, \qquad p_{1R} = \lfloor p_{\text{int}\,R} \cdot 2^{-1} \rfloor$$

Per quanto riguarda la seconda unità si ha

$$\tilde{p}(x) = p_1 \cdot \Delta x + \tilde{c}_0 = p_{1R} \cdot 2^{-16} \cdot \Delta x_R \cdot 2^{-16} + \tilde{c}_{0R} \cdot 2^{-17}$$

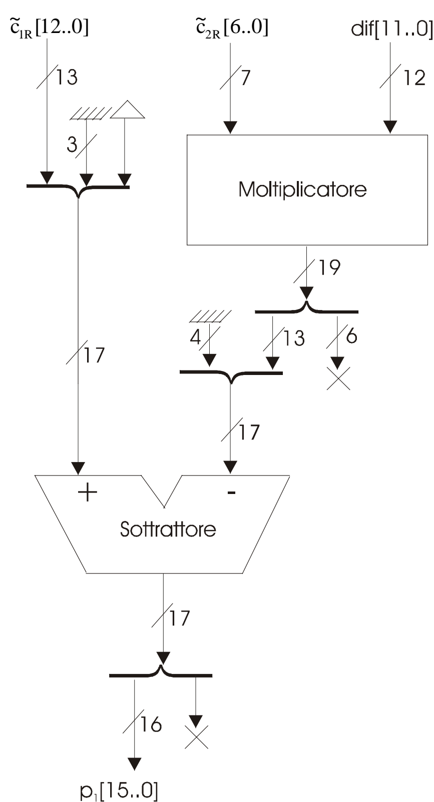
Dovendo restituire il risultato su 16 bit, non si può riutilizzare l'accorgimento usato in precedenza per gestire l'arrotondamento. È in ogni modo già stato deciso dal software quale sia il valore di $\tilde{c}_{0R}$ per minimizzare l'errore.

$$\tilde{p}(x) = 2^{-17} \cdot (p_{1R} \cdot \Delta x_R \cdot 2^{-15} + \tilde{c}_{0R})$$

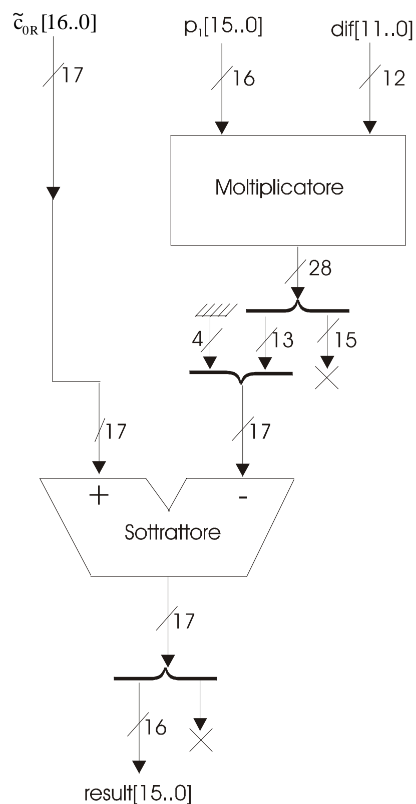
Si osserva che vengono in questo caso eliminati i 15 bit meno significativi del prodotto $p_{1R} \cdot \Delta x_R$ e troncando successivamente il risultato su 16 bit, si ottiene il valore finale.

Dalle considerazioni fatte si deduce infine lo schema che segue

1º Moltiplicatore-Sommatore



2º Moltiplicatore-Sommatore



Nei grafici nelle due pagine seguenti si mostra l'errore commesso in funzione dell'operando di ingresso x, con $x \in [1,2[$. Si nota che per alcuni sottointervalli l'inviluppo presenta l'ondulazione caratteristica di come sono stati determinati i polinomi. Per quelli di indice superiore a 4 l'ondulazione del polinomio ideale è così bassa rispetto agli errori di troncamento che nell'inviluppo non è più riconoscibile.

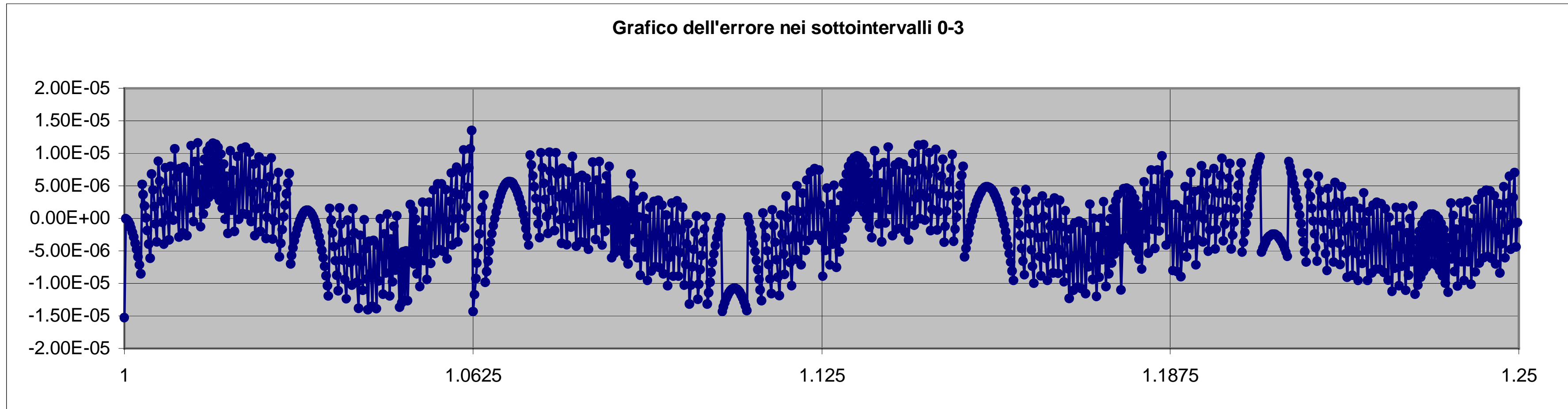


Grafico dell'errore nei sottointervalli 0-3
2.00E-05
1.50E-05
1.00E-05
5.00E-06
0.00E+00
-5.00E-06
-1.00E-05
-1.50E-05
-2.00E-05
1
1.0625
1.125
1.1875
1.25

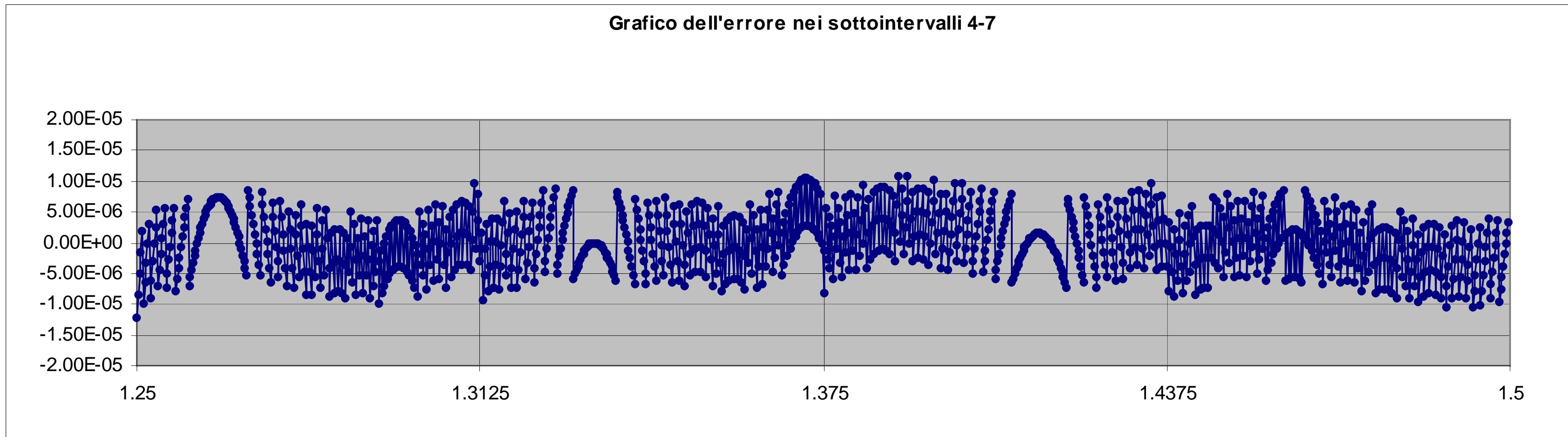


Grafico dell'errore nei sottointervalli 4-7
2.00E-05
1.50E-05
1.00E-05
5.00E-06
0.00E+00
-5.00E-06
-1.00E-05
-1.50E-05
-2.00E-05
1.25
1.3125
1.375
1.4375
1.5

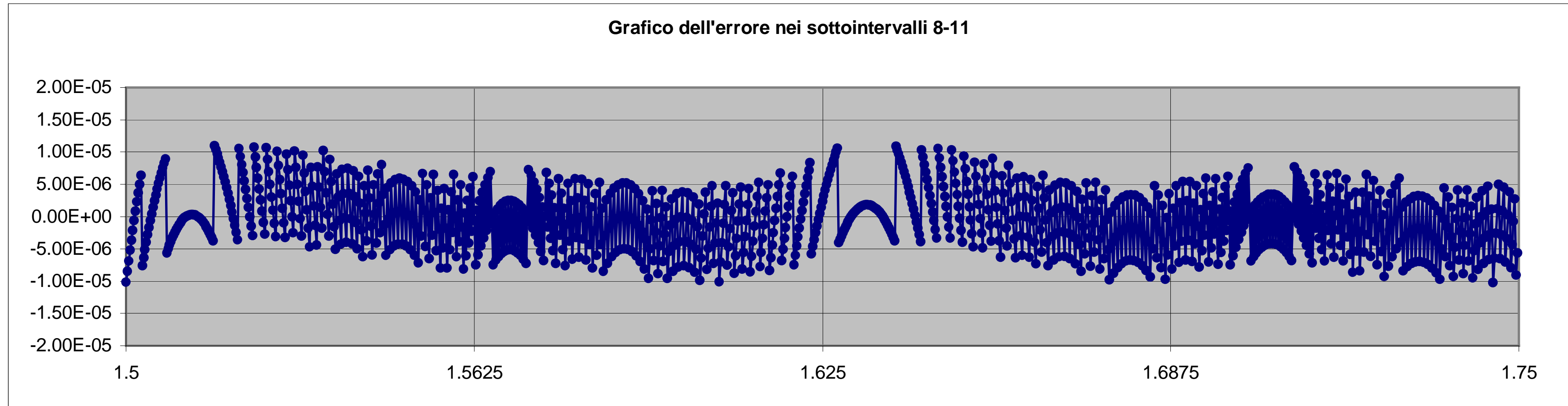


Grafico dell'errore nei sottointervalli 8-11
2.00E-05
1.50E-05
1.00E-05
5.00E-06
0.00E+00
-5.00E-06
-1.00E-05
-1.50E-05
-2.00E-05
1.5
1.5625
1.625
1.6875
1.75

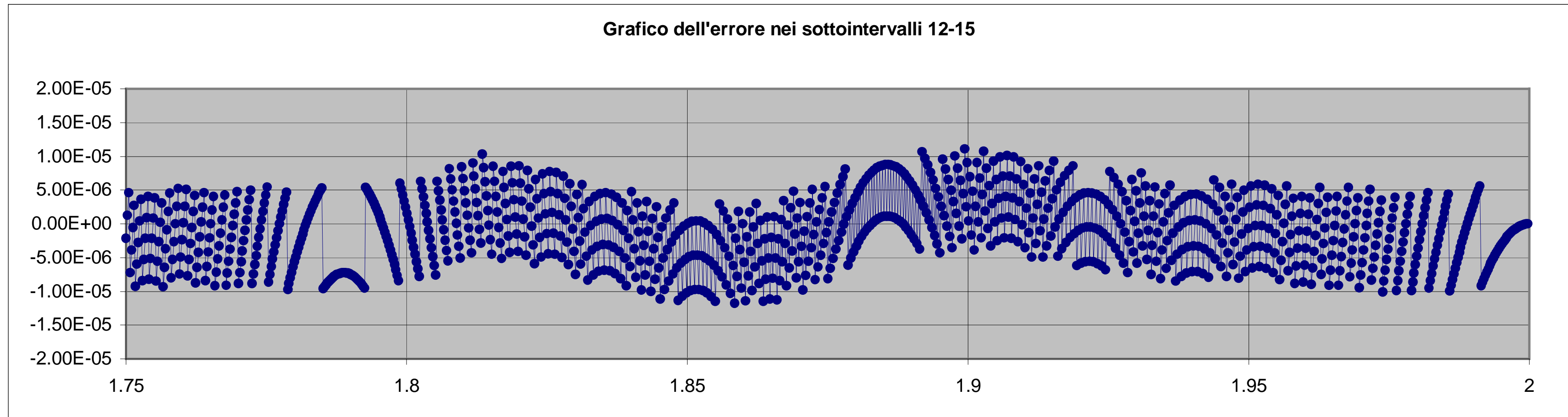


Grafico dell'errore nei sottointervalli 12-15
2.00E-05
1.50E-05
1.00E-05
5.00E-06
0.00E+00
-5.00E-06
-1.00E-05
-1.50E-05
-2.00E-05
1.75
1.8
1.85
1.9
1.95
2

3.2 Il tool di sviluppo

Le considerazioni fin adesso fatte prescindono dalla tecnologia a disposizione per sintetizzare questo dispositivo, ed in particolare possono essere ritenute valide sia in ambito di progettazione ASIC sia in ambito di progettazione a componenti Standard.

Dobbiamo ora fare comunque una scelta su come portare avanti questo progetto e su quali tecnologie. La scelta che facciamo si orienta verso i dispositivi programmabili (Programmable Logic Array) e ciò per duplice motivo. In primo luogo a questo livello della sintesi è sempre utile partire con un prototipo dal quale si possono ottenere dei risultati sperimentali in tempi brevi e con costi ridotti, visto che un dispositivo programmabile può essere riprogrammato e dunque riutilizzato a piacere. In secondo luogo se il progetto porta a dei risultati concreti, esso può essere utilizzato come oggetto libreria per progetti nello stesso ambito. Inoltre la scelta che si fa non preclude la possibilità di implementare una diversa architettura e/o sviluppare il progetto su tecnologie differenti in un secondo momento.

L'ambiente di sviluppo di cui si fa uso d'ora in avanti è Max + Plus II dell'americana Altera. Quest'ultima produce (tra le altre cose) un certo numero di dispositivi programmabili; cercheremo nel contesto del nostro lavoro di implementare il divisore su una FLEX10K20-RC240-4.

3.3 Descrizione dell'hardware in AHDL

Nelle sezioni seguenti riportiamo il codice AHDL dell'hardware di ciascuno dei blocchi funzionali che costituiscono nel loro insieme il dispositivo divisore.

3.3.1 I blocchi Moltiplicatori Sommatori

La descrizione in AHDL dei due blocchi moltiplicatori sommatori è immediata, qualora si faccia uso degli oggetti di libreria parametrizzabili (Altera Megafunctions) presenti nel tool di sviluppo.

```
Include "Molt1.Inc";
Include "Sub1.Inc";
SUBDESIGN MoltSomm1 (
     clock,c2[6..0],dif[11..0],c1[12..0]:input;
```

```
      result[15..0]:output;
      )
VARIABLE
      multiplier:Molt1;
      sub:Sub1;
      registerMult[12..0]:DFF;

BEGIN
      multiplier.clock = clock;
      multiplier.dataa[6..0] = c2[6..0];
      multiplier.datab[11..0] = dif[11..0];

      registerMult[12..0].CLK = clock;
      registerMult[12..0].D=multiplier.result[12..0];

      sub.dataa[16..4] = c1[12..0];
      sub.dataa[3..0] = 1;
      sub.datab[16..13] = 0;
      sub.datab[12..0] = registerMult[12..0].Q;

      result[15..0] = sub.result[16..1];
END;

INCLUDE "Molt2.Inc";
INCLUDE "Sub2.Inc";
SUBDESIGN MoltSomm2 (
      clock,parz[15..0],dif[11..0],c0[16..0]:input;
      result[15..0]:output;
      )
VARIABLE
      multiplier:Molt2;
      sub:Sub2;
      registerMult[12..0]:DFF;
      registerdifa[11..0],registerdifb[11..0]:DFF;
BEGIN
      registerdifa[11..0].CLK = clock;
      registerdifb[11..0].CLK = clock;
      registerdifa[11..0].D = dif[11..0];
      registerdifb[11..0].D = registerdifa[11..0].Q;

      multiplier.clock = clock;
      multiplier.dataa[15..0] = parz[15..0];
      multiplier.datab[11..0] = registerdifb[11..0].Q;

      registerMult[12..0].CLK = clock;
      registerMult[12..0].D=multiplier.result[12..0];

      sub.dataa[16..0] = c0[16..0];
      sub.datab[16..13] = 0;
      sub.datab[12..0] = registerMult[12..0].Q;

      result[15..0] = sub.result[16..1];
END;
```

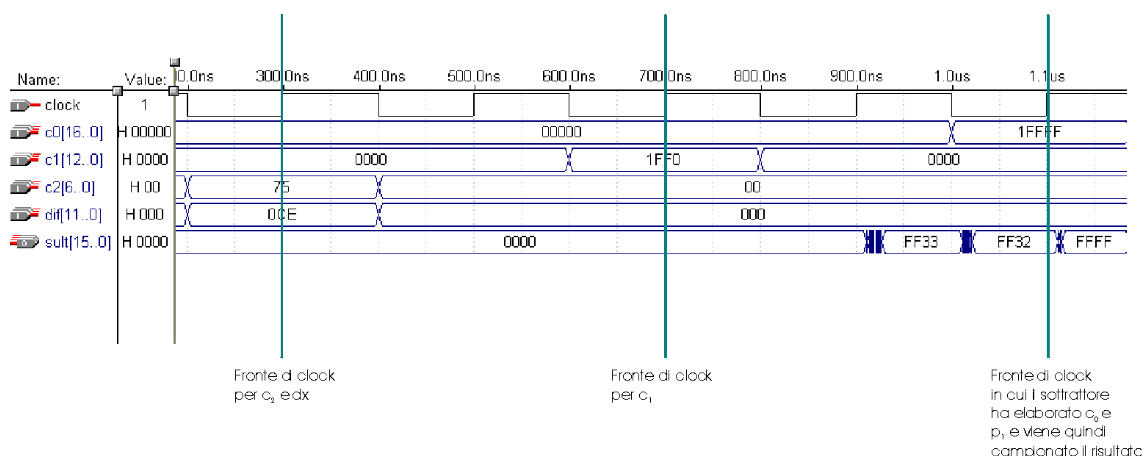
Si nota la distribuzione del clock anche ai blocchi moltiplicatori, che di per sé sarebbero normali reti combinatorie. Il clock viene portato ai moltiplicatori perché si decide a questo stadio della sintesi di inserire dei registri di pipeline all'interno dei moltiplicatori e un registro tra moltiplicatore e sottrattore. Questa possibilità ci viene data nonostante che i blocchi moltiplicatori siano precostituiti. Si può cioè decidere tra i vari parametri del moltiplicatore il numero di registri da distribuire all'interno del blocco funzionale. Più precisamente, sia per il primo moltiplicatore sia per il secondo viene deciso di inserire un solo registro interno, mentre le unità sottrattore sono normali reti combinatorie.

3.3.2 Il dispositivo approssimatore

Il successivo blocco che sintetizziamo lo chiamiamo Approssimatore, ed è diretta conseguenza della connessione dei due blocchi appena descritti.

```
Function MoltSomm1
(c1[12..0],c2[6..0],clock,dif[11..0]) Returns (result[15..0]);

Function MoltSomm2
(c0[16..0],parz[15..0],clock,dif[11..0]) Returns(result[15..0]);
SUBDESIGN approssimatore (
       c0[16..0],c1[12..0],c2[6..0],clock,dif[11..0]:input;
       result[15..0]:output;
       )
VARIABLE
      molt2: MoltSomm2;
      molt1: MoltSomm1;
BEGIN
molt1.clock = clock;
molt1.c1[12..0] = c1[12..0];
molt1.c2[6..0] = c2[6..0];
molt1.dif[11..0] = dif[11..0];

molt2.clock = clock;
molt2.parz[15..0] = molt1.result[15..0];
molt2.c0[16..0] = c0[16..0];
molt2.dif[11..0] = dif[11..0];

result[15..0] = molt2.result[15..0];
END;
```

A questo punto il cuore del sistema è pronto, rimane da descrivere il blocco che abbiamo definito come coordinatore. Facciamo allora una considerazione su come devono essere inseriti i coefficienti e l'operando all'ingresso dell'approssimatore e approfittiamo dell'occasione per fare anche una verifica numerica della correttezza del risultato ottenuto in uscita per un possibile operando in ingresso. Come si può dedurre dal circuito descritto e dalla figura che segue, il coefficiente c_0 deve essere ritardato di 4 cicli di clock, c_1 di 2 cicli, e c_2 e Δx in fase.



$$\Delta x_R = 0CEh = 206$$

$$\tilde{c}_{2R} = 75h = 117$$

$$-\tilde{c}_{1R} = 1FF0h = 8176$$

$$\tilde{c}_{0R} = 1FFFF = 131071$$

Si ottiene

$$p_{\text{int}\,R} = -\tilde{c}_{1R} \cdot 2^4 + 1 - \left\lfloor \tilde{c}_{2R} \cdot \Delta x_R \cdot 2^{-6} \right\rfloor = 130441$$

$$p_{1R} = \left\lfloor p_{\text{int}\,R} \cdot 2^{-1} \right\rfloor = 65220$$

$$\tilde{p}_R = \left\lfloor \left\{ \tilde{c}_{0R} - \left\lfloor p_{1R} \cdot \Delta x_R \cdot 2^{-15} \right\rfloor \right\} \cdot 2^{-1} \right\rfloor = \left\lfloor \{131071 - 410\} \cdot 2^{-1} \right\rfloor = 65330 = FF32h$$

Per l'errore si ha

$$e = 65330 \cdot 2^{-16} - \frac{1}{1 + 206 \cdot 2^{-16}} \cong -9{,}85 \cdot 10^{-6}$$

3.3.3 Il blocco coordinatore

La ROM che in un progetto ASIC potrebbe essere facilmente sintetizzata con 16 righe di 37 bit ciascuna, non può essere implementata con altrettanta facilità in un dispositivo programmabile. Abbiamo a disposizione dei blocchi (Embedded Array Blocks) di 256 righe di 8 bit, che possono essere trasformati in blocchi da 512 X 4, da 1024 X 2, oppure da 2048 X 1. Il nostro problema è che dobbiamo invece estendere il bus dati, da 8 bit a 37. Scegliamo di utilizzare 4 blocchi EAB ottenendo 32 bit, e gli altri 5 verranno sintetizzati come reti combinatorie. In questo stesso elemento di progetto inseriamo anche i registri per lo skewing dei coefficienti in ingresso all'approssimatore.

Prima di sintetizzare la ROM conviene decidere quali siano i bit dei coefficienti da sintetizzare come rete combinatoria. A tal fine si usa la tabella dei coefficienti in formato binario.

	Indirizzo				$\tilde{c}_{2R}$							$\tilde{c}_{1R}$													$\tilde{c}_{0R}$																
0	0	0	0	0	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	0	0	0	1	0	1	1	1	0	0	0	1	0	0	1	1	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1
2	0	0	1	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0
3	0	0	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	1	1	1	0	0	1	0	1	0	0	0
4	0	1	0	0	0	1	1	1	1	0	1	1	0	1	0	0	0	1	1	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
5	0	1	0	1	0	1	1	0	1	0	1	1	0	0	1	0	1	0	0	0	1	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0
6	0	1	1	0	0	1	0	1	1	1	0	1	0	0	0	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	1
7	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	1	0	1	1	1	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1	0	1	1	0	0
8	1	0	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0	0	1	1	0	0	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
9	1	0	0	1	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	1	1	0	1	0	1	0	1	0	0	0	1	1	1	1	0	1	0	1	1	1	0
10	1	0	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1	0	0	1	0	0	1	1	1	0	1	1	0	0	0	1	0	1	0	0
11	1	0	1	1	0	0	1	1	0	0	1	0	1	0	1	1	0	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	1	1	0	1	0	0	0
12	1	1	0	0	0	0	1	0	1	1	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
13	1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	1	1	0	0	0	1	1	0	1	0	0	1	1	1	1	1	0	0
14	1	1	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	1	1	1	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
15	1	1	1	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
	3	**2**	**1**	**0**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Tabella 3.3

Si osserva subito che il bit n.16 di $\tilde{c}_{0R}$ è sempre a uguale 1, quindi non necessiterà di alcuna logica combinatoria di sintesi, in altre parole si risparmia un ulteriore bit della look-up table ottenendo come numero di bit nella LUT

$$\text{N.Bit} = 36 \times 16 = 576$$

Sintetizzando tutti i bit della tabella come reti combinatorie si osserva che quelli di minor costo sono quelli evidenziati in figura. Non è un caso che siano inoltre i bit più significativi dei coefficienti, data la regolarità della funzione che stiamo sintetizzando. Segue il codice AHDL di descrizione del blocco.

```
Include "RomLib.inc";

SUBDESIGN Coordinatore (
      clock,address[3..0],dif[11..0]:input;
      c2[6..0],c1[12..0],c0[16..0],difrit[11..0]:output;
      )
VARIABLE
      memoria:RomLib;

      rc0,rc2b,rc2a,rc1:NODE;

      regdif[11..0]:DFF;
      regaddress[3..0]:DFF;
      regc0d[15..0]:DFF;
      regc0c[15..0]:DFF;
      regc0b[15..0]:DFF;
      regc0a[15..0]:DFF;
      regc1b[12..0]:DFF;
      regc1a[12..0]:DFF;

BEGIN
      regaddress[3..0].CLK = clock;
      regdif[11..0].CLK = clock;
      regc0d[15..0].CLK = clock;
      regc0c[15..0].CLK = clock;
      regc0b[15..0].CLK = clock;
      regc0a[15..0].CLK = clock;
      regc1b[12..0].CLK = clock;
      regc1a[12..0].CLK = clock;

      regaddress[3..0].D = address[3..0];

      memoria.address[3..0] = regaddress[3..0].Q;

      TABLE regaddress[3..0].Q => rc2b,rc2a,rc1,rc0;
```

```
        B"0000"=>1,1,1,1;
        B"0001"=>1,1,1,1;
        B"0010"=>1,0,1,1;
        B"0011"=>1,0,1,1;
        B"0100"=>0,1,1,1;
        B"0101"=>0,1,1,1;
        B"0110"=>0,1,1,0;
        B"0111"=>0,1,0,0;
        B"1000"=>0,1,0,0;
        B"1001"=>0,1,0,0;
        B"1010"=>0,0,0,0;
        B"1011"=>0,0,0,0;
        B"1100"=>0,0,0,0;
        B"1101"=>0,0,0,0;
        B"1110"=>0,0,0,0;
        B"1111"=>0,0,0,0;
    END TABLE;


    c2[6] = rc2b;
    c2[5] = rc2a;
    c2[4..0] = memoria.Q[31..27];

    regc1b[12].D = rc1;
    regc1b[11..0].D = memoria.Q[26..15];
    regc1a[12..0].D = regc1b[12..0].Q;

    regc0d[15].D = rc0;
    regc0d[14..0].D = memoria.Q[14..0];
    regc0c[15..0].D = regc0d[15..0].Q;
    regc0b[15..0].D = regc0c[15..0].Q;
    regc0a[15..0].D = regc0b[15..0].Q;

    regdif[11..0].D = dif[11..0];

    c1[12..0] = regc1a[12..0].Q;
    c0[15..0] = regc0a[15..0].Q;
    c0[16]=vcc;
    difrit[11..0] =regdif[11..0].Q;

    END;
```

3.3.4 Il divisore (Hierarchy Project Top)

Non rimane che connettere gli elementi sintetizzati e ottenere ciò che era il nostro obiettivo.

```
Function Coordinatore (clock,address[3..0],dif[11..0])
Returns(c0[16..0],c1[12..0],c2[6..0],difrit[11..0]);
Function Approssimatore
```

```
(clock,c0[16..0],c1[12..0],c2[6..0],dif[11..0])
Returns (result[15..0]);

SUBDESIGN divisore (
      clock,operand[15..0]:input;
      result[15..0]:output;
      )
VARIABLE
      approx : approssimatore;
      romCoef : coordinatore;
      register[15..0]:DFF;

BEGIN
      romCoef.clock = clock;
      romCoef.address[3..0] = operand[15..12];
      romCoef.dif[11..0] = operand[11..0];

      approx.clock = clock;
      approx.c0[16..0] = romCoef.c0[16..0];
      approx.c1[12..0] = romCoef.c1[12..0];
      approx.c2[6..0] = romCoef.c2[6..0];
      approx.dif[11..0] = romCoef.difrit[11..0];

      register[15..0].CLK = clock;
      register[15..0].D = approx.result[15..0];
      result[15..0] = register[15..0].Q;
END;
```

Per concludere si riportano nella seguente tabella i tempi di risposta dei singoli sottoblocchi, e si stima il throughput del dispositivo divisore.

	T_{SETUP}	$T_{PROPDELAY}$	T_{RC}	$Max(T_{RC1}+T_{SETUP}, T_{RC2}+T_{SETUP})$
DFF	3.2 ns	13 ns		
Molt1				45.5 ns
Sub1			58 ns	
Molt2				83.1 ns
Sub2			58 ns	
Rom(4_to_8)			31.1 ns	

Tabella 3.4

I valori riportati in tabella sono estratti mediante l’apposita funzionalità nel tool di sviluppo, e sono caratteristici del dispositivo EPF10K20-RC240-4 già nominato. Si osser-

va che avremo come limite inferiore per il clock $(83+13) \approx 96$ ns .Ci si aspetta quindi di ottenere in uscita un throughput poco superiore a $10 \frac{\text{Msample}}{\text{sec}}$.

Utilizzando il "Timing Analyzer" fornito in dotazione si ottiene un minimo periodo di clock di 90 ns corrispondenti a una massima frequenza di lavoro di 11,1 MHz. Per quanto riguarda la latenza si ha in uscita il risultato dopo 6 cicli di clock.

Osservando poi che il tempo $T_{RC} + T_{SETUP}$ del primo moltiplicatore Molt1 è poco più della metà del tempo del secondo, possiamo eliminare il registro all'interno del primo moltiplicatore, riducendo così la latenza del dispositivo di un ciclo di clock e lasciando pressoché invariato il throughput.

CAPITOLO 4

Le prove sperimentali

4.1 La procedura per la stima dell'errore

Vogliamo in questo capitolo verificare sperimentalmente che il dispositivo funziona, e possibilmente stimare, anche in via sperimentale, il massimo modulo dell'errore commesso.

Riepiloghiamo i risultati ottenuti:

$x \in [1,2[$ è l'operando in ingresso al divisore

$y \in [0,1]$ è il risultato dell'operazione

$$y = \frac{1}{x} + \varepsilon, \quad \varepsilon \text{ errore commesso}$$

$$y \cdot x = \left(\frac{1}{x} + \varepsilon \right) \cdot x = 1 + \varepsilon \cdot x$$

Quindi se sottraiamo 1 alla quantità $y \cdot x$ otteniamo una prima stima dell'errore. Se poi moltiplichiamo ancora quest'ultima quantità per y, otteniamo

$$(y \cdot x - 1) \cdot y = \varepsilon \cdot x \cdot \left(\frac{1}{x} + \varepsilon \right) = \varepsilon + x \cdot \varepsilon^2 \cong \varepsilon$$

Sulla base di queste osservazioni cerchiamo di costruire il dispositivo cui abbiamo accennato.

4.2 La sintesi della "macchina di test"

Con riferimento a ciò che è stato detto nella precedente sezione, siano

$$x = x_R \cdot 2^{-16}, \quad x_R \text{ numero naturale su 17 bit}$$

$$y = y_R \cdot 2^{-16}, \quad y_R \text{ numero naturale su 16 bit}$$

$$y \cdot x = y_R \cdot x_R \cdot 2^{-32}, \; y_R \cdot x_R \text{ numero naturale su 33 bit}$$

Per valutare il prodotto $y \cdot x$ si usa un moltiplicatore senza segno, avente in ingresso un operando su 17 bit e uno su 16. Se l'errore massimo ε che si commette è di 2^{-16} (1 LSB) si ha in uscita un numero che dista al massimo da 1 della quantità $\varepsilon_{max} \cdot x_{max} = 2^{-15}$, quindi il numero in uscita al moltiplicatore è 10000….h oppure 0FFFF….h. Il primo bit del risultato costituisce la codifica della parte intera, che sappiamo già che sarà pari a 0 o a 1 a seconda che l'errore sia positivo o negativo.

Il numero che a noi interessa vale al massimo 2^{-15}, si prendono quindi i bit di peso 2^{-13}, 2^{-14}, e ancora altri 15 bit ottenendo il numero

$$z = \varepsilon \cdot x = z_R \cdot 2^{-29}$$

32	31	30	29	28	…	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	-1	-2	-3	-4	…	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27	-28	-29	-30	-31	-32

Da un punto di vista operativo si scartano i primi 12+1=13 bit del risultato, e si porta questa selezione a un estrattore di valore assoluto (una barriera di invertitori nel caso in cui l'operando sia negativo o un corto circuito altrimenti) e poi di nuovo in ingresso al moltiplicatore, mantenendo su l'altro ingresso sempre il risultato in uscita dal divisore. Dopo questo secondo passo si ha in uscita il modulo dell'errore commesso. In tal modo avremo

$$u = z \cdot y = z_R \cdot 2^{-29} \cdot y_R \cdot 2^{-16} = u_R \cdot 2^{-45}$$

Si rappresenta nella tabella seguente la porzione di bit che verrà portata in uscita dalla macchina di test.

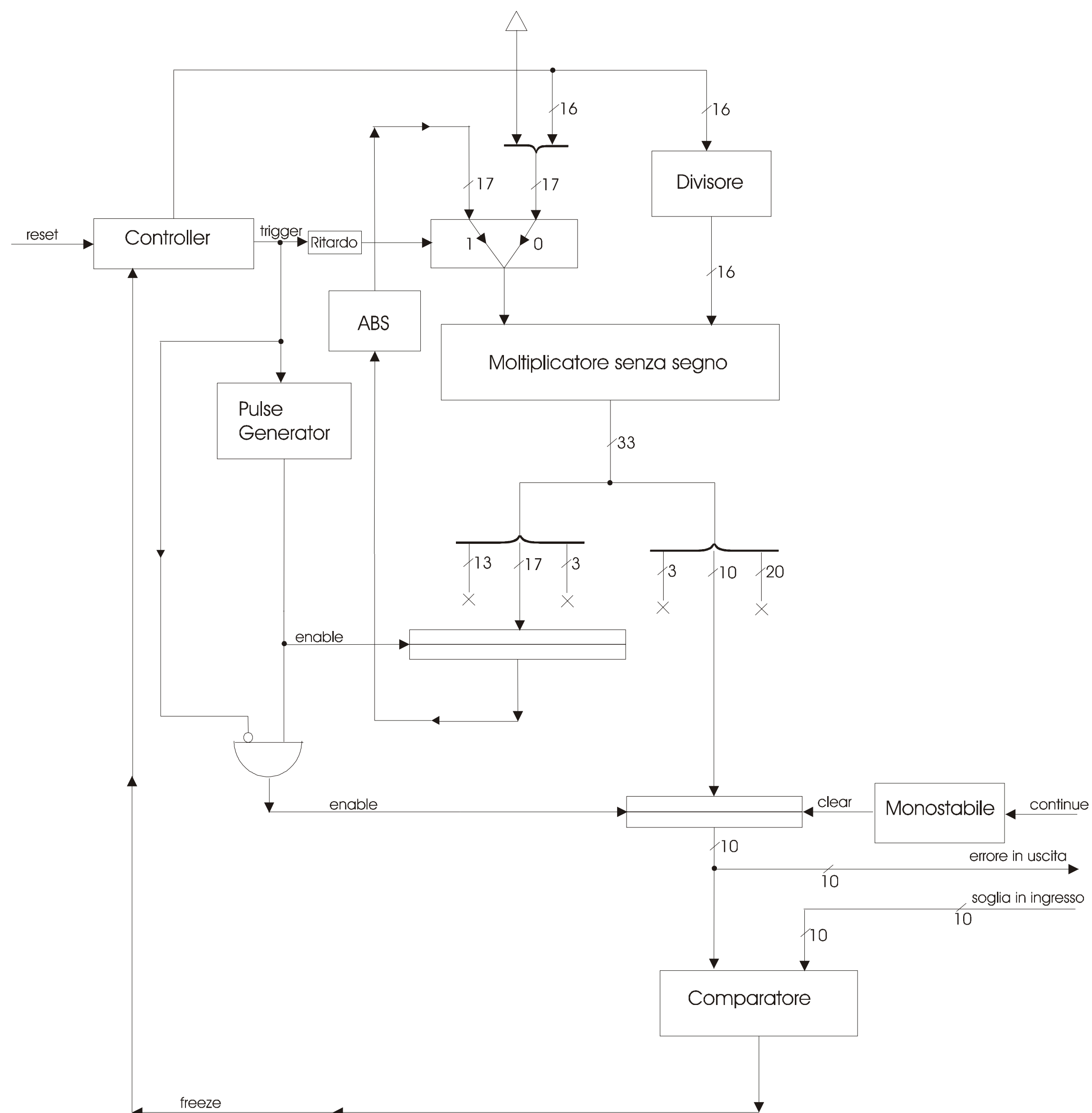
32	31	30	29	28	27	26	25	24	23	22	21	20													2	1	0
-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25													-43	-44	-45

Si ottiene in definitiva

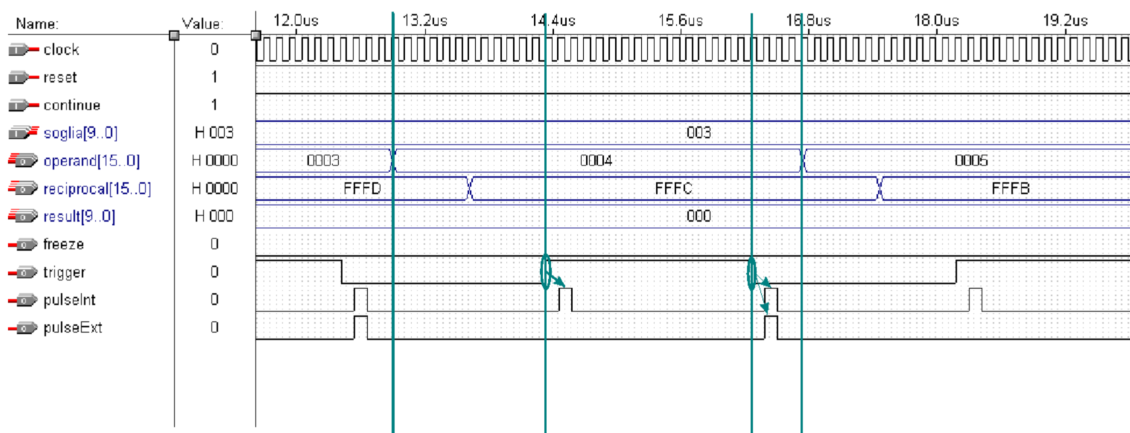
$$\left| \varepsilon \right| = \varepsilon_R \cdot 2^{-25}$$

Con questo dispositivo potremo rilevare in uscita un errore che al massimo può valere in modulo $2^{-25} \cdot 2^{+10} = 2^{-15} \approx 3{,}05 \cdot 10^{-5}$. Quindi nell'ipotesi che il circuito divisore funzioni correttamente, cioè che commetta un errore inferiore a 1 LSB, riusciremo a stimare il modulo massimo dell'errore commesso.

Nello schema che segue si rappresenta l'architettura della macchina di test



Con l'aiuto di una temporizzazione descriviamo il funzionamento di questa macchina.



Diciamo subito che l'unità controller ha al suo interno 2 contatori, uno produce in sequenza tutti i vettori di test (gli operandi) per i quali si vuol valutare il modulo dell'errore, l'altro, invece, serve per scandire il numero di cicli di clock necessari al divisore per restituire il risultato. Il blocco PulseGen riceve in ingresso il clock e un segnale di trigger prodotto dal controller. Il fronte in salita di quest'ultimo segnale avverte che sono passati un numero di cicli di clock sufficienti all'elaborazione dei segnali in ingresso al moltiplicatore. Specificamente è stato calcolato il prodotto tra il nuovo operando x e il suo reciproco. Il successivo fronte in discesa del trigger avverte invece che è pronto il risultato della seconda moltiplicazione, ovvero che è stato calcolato il modulo dell'errore. Il due segnali pulseInt e pulseExt vengono quindi prodotti da pulseGen e abilitano i registri al campionamento. Come si può osservare dalla figura l'operando successivo viene introdotto dal controller 3 cicli di clock dopo. Il motivo di questa scelta è dovuto al fatto che prima di incrementare l'operando, e cominciare quindi un altro test, vogliamo verificare, mediante il comparatore, che la soglia d'errore massimo permesso non è stata superata. A tal fine introduciamo l'uscita del comparatore in ingresso al controller, permettendo a quest'ultimo di "congelare" i contatori al suo interno. Con questo accorgimento l'operatore può leggere il valore dell'operando che ha causato il superamento della soglia. Se si vuole poi poter riavviare il test dal punto in cui è stato interrotto si può portare alla macchina di test un segnale di "sblocco" che chiamiamo *continue*. Una transizione di questo segnale da livello alto al livello basso viene "sentita" da un monostabile che produce un impulso. Quest'ultimo causa l'azzeramento

dell'errore memorizzato nel registro e riabilita quindi i contatori all'interno del controllore. La presenza del monostabile è giustificata dal fatto che un eventuale pulsante per l'invio del segnale di sblocco può produrre inaspettatamente più di un fronte; facciamo in modo quindi che il monostabile invii l'impulso in corrispondenza del primo fronte ricevuto, e si ponga in uno stato di inibizione per un tempo prefissato.

Si riporta qui sotto la descrizione in AHDL dell'hardware.

```
SUBDESIGN Controller (
      clock,freeze,reset:input;
      operand[15..0],trig:output;
      )
VARIABLE
      count[4..0]:DFF;
      regoperand[15..0]:DFF;
BEGIN
      count[4..0].CLK = clock;
      regoperand[15..0].CLK = clock;
      count[4..0].CLRN = reset;
      regoperand[15..0].CLRN = reset;

      If freeze Then
            count[4..0].D = count[4..0].Q;
                  Else
            count[4..0].D = count[4..0].Q + 1;
            End If;
      If count[4..0].Q == 3 & !freeze Then
                  regoperand[15..0].D = regoperand[15..0].Q+1;
                  Else
                  regoperand[15..0].D = regoperand[15..0].Q;
                  End If;
      trig = count[4].Q;
      operand[15..0] = regoperand[15..0].Q;
END;

SUBDESIGN pulseGen (
      clock,trig:input;
      pulseIntReg,pulseOutReg:output;
      )
VARIABLE
      ffin,ffout1:DFF;
BEGIN
      ffin.CLK = clock;
      ffout1.CLK = clock;
      ffin.D = trig;
      ffout1.D = ffin.Q $ trig;
      pulseIntReg = ffout1.Q;
      pulseOutReg = ffout1.Q & !trig;
END;
```

```
Include "MoltTest.inc";
Include "Mux17.inc";

Function ABS (datain[16..0]) Returns (dataout[16..0]);

SUBDESIGN multiplier (
      clock,trig,data0[16..0],dataa[15..0],pulse:input;
      result[9..0]:output;
      )
VARIABLE
      multiplex:mux17;
      molt:MoltTest;
      reg[16..0]:DFFE;
      rit0,rit1:DFF;
      ass:ABS;
BEGIN
      reg[16..0].CLK = clock;
      rit0.CLK = clock;
      rit1.CLK = clock;

      rit0.D = trig;
      rit1.D = rit0.Q;

      multiplex.data0_[16..0] = data0[16..0];
      multiplex.data1_[16..0] = ass.dataout[16..0];
      multiplex.sel = rit1.Q;

      molt.dataa[15..0] = dataa[15..0];
      molt.datab[16..0] = multiplex.result[16..0];

      reg[16..0].ENA = pulse;
      reg[16..0].D = molt.result[16..0];

      ass.datain[16..0] =reg[16..0].Q;

      result[9..0] = molt.result[26..17];
END;

SUBDESIGN ABS (
      datain[16..0]:input;
      dataout[16..0]:output;
      )
BEGIN
      If datain[16] Then
            dataout[16..0]=!datain[16..0];
            Else
            dataout[16..0]=datain[16..0];
            End If;
END;

CONSTANT NUMBIT = 23;
% DT=336 ms per un clock a 12,5 MHz. %
SUBDESIGN Monostabile (
```

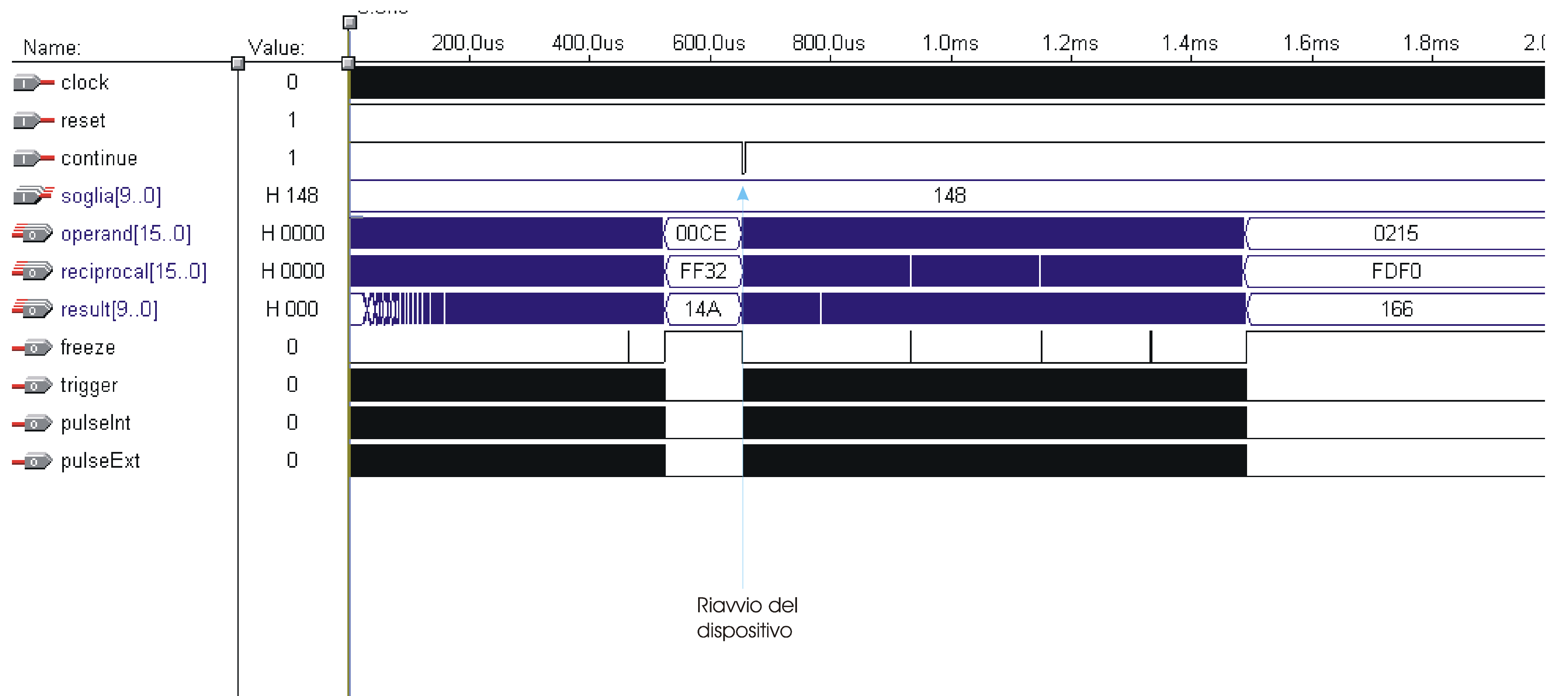
```
      clock, continue: input;
      pulse: output;
      )
VARIABLE
      counter[NUMBIT..0]:DFF;
      ff:DFF;
BEGIN
      counter[NUMBIT..0].CLRN = continue;
      counter[NUMBIT..0].CLK = clock;
      ff.CLK = clock;
      If counter[NUMBIT].Q Then
            counter[NUMBIT..0].D = counter[NUMBIT..0].Q;
      Else
            counter[NUMBIT..0].D = counter[NUMBIT..0].Q+1;
      End If;
      ff.D = counter[NUMBIT].Q;
      pulse = !ff.Q # counter[NUMBIT].Q;
END;

Include "Comp.inc";
Function Controller (reset,clock,freeze)
      Returns (operand[15..0],trig);
Function PulseGen (clock,trig)
      Returns (pulseIntReg,pulseOutReg);
Function Multiplier (clock,trig,pulse,data0[16..0],dataa[15..0])
      Returns (result[9..0]);
Function Monostabile (clock, continue) Returns (pulse);
Function Divisore (operand[15..0],clock)
      Returns (result[15..0]);

SUBDESIGN testonsingleboard (
    clock,reset,soglia[9..0],continue:input;
    result[9..0],operand[15..0],reciprocal[15..0],freeze:output;
    )
VARIABLE
      molt:multiplier;
      contr:controller;
      pulse:pulsegen;
      regusc[9..0]:DFFE;
      compar:comp;
      mono:Monostabile;
      divis:Divisore;

BEGIN
      contr.reset = reset;
      contr.clock = clock;

      pulse.clock = clock;
      pulse.trig = contr.trig;

      divis.clock = clock;
      divis.operand[15..0] = contr.operand[15..0];
```

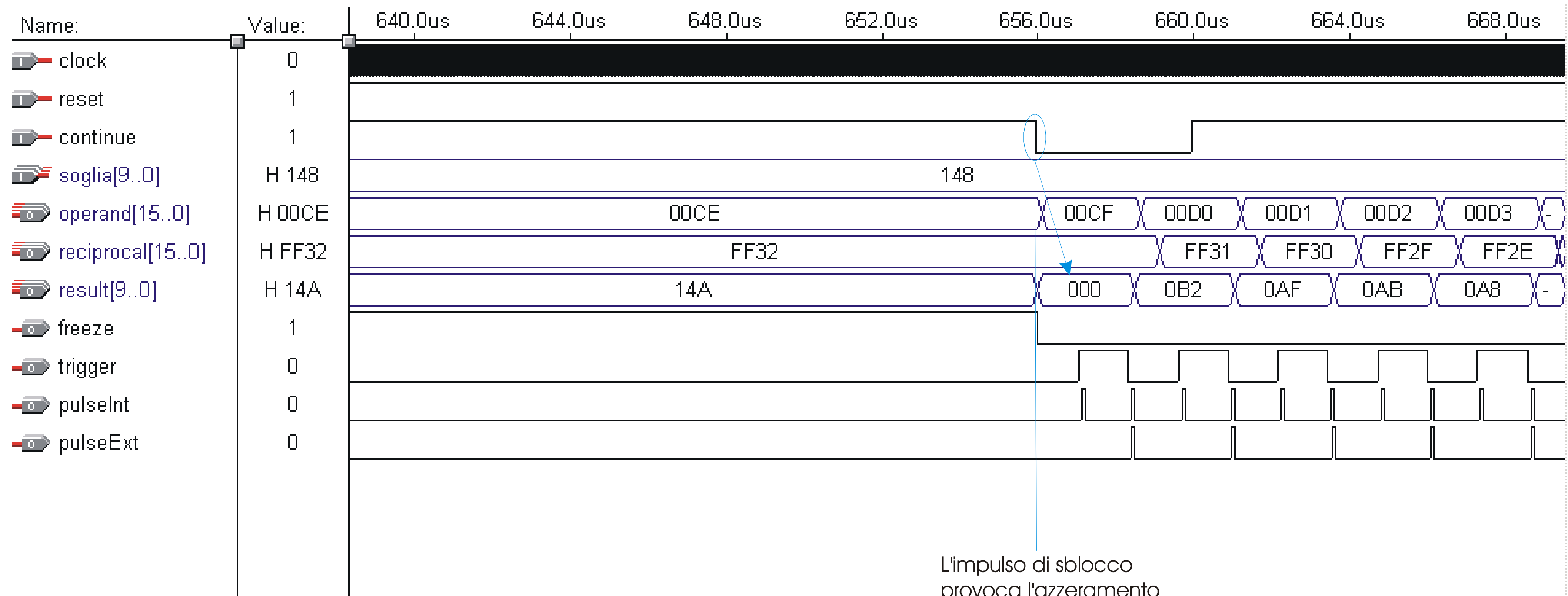
```
        molt.clock = clock;
        molt.trig = contr.trig;
        molt.pulse = pulse.pulseIntReg;
        molt.dataa[15..0] = divis.result[15..0];
        molt.data0[16] = vcc;
        molt.data0[15..0] = contr.operand[15..0];

        regusc[9..0].CLK = clock;
        regusc[9..0].ENA = pulse.pulseOutReg;
        regusc[9..0].D = molt.result[9..0];
        regusc[9..0].CLRN = continue;

        mono.continue = continue;
        mono.clock = clock;

        result[9..0] = regusc[9..0].Q;
        compar.dataa[9..0] = regusc[9..0].Q;
        compar.datab[9..0] = soglia[9..0];
        contr.freeze = compar.agb;
        operand[15..0] = contr.operand[15..0];

        reciprocal[15..0] = divis.result[15..0];
        freeze = compar.agb;
END;
```

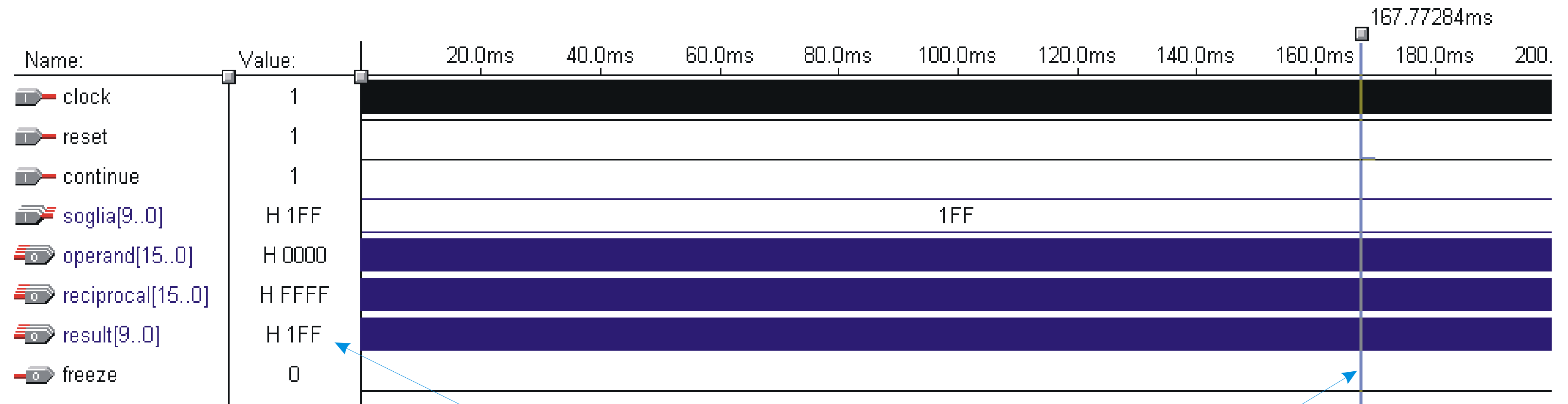
Concludiamo, ora, mostrando tre temporizzazioni. Nella prima si stabilisce per la soglia un valore di $9{,}8 \cdot 10^{-6} \cong 0{,}64\ \text{LSB}$ corrispondente ad un valore esadecimale di 148h e si mostra l’arresto dei contatori con possibilità di lettura dei risultati. Nella seconda si mostra un ingrandimento della finestra temporale al riavvio del dispositivo. Nella terza infine, si fa vedere che impostando il periodo di clock a 80 ns, e una soglia di 1 LSB, il dispositivo non rileva mai il superamento della soglia impostata per un intervallo di tempo sueriore a

$$2^{16} \cdot \text{N} \cdot \text{T}_{\text{ck}} = 65536 \cdot 32 \cdot 80\ \text{ns} \cong 168\ \text{ms},$$

dove N è il numero di cicli di clock impiegati per il test di un singolo valore in ingresso e 2^{16} è il numero di operandi sui quali è stato fatto il test.


Name:
Value:
200.0us
400.0us
600.0us
800.0us
1.0ms
1.2ms
1.4ms
1.6ms
1.8ms
clock
0
reset
1
continue
1
soglia[9..0]
H 148
148
operand[15..0]
H 0000
00CE
0215
reciprocal[15..0]
H 0000
FF32
FDF0
result[9..0]
H 000
14A
166
freeze
0
trigger
0
pulseInt
0
pulseExt
0
Riavvio del dispositivo


Name:
Value:
640.0us
644.0us
648.0us
652.0us
656.0us
660.0us
664.0us
668.0us
clock
0
reset
1
continue
1
soglia[9..0]
H 148
148
operand[15..0]
H 00CE
00CE
00CF
00D0
00D1
00D2
00D3
reciprocal[15..0]
H FF32
FF32
FF31
FF30
FF2F
FF2E
result[9..0]
H 14A
14A
000
0B2
0AF
0AB
0A8
freeze
1
trigger
0
pulseInt
0
pulseExt
0
L'impulso di sblocco provoca l'azzeramento del registro, e il conseguente fronte in discesa del segnale di freeze



Si mostra come dopo circa
168 ms il dispositivo ricominci
il ciclo di conteggio. Si ha inoltre,
per l'ingresso 0000h il massimo
modulo dell'errore: 1FFh pari a 1 LSB.

CONCLUSIONI

Concludiamo il lavoro descrivendo brevemente ciò che è stato fatto, e facendo anche alcune considerazioni su quali ne siano i possibili sviluppi.

Si è iniziato studiando alcuni dei vari metodi presenti in letteratura per l'implementazione dell'operazione aritmetica di divisione. Dopo questa fase introduttiva, è stato notato che l'approssimazione polinomiale di $\frac{1}{x}$ fornisce ottimi risultati riguardo al trade-off tra le risorse hardware impiegate e la latenza del dispositivo. In base a questo fatto si è deciso di approfondire le tecniche per l'approssimazione di una funzione generica mediante l'uso di un polinomio di secondo grado. Per questo intento ci siamo serviti di un algoritmo (di Remes), che, basandosi sulla teoria di Chebycev, permette di determinare il polinomio, di grado fissato, di migliore approssimazione uniforme su un assegnato intervallo. Data la generalità dell'algoritmo, si può apprezzare la sua estendibilità ad altri contesti, quali il calcolo delle funzioni $\sqrt{x}$, $\frac{1}{x^2}$, $\frac{1}{\sqrt{x}}$, ma anche di altre funzioni elementari in genere. Osservando poi che il nucleo del dispositivo sintetizzato calcola il valore del polinomio a prescindere da quale sia lo scopo per cui lo fa, possiamo pensare di poter riutilizzare le considerazione progettuali fatte, per i contesti alternativi citati; basta, infatti, conoscere i coefficienti e l'intervallo nel quale il valore del polinomio deve essere calcolato. In definitiva, in base alla generalità delle considerazioni fatte, sia nello studio dell'algoritmo, sia nella progettazione dell'hardware, si intuisce che dette considerazioni possono essere facilmente estese per la progettazione di un dispositivo che approssima una qualsiasi funzione elementare.

Mettiamo in evidenza, ora, quelli che sono gli aspetti nuovi di questo lavoro. La problematica del controllo dell'errore massimo che si commette quando si approssima una funzione mediante un polinomio i cui coefficienti siano l'arrotondamento di quelli del polinomio ideale di minimo scarto è complicata e non può essere altrimenti sviluppata se non introducendo delle maggiorazioni sulla stima dell'errore commesso, che portano, in genere, a dei risultati non ottimali.

È stato dato allora, in questo contesto, un ampio spazio alla ricerca di soluzioni per piccole modifiche alla soluzione ottenuta mediante un generico arrotondamento. In altre parole, osservando che i coefficienti arrotondati non sono più quelli di minimo scarto, e che l'errore introdotto dall'arrotondamento può essere quantitativamente più grande rispetto a quello prodotto dall'approssimazione della funzione mediante il polinomio di minimo scarto, abbiamo valutato l'errore in corrispondenza di molte terne di coefficienti, tutte in un intorno di quella del polinomio ideale.

Più specificamente, fissato il numero di bit per la rappresentazione dei coefficienti, abbiamo modificato quelli che si ottengono attraverso il conseguente arrotondamento, per aggiunte o sottrazioni di quantità multiple intere della unità di base e si è valutato l'errore per ogni possibile operando in ingresso. La minimizzazione del massimo modulo di questi errori, è stato il criterio secondo il quale abbiamo scelto una "terna di spostamento" piuttosto che un'altra.

L'ulteriore elemento nuovo di questo lavoro è la parametrizzazione del dispositivo di calcolo. Ciò permette di scrivere il software di simulazione del dispositivo stesso in forma parametrica, di valutare il massimo errore commesso fissati i parametri, e poi in maniera automatizzata, modificare i parametri così definiti nel tentativo di trovare una soluzione che riduca le risorse hardware impiegate. L'esito di questa ricerca per tentativi, con l'ausilio di un personal computer, ha dato dei risultati positivi in termini di risparmio sul numero di bit della look-up table, ma anche in termini di dimensioni dei blocchi moltiplicatori-sommatori. La riduzione delle risorse impiegate in un progetto di questo tipo è fondamentale, non solo per ragioni di carattere economico (riduzione dell'area di silicio impiegata), ma anche per ragioni di performance del dispositivo. È infatti una regola generale che tanto più un dispositivo è semplice e compatto, tanto più è economico ed efficiente. Un ulteriore vantaggio derivante da una parametrizzazione accurata è la possibilità di modificare le specifiche del progetto con estrema facilità e con poco sforzo.

Un possibile sviluppo di questo lavoro può essere quindi, sulla base dell'ultima osservazione fatta, la progettazione di un software come strumento vero e proprio, cioè estendibile alla ricerca di soluzioni per il calcolo di altre funzioni, ma anche estendibile dal punto di vista dell'architettura presa in esame; ossia parametrizzando architetture alternative e analizzando e confrontando i risultati ottenuti. Concludiamo osservando che

il limite di questo approccio risiede nel fatto che la mole di calcoli che vengono effettuati cresce enormemente al crescere del numero di bit dell'operando in ingresso e del numero di parametri che si definiscono; tuttavia, la potenza di calcolo dei personal computer è aumentata in maniera così considerevole nel corso degli anni, da far ritenere valido anche un approccio così dispendioso dal punto di vista computazionale.

APPENDICE

Il software di simulazione

Per lo sviluppo dell'algoritmo presentato nel corso di questo lavoro abbiamo fatto uso dell'ambiente Borland C++ 5.0. Abbiamo sviluppato alcuni oggetti seguendo l'ordine delle problematiche presentate nei capitoli 2 e 3.

Abbiamo deciso di sviluppare, anzitutto, una classe Polinomio per la determinazione del polinomio di migliore approssimazione uniforme. Questa classe ha al suo interno un oggetto di tipo Intervallo, un puntatore alla funzione da approssimare e un puntatore alla derivata della stessa funzione. Vengono quindi sviluppate alcune procedure e funzioni per l'implementazione dell'algoritmo di Remes.

Il secondo problema che abbiamo dovuto affrontare è lo sviluppo dei calcoli in aritmetica finita. In un primo momento ci siamo preoccupati soltanto dell'arrotondamento dei coefficienti, e abbiamo sviluppato il calcolo del valore del polinomio su un numero di bit più che sufficiente a ritenere trascurabili gli errori di troncamento dei risultati parziali e del risultato finale. Successivamente abbiamo deciso di implementare una classe apposita per la parametrizzazione del dispositivo, e l'abbiamo chiamata DispositivoCalcolatore. Quest'ultimo oggetto, inserito all'interno della classe Polinomio, viene inizializzato con i parametri descritti nel capitolo 3: $\mathrm{bitC}_0, \mathrm{bitC}_1, \mathrm{bitC}_2, \mathrm{bitDif}, \mathrm{bitMolt}_1, \mathrm{bitMolt}_2$. Una volta inizializzati i valori dei parametri del dispositivo, si è svolta la ricerca dei coefficienti che producessero il risultato migliore. In tal modo abbiamo valutato per quella configurazione di parametri il massimo errore commesso.

La terza problematica affrontata è quella di provare a sintetizzare il dispositivo con centinaia di configurazioni diverse, e valutare per ciascuna di queste costi e benefici. Abbiamo allora descritto due classi: Descrittore e Container. Descrittore è semplicemente una classe in cui sono memorizzati i parametri, e l'errore massimo. Container è un contenitore, specificamente un vettore, di oggetti di tipo Descrittore, ordinati secondo un criterio di costo del dispositivo associato a quella configurazione. In quest'ultimo

oggetto sono implementate delle funzioni di ausilio, che permettono di confrontare le configurazioni di parametri e il relativo errore, ed eliminare quelle per le quali un maggior costo non si traduce in un minore errore commesso.
Sono state scritte infine le funzioni di visibilità globale *ricercaOttimo*, s*ingleTry*, e *filtra*. *RicercaOttimo* legge da un file gli estremi entro i quali far variare i valori dei parametri, e prova tutte le possibili combinazioni che soddisfano tali vincoli. *SingleTry* invece determina, come *RicercaOttimo*, le terne di coefficienti, limitandosi però a una sola configurazione di parametri. Infine *filtra* legge da un file tutte le prove fatte sulle configurazioni di parametri e facendo uso delle classi Container e Descrittore memorizza in un file le configurazioni più significative.

Si riporta di seguito l'header file, e poi il relativo listato del file contenente il codice.

Simulatore.h

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

class Intervallo {
            public:
            Intervallo () { estSx = estDx = 0; }
            Intervallo (long double sx, long double dx) {
                        estSx = sx;
                        estDx = dx;
                        }
            Intervallo& operator = (const Intervallo& i);
            long double estSx,estDx;
            };

class DispositivoCalcolatore {
    public:
            DispositivoCalcolatore();
            void setCoeff(long double c0ing, long double c1ing, long double c2ing);
            void setBitsCoeff (int bitC0ing, int bitC1ing, int bitC2ing);
```

```
            void setBitsOper (int bitDifing, int bitMolt1ing, int bitMolt2ing);
            void setBitsPartiIntere (int bitPintC0ing, int bitPintC1ing, int bitPintC2ing);
            void setNBit (int nb);
            void inizializzaVariabili();
            void inizializzaCoefficienti();
            long double calcola (long double x);
  private:
            long double c0,c1,c2;
            long double c0int,c1int,c2int,difint;
            int bitDif,nBit,bitC0,bitC1,bitC2,bitMolt1,bitMolt2,scala1,scala2;
            int bitPintC0,bitPintC1,bitPintC2;
            long int arrotonda (long double x);
            long int rappresentaCoef(long double x, int bit);
            long int rappresentaDif(long double x, int bit, long double amp);
            long double tronca(long double x,int nb);
            long double riduci(long double x,int nb,bool aggLSB);
            long double molt1,molt1Tronc,opc1,opm1,prodp1,molt2,molt2Tronc,opc0,opm2,prodp2;
            long double result;
            int SHRC1,SHRM1,ridC1,ridM1,scalaP1,bitSomm1,MSBC1,MSBM1;
            int SHRC0,SHRM2,ridC0,ridM2,scalaP2,bitSomm2,MSBC0,MSBM2;
            int bitPintP1,bitPintP2;
            };

class Polinomio  {
  public:
            Polinomio () { pf = 0; }
            Polinomio (ofstream* ro, ofstream* so,long double (*puntf) (long double),long double (*puntdf) (long dou-
ble), Intervallo i);
            void init(ofstream* ro, ofstream* so,long double (*puntf) (long double),long double (*puntdf) (long double),
Intervallo i);
            static void setAmpC0 (long double amp) { ampc0 = amp; }
            static void setAmpC1 (long double amp) { ampc1 = amp; }
            static void setAmpC2 (long double amp) { ampc2 = amp; }
            void setNBit (int n);
            void setBitC0 (int bit);
            void setBitC1 (int bit);
            void setBitC2 (int bit);
            void setBitDif (int bit);
            void setBitMult1 (int bit);
```

```
        void setBitMult2 (int bit);
        void setBitPartiIntere (int p0, int p1, int p2);
        void setBitsDesc();
        char* getBitsDesc() { return bitsDesc; }
        void getBitPartiIntere(int& i0, int& i1, int& i2);
        long double getC0 () { return c0; }
        long double getC1 () { return c1; }
        long double getC2 () { return c2; }
        long double getMinMax () { return minmax; }
        void visualizzaPolinomio (ofstream* resOut);
        void visualizzaCoefficienti (ofstream* resOut);
        void visualizzaRisultati(ofstream* resOut);
        void visualizzaNodi (ofstream* resOut);
        void visualizzaSoglie (ofstream* resOut);
        void cicloRemes();
        void arrotondaCoefficienti();
        static long double soglie[];
        void calcolaMaxPerturbazione(long double delta);
        void setInd(int ix) { indice = ix; }
        int getInd() { return indice; }
        void setDispositivo();
        void setCoeffOtt(long int c0in, long int c1in, long int c2in);
        long double calcola (long double);
  private:
        void risolviSistema (long double []);
        long double trovaSeparazione (long double (Polinomio::*pfunz) (long double),long double a, long double b);
        long double trovaSeparazione (long double (Polinomio::*pfunz) (long double),long double a, long double b,
bool& fail);
        long double bisezione (long double (Polinomio::*pfunz) (long double), long double a, long double b);
        void nuoviNodi ();
        void calcolaRatio ();
        void trasformaCoefficienti ();
        long double massimoScarto(bool&);
        long double massimoScarto(long double&);
        long double q(long double);
        long double dist(long double);
        long double dq(long double);
        long double ddist(long double);
        long double distNorm(long double);
```

```
            long double ddistNorm(long double);
            long double dqNorm(long double);
            long double c0,c1,c2,L,x[4],ratio,erroreArr,minmax;
            long double maxPerturbazione;
            int nBit;
            long double c0ott,c1ott,c2ott;
            int terna[3],indice;
            int bitC0,bitC1,bitC2, bitDif, bitMult1, bitMult2,bitPintC0,bitPintC1,bitPintC2;
            char bitsDesc[100];
            static long double ampc0,ampc1,ampc2;
            long double ampdif,puntodimax;
            Intervallo intervallo;
            long double (*pf) (long double);
            long double (*pdf) (long double);
            ofstream *resOut,*sumOut;
            DispositivoCalcolatore disp;
            };

class Approssimatore {
  public:
            Approssimatore ();
            ~Approssimatore();
            Approssimatore (long double (*pfunz) (long double), long double (*pdfunz)(long double), Intervallo i);
            void setNBit (int n) {
                        nBit = n;
                        pot = 1;
                        for (int ix=0; ix<nBit; ix++) pot=pot*2;
                        numSottoIntervalli = pot;
                        }
            void setFileRiassunto (ofstream* f) { sumOut = f; }
            void setFileRisultati (ofstream* f) { resOut = f; }
            void setDescrizione (char* desc) { descrizione = desc; }
            void setBitC0C1C2 (int b0, int b1, int b2) { bitC0 = b0; bitC1 = b1; bitC2 = b2; }
            void setBitDifMult12 (int b0, int b1, int b2) { bitDif = b0; bitMult1 = b1; bitMult2 = b2; }
            void esegui();
            void setIndSottoIntervalli (int*,int);
            long double getMinmax () { return minmax; }
            void scriviTabellaBinario();
  private:
```

```
            long double (*pdf) (long double), (*pf) (long double);
            void creaSottoIntervalli();
            long double delta,minmax;
            int nBit;
            int bitC0,bitC1,bitC2,bitDif,bitMult1,bitMult2;
            ofstream *resOut,*sumOut;
            int* vet;
            int pot,numSottoIntervalli;
            Polinomio* pol;
            Intervallo intervallo;
            char* descrizione;
            void scrivi(char*,ofstream*,int);
            };

class Descrittore {
    friend ofstream& operator << (ofstream& out, const Descrittore& d);
    private:
            int vett[6];
            int sommabit;
            int Lut;
            double minmax;
    public:
            Descrittore();
            Descrittore(int bc0, int bc1, int bc2, int bdif, int bm1, int bm2);
            void init(int bc0, int bc1, int bc2, int bdif, int bm1, int bm2);
            void init(int* v);
            bool operator < (const Descrittore& d);
            Descrittore& operator = (const Descrittore &);
            bool match (const Descrittore&);
            bool equals (const Descrittore&);
            bool improves (const Descrittore&);
            void setMinmax (double m);
            };

class Container {
     private:
            int capacity,size,delta;
            Descrittore* box;
            void insert(Descrittore d);
```

```
            void replace(Descrittore d, int ind);
            void scambia (int ix, int ij);
            void removeMatchingItems (Descrittore d);
            bool isRedundant(int ind);
public:
            Container();
            ~Container();
            void add(Descrittore d);
            int getSize();
            Descrittore getAt(int ind);
            void filtra();
            };
```

Simulatore.cpp

```
#include "simulatore.h"

// UTILITIES

void decToBin (int n, char* p,int bit) {
  char *buf = new char[200];
  int cont = 0;
  while (n>0) {
    int ap = n%2;
    if (ap) buf[cont++]='1';
    else buf[cont++]='0';
    n = n/2;
    }
  for (int ix=0; ix<bit-cont; ix++) p[ix]='0';
  for (int ix=bit-cont; ix<bit; ix++) p[ix]=buf[bit-1-ix];
  }


int trovaBitParteIntera (long double x) {
  int cont=0;
     while (x>=1) {
     x=x/2;
```

```
      cont++;
      }
  return cont;
  }


long int tronca(long double x, int bit) {
  bool pos = x>0 ? true : false;
     long double y = fabsl(x);
               y = y*powl(2,-bit);
       y = floorl(y);
  if (pos) return y;
  return -y;
  }


    /* INTERVALLO */


Intervallo& Intervallo::operator = (const Intervallo& i) {
     estSx = i.estSx;
  estDx = i.estDx;
  return *this;
  }


     /* DISPOSITIVOCALCOLATORE */

DispositivoCalcolatore::DispositivoCalcolatore () {
  c0 = c1 = c2 = 0;
  bitDif = nBit = bitC0 = bitC1 = bitC2 = bitMolt1 = bitMolt2 = 0;
  scala1 = scala2 = bitPintC0 = bitPintC1 = bitPintC2 = 0;
  c0int = c1int = c2int = difint = 0;
  }

void DispositivoCalcolatore::setCoeff(long double c0ing, long double c1ing, long double c2ing) {
  c0 = c0ing;
  c1 = c1ing;
  c2 = c2ing;
```

```
  if (bitC0!=0 || bitC1!=0 || bitC2!=0) inizializzaCoefficienti();
  }

void DispositivoCalcolatore::setBitsCoeff (int bitC0ing, int bitC1ing, int bitC2ing) {
  bitC0 = bitC0ing;
  bitC1 = bitC1ing;
  bitC2 = bitC2ing;
  }

void DispositivoCalcolatore::setBitsOper (int bitDifing, int bitMolt1ing, int bitMolt2ing) {
  bitDif = bitDifing;
  bitMolt1 = bitMolt1ing;
  bitMolt2 = bitMolt2ing;
  }

void DispositivoCalcolatore::setBitsPartiIntere (int bitPintC0ing, int bitPintC1ing, int bitPintC2ing) {
  bitPintC0 = bitPintC0ing;
  bitPintC1 = bitPintC1ing;
  bitPintC2 = bitPintC2ing;
  }

void DispositivoCalcolatore::setNBit (int nb) {
    nBit = nb;
    }

void DispositivoCalcolatore::inizializzaCoefficienti() {
  c0int = rappresentaCoef(c0,bitC0);
  c1int = rappresentaCoef(c1,bitC1);
  c2int = rappresentaCoef(c2,bitC2);
  opc1 = riduci (c1int,ridC1,true); // Nota invece che calcolarlo tutte le volte,
                                    // lo calcolo quando viene modificato il coefficiente.
                                    // Cioè sarebbe parte della funzione calcola.
  opc0 = riduci (c0int,ridC0,true);
  }

void DispositivoCalcolatore::inizializzaVariabili() {
    inizializzaCoefficienti ();

            // Si calcola a priori l'allineamento degli
```

```
                // operandi sul primo prodotto parziale

bitSomm1 = bitMolt1+1; //bit del primo sommatore/sottrattore
scala1 = bitMolt1+1-bitPintC2; //peso del bit meno significativo del prodotto
MSBC1 = bitPintC1;
MSBM1 =  bitMolt1+1-scala1;
if (MSBC1>MSBM1) {
  SHRC1 = 0;
  SHRM1 = MSBC1-MSBM1;
  }
else {
  SHRC1 = MSBM1-MSBC1;
  SHRM1 = 0;
   }
ridC1 = SHRC1+bitC1+bitPintC1-bitSomm1; //se è negativo si può aggiungere 1 LSB
                                          // quando si effettua l'operazione.
ridM1 = SHRM1+bitMolt1+1-bitSomm1;
scalaP1 = bitMolt1-SHRC1;
bitPintP1 = bitPintC1+SHRC1;


     // Si calcola a priori l'allineamento degli
     // operandi sul secondo prodotto parziale

bitSomm2 = bitMolt2+1;
scala2 = bitMolt2+1-bitPintP1;
MSBC0 = bitPintC0;
MSBM2 = bitMolt2+1-scala2;
if (MSBC0>MSBM2) {
  SHRC0 = 0;
  SHRM2 = MSBC0-MSBM2;
   }
else {
   SHRC0 = MSBM2-MSBC0;
  SHRM2 = 0;
   }
ridC0 = SHRC0+bitC0+bitPintC0-bitSomm2;
ridM2 = SHRM2+bitMolt2+1-bitSomm2;
scalaP2 = bitMolt2-SHRC0;
```

```
  bitPintP2 = bitPintC0 + SHRC0;
  }

long int DispositivoCalcolatore::arrotonda (long double x) {
  bool pos = x>0 ? true : false;
  long double z = fabsl(x);
  long double sup = ceill(z);
  long double inf = floor(z);
  long double dif1 = sup-z;
  long double dif2 = z-inf;
  long int ris;
  if (dif1>dif2) ris =  (long int) inf;
  else ris = (long int) sup;
  if (pos) return ris;
  else return -ris;
  }

long int DispositivoCalcolatore::rappresentaDif (long double x,  int bit, long double amp) {
  long double z = x/amp;
  int pi = trovaBitParteIntera(z);
  long double y = z*powl(2,bit-pi);
  return arrotonda(y);
  }


long int DispositivoCalcolatore::rappresentaCoef (long double x,  int bit) {
  long double y = x*powl(2,bit);
  return arrotonda(y);
  }

long double DispositivoCalcolatore::tronca (long double x, int nb) {
  bool pos = x>0 ? true : false;
  long double y = fabsl(x);
  y = y*powl(2,-nb);
  y = floorl(y);
  if (nb<0) cout << "Warning: tronca(x,nbit<0)\n";
  if (pos) return y;
  return -y;
  }
```

```
long double DispositivoCalcolatore::riduci (long double x, int nb, bool aggLSB) {
  bool pos = x>0 ? true : false;
  long double y = fabsl(x);
  y = y*powl(2,-nb);
  y = floorl(y);
  if (nb<0 && aggLSB) y=y+1;
  if (pos) return y;
  return -y;
  }


long double DispositivoCalcolatore::calcola(long double x) {
  difint = rappresentaDif (x,bitDif,powl(2,-nBit));
  molt1 = difint*c2int;
  molt1Tronc = tronca(molt1,bitPintC2+bitC2+bitDif+nBit-bitMolt1-1);
  opm1 = riduci (molt1Tronc,ridM1,false);
  prodp1 = opc1+opm1;
  prodp1 = tronca(prodp1,1);

  molt2 = difint*prodp1;
  molt2Tronc = tronca (molt2,bitPintP1+bitMolt1+bitDif+nBit-bitMolt2-1);
  opm2 = riduci (molt2Tronc,ridM2,false);
  prodp2 = opc0+opm2;
  prodp2 = tronca(prodp2,1);
  result = prodp2*powl(2,-scalaP2);
  return result;
  }

    /* POLINOMIO                    */

long double Polinomio::soglie[] = { 0.24L, 0.166L };
long double Polinomio::ampc0 = 1;
long double Polinomio::ampc1 = 1;
long double Polinomio::ampc2 = 1;

void Polinomio::setBitC0 (int bit) { bitC0 = bit; }
void Polinomio::setBitC1 (int bit) { bitC1 = bit; }
```

```
void Polinomio::setBitC2 (int bit) { bitC2 = bit; }
void Polinomio::setBitDif (int bit) { bitDif = bit; }
void Polinomio::setBitMult1 (int bit) { bitMult1 = bit; }
void Polinomio::setBitMult2 (int bit) { bitMult2 = bit; }
void Polinomio::setNBit (int n) { nBit = n; }
void Polinomio::setBitPartiIntere (int p0, int p1, int p2) {
  bitPintC0 = p0;
  bitPintC1 = p1;
  bitPintC2 = p2;
  }

void Polinomio::getBitPartiIntere (int& i0, int& i1, int& i2) {
     i0 = bitPintC0;
     i1 = bitPintC1;
     i2 = bitPintC2;
     }

Polinomio::Polinomio (ofstream* ro, ofstream* so, long double (*puntf) (long double), long double (*puntdf) (long
double),Intervallo i) {
     init (ro,so,puntf,puntdf,i);
     }

void Polinomio::init (ofstream* ro, ofstream* so, long double (*puntf) (long double), long double (*puntdf) (long
double),Intervallo i) {
  pf = puntf;
  pdf = puntdf;
  intervallo = i;
  ampdif = i.estDx-i.estSx;
  sumOut = so;
  resOut = ro;
  x[0] = intervallo.estSx;
  x[3] = intervallo.estDx;
  x[1] = (3*x[0]+x[3])/4;
  x[2] = (x[0]+3*x[3])/4;
  }

void Polinomio::setBitsDesc() {
  char tmp[50];
  itoa (bitC0,bitsDesc,10);
```

```
  strcat (bitsDesc,"-");
  itoa (bitC1,tmp,10);
  strcat (bitsDesc,tmp);
  strcat (bitsDesc,"-");
  itoa (bitC2,tmp,10);
  strcat (bitsDesc,tmp);
  strcat (bitsDesc,"-");
  itoa (bitDif,tmp,10);
  strcat (bitsDesc,tmp);
  strcat (bitsDesc,"-");
  itoa (bitMult1,tmp,10);
  strcat (bitsDesc,tmp);
  strcat (bitsDesc,"-");
  itoa (bitMult2,tmp,10);
  strcat (bitsDesc,tmp);
  }

void Polinomio::visualizzaPolinomio (ofstream* resOut)  {
      *resOut <<"Intervallo: [" << intervallo.estSx << "," << intervallo.estDx << "]" << endl;
     }

void Polinomio::visualizzaCoefficienti (ofstream* resOut) {
    *resOut << "c0: " << c0 << "\t";
    *resOut << "c1: " << c1 << "\t";
    *resOut << "c2: " << c2 << "\t";
    *resOut << "L: " << L << "\n";
     }

void Polinomio::visualizzaRisultati (ofstream* resOut) {
    *resOut << x[0] << "\t" << x[3] << "\t";
    *resOut << c0*powl(2,bitC0) << "\t";
    *resOut << c1*powl(2,bitC1) << "\t";
    *resOut << c2*powl(2,bitC2) << "\t";
    *resOut << L << "\t";
    *resOut << erroreArr << "\t";
    *resOut << (erroreArr-L) << "\t";
    *resOut << minmax << "\t";
    *resOut << terna[0] << "\t";
    *resOut << terna[1] << "\t";
```

```
      *resOut << terna[2] << "\t";
      *resOut << (1-(minmax/erroreArr)) << "\n";
       }


void Polinomio::visualizzaNodi(ofstream* resOut) {
   *resOut << "Nodi: \n\n";
   for (int ix=0; ix<4; ix++) {
      *resOut << "x[" << ix << "]: " << x[ix] << "\n";
       }
   *resOut << "Ratio: " << ratio << endl;
   *resOut << "\n\n";
   }

void Polinomio::visualizzaSoglie (ofstream* resOut) {
   *resOut << "Soglie\tK=4\t" << soglie[0]*maxPerturbazione << "\tK=2\t" << soglie[1]*maxPerturbazione << endl;
   }

void Polinomio::risolviSistema(long double n[4]) {
   long double det;
   long double nq[4];
   for (int ix=0; ix<4; ix++) {
       nq[ix]=n[ix]*n[ix];
       }
   det = (n[0]-n[2])*(n[1]-n[3])*(n[0]-n[1]+n[2]-n[3]);

   c0 = nq[0]*( (-pf(n[2])-pf(n[3]))*n[1] + (pf(n[1])-pf(n[3]))*n[2] + (pf(n[1])+pf(n[2]))*n[3] );
   c0 = c0 + nq[1]* ( (+pf(n[2])+pf(n[3]))*n[0] + (-pf(n[0])-pf(n[3]))*n[2] + (-pf(n[0])+pf(n[2]))*n[3] );
   c0 = c0 + nq[2]* ( (-pf(n[1])+pf(n[3]))*n[0] + (+pf(n[0])+pf(n[3]))*n[1] + (-pf(n[0])-pf(n[1]))*n[3] );
   c0 = c0 + nq[3]* ( (-pf(n[1])-pf(n[2]))*n[0] + (+pf(n[0])-pf(n[2]))*n[1] + (+pf(n[0])+pf(n[1]))*n[2] );
   c0 = -c0/(2*det);

   c1 = nq[0]*(pf(n[1])-pf(n[3]));
   c1 = c1 + nq[1]*(-pf(n[0])+pf(n[2]));
   c1 = c1 + nq[2]*(-pf(n[1])+pf(n[3]));
   c1 = c1 + nq[3]*(+pf(n[0])-pf(n[2]));
   c1 = c1/det;

   c2 = n[0]*(pf(n[1])-pf(n[3]));
```

```
  c2 = c2 + n[1]*(-pf(n[0])+pf(n[2]));
  c2 = c2 + n[2]*(-pf(n[1])+pf(n[3]));
  c2 = c2 + n[3]*(+pf(n[0])-pf(n[2]));
  c2 = -c2/det;

  L =     nq[0]*(                          (+pf(n[2])-pf(n[3]))*n[1] + (-pf(n[1])+pf(n[3]))*n[2] + (+pf(n[1])-pf(n[2]))*n[3] );
  L = L + nq[1]*( (-pf(n[2])+pf(n[3]))*n[0] +                          (+pf(n[0])-pf(n[3]))*n[2] + (-pf(n[0])+pf(n[2]))*n[3] );
  L = L + nq[2]*( (+pf(n[1])-pf(n[3]))*n[0] + (-pf(n[0])+pf(n[3]))*n[1] +                          (+pf(n[0])-pf(n[1]))*n[3] );
  L = L + nq[3]*( (-pf(n[1])+pf(n[2]))*n[0] + (+pf(n[0])-pf(n[2]))*n[1] + (-pf(n[0])+pf(n[1]))*n[2] );
  L = L/(2*det);
  }

void Polinomio::cicloRemes () {
  visualizzaPolinomio(resOut);
  ratio = 2;
  risolviSistema(x);
  visualizzaCoefficienti (resOut);
  visualizzaNodi(resOut);

  while ( ratio > 1.000005L ) {
        nuoviNodi();
        visualizzaNodi(resOut);
        risolviSistema(x);
        visualizzaCoefficienti (resOut);
        }
  trasformaCoefficienti();
  ofstream outTab ("tabellaCRemes.txt",ios::app);
  outTab.precision(16);
  outTab << indice << "\t" << c0 << "\t" << c1 << "\t" << c2 << "\t" << L << "\n";
  outTab.close();
  *resOut << "\nCoefficienti: \n";
  visualizzaCoefficienti(resOut);
  *resOut << "Errore Max: " << L << endl;
  *resOut << "\n\n\n\n";
  }


long double Polinomio::q(long double x) { return (c2*x+c1)*x+c0; }
long double Polinomio::dq(long double x) {   return c1+2*c2*x;   }
```

```
long double Polinomio::dist(long double x) { return pf(x)-q(x); }
long double Polinomio::ddist (long double x) { return pdf(x)-dq(x); }

long double Polinomio::distNorm(long double xs) {
    long double dif = xs-x[0];
    return pf(xs)-((c2*dif+c1)*dif+c0);
    }

long double Polinomio::dqNorm (long double v) { return c1+ 2*c2*(v-x[0]); }
long double Polinomio::ddistNorm (long double v) { return pdf(v)-dqNorm(v); }



typedef long double (Polinomio::*PUNTAFUNZIONE) (long double);
long double Polinomio::trovaSeparazione (PUNTAFUNZIONE pfunz, long double a, long double b) {
  long double passo = (b-a)/2;
  int ix=1;
  bool trovato = false;
  long double segno = (this->*pfunz)(a);
  long double v;
  int ij;
  while (!trovato) {
     long double numero = powl(2,ix-1);
     for (ij=0; ij<numero; ij++) {
              v = (this->*pfunz)(a+passo*(2*ij+1));
              if (segno*v<0) return a+passo*(2*ij+1);
              }
    passo = passo/2;
    ix++;
    }
  return a;
  }

long double Polinomio::trovaSeparazione (PUNTAFUNZIONE pfunz, long double a, long double b, bool& fail) {
  long double orig = b-a;
  long double passo = (b-a)/2;
  int ix=1;
  bool trovato = false;
  long double segno = (this->*pfunz)(a);
```

```
  long double v;
  int ij;
  fail = false;
  while (!trovato && passo>(1e-3)*orig) {
                long double numero = powl(2,ix-1);
                for (ij=0; ij<numero; ij++) {
                v = (this->*pfunz)(a+passo*(2*ij+1));
                if (segno*v<0) return a+passo*(2*ij+1);
                }
    passo = passo/2;
    ix++;
    }
  if (!trovato) fail = true;
  return a;
  }

long double Polinomio::bisezione (PUNTAFUNZIONE pfunz,long double a, long double b) {
  long double risol = 1e-17;
  long double value;
  while ( b-a > risol ) {
     value = (this->*pfunz)((b+a)/2);
    if (value==0) return (b+a)/2;
       if ( (this->*pfunz)(a)*value<0 ) b = (b+a)/2;
     else a = (b+a)/2;
     }
  return a;
  }

void Polinomio::nuoviNodi () {
     long double c,s1,s2;
     c = ddist(x[0])*ddist(x[1]);
     if (c<0)  {
                s1 = bisezione(&Polinomio::ddist,x[0],x[1]);
                s2 = bisezione(&Polinomio::ddist,x[1],x[3]);
                }
     else {
                long double nodo = trovaSeparazione (&Polinomio::ddist,x[1],x[2]);
                s1 = bisezione (&Polinomio::ddist,x[1],nodo);
                s2 = bisezione (&Polinomio::ddist,nodo,x[3]);
```

```
            }
  x[1] = s1;
  x[2] = s2;
  calcolaRatio();
  }

void Polinomio::calcolaRatio() {
  long double max,min;
  max = fabsl(dist(x[0]));
  min = fabsl(dist(x[0]));
  for (int ix=1; ix<4; ix++) {
      long double v = fabsl(dist(x[ix]));
      if (v<min) min = v;
      else if (v>max) max = v;
       }
  ratio = max/min;
  }

void Polinomio::trasformaCoefficienti() {
  long double b1,b0;
  b1 = c1+2*c2*x[0];
  b0 = (c2*x[0]+c1)*x[0]+c0;
  c1 = b1;
  c0 = b0;
  }

long double round(long double x) {
  long double c = fabsl(x);
  long double remainder = c-floorl(c);
  long double ris;
  if (remainder>0.5) ris = c+1-remainder;
  else ris = c-remainder;
  if (x>0) return ris;
  else return -ris;
  }

long double Polinomio::massimoScarto(bool& fail) {
      long double s1,s2;
      long double nodo = trovaSeparazione (&Polinomio::ddistNorm,x[0],x[3],fail);
```

```
     if (!fail) {
         s1 = bisezione (&Polinomio::ddistNorm,x[0],nodo);
         s2 = bisezione (&Polinomio::ddistNorm,nodo,x[3]);
         }
     long double v1,v;
     v = fabsl(distNorm(x[0]));
     if (!fail) {
        v1 = fabsl(distNorm(s1));
        if (v1>v) v=v1;
        v1 = fabsl(distNorm(s2));
        if (v1>v) v=v1;
        }
     v1 = fabsl(distNorm(x[3]));
     if (v1>v) v=v1;
     return v;
     }

long double Polinomio::massimoScarto(long double& p) {
        long double max = fabsl(disp.calcola(0)-pf(x[0]));
        int multiplo = 1;
        long double delta = (x[3]-x[0])*multiplo*powl(2,-bitDif);
        long double xs = 0;
        long double end = x[3]-x[0];
        long double v;
        p = x[0];
        while (xs<end) {
                v = fabsl(disp.calcola(xs)-pf(x[0]+xs));
                 if (v>max) {
                        max = v;
                        p = xs+x[0];
                        }
                xs += delta;
                }
        return max;
        }

void Polinomio::calcolaMaxPerturbazione (long double delta) {
  maxPerturbazione = powl (2,-(bitC0+1)) + powl (2,-(bitC1+1))*delta + powl (2,-(bitC2+1))*powl(delta,2);
  }
```

```
void Polinomio::arrotondaCoefficienti () {
  c0 = round(c0*powl(2,bitC0))*powl(2,-bitC0);
  c1 = round(c1*powl(2,bitC1))*powl(2,-bitC1);
  c2 = round(c2*powl(2,bitC2))*powl(2,-bitC2);
  setDispositivo();
  erroreArr = fabsl(massimoScarto(puntodimax));
  L = fabsl(L);
  long double perturb = erroreArr-L;
  int k;
  calcolaMaxPerturbazione(powl(2,-nBit));
  if (perturb > soglie[0]*maxPerturbazione) k=4;
  else if (perturb > soglie[1]*maxPerturbazione) k=2;
     else k=1;
  long double w0,w1,w2,v;
  w0 = c0;
  w1 = c1;
  w2 = c2;
  c0ott = c0;
  c1ott = c1;
  c2ott = c2;
  terna[0] = terna[1] = terna[2] = 0;
  minmax = erroreArr;
  *resOut << "\n";
  for (int i0=-k; i0<=k; i0++) {
    cout << getBitsDesc() << ":\t" << getInd() << ".\t "<< i0 << "\n" << endl;
       for (int i1=-k; i1<=k; i1++)
              for (int i2=-k; i2<=k; i2++) {
                     c0 = w0 + i0*powl (2,-bitC0);
                     c1 = w1 + i1*powl (2,-bitC1);
                     c2 = w2 + i2*powl (2,-bitC2);
                     disp.setCoeff(c0,c1,c2);
                     v = massimoScarto(puntodimax);
                     if (v>4e-5) cout << getBitsDesc() << ":\t" << getInd() << ".\t " << v << "\t\t" << i0 << " " <<
i1 << " " << i2 << "\n";
                     if (v<minmax) {
                            minmax = v;
                            c0ott = c0;
                            c1ott = c1;
```

```
                                c2ott = c2;
                                terna[0] = i0;
                                terna[1] = i1;
                                terna[2] = i2;
                                }
                        *resOut << v << " \tterna ideale: {" << terna[0] << "," << terna[1] << "," << terna[2] << "}"
<< endl;
                        }
                cout << "\n\n";
                }
    c0 = c0ott;
    c1 = c1ott;
    c2 = c2ott;
    massimoScarto(puntodimax);
    }

  void Polinomio::setDispositivo () {
    disp.setCoeff(c0,c1,c2);
    disp.setBitsCoeff(bitC0,bitC1,bitC2);
    disp.setBitsOper(bitDif,bitMult1,bitMult2);
    disp.setBitsPartiIntere(bitPintC0,bitPintC1,bitPintC2);
    disp.setNBit(nBit);
    disp.inizializzaVariabili();
    disp.inizializzaCoefficienti();
    }

  void Polinomio::setCoeffOtt(long int c0in, long int c1in, long int c2in) {
    c0 = c0in*powl(2,-bitC0);
    c1 = c1in*powl(2,-bitC1);
    c2 = c2in*powl(2,-bitC2);
    }

  long double Polinomio::calcola (long double x) { return disp.calcola(x); }

                /* APPROSSIMATORE                         */

  Approssimatore::Approssimatore () {
     pf = pdf = 0;
     descrizione = 0;
```

```
        vet = 0;
        }

    Approssimatore::Approssimatore ( long double (*pfunz) (long double), long double (*pdfunz) (long double), In-
tervallo i) {
      pf = pfunz;
      pdf = pdfunz;
      intervallo = i;
      descrizione = 0;
      vet = 0;
      }

    Approssimatore::~Approssimatore() {
      delete [] pol;
      delete [] vet;
      }

    void Approssimatore::setIndSottoIntervalli(int* pv, int num) {
      vet = new int[num+2];
      for (int ix=0; ix<num; ix++) vet[ix+1]=pv[ix];
      vet[0] = 0;
      vet[num+1] = pot;
      numSottoIntervalli = num+1;
      }

    void Approssimatore::creaSottoIntervalli() {
        if (vet==0) {
                pol = new Polinomio[pot];
                delta = (intervallo.estDx-intervallo.estSx)/pot;
                for (int ix=0; ix<pot; ix++) {
                        Intervallo i(intervallo.estSx+delta*ix,intervallo.estSx+delta*(ix+1));
                        pol[ix].init(resOut,sumOut,pf,pdf,i);
                        pol[ix].setNBit(nBit);
                        pol[ix].setBitC0(bitC0);
                        pol[ix].setBitC1(bitC1);
                        pol[ix].setBitC2(bitC2);
                        pol[ix].cicloRemes();
                        cout << (ix+1) << "." << endl;
                        }
```

```
                    }
        else {
          pol = new Polinomio[numSottoIntervalli];
          delta = (intervallo.estDx-intervallo.estSx)/pot;
          for (int ix=0; ix<numSottoIntervalli; ix++) {
                    Intervallo i(intervallo.estSx+delta*vet[ix],intervallo.estSx+delta*vet[ix+1]);
                    pol[ix].init(resOut,sumOut,pf,pdf,i);
                    pol[ix].cicloRemes();
                    cout << (ix+1) << "." << endl;
                    }
          }
   }

void Approssimatore::esegui () {
   *sumOut << descrizione << "\n";
   cout << "Numero sottointervalli: " << pot << "\n\n";
   *sumOut << "Numero sottointervalli: " << pot << "\n";
   creaSottoIntervalli();
   long double v0,v1,v2,tmp0,tmp1,tmp2;
   v0 = fabsl(pol[0].getC0());
   v1 = fabsl(pol[0].getC1());
   v2 = fabsl(pol[0].getC2());
   for (int ix=1; ix<numSottoIntervalli; ix++) {
                    tmp0 = fabsl(pol[ix].getC0());
                    tmp1 = fabsl(pol[ix].getC1());
                    tmp2 = fabsl(pol[ix].getC2());
                    if (tmp0>v0) v0 = tmp0;
                    if (tmp1>v1) v1 = tmp1;
                    if (tmp2>v2) v2 = tmp2;
                    }
   int corz0 = trovaBitParteIntera(v0);
   int corz1 = trovaBitParteIntera(v1);
   int corz2 = trovaBitParteIntera(v2);
   for (int ix=0; ix<numSottoIntervalli; ix++) {
          pol[ix].calcolaMaxPerturbazione(powl(2,-nBit)*(intervallo.estDx-intervallo.estSx));
          pol[ix].setBitPartiIntere(corz0,corz1,corz2);
          }
  cout << "Arrotondamento coefficienti\n\n";
  for (int ix=0; ix<numSottoIntervalli; ix++) {
```

```
        pol[ix].setNBit (nBit);
        pol[ix].setBitC0(bitC0);
        pol[ix].setBitC1(bitC1);
        pol[ix].setBitC2(bitC2);
        pol[ix].setBitDif(bitDif);
        pol[ix].setBitMult1(bitMult1);
        pol[ix].setBitMult2(bitMult2);
        pol[ix].setInd(ix);
        pol[ix].setBitsDesc();
        pol[ix].arrotondaCoefficienti();
        long double v = pol[ix].getMinMax();
       if (ix==0) minmax = v;
      else if (minmax<v) minmax = v;
       }
v0 = fabsl(pol[0].getC0());
v1 = fabsl(pol[0].getC1());
v2 = fabsl(pol[0].getC2());
for (int ix=1; ix<numSottoIntervalli; ix++) {
    tmp0 = fabsl(pol[ix].getC0());
    tmp1 = fabsl(pol[ix].getC1());
    tmp2 = fabsl(pol[ix].getC2());
    if (tmp0>v0) v0 = tmp0;
    if (tmp1>v1) v1 = tmp1;
    if (tmp2>v2) v2 = tmp2;
    }
corz0 = trovaBitParteIntera(v0);
corz1 = trovaBitParteIntera(v1);
corz2 = trovaBitParteIntera(v2);
*sumOut << "bitC0: " << bitC0 << "\tbitParteIntera\t" << corz0 << "\n";
*sumOut << "bitC1: " << bitC1 << "\tbitParteIntera\t" << corz1 << "\n";
*sumOut << "bitC2: " << bitC2 << "\tbitParteIntera\t" << corz2 << "\n";
pol[0].visualizzaSoglie (sumOut);
*sumOut << "Ind.\tx[0]\tx[3]\tc0\tc1\tc2\tErr. intr.\tErr. Arr.\tErr. Agg.\tErr. fin.\t\tTerna\t\tGuadagno\n";
for (int ix=0; ix<numSottoIntervalli; ix++) {
                pol[ix].setBitPartiIntere(corz0,corz1,corz2);
                *sumOut << (ix) << "\t";
                pol[ix].visualizzaRisultati(resOut);
                pol[ix].visualizzaRisultati(sumOut);
                }
```

```
  *sumOut << "\nErrore Massimo su tutti gli intervalli: " << minmax << endl;
  }

void Approssimatore::scrivi(char* p, ofstream* pout, int nc) {
  int ix=0;
  while (ix<nc) *pout << p[ix++] << "\t";
  }

void Approssimatore::scriviTabellaBinario() {
     // Scrive su un file la tabella dei coefficienti
     // in formato binario
  ofstream out("TabellaCI.txt");
  char *pc0,*pc1,*pc2,*pix;
  int bpintC0,bpintC1,bpintC2;
  pol[0].getBitPartiIntere(bpintC0,bpintC1,bpintC2);
  pix = new char[nBit];
  pc2 = new char[bitC2+bpintC2];
  pc1 = new char[bitC1+bpintC1];
  pc0 = new char[bitC0+bpintC0];
  out << "Ind.\taddress\tC2\tC1\tC0\n";
  for (int ix=0; ix<numSottoIntervalli; ix++) {
                decToBin (ix,pix,nBit);
                decToBin (abs(pol[ix].getC2()*powl(2,bitC2)),pc2,bitC2+bpintC2);
                decToBin (abs(pol[ix].getC1()*powl(2,bitC1)),pc1,bitC1+bpintC1);
                decToBin (abs(pol[ix].getC0()*powl(2,bitC0)),pc0,bitC0+bpintC0);
                out << ix << "\t";
                scrivi(pix,&out,nBit);
                out << "\t";
                scrivi(pc2,&out,bitC2+bpintC2);
                out << "\t";
                scrivi(pc1,&out,bitC1+bpintC1);
                out << "\t";
                scrivi(pc0,&out,bitC0+bpintC0);
                out << "\n";
                }
  delete [] pix;
  delete [] pc2;
  delete [] pc1;
  delete [] pc0;
```

```
  out.close();
   }

// CLASSI PER LA GESTIONE E IL FILTRAGGIO DEI RISULTATI

Descrittore::Descrittore () {}

Descrittore::Descrittore (int bc0, int bc1, int bc2, int bdif, int bm1, int bm2) { init(bc0,bc1,bc2,bdif,bm1,bm2); }

Descrittore& Descrittore::operator = (const Descrittore& d) {
  for (int ix=0; ix<6; ix++) vett[ix] = d.vett[ix];
  sommabit = d.sommabit;
  Lut = d.Lut;
  minmax = d.minmax;
  return *this;
  }

void Descrittore::init (int bc0,int bc1, int bc2, int bdif, int bm1, int bm2) {
      vett[0] = bc0;
      vett[1] = bc1;
      vett[2] = bc2;
      vett[3] = bdif;
      vett[4] = bm1;
      vett[5] = bm2;
      sommabit = 0;
      Lut = 0;
      for (int ix=0; ix<6; ix++) {
            if (ix<3) Lut += vett[ix];
            sommabit += vett[ix];
        }
     }

void Descrittore::init (int* v) {
     sommabit = 0;
     Lut = 0;
     for (int ix=0; ix<6; ix++) {
         vett[ix] = v[ix];
         sommabit += vett[ix];
         if (ix<3) Lut += vett[ix];
```

```
          }
        }

bool Descrittore::operator < (const Descrittore& d1) {
         if (Lut < d1.Lut) return true;
        else if (Lut == d1.Lut) if (sommabit < d1.sommabit) return true;
                              else if (sommabit == d1.sommabit) return minmax < d1.minmax;
         return false;
         }

bool Descrittore::match (const Descrittore& d) { return d.Lut == Lut && sommabit == d.sommabit; }

bool Descrittore::equals (const Descrittore& d) { return fabsl((d.minmax-minmax)/minmax)<0.005; }

bool Descrittore::improves (const Descrittore& d) { return minmax<d.minmax; }

void Descrittore::setMinmax (double m) { minmax = m; }

ofstream& operator << (ofstream& out, const Descrittore& d) {
  for (int ix=0; ix<6; ix++) out << d.vett[ix] << "\t";
  out << d.minmax << "\t" << d.Lut << "\t" << d.sommabit << "\n";
  return out;
  }



void Container::insert(Descrittore d) {
      if (size==capacity-1) {
                  capacity += delta;
                  Descrittore* tmp = new Descrittore[capacity];
                  for (int ix=0; ix<size; ix++) tmp[ix]=box[ix];
                  delete [] box;
                  box = tmp;
                  }
      if (!size || (d < box[size-1])) {
                  box[size++]=d;
                   return;
                  }
       int inf = 0;
```

```
        int sup = size-1;
        int mid = (inf+sup)/2;
        while (sup-inf > 1){
                if (d < box[mid]) inf = mid;
                else sup = mid;
                mid = (inf+sup)/2;
                }
        int ind;
        if (box[inf] < d) ind = inf;
        else ind = sup;
        for (int ix=size; ix>ind; ix--) box[ix]=box[ix-1];
        box[ind]=d;
        size++;
        }

void Container::replace (Descrittore d, int ind) {
        if (ind<size) box[ind]=d;
        }

void Container::scambia (int ix, int ij) {
        Descrittore tmp;
        tmp = box[ix];
        box[ix] = box[ij];
        box[ij] = tmp;
        }

void Container::removeMatchingItems(Descrittore d) {
        bool* vbool = new bool[size];
        int cont = 0;
        for (int ix=0; ix<size; ix++) {
                if (d.match(box[ix])) {
                        cont++;
                        vbool[ix]=true;
                        }
                else vbool[ix]=false;
                }
        int ij=0;
        for (int ix=0; ix<size; ix++) {
                if (!vbool[ix]) {
```

```
                box[ij++]=box[ix];
                }
        }
    size = size-cont;
    }

bool Container::isRedundant(int ind) {
     for (int ix=ind+1; ix<size; ix++) {
                bool a = !box[ind].improves(box[ix]);
                bool b = box[ix]<box[ind];
                if (a && b) return true;
                }
     return false;
     }

Container::Container () {
    delta = 10;
    size = 0;
    capacity = 10;
    box = new Descrittore[capacity];
    }

Container::~Container () { delete[] box; }

void Container::add (Descrittore d) {
    int ix=0;
    bool trovato = false;
    while (ix<size && !trovato) {
        if (d.match(box[ix])) {
                if (d.equals(box[ix])) insert(d);
                else if (d < box[ix]) {
                        removeMatchingItems(d);
                        insert(d);
                        }
                return;
                }
        ix++;
        }
   insert(d);
```

```
    }

int Container::getSize() { return size; }

Descrittore Container::getAt(int ind) { return box[ind]; }

void Container::filtra() {
     bool *vbool = new bool[size];
     for (int ix=0; ix<size; ix++) if (isRedundant(ix)) vbool[ix]=true;
                                  else vbool[ix]=false;
     int ij=0;
     int cont = 0;
     for (int ix=0; ix<size; ix++) {
            if (!vbool[ix]) box[ij++]=box[ix];
            else cont++;
            }
     size -= cont;
     }

// FUNZIONI DI VISIBILITA GLOBALE

void filtra () {
     ifstream in("risultatiglobali.txt");
     char s[101];
  Descrittore d;
  Container contenitore;
  while (!in.eof()) {
     int vett[6];
     double minmax;
     for (int ix=0; ix<6; ix++) in >> vett[ix];
     d.init(vett);
     in >> minmax;
     in.getline(s,100);
     d.setMinmax(minmax);
     contenitore.add(d);
     }
  in.close();
  contenitore.filtra();
  ofstream out ("filtrati.txt");
```

```
  for (int ix=0; ix<contenitore.getSize(); ix++) out << contenitore.getAt(ix);
  out.close();
  };


long double funz (long double x) { return powl(x,-1); }
long double dfunz (long double x) { return -powl(x,-2); }

bool controlla(int vett[]) {
  ifstream in("datigeneralinew.txt");
  if (!in) return false;
  int v;
  while (!in.eof()) {
       bool trovato = true;
       for (int ix=0; ix<6; ix++) {
                    in >> v;
                    if (v!=vett[ix]) trovato = false;
                    }
     if (trovato) {
                    in.close();
                    return true;
                    }
     char s[101];
     in.getline(s,100);
     }
 in.close();
 return false;
 }


void ricercaOttimo() {
  long double (*pf) (long double);
  long double (*pdf) (long double);
  pf = funz;
  pdf = dfunz;
  ifstream in ("DominioRicerca.txt");
  char s[101];
  int st[6];
  in.getline (s,100);
```

```
for (int ix=0; ix<6; ix++) in >> st[ix];
in.getline(s,100);
in.getline(s,100);
int jrEnd[6];
int jr[6];
for (int ix=0; ix<6; ix++) in >> jrEnd[ix];
in.close();
for (int ix=0; ix<6; ix++) jr[ix]=0;
bool finito = false;
while (!finito) {
  ifstream ib("batchfilenew.txt");
  if (ib && !ib.eof()) {
        for (int ix=0; ix<6; ix++) ib >> jr[ix];
        for (int ix=5; ix>=0; ix--) {
              if (jr[ix]==jrEnd[ix]) {
                    if (ix==0) return;
                    jr[ix]=0;
                    }
              else {
                    jr[ix]++;
                    break;
                    }
              }
        }
   else { for (int ix=0; ix<6; ix++) jr[ix]=0; }
   ib.close();
   if (!finito){
        ofstream ob("batchfilenew.txt");
        for (int ix=0; ix<6; ix++) ob << jr[ix] << "\t";
        ob.close();
        int vett[6];
        for (int ix=0; ix<6; ix++) vett[ix]=st[ix]+jr[ix];
              if (!controlla(vett)) {
                    ofstream sumOut("riassunto.txt");
                    ofstream resOut("risultati.txt");
                    sumOut.precision(16);
                    resOut.precision(16);

                    Intervallo i(1,2);
```

```
                                        Approssimatore approx(pf,pdf,i);
                                        approx.setDescrizione (" f(x)=1/x");
                                        approx.setNBit(4);
                                        approx.setBitC0C1C2 (jr[0]+st[0],jr[1]+st[1],jr[2]+st[2]);
                                        approx.setBitDifMult12 (jr[3]+st[3],jr[4]+st[4],jr[5]+st[5]);
                                        approx.setFileRiassunto(&sumOut);
                                        approx.setFileRisultati (&resOut);
                                        approx.esegui();
                                        ofstream dati("datigeneralinew.txt",ios::app);
        dati << (jr[0]+st[0]) << "\t" << (jr[1]+st[1]) << "\t" << (jr[2]+st[2]) << "\t";
        dati << (jr[3]+st[3]) << "\t" << (jr[4]+st[4]) << "\t" << (jr[5]+st[5]) << "\t" << approx.getMinmax() <<"\n";
                                        dati.close();
                                        resOut.close();
                                        sumOut.close();
                                        }
                              else {
                                 cout << "TROVATO: ";
                                 for (int ix=0; ix<6; ix++) cout << (jr[ix]+st[ix]) << " ";
                                 cout << "\n";
                                 }
                              }
                    }
          }

void singleTry (int nb, int bc0, int bc1, int bc2, int bdif, int bm1, int bm2) {
  long double (*pf) (long double);
  long double (*pdf) (long double);
     pf = funz;
     pdf = dfunz;
  ofstream sumOut("riassuntosingle.txt");
  ofstream resOut("risultatisingle.txt");
     sumOut.precision(16);
  resOut.precision(16);
  Intervallo i(1,2);
  Approssimatore approx(pf,pdf,i);
  approx.setDescrizione (" f(x)=1/x");
  approx.setNBit(nb);
  approx.setBitC0C1C2 (bc0,bc1,bc2);
```

```
  approx.setBitDifMult12 (bdif,bm1,bm2);
  approx.setFileRiassunto(&sumOut);
  approx.setFileRisultati (&resOut);
  approx.esegui();
  approx.scriviTabellaBinario();
  resOut.close();
  sumOut.close();
  }

int main() {
     ricercaOttimo();
  return 0;
     }
```

BIBLIOGRAFIA

[1] STUART F. OBERMAN and MICHAEL J. FLYNN, *Division Algorithms and Implementations,* IEEE Transactions on Computers, Vol. 46, No. 8, August 1997.

[2] DEBJIT DAS SARMA and DAVID W. MATULA, *Faithful Bipartite Rom Reciprocal Tables,* Proc. 12th Symp. Computer Arithmetic, pages 17-28, July 1995.

[3] KAI HWANG, *Computer Arithmetic,* Cap. 5, John Wiley & Sons, 1979

[4] NAOFUMI TAKAGI, *Powering by a Table look-up and a Multiplication with Operand Modification,* IEEE Transactions on Computers Vol. 47, No. 11, November 1998.

[5] VIJAY K. JAIN, SUHRID A. WADEKAR and LEI LIN, *A Univerasal Nonlinear Component and Its Application to WSI,* IEEE Transactions on Components, Hybrids, and Manufacturing Technology, Vol. 16, No. 7, November 1993.

[6] JUN CAO and BELLE W. Y. WEI, *High Performance Hardware for Function Generation*, Proc. 13th Symp. Computer Arithmetic, pages 184-188, 1997.

[7] NIKOLAI S. BACHVALOV, *Metodi Numerici*, Cap. 4, Editori Riuniti Roma, 1981

[8] KENDALL and ATKINSON, *Notes on the Remes Algorithm* http://www.math.uiowa.edu/~atkinson/m170.dir/remez.pdf

[9] J.A. PIÑEIRO, J.D. BRUGUERA, J. M. MULLER, *Powering by Table Look-Up using a second-degree minimax approximation with fused accumulation tree,* Department of Electrical and Computer Engineering Univ. Santiago de Compostela, Spain. (Internal Report October 2000).
http://www-gpaa.dec.usc.es/files/articulos/2000/gac2000-i05.ps.gz