OPERAZIONI FLOATING POINT NEI DSP DELLA FAMIGLIA ADSP-2106X

Sergio Sigala 026790
ssigala@globalnet.it
0347-4443326
0364-534107

24 luglio 2001

UNIVERSITA' DEGLI STUDI DI BRESCIA
FACOLTA' DI INGEGNERIA
A.A. 2000/2001
Corso di Elettronica Dei Sistemi Digitali
Professoressa Alessandra Flammini

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

In questa sezione si descrivono gli obiettivi dell'elaborato e si giustifica la necessità dei formati numerici con virgola mobile.

1.1 Presentazione dell'elaborato

Questo lavoro analizza come vengono implementati i calcoli in virgola mobile su una classe particolare di microprocessori, i *DSP*. E' organizzato logicamente in due parti.

Nella prima parte viene introdotta e discussa la rappresentazione floating point e vengono presentati alcuni algoritmi tipici usati per eseguire le operazioni floating point fondamentali. Nella seconda si prende come riferimento una famiglia di DSP commerciali: si descrive la sua architettura e si analizzano le istruzioni macchina floating point messe a disposizione. Successivamente si considera il compilatore C distribuito dal produttore e per concludere si analizza l'implementazione delle funzioni matematiche nella libreria C fornita a corredo.

1.2 Il mondo dei DSP

I dispositivi DSP (acronimo di digital signal processor) sono una classe speciale di microprocessori ottimizzati per eseguire i calcoli in tempo reale impiegati nell'elaborazione numerica dei segnali. Anche se è possibile usare microprocessori general-purpose veloci per eseguire questi calcoli, essi non sono particolarmente adatti per questo tipo di compiti: il sistema risultante può essere difficile da implementare e costoso da costruire. Al contrario, le architetture dei DSP moderni semplificano la fase di disegno e implementazione di questi sistemi e rendono conveniente l'elaborazione numerica dei segnali anche dal punto di vista economico.

Gli algoritmi usati per l'elaborazione dei segnali possono essere ottimizzati efficacemente se vengono scritti per un'architettura pensata e realizzata proprio per essi. Per gestire efficacemente l'elaborazione dei segnali, un generico microprocessore deve avere queste caratteristiche:

- unità di calcolo veloci e flessibili;
- flusso di dati da e per le unità di calcolo rapido;
- unità di calcolo con precisione e range dinamico elevati;
- sistema di generazione indirizzi veloce e flessibile;
- esecuzione efficiente del codice e dei salti.

1.3 Descrivere numericamente la realtà

I sistemi digitali sono molto veloci nel manipolare i numeri; purtroppo però possono farlo solo con una certa approssimazione. Come esempio si consideri il caso della costante matematica *e*: un computer che rapprenta i numeri reali in virgola mobile a 32 bit approssimerà *e* con 2.71828182.

In molte applicazioni l'errore commesso può essere trascurabile, mentre in altri casi può essere disastroso. Basti pensare alla propagazione degli errori dovuta ad arrotondamento nei dati di in un programma che elabora iterativamente le stesse informazioni per molti cicli. Un'altro esempio dell'incapacità dei sistemi reali di elaborare numeri con precisione arbitraria è la somma delle quantità 1 e 0.00000001: il computer precedente darà come risultato 1.

Il problema di avere precisione di rappresentazione finita è dovuto, nei sistemi digitali, al fatto che i numeri vengono memorizzati in celle di memoria di dimensione fissata e dunque l'unico modo per avere maggiore precisione è impiegare più bit nella rappresentazione. D'altronde impiegare più bit necessita di più memoria e dunque bisogna cercare un compromesso.

I numeri sono classificabili in più insiemi, tra i quali i più noti sono quelli dei naturali, degli interi e dei reali. Ogni dispositivo che deve manipolare una o più di queste categorie di numeri deve disporre, per ciascuna categoria interessata, di almeno una rappresentazione adatta alla memorizzazione e di opportuni algoritmi di elaborazione, sotto forma di librerie di routine matematiche oppure di istruzioni macchina specifiche. Per caratterizzare una certa rappresentazione sono stati introdotti diversi parametri, tra i quali: *range dinamico*, *errore relativo* e *risoluzione*, che vengono definiti successivamente.

Data per scontata la conoscenza delle rappresentazioni intere (che qui chiameremo fixed point), con e senza segno, nel seguito ci si concentrerà sulle rappresentazioni floating point.

2 I numeri floating point

In questa sezione vengono introdotti i numeri a virgola mobile e se ne spiega l'utilità.

2.1 Introduzione

Lo schema di codifica e gli algoritmi di calcolo per i numeri floating point sono notoriamente più complessi di quelli impiegati per i fixed point. L'idea di base della codifica floating point è la stessa usata nella notazione scientifica classica dei numeri, secondo la quale ogni numero deve essere scritto come:

$$(-1)^s \cdot man \cdot 10^{exp}$$

dove $s \in \{0,1\}$ è il segno, *man* è la mantissa e *exp* l'esponente del numero. La differenza principale è che la base è 2 invece che 10, poiché i microprocessori lavorano in base 2 molto efficientemente. Di ogni numero è quindi sufficiente memorizzare segno, mantissa e esponente; in particolare per rappresentare il segno è necessario un solo bit. Poiché i numeri dovranno essere memorizzati in un formato ben definito le ampiezze, espresse in numero di bit, della mantissa e dell'esponente vanno fissate a priori e caratterizzano direttamente una particolare rappresentazione e le sue proprietà.

2.2 Numeri normalizzati

Ciascun numero possiede differenti rappresentazioni nella notazione scientifica. Per esempio, tutte le forme seguenti sono perfettamente equivalenti:

$$13.2 \cdot 10^4 = 1.32 \cdot 10^5 = 0.132 \cdot 10^6 = 0.0132 \cdot 10^7$$

Per evitare questa molteplicità, per ciascun numero si sceglie una particolare rappresentazione tra quelle equivalenti che si chiama *normale*, dando luogo a una classe di numeri chiamati *normalizzati*. Nei numeri normalizzati l'esponente è scelto in modo da avere una sola cifra significativa prima della virgola, pertanto ciascun numero possiede una sola rappresentazione normalizzata. Questa forma viene cercata anche nei numeri floating point, per semplificare gli algoritmi e per massimizzare il numero di bit significativi nella mantissa. Tutte le altre rappresentazioni sono dette *denormalizzate*.

2.3 Benefici dei floating point

I formati numerici gestiti da un microprocessore determinano la sua abilità a gestire dati con differente range dinamico e precisione. Di seguito si illustrano le caratteristiche essenziali che differenziano i floating point dai fixed point, in riferimento all'implementazione che ne viene fatta nei DSP.

2.3.1 Range dinamico

La proprietà fondamentale della rappresentazione floating point è quella di essere in grado di rappresentare facilmente numeri molto grandi o molto piccoli mantenendo inalterato il formato di rappresentazione. Ad esempio, con una rappresentazione floating point a 32 bit è possibile memorizzare sia $6.022 \cdot 10^{23}$, il numero di Avogadro, che $1.38 \cdot 10^{-23}$, la costante di Boltzmann, cosa impossibile da farsi con un formato fixed point.

Con range dinamico (o range di rappresentazione) usualmente si intende l'insieme $[min, max]$ di valori rappresentabili da una certa rappresentazione; talvolta con questo termine si indica la loro differenza $max - min$.

2.3.2 Errore relativo

Dato un numero x in una certa rappresentazione, definiamo:

- *successore* di x il minimo tra i numeri maggiori di x che ha rappresentazione distinta da quella di x, se esiste;
- *predecessore* di x il massimo tra i numeri minori di x che ha rappresentazione distinta da quella di x, se esiste;
- *numeri adiacenti* di x l'insieme formato dal predecessore e dal successore di x;
- *gap* (passo o salto) la differenza tra i valori di un numero e di un suo adiacente.

Nei numeri fixed point il gap tra un numero e un suo adiacente è ovviamente sempre 1. Nella notazione floating point, invece, il gap tra due numeri adiacenti varia all'interno del range di rappresentazione. Se prendiamo un certo numero floating point, ci accorgiamo che il gap che lo separa da un adiacente è di qualche milione di volte inferiore al valore del numero stesso. Questo è il concetto chiave della notazione floating point: grandi numeri hanno grossi gap con i loro adiacenti, mentre piccoli numeri hanno gap ridotti.

L'errore relativo è una misura del valore di questo gap rispetto al numero che si sta considerando e per i floating point risulta essere approssimativamente costante su tutto il range di rappresentazione.

2.3.3 Risoluzione

Per risoluzione si intende l'errore massimo di rappresentazione rispetto al valore di fondo scala ed è espressa dalle dimensioni del campo della mantissa. Poiché il campo della mantissa occupa una parte notevole (in numero di bit) nella rappresentazione del numero, se ne deduce che la risoluzione di un numero floating point non è molto peggiore della risoluzione di un numero fixed point rappresentato con il medesimo numero di bit.

2.3.4 Rapporto segnale/rumore

Come detto precedentemente, il gap tra un numero floating point e un suo adiacente è di qualche milione di volte inferiore al valore del numero stesso. Per memorizzare un numero reale x arbitrario, questo deve essere arrotondato in difetto o in eccesso a un valore rappresentabile y che differisce da x per un valore massimo pari alla metà del gap:

$$|x-y| \leq \frac{gap}{2}$$

In altre parole, ogni volta che si memorizza un numero in formato floating point viene aggiunta una componente di rumore dovuta all'arrotondamento. La stessa cosa accade quando si memorizza un numero in formato fixed point, con la importante differenza che il rumore additivo è molto più rilevante, poiché il gap tra i numeri fixed point è superiore.

2.3.5 Semplicità d'uso

Idealmente, i DSP con ALU floating point dovrebbero essere più facili da usare perché il programmatore può concentrarsi sugli algoritmi ad alto livello piuttosto che sui problemi che possono nascere durante la loro codifica a basso livello (tra i quali overflow e errori di troncamento).

2.3.6 Efficienza di calcolo

Nei DSP le operazioni floating point impiegano lo stesso numero di cicli di clock delle operazioni fixed point (tipicamente 1), dunque le operazioni floating point

non sono onerose in termini computazionali e cercare di eseguirne il numero minore possibile non costituisce un'ottimizzazione. Questo ha la conseguenza benefica che le routine ottimizzate per DSP che fanno uso di calcoli floating point sono spesso più facilmente comprensibili di analoghe routine ottimizzate per DSP fixed point.

3 Lo standard IEEE754

In questa sezione si introduce lo standard di riferimento per i numeri a virgola mobile.

3.1 Introduzione

Anche se vi sono molti formati floating point simili in uso, i più comuni sono quelli definiti nello standard ISO/IEEE STD 754-1985, al quale nel seguito ci si riferirà con la notazione *standard IEEE* o semplicemente con *standard*. Lo standard descrive sia le rappresentazioni che il modo nel quale le quattro operazioni aritmetiche *elementari* vanno eseguite.

3.2 Tipi di rappresentazione

Lo standard definisce il formato per i numeri floating point a 32 bit (*single precision*) e a 64 bit (*double precision*), oltre che certi tipi di rappresentazioni estese; vediamo singolarmente le prime due rappresentazioni e le rispettive proprietà.

3.2.1 Rappresentazione a 32 bit

Come mostrato nella figura 1, nella rappresentazione single precision i bit $[22,0]$ formano il valore f che rappresenta la mantissa, i bit $[30,23]$ formano il numero e che rappresenta l'esponente e il bit 31 è il bit di segno s.

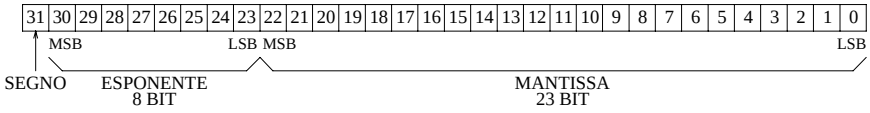

Figura 1: Formato single precision.

A seconda delle varie configurazioni dei bit della rappresentazione si identificano diverse categorie di numeri, come indicato nella tabella 1.

Il numero e varia nel range $[0,255]$ perciò, per rappresentare gli esponenti negativi, si conviene di sottrargli 127 (*bias*). In questo modo si ottiene un'esponente effettivo (detto anche esponente *unbiased*) variabile nell'intervallo [-127,128], che può finalmente essere utilizzato per calcolare il fattore di scala. Il numero 127 è il valore medio dell'intervallo $[0,255]$, scelto in modo da avere circa lo stesso range di escursione per le potenze negative e per quelle positive.

I valori estremi di e, 0 e 255, vengono usati rispettivamente per rappresentare i numeri speciali ± 0 e $\pm\infty$, quando f è nullo; il segno di questi numeri dipende

Numero rappresentato	f	e	s
-0	0	0	1
$+0$	0	0	0
$-\infty$	0	255	1
$+\infty$	0	255	0
N- (normalizzati negativi)	$\neq 0$	$\neq 0, 255$	1
N+ (normalizzati positivi)	$\neq 0$	$\neq 0, 255$	0
D- (denormalizzati negativi)	$\neq 0$	0	1
D+ (denormalizzati positivi)	$\neq 0$	0	0
NaN	$\neq 0$	255	0/1

Tabella 1: Classi di numeri per il formato single precision.

sempre da *s*. Per i numeri regolari il range di *e* è dunque $[1, 254]$, il che implica che l'esponente può variare nel range $[-126, 127]$.

Il campo f coincide con la mantissa effettiva solo quando il numero è *denormalizzato*. Si cerca di lavorare sempre con numeri normalizzati, in cui l'esponente viene aggiustato per avere il bit più significativo della mantissa sempre a 1. In tale situazione il bit più significativo non reca alcuna informazione utile per ricostruire il numero memorizzato e pertanto viene eliminato dalla rappresentazione f della mantissa e si dice che tale bit è *implicito* o *hidden*. In questo modo si risparmia un bit che può essere sfruttato per aumentare il numero di bit della mantissa, aumentando la risoluzione. Inoltre, da un confronto dei valori *e* di due numeri normalizzati si può determinare rapidamente la differenza tra i loro ordini di grandezza. I 23 bit memorizzati in f rappresentano la parte frazionaria della mantissa, cioè i bit alla destra del punto decimale.

Data la rappresentazione di un numero normalizzato, il suo valore *v* è:

$$v = (-1)^s (1.f) 2^{e-127}$$

Il fattore di scala è compreso nel range $[2^{-126}, 2^{127}]$, che corrisponde a circa $[10^{-38}, 10^{38}]$. I più piccoli normalizzati rappresentabili sono $\pm 1.00\ldots00 \cdot 2^{-126} \simeq \pm 1.2 \cdot 10^{-38}$, mentre i più grandi sono $\pm 1.11\ldots11 \cdot 2^{127} \simeq \pm 3.4 \cdot 10^{38}$. I 24 bit di una mantissa normalizzata consentono di avere circa la stessa precisione di un numero decimale con 8 cifre significative.

Vengono riservate alcune configurazioni di bit per rappresentare numeri denormalizzati, che possono essere prodotti da alcune operazioni. Data la rappresentazione di un numero denormalizzato, il suo valore *v* è:

$$v = (-1)^s (0.f) 2^{-126}$$

L'esponente viene fissato a 126 in modo tale che il minimo dei normalizzati ($1.00\ldots00\cdot 2^{-126}$) sia il successore del massimo dei denormalizzati ($0.11\ldots11\cdot 2^{-126}$). Poiché il fattore di scala dei denormalizzati è fisso a 2^{-126} il loro range è più ristretto di quello dei normalizzati: si estende tra $\pm 0.00\ldots01\cdot 2^{-126}\simeq\pm 1.4\cdot 10^{-45}$ e $\pm 0.11\ldots11\cdot 2^{-126}\simeq\pm 1.2\cdot 10^{-38}$.

Se i denormalizzati venissero trattati nello stesso modo dei normalizzati, cioè interpretati come $v=(-1)^s(1.f)2^{e-127}$, il più piccolo numero rappresentabile sarebbe $\pm 1.00\ldots01\cdot 2^{-127}\simeq\pm 5.9\cdot 10^{-38}$, aggiungendo il range di rappresentazione $[\pm 5.9\cdot 10^{-38},\pm 1.2\cdot 10^{-38}]$ a quello già offerto dai normalizzati. Purtroppo l'aumento di range che si ha con un'interpretazione normalizzata dei denormalizzati è più piccolo di quello fornibile dall'interpretazione denormalizzata.

Più avanti vedremo che le istruzioni macchina dei dispositivi commerciali e le complementari routine matematiche fornite a loro corredo gestiscono solo parzialmente i denormalizzati.

Esistono anche configurazioni di bit che non ricadono nelle classi precedenti, chiamate comunemente *NaN* (not a number), per significare che non rappresentano alcun numero. Ad esempio, una configurazione *NaN* viene sfruttata per rappresentare il risultato delle operazioni $0\cdot\infty$ e $0/0$.

3.2.2 Rappresentazione a 64 bit

Come mostrato nella figura 2, nella rappresentazione double precision i bit $[51,0]$ formano il valore f che rappresenta la mantissa, i bit $[63,52]$ formano il valore e che rappresenta l'esponente e il bit 63 è il bit di segno s.

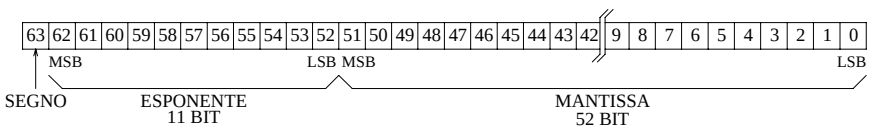

Figura 2: Formato double precision.

A seconda delle varie configurazioni dei bit della rappresentazione si identificano diverse categorie di numeri, come indicato nella tabella 2. Poiché e varia nel range $[0,2047]$, il valore del bias viene fissato al valore intermedio 1023.

Data la rappresentazione di un numero normalizzato, il suo valore v è:

$$v=(-1)^s(1.f)2^{e-1023}$$

Il fattore di scala varia all'interno dell'intervallo $[2^{-1022},2^{1023}]$, che corrisponde a circa $[10^{-308},10^{308}]$. I più piccoli normalizzati rappresentabili risultano $\pm 1.00\ldots00\cdot 2^{-1022}\simeq\pm 2.2\cdot 10^{-308}$, mentre i più grandi $\pm 1.11\ldots11\cdot 2^{1023}\simeq$

Numero rappresentato	f	e	s
-0	0	0	1
$+0$	0	0	0
$-\infty$	0	2047	1
$+\infty$	0	2047	0
N- (normalizzati negativi)	$\neq 0$	$\neq 0,2047$	1
N+ (normalizzati positivi)	$\neq 0$	$\neq 0,2047$	0
D- (denormalizzati negativi)	$\neq 0$	0	1
D+ (denormalizzati positivi)	$\neq 0$	0	0
NaN	$\neq 0$	2047	0/1

Tabella 2: Classi di numeri per il formato double precision.

$\pm 1.8 \cdot 10^{308}$. I 53 bit di una mantissa normalizzata consentono di avere circa la stessa precisione di un numero decimale con 16 cifre significative.

Data la rappresentazione di un numero denormalizzato, il suo valore *v* è:

$$v = (-1)^s (0.f)\, 2^{-1022}$$

Poiché il fattore di scala dei denormalizzati è fisso a 2^{-1022} il loro range è più piccolo di quello dei normalizzati: tra $\pm 0.00\ldots01 \cdot 2^{-1022} \simeq \pm 4.9 \cdot 10^{-324}$ e $\pm 0.11\ldots11 \cdot 2^{-1022} \simeq \pm 2.2 \cdot 10^{-308}$.

3.2.3 Rappresentazioni estese

Lo standard propone anche alcuni tipi di rappresentazioni estese, per entrambi i formati single precision e double precision, che hanno il duplice scopo di aumentare la precisione e il range degli esponenti. L'uso di formati estesi aiuta a ridurre l'accumolo di errore d'arrotondamento in una sequenza di calcoli, migliora l'accuratezza delle operazioni aritmetiche elementari e rende più infrequenti le occasioni di underflow o overflow.

Consideriamo brevemente il caso della rappresentazione extended precision a 40 bit, mostrata nella figura figura 3, perché viene utilizzata frequentemente nei DSP. I bit $[30,0]$ formano il valore f che rappresenta la mantissa, i bit $[38,31]$ formano il valore *e* che rappresenta l'esponente e il bit 39 è il bit di segno *s*. Questo formato estende la single precision, l'unica differenza è la dimensione più ampia della mantissa; il range dell'esponente rimane invariato.

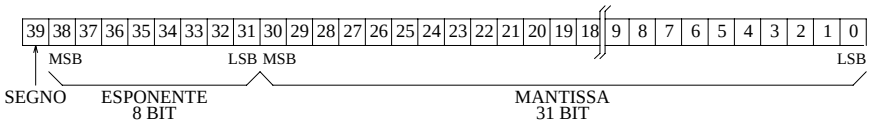

Figura 3: Formato extended precision a 40 bit.

3.3 Le eccezioni

Durante l'esecuzione di operazioni aritmetiche in virgola mobile possono verificarsi a volte delle condizioni di errore o *eccezioni*. La più conosciuta è la divisione per 0, ma lo standard ne definisce altre, generate da operazioni che producono risultati non validi oppure fuori dal range dei numeri floating point normalizzati.

Un modo piuttosto comune di gestire queste condizioni di errore è quello di sospendere immediatamente l'esecuzione del programma (mediante *TRAP*, o interrupt software) e passare il controllo a delle routine apposite definibili dall'utente per gestire l'eccezione. Se l'utente non implementa queste routine possono essere invocate quelle di default del sistema.

Un altro metodo è quello di fare in modo che l'operazione che ha provocato l'eccezione restituisca uno speciale *error code* che segnali il problema. Questo secondo metodo è usato raramente per due motivi:

- Affinché l'utente sia in grado di riconoscere l'eccezione, l'error code ritornato deve essere distinto da qualunque valore numerico ordinario; inoltre, le varie eccezioni dovrebbero produrre error code diversi, ma spesso è difficile ottenere i bit aggiuntivi necessari per codificare il tipo di eccezione perché questo comporterebbe una riduzione del range dei numeri rappresentabili.

- Dopo l'esecuzione di qualsiasi operazione bisognerebbe inserire un salto condizionato per controllare se si è verificata un'eccezione, complicando e rallentando il programma.

Secondo lo standard bisogna identificare e poter gestire diverse eccezioni, descritte in dettaglio di seguito.

3.3.1 Overflow

Generata quando il risultato ha un valore assoluto troppo grande per poter essere rappresentato, cioè un valore minore del minimo numero negativo rappresentabile o maggiore del massimo numero positivo rappresentabile. Per esempio, la condizione si verifica quando si somma un numero molto elevato a se stesso; un'altro

esempio è la conversione di un numero elevato da double precision a single precision. L'uso della precisione estesa per tutti i risultati intermedi fa in modo che questa eccezione si verifichi molto raramente.

3.3.2 Underflow

Generata da un risultato troppo vicino allo 0, cioà un valore compreso tra il massimo numero negativo rappresentabile e il minimo numero positivo rappresentabile. Dividendo 1.0 per un numero elevato si ottiene una condizione di questo tipo; un'altro esempio è la conversione di un piccolo numero positivo non nullo da double precision a single precision. Anche in questo caso l'eccezione si verifica molto raramente se si utilizza la precisione estesa per rappresentare tutti i risultati intermedi.

Nel caso di underflow, una tipica routine di gestione dell'eccezione potrebbe silenziosamente arrotondare il risultato a 0 e ritornare il controllo al programma utente, per continuare nei calcoli successivi.

Poiché vi è perfetta simmetria tra i numeri negativi e positivi, l'operazione di valore assoluto non può mai generare overflow o underflow; tale proprietà non è vera, per esempio, per i formati fixed point.

Si focalizzi ora l'attenzione sui numeri normalizzati, positivi o negativi: se, per ogni formato standard, si moltiplica il più piccolo per il corrispondente più grande si trova che il risultato è approssimativamente 4. In altre parole il numero più grande in valore assoluto è approssimativamente 4 volte il reciproco del più piccolo; pertanto le operazioni del tipo $1/x$, $2/x$, $3/x$ e π/x non possono produrre un overflow, ma piuttosto un underflow.

3.3.3 Loss of precision, significance loss

Risultato inesatto rispetto alla tecnica di arrotondamento scelta. Questa condizione si verifica quando il risultato di un'operazione floating point non può essere rappresentato in modo preciso nel formato di destinazione scelto. Alcuni esempi che producono questa situazione:

- calcolo $1.0/3.0$: fornisce in base 2 una frazione periodica $0.010101\ldots$ che non può essere rappresentata esattamente in alcun formato floating point binario;
- conversione dalla double precision verso la single precision del numero 9.87654321: la single precision non prevede abbastanza bit per tutte le nove cifre significative;
- moltiplicazione di due valori uguali: la precisione viene dimezzata;

- sottrazione di due valori molto vicini fra loro: la precisione viene persa in gran parte.

La maggior parte dei programmatori che utilizzano l'aritmetica floating point non considerano un risultato inesatto come un errore, ma piuttosto prevedono che si verifichi una specie di troncamento o arrotondamento e quindi ignorano questa eccezione, se viene supportata, *mascherandola*. In tale situazione il dispositivo dovrebbe troncare o arrotondare il risultato secondo il modo di arrotondamento correntemente selezionato, che può essere scelto tra molti possibili, come vedremo.

3.3.4 Division by 0

Si verifica quando un numero non nullo viene diviso per 0. Invece di sospendere l'esecuzione del programma, può essere comodo fare in modo che il risultato assuma i valori speciali $-\infty$ o $+\infty$, a seconda del segno del divisore.

3.3.5 Invalid operation

Operazione non valida; questa categoria include tutti i casi che non ricadono nelle classi precedenti, per esempio $0/0$ oppure $\sqrt{-1}$. Queste operazioni possono produrre un particolare risultato *NaN*, detto *undefined*.

La generazione di eccezioni del tipo divisione per zero e operazione non valida spesso sono sintomi del malfunzionamento del programma, quindi potrebbe essere saggio terminarne l'esecuzione e passare il controllo al sistema operativo, se esiste.

3.4 Modi di arrotondamento

Lo standard ne definisce quattro: *arrotondamento verso lo 0, arrotondamento al più vicino e al pari, arrotondamento verso* $-\infty$, *arrotondamento verso* $+\infty$. Il loro funzionamento è illustrato nella figura 4. Le crocette sull'asse indicano i risultati veri, di precisione infinita, delle operazioni aritmetiche. I segmenti rappresentano invece i numeri floating point più vicini, minori o maggiori dei risultati veri.

3.4.1 Arrotondamento verso lo 0

Conosciuto anche come *troncamento* o *chopping*: il risultato viene rappresentato nel formato di destinazione ignorando alcuni bit meno significativi. Ciò equivale a scegliere come risultato il più vicino numero floating point più piccolo in valore assoluto del risultato vero.

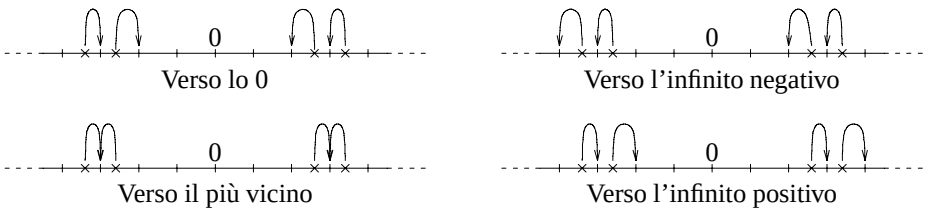

Figura 4: Modi di arrotondamento.

Il range dell'errore commesso è $[-1, 0]$ LSB rispetto ai bit rimanenti; poiché non è simmetrico rispetto allo 0 si dice che l'approssimazione è *biased*. Questo tipo di arrotondamento è più utile nell'aritmetica intera dove, ad esempio, $3/2$ viene arrotondato in 1 e $-3/2$ in -1.

3.4.2 Arrotondamento al più vicino e al pari

Per minimizzare l'errore di arrotondamento si sceglie come risultato il numero floating point più vicino al numero vero; se il risultato vero si trova a metà del gap tra una coppia di numeri floating point, si deve scegliere quello pari, cioè quello che ha il bit meno significativo nullo. Il range dell'errore é $[-1/2, 1/2]$ LSB rispetto ai bit rimanenti e perciò l'arrotondamento è *unbiased*.

Questo risulta essere il metodo d'arrotondamento migliore ma purtroppo anche il più oneroso da implementare, perché complesso e può richiedere un'eventuale rinormalizzazione del risultato.

3.4.3 Arrotondamento verso $-\infty$

Si approssima il risultato al numero floating point appena minore.

3.4.4 Arrotondamento verso $+\infty$

Si approssima il risultato al numero floating point appena maggiore.

Gli arrotondamenti verso $-\infty$ e verso $+\infty$ vengono utilizzati per realizzare l'*aritmetica degli intervalli*. Un algoritmo che utilizza l'aritmetica floating point ordinaria fornisce un risultato che è un'approssimazione del risultato reale e risulta difficile analizzare l'algoritmo per verificare se l'approssimazione fornita sia sufficiente.

Un'approccio alternativo, che elimina l'esigenza dell'analisi della bontà degli arrotondamenti, consiste nell'eseguire ciascuna operazione aritmetica due volte, la prima volta eseguendo un arrotondamento verso $-\infty$ e la seconda con arroton-

damento verso $+\infty$. In questo modo si garantisce che il risultato vero sia compreso nel range dei due risultati ottenuti.

Purtroppo non è possibile ricavare i due valori estremi del range del risultato dell'esecuzione di un intero algoritmo semplicemente eseguendolo dapprima con l'arrotondamento verso $-\infty$ e poi con l'arrotondamento verso $+\infty$. Infatti, per fare un esempio, un problema che si presenta è nella moltiplicazione per un numero negativo: l'approssimazione verso $-\infty$ viene convertita in approssimazione verso $+\infty$ e viceversa; tenere traccia di tutte queste situazioni è difficile.

3.4.5 Altri modi d'arrotondamento

Per completezza descriviamo anche il *Von Neumann rounding*, non definito dallo standard. Se i bit che devono essere eliminati sono tutti nulli essi sono semplicemente scartati, senza effetti sui bit rimanenti. Se invece c'è almeno un bit 1, il bit meno significativo del blocco di bit rimanenti viene posto a 1.

L'errore commesso è nell'intervallo di $[-1, 1]$ LSB rispetto ai bit rimanenti e dunque l'arrotondamento è unbiased.

3.4.6 Discussione

Le approssimazioni unbiased possono essere vantaggiose rispetto a quelle biased nel caso di calcoli abbastanza lunghi, perché statisticamente gli errori tendono a compensarsi e pertanto è possibile ottenere risultati con migliore accuratezza.

Secondo lo standard, i risultati delle singole operazioni elementari devono essere calcolati con un'errore entro $[-1/2, 1/2]$ LSB rispetto al valore vero. Per ottenere ciò è necessario, in generale, impiegare il metodo d'arrotondamento al più vicino e al pari; si dimostra che per implementare tale metodo con questo errore massimo è sufficiente aumentare le dimensioni delle mantisse temporanee usate durante i passi intermedi delle operazioni elementari di 3 bit, detti *guard bits* (bit di guardia).

I primi due bit costituiscono semplicemente un'aggiunta di due bit di risoluzione; il terzo bit, chiamato *sticky bit*, è l'OR logico di tutti i bit meno significativi che vengono scartati durante i passi intermedi.

All'inizio dell'esecuzione dell'operazione i guard bits degli operandi vengono inizializzati a zero; alla fine dell'esecuzione il troncamento della mantissa con arrotondamento al più vicino e al pari si effettua considerando la terna dei 3 bit come un numero binario g in questo modo:

- se $g < 100$ si setta il LSB della mantissa troncata a 0 (arrotonda in difetto);
- se $g = 100$ si decide in base al valore del LSB della mantissa troncata come segue:

  - se è 0 non si fa nulla;
  - se è 1 si somma 1 LSB alla mantissa troncata;

- se $g > 100$ si setta il LSB della mantissa troncata a 1 (arrotonda in eccesso).

3.5 Osservazioni

Secondo lo standard qualsiasi implementazione commerciale di un dispositivo che elabora numeri floating point deve avere almeno una rappresentazione single precision conforme. La rappresentazione double precision non è obbligatoria.

Lo standard richiede inoltre che, oltre alle quattro operazioni elementari, siano implementate anche la divisione con resto, la radice quadrata e le conversioni tra rappresentazioni binarie e decimale.

3.5.1 Ordinamento dei campi

L'ordinamento dei campi di un numero in virgola mobile è coerente con l'idea che la parte più importante o più *significativa* debba essere memorizzata nei campi più alti. Il campo del segno è il più importante nel confronto tra due numeri floating point, successivamente vi è l'esponente e infine la mantissa.

3.5.2 Numeri denormalizzati

Come visto, lo standard ammette che in determinate circostanze certi numeri reali possano essere rappresentati in forma denormalizzata, al fine di ridurre al minimo l'ampiezza dell'intervallo tra lo 0 e più vicini valori rappresentabili, che risultano essere proprio una coppia di numeri denormalizzati. Questo strappo alla regola permette di approssimare meglio lo 0 usando parti frazionarie sempre più piccole, che portano a un *gradual underflow*.

Il fatto che siano occasionalmente ammessi numeri reali rappresentati in forma denormalizzata è un problema per i progettisti di unità di calcolo floating point veloci, che devono considerare anche queste situazioni. Di conseguenza, in molti dispositivi i numeri denormalizzati non sono supportati e la presenza di un operando denormalizzato in un'operazione aritmetica genera un'eccezione.

3.5.3 Doppia rappresentazione dello 0

Come visto, le infinità e lo 0 possono essere rappresentate con un segno arbitrario; lo standard si preoccupa abbastanza di queste situazioni. Disporre di rappresentazioni distinte è utile perché consente di implementare più facilmente le funzioni

matematiche. Ad esempio, in una funzione di divisione possono essere codificate le seguenti regole convenzionali, dove si intende che $0 < x < +\infty$:

$$\begin{aligned} x/+0 &= +\infty \\ -x/+0 &= -\infty \\ x/-0 &= -\infty \\ -x/-0 &= +\infty \end{aligned}$$

3.5.4 Diversi tipi di *NaN*

Lo standard distingue i *silent NaN* dai *signaling NaN*. Operandi di tipo silent *NaN* dovrebbero produrre risultati di tipo silent *NaN*, senza generare eccezioni. La propagazione passiva di questi silent *NaN* attraverso calcoli successivi appare quindi come un'infezione silenziosa. Viceversa, la presenza di un signaling *NaN* come operando di un'operazione deve generare un'eccezione. I signaling *NaN* possono essere usati per rappresentare numeri speciali, le cui operazioni aritmetiche vengono definite via software: la generazione di un'eccezione consente al relativo gestore di calcolare il risultato corretto e di restituirlo al programma, che può riprendere la sua normale esecuzione.

3.5.5 Vantaggi dell'aderenza allo standard

Si riportano alcuni dei vantaggi di cui gode l'utilizzatore di un microprocessore che aderisce allo standard:

- scambio dati con altri dispositivi più facile perché non si devono effettuare conversioni; al massimo l'unica preoccupazione può essere quella di modificare l'ordine dei byte (*little endian* contro *big endian*);
- non è necessario apprendere un nuovo formato di rappresentazione;
- il codice esistente può essere riadattato più facilmente.

Ora vengono riassunti i punti che possono giustificare, in talune situazioni, la non aderenza o l'aderenza parziale allo standard:

- in alcuni dispositivi la perfetta aderenza allo standard può comportare minori prestazioni nei calcoli (certi tipi di DSP);
- in alcune applicazioni l'aderenza non è necessaria (ad esempio nelle calcolatrici tascabili);

Risultato intermedio	Difetto	Eccesso	Al pari
1.24	1.2	1.2	1.2
1.25	1.2	1.3	1.2
1.26	1.3	1.3	1.3
1.34	1.3	1.3	1.3
1.35	1.3	1.4	1.4
1.36	1.4	1.4	1.4

Tabella 3: Primo arrotondamento.

- in semplici applicazioni l'implementazione di tutto lo standard risulta inutile;
- la complessità di gestione dell'underflow graduale, dei codici per le infinità e per i *NaN* spingono gli implementatori a supportare, spesso, solo parti dello standard.

Dunque spesso, nelle implementazioni commerciali, non tutto lo standard è rispettato e a volte i comportamenti non conformi non vengono indicati chiaramente.

3.6 Esempi importanti

Vediamo due esempi che illustrano la superiorità dell'arrotondamento al più vicino e al pari rispetto alle altre tecniche.

Primo esempio

Consideriamo il caso di un ipotetico dispositivo in grado di eseguire calcoli tra numeri floating point decimali, ma il cui formato di rappresentazione finale può contenere solo una cifra frazionaria. Dati alcuni risultati intermedi che devono essere memorizzati nell'operando di destinazione, la tabella 3 illustra come opererebbero alcune strategie di arrotondamento al più vicino. Un numero che ha la seconda cifra frazionaria pari a 5 viene arrotondato verso il predecessore secondo la strategia in difetto, verso il successore secondo la strategia in eccesso.

Si potrebbe pensare che un arrotondamento al più vicino, dove si sceglie a caso il rappresentante nel caso il risultato intermedio sia nel mezzo di un gap, sia statisticamente valido e non vi siano modi di arrotondamento migliori. La spiegazione del perché un'arrotondamento deterministico come quello al pari è migliore si deduce considerando le operazioni successive a cui possono essere sottoposti questi risultati.

Operando	Risultato intermedio	Difetto	Eccesso	Al pari
1.2	0.6	0.6	0.6	0.6
1.3	0.65	0.6	0.7	0.6
1.4	0.7	0.7	0.7	0.7

Tabella 4: Secondo arrotondamento.

Intermedio iniziale	Difetto	Eccesso	Al pari	Risultato vero
1.24	0.6	0.6	0.6	0.62
1.25	0.6	0.7	0.6	0.625
1.26	0.6	0.7	0.6	0.63
1.34	0.6	0.7	0.6	0.67
1.35	0.6	0.7	0.7	0.675
1.36	0.7	0.7	0.7	0.68

Tabella 5: Confronti tra i risultati finali.

Consideriamo solo il caso della divisione per 2.0: la tabella 4 mostra i risultati che si otterrebbero, mentre la tabella 5 compara i valori intermedi iniziali, i risultati finali della seconda operazione e i risultati veri. In sostanza, l'arrotondamento al più vicino e al pari propaga gli errori, inevitabili, in modo più limitato.

Secondo esempio

Esaminiamo il caso di un dispositivo con rappresentazione floating point decimale con parte significativa di cinque cifre, che esegue calcoli con arrotondamento in eccesso. Vediamo come opererebbe su una successione di somme e differenze sui numeri $a = 1.0000$ e $b = 0.55555$:

$$\begin{aligned} a+b &= 1.5556 \\ -b &= 1.0001 \\ +b-b &= 1.0002 \\ +b-b &= 1.0003 \end{aligned}$$

I risultati sono crescenti, nonostante si aggiunga e si sottragga sempre lo stesso valore. Questa deviazione scompare se si calcola la stessa somma con arrotondamento al più vicino e al pari:

$$a+b = 1.5556$$

Descrizione	Codice esadecimale	Valore decimale
+0	`0x00000000`	0.0
-0	`0x80000000`	-0.0
+1	`0x3F800000`	1.0
-1	`0xBF800000`	-1.0
+2	`0x40000000`	2.0
massimo norm. positivo	`0x7F7FFFFF`	3.40282347E+38
minimo norm. positivo	`0x00800000`	1.17549435E-38
massimo denorm. positivo	`0x007FFFFF`	1.17549421E-38
minimo denorm. positivo	`0x00000001`	1.40129846E-45
$+\infty$	`0x7F800000`	$+\infty$
$-\infty$	`0xFF800000`	$-\infty$

Tabella 6: Costanti single precision.

$$\begin{aligned} -b &= 1.0000 \\ +b-b &= 1.0000 \\ +b-b &= 1.0000 \end{aligned}$$

3.7 Costanti caratteristiche

Le tabelle 6 e 7 riassumono le costanti floating point più usate, con il loro codice esadecimale equivalente.

Descrizione	Codice esadecimale	Valore decimale
+0	0x0000000000000000	0.0
-0	0x8000000000000000	-0.0
+1	0x3FF0000000000000	1.0
-1	0xBFF0000000000000	-1.0
+2	0x4000000000000000	2.0
max norm. pos.	0x7FEFFFFFFFFFFFFF	1.7976931348623157E+308
min norm. pos.	0x0010000000000000	2.2250738585072014E-308
max denorm. pos.	0x000FFFFFFFFFFFFF	2.2250738585072009E-308
min denorm. pos.	0x0000000000000001	4.9406564584124654E-324
$+\infty$	0x7FF0000000000000	$+\infty$
$-\infty$	0xFFF0000000000000	$-\infty$

Tabella 7: Costanti double precision.

4 Tipologie di operazioni

In questa sezione vengono descritte le operazioni che, di solito, devono essere disponibili in ogni applicazione, sotto forma di istruzioni macchina specifiche o di routine software.

Per semplicità gli algoritmi proposti sono a livello di astrazione piuttosto elevato e talvolta non considerano le rappresentazioni fisiche degli operandi. Nonostante ciò vengono indicate le eccezioni che possono essere generate, poiché questa capacità risulta importante per valutare la bontà di ogni implementazione. Chiaramente, l'implementazione ideale di queste funzioni dovrebbe operare correttamente su tutte le possibili combinazioni di operandi d'ingresso che non siano dei NaN, fornendo sempre dei risultati consistenti. In caso un operando sia di tipo NaN dovrebbero ritornare un risultato NaN.

4.1 Operazioni elementari

Si tratta delle operazioni di addizione, sottrazione, moltiplicazione e divisione tra numeri reali.

4.1.1 Addizione e sottrazione

Viene illustrato un algoritmo tipico per l'addizione e la sottrazione. Scriviamo gli operandi e il risultato come:

- $v_1 = 2^{e_1} m_1$ per il primo operando;

- $v_2 = 2^{e_2} m_2$ per il secondo operando;
- $v_3 = 2^{e_3} m_3$ per il risultato.

dove, ovviamente, m_1, e_1, m_2, e_2, m_3 e e_3 non corrispondono ai reali contenuti dei campi delle rappresentazioni dei numeri. Poiché l'operazione può essere definita come:

$$v_3 = v_1 \pm v_2 = \begin{cases} (m_1 \pm m_2 2^{-(e_1 - e_2)}) 2^{e_1} & \text{se } e_1 \geq e_2 \\ (m_1 2^{-(e_2 - e_1)} \pm m_2) 2^{e_2} & \text{se } e_1 < e_2 \end{cases}$$

l'algoritmo che si può applicare è:

1. identifica l'operando con l'esponente più piccolo (in valore assoluto);
2. calcola l'esponente $e = \max\{|e_1|, |e_2|\}$;
3. calcola $d = |e_1 - e_2|$;
4. scorri verso destra di d bit la mantissa dell'operando con l'esponente più piccolo;
5. calcola la mantissa $m = m_1 \pm m_2$;
6. se $m = 0$, setta v_3 con la rappresentazione dello 0 ed esci;
7. normalizza m, decrementando o incrementando e quanto necessario;
8. se e è in underflow, setta v_3 con la rappresentazione dello 0 ed esci;
9. se e è in overflow positivo, setta v_3 con $+\infty$ ed esci;
10. se e è in overflow negativo, setta v_3 con $-\infty$ ed esci;
11. setta v_3 con i valori di m e e.

Possono venire generate queste eccezioni:

- underflow;
- overflow.

4.1.2 Moltiplicazione

L'algoritmo è più semplice di quello dell'addizione e sottrazione poiché non è necessario allineare le mantisse. Scriviamo gli operandi e il risultato come:

- $v_1 = 2^{e_1} m_1$ per il primo operando;
- $v_2 = 2^{e_2} m_2$ per il secondo operando;
- $v_3 = 2^{e_3} m_3$ per il risultato.

dove, ovviamente, m_1, e_1, m_2, e_2, m_3 e e_3 non corrispondono ai reali contenuti dei campi delle rappresentazioni dei numeri. Osservando l'identità:

$$v_3 = v_1 v_2 = m_1 2^{e_1} m_2 2^{e_2} = m_1 m_2 2^{(e_1+e_2)}$$

l'algoritmo che si può usare è:

1. se un operando è nullo setta v_3 con la rappresentazione dello 0 ed esci;
2. calcola la mantissa $m = m_1 m_2$;
3. calcola l'esponente $e = e_1 + e_2$;
4. normalizza m, incrementando e quanto necessario;
5. se e è in underflow, setta v_3 con la rappresentazione dello 0 ed esci;
6. se e è in overflow positivo, setta v_3 con $+\infty$ ed esci;
7. se e è in overflow negativo, setta v_3 con $-\infty$ ed esci;
8. setta v_3 con i valori di m e e.

Possono venire generate queste eccezioni:

- underflow;
- overflow.

4.1.3 Divisione

L'algoritmo della divisione è simile, ad alto livello, a quello della moltiplicazione ma praticamente risulta più complicato da codificare. Scriviamo gli operandi e il risultato come:

- $v_1 = 2^{e_1} m_1$ per il primo operando;
- $v_2 = 2^{e_2} m_2$ per il secondo operando;
- $v_3 = 2^{e_3} m_3$ per il risultato.

dove, ovviamente, m_1, e_1, m_2, e_2, m_3 e e_3 non corrispondono ai reali contenuti dei campi delle rappresentazioni dei numeri. Osservando l'identità:

$$v_3 = \frac{v_1}{v_2} = \frac{m_1 2^{e_1}}{m_2 2^{e_2}} = \frac{m_1}{m_2} 2^{(e_1 - e_2)}$$

l'algoritmo che si può usare è:

1. se m_2 è nulla, setta v_3 con $\pm\infty$ (a seconda del segno di v_1) ed esci;
2. calcola la mantissa $m = \frac{m_1}{m_2}$;
3. calcola l'esponente $e = e_1 - e_2$;
4. normalizza m, decrementando e quanto necessario;
5. se e è in underflow, setta v_3 con la rappresentazione dello 0 ed esci;
6. se e è in overflow positivo, setta v_3 con $+\infty$ ed esci;
7. se e è in overflow negativo, setta v_3 con $-\infty$ ed esci;
8. setta v_3 con i valori di m e e.

Possono venire generate queste eccezioni:

- division by zero;
- invalid operation;
- underflow;
- overflow.

Vedremo che non sempre l'istruzione di divisione viene implementata direttamente come istruzione macchina, ma a volte si preferisce costruire la divisione moltiplicando il primo operando per il reciproco del secondo.

4.2 Operazioni trigonometriche

Data la loro notevole complessità, le funzioni trigonometriche e le trascendenti vengono implementate come istruzioni macchina (più lente rispetto alle operazioni elementari) solo in alcune famiglie di microprocessori general-purpose.

Per quanto riguarda le trigonometriche, verranno considerate le sole funzioni sin, cos e arctan; tutte le altre possono essere calcolate con procedimenti analoghi, oppure applicando semplici formule del tipo:

$$\tan\Theta = \frac{\sin\Theta}{\cos\Theta}$$

Gli algoritmi che di solito vengono utilizzati si dividono in due generi, aventi in comune il primo passo. Chiamiamo Θ l'argomento delle funzioni.

4.2.1 Riduzione dell'argomento

Applicando alcune proprietà di simmetria, di periodicità o certe identità fondamentali ci si riconduce a calcoli equivalenti, ma più convenienti, come illustrato di seguito.

- Le funzioni sin e cos sono periodiche di periodo 2π, quindi si divide Θ per 2π e si prende il resto, ϑ. Si ottiene $\vartheta \in [0, 2\pi[$. L'intevallo può essere reso più piccolo, se si considerano anche le simmetrie di cui godono queste funzioni. Si ricorda, inoltre, che ciascuna funzione può essere ricondotta all'altra, se si tiene conto di una differenza d'angolo di $\pi/2$.

- Per la funzione arctan si considera l'identità:

$$\arctan\Theta = \begin{cases} \arctan\vartheta & \text{con } \vartheta = \Theta, \text{ se } |\Theta| \leq 1 \\ \frac{\pi}{2} - \arctan\vartheta & \text{con } \vartheta = 1/\Theta, \text{ se } \Theta > 1 \\ -\frac{\pi}{2} + \arctan\vartheta & \text{con } \vartheta = 1/\Theta, \text{ se } \Theta < 1 \end{cases}$$

  In tale modo si ottiene $\vartheta \in [-1, 1]$.

Le trasformazioni sono necessarie per evitare casi di non convergenza con la prima tecnica di approssimazione e per ridurre le dimensioni della tabella nel caso della seconda tecnica.

4.2.2 Approssimazione polinomiale

Ogni funzione trigonometrica $f(x)$ che si desidera implementare viene sviluppata in serie di Taylor con centro in 0 (detta anche serie di Mac Laurin) per un numero k di termini sufficienti ad ottenere un'accuratezza che uguagli la risoluzione della rappresentazione utilizzata:

$$f(x) \simeq \sum_{i=0}^{k} \frac{f^i(0)x^i}{i!}$$

Imponendo che l'argomento sia sempre $|x| < 1$ si garantisce la rapida convergenza a 0 degli elementi dello sviluppo e quindi l'approssimazione del valore della funzione tramite una sommatoria finita di termini è valida.

I coefficenti $a_i = f^i(0)/i!$ della serie di potenze vengono memorizzati in una tabella. Il calcolo che deve eseguire la routine di approssimazione polinomiale si riduce in una successione di operazioni del tipo multiply & accumulate per trovare il valore della funzione:

$$f(\vartheta) = \sum_{i=0}^{k} a_i\vartheta^i = a_0 + a_1\vartheta + a_2\vartheta^2 + \ldots a_k\vartheta^k = a_0 + \vartheta(a_1 + \vartheta(a_2 + \ldots \vartheta a_k))$$

4.2.3 Approssimazione con look-up table

Le tecniche tabellari sono notevolmente più rapide e semplici, ma necessitano di parecchio spazio in memoria. Supponendo di voler implementare la funzione $f(x)$ si costruisce un vettore unidimensionale formato da diverse coppie ordinate del tipo $< x_k, f(x_k) >$, con $\{x_k\}$ successione monotona strettamente crescente. In questo modo, dato un qualsiasi valore di ϑ, scandendo il vettore si trovano le due coppie $< x_i, f(x_i) >$ e $< x_{i+1}, f(x_{i+1}) >$ tali che $x_i \leq \vartheta \leq x_{i+1}$. Il valore $f(\vartheta)$ cercato si può trovare interpolandolo linearmente (o in modo più raffinato) tra i valori $f(x_i)$ e $f(x_{i+1})$.

Se i punti x_k della successione $\{x_k\}$ sono equidistanti di un valore Δx si può ridurre il vettore di coppie a un semplice vettore di numeri $\{f(k\Delta x)\}$. Dato il valore di ϑ, si calcola velocemente l'indice i del primo elemento da interpolare come:

$$i = \lfloor \frac{\vartheta}{\Delta x} \rfloor$$

Il valore $f(\vartheta)$ cercato si può determinare interpolandolo tra i valori memorizzati agli indici i e $i+1$.

4.3 Operazioni trascendenti

Sono principalmente le funzioni logaritmo ed esponenziale naturali; molte altre funzioni si possono ricondurre a queste facilmente.

Importante: bisogna ricordare che i calcoli che vengono eseguiti nelle routine reali sono parecchio più complessi perché si deve tenere conto della limitata capacità che hanno i formati floating point nell'approssimare i numeri reali, sia in range che in risoluzione.

4.3.1 Logaritmo naturale

Si consideri un numero reale $x > 0$ e lo si rappresenti in forma normalizzata come:

$$x = m2^n$$

Applicando una nota proprietà dei logaritmi si ottiene:

$$\log x = \log(m2^n) = \log m + n\log 2$$

Il valore $\log m$ può essere calcolato usando lo sviluppo in serie di Taylor del logaritmo, infatti si ha $1 \leq m < 2$ e questo è sufficiente ad assicurare la rapida convergenza a 0 degli elementi dello sviluppo:

$$\log m = \log(1 + (m-1)) \simeq \sum_{i=1}^{k} \frac{(-1)^{i+1}(m-1)^i}{i}$$

Logaritmi non naturali sono riconducibili a quelli naturali con degli opportuni coefficenti moltiplicativi di scalatura: per ottenere il logaritmo binario o decimale, il risultato va moltiplicato per $1/\log 2$ o $1/\log 10$, rispettivamente.

4.3.2 Esponenziale

Si consideri un qualsiasi numero reale x. Il primo passo da compiere è ricondurre x ad un altro valore g, piccolo e simmetrico rispetto allo 0. Per fare ciò si può calcolare:

$$N = \frac{x}{\log 2}$$

e sucessivamente si può ricavare:

$$g = x - N\log 2$$

Il numero g è piccolo e non nullo grazie alla precisione finita delle operazioni floating point. Poiché l'accuratezza dell'esponenziale dipende in modo critico

dall'accuratezza di g, nelle implementazioni reali i calcoli che vengono svolti per determinare g sono più complessi. A questo punto è verificata la disuguaglianza:

$$|g| \leq \frac{\log 2}{2} \simeq 0.347\ldots$$

Dunque è assicurata la rapida convergenza a 0 degli elementi dello sviluppo in serie di Taylor di $\exp^g$. Rimane da calcolare:

$$\exp^x = \exp^{N\log 2+g} = \left(\exp^{\log 2}\right)^N \exp^g = 2^N \exp^g$$

4.3.3 Altre trascendenti

Sono principalmente le potenze e le radici. Si applicano, attraverso algoritmi piuttosto raffinati, le formule seguenti per ricondursi a calcoli già visti:

$$x^y = \exp^{y\log x}$$

$$\sqrt[y]{x} = x^{1/y}$$

4.4 Altre funzioni

Si riportano le descrizioni di altre funzioni, non riconducibili a quelle precedenti e spesso indispensabili.

4.4.1 Conversioni tra formati

A volte risulta necessario disporre di funzioni che eseguono la conversione da un formato di rappresentazione all'altro:

- capita di dover elaborare dati floating point provenienti da fonti diverse, con formati diversi;
- a volte si convertono i dati da 32 a 64 bit o in qualche forma a precisione estesa prima di effettuare una lunga serie di calcoli, per limitare la propagazione degli errori;
- alla fine della elaborazione si può voler memorizzare i dati in formato a 32 bit per risparmiare memoria;
- spesso nei linguaggi di programmazione ad alto livello si usano variabili sia a 32 che 64 bit all'interno degli stessi calcoli e il compilatore deve disporre di funzioni di conversione.

4.4.2 Conversione da 32 a 64 bit

Detti s_0, e_0, f_0 i campi del numero di partenza e s_1, e_1, f_1 quelli del numero d'arrivo, l'algoritmo di conversione è:

1. copia f_0 in f_1, allineandolo a sinistra e azzerando i bit meno significativi;
2. copia e_0 in e_1, estendendo il segno;
3. copia s_0 in s_1.

Chiaramente non si perde alcuna informazione durante la conversione poiché si hanno più bit a disposizione.

4.4.3 Conversione da 64 a 32 bit

Detti s_0, e_0, f_0 i campi del numero di partenza e s_1, e_1, f_1 quelli del numero d'arrivo, l'algoritmo di conversione è:

1. copia i bit più significativi di f_0 in f_1, arrotondando f_1 con il metodo di arrotondamento attualmente selezionato;
2. copia i bit meno significativi di e_0 in e_1, controllando il verificarsi di situazioni di underflow o overflow;
3. copia s_0 in s_1.

Poiché il formato d'arrivo è più compatto di quello di partenza possono generarsi queste eccezioni:

- underflow;
- overflow;
- loss of precision.

4.4.4 Normalizzazione di denormalizzati

Qualsiasi numero considerato denormalizzato (cioè per il quale si assume che il bit implicito sia nullo) può essere reso normalizzato, sfruttando le proprietà del formato floating point, con questo algoritmo:

1. identificare la posizione del bit a 1 più significativo di f;
2. scorrere a sinistra f fino a fare uscire il bit dal campo, rendendolo implicito;

3. decrementare *e* di un valore pari al numero di shift eseguiti.

Nella figura 5 viene mostrato un numero considerato denormalizzato e in figura 6 lo stesso numero dopo la normalizzazione. L'operazione può produrre l'underflow di *e*, se il numero denormalizzato è troppo piccolo.

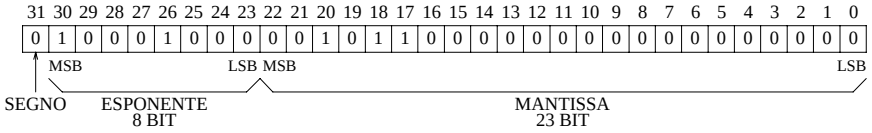

Figura 5: Un numero considerato denormalizzato.

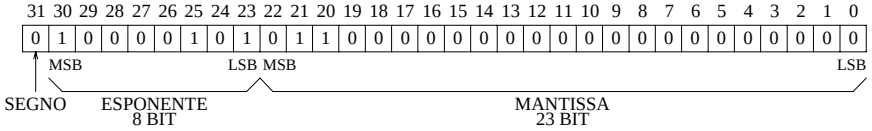

Figura 6: Il numero dopo la normalizzazione.

I DSP della famiglia TMS320C4X possono eseguire la normalizzazione in hardware, usando l'istruzione `NORM`, mentre i DSP della famiglia ADSP-2106X non dispongono di un'istruzioni macchina simile.

4.5 Osservazioni

Si riportano alcune considerazioni conclusive. Un libro eccellente che discute in dettaglio come scrivere le librerie matematiche è il SOFTWARE MANUAL FOR THE ELEMENTARY FUNCTIONS. Molte implementazioni correnti, tra cui quella in esame, fanno uso degli algoritmi e delle tecniche descritti dagli autori.

4.5.1 Dimensione della mantissa temporanea

Durante l'implementazione fisica degli algoritmi (hardware o software) bisogna considerare che la mantissa dei risultati intermedi delle varie operazioni dovrà essere più ampia rispetto alle mantisse proprie degli operandi, per due motivi:

- in modo che essa non vada in underflow o overflow;
- affinché sia possibile applicare le tecniche di arrotondamento previste dallo standard, quali quella di arrotondamento al più vicino e al pari.

4.5.2 Mancanza della divisione

Molti microprocessori general-purpose con unità floating point dispongono di istruzioni dedicate al calcolo di divisioni.

Invece, in genere, i DSP sono privi di istruzioni di divisione perché le divisioni veloci sono operazioni complicate e costose da realizzare, richiedendo parecchio spazio su silicio. Inoltre, il fatto che vengano usate raramente nelle applicazioni reali (in termini di numero di chiamate) concorre a giustificarne l'assenza.

Spesso i DSP supportano indirettamente le divisioni floating point, sfruttando le istruzioni di moltiplicazione e di reciproco: la divisione si realizza moltiplicando il primo operando per una stima del reciproco del secondo operando.

Si è parlato di stima del reciproco perché, di solito, l'operazione di reciproco viene soltanto approssimata. L'esponente del risultato viene calcolato esattamente con un semplice cambiamento di segno dell'esponente dell'operando, mentre solo i bit più significativi della mantissa del risultato sono accurati e vengono determinati leggendoli da una look-up table che ha come ingresso i bit più significativi della mantissa dell'operando.

Se necessario si può ottenere un reciproco molto più preciso applicando l'algoritmo iterativo di *Newton-Raphson*, che vedremo successivamente.

4.5.3 Operazioni in precisione multipla

A volte si desidera operare con numeri floating point il cui formato è più grande di quello gestibile direttamente dall'hardware. Tipicamente si dispone di un dispositivo in grado di effettuare calcoli fixed point e floating point a 32 bit e si vuole estendere le sue capacità di calcolo a numeri a 64 bit. Per fare ciò si scrivono delle routine matematiche, dette a precisione multipla, una per ogni funzione da estendere, che funzionano secondo questo algoritmo:

1. ricevi in ingresso gli operandi a 64 bit;
2. spezza gli operandi in più componenti a 32 bit (o in qualche formato intermedio a precisione estesa, tipicamente a 40 bit);
3. esegui delle operazioni tra le componenti ottenendo dei risultati parziali;
4. ricomponi i risultati parziali per formare il risultato definitivo a 64 bit.

4.5.4 Comportamento con operandi denormalizzati

Non è possibile utilizzare le routine matematiche definite per i numeri normalizzati anche per elaborare operandi denormalizzati in quanto la forma denormalizzata

è in virgola fissa e il bit implicito è necessariamente presente nella mantissa. Infatti, nel caso di una somma dove un operando è denormalizzato la mantissa di quest'ultimo dovrebbe essere shiftata verso destra fino ad uguagliare gli esponenti degli operandi (eseguendo un *allineamento*); in questo processo alla mantissa dell'operando verrebbe aggiunto anche il bit implicito dei normalizzati, alterandone di conseguenza il valore e provocando quindi l'inesattezza del risultato dell'intera operazione.

Considerazioni simili valgono anche per le altre operazioni fondamentali: per elaborare correttamente i denormalizzati sono dunque necessarie routine apposite oppure bisogna inserire ulteriori controlli e diramazioni nelle routine generali.

5 La famiglia ADSP-2106X

Questa sezione descrive brevemente la famiglia di DSP che costituisce l'implementazione commerciale analizzata in questo elaborato. La documentazione di riferimento, sfruttata anche nelle sezioni successive, è costituita dai manuali e dai file sorgente che accompagnano il kit di valutazione EZ-KIT LITE prodotto dalla Analog Devices per il processore ADSP-21061.

5.1 Generalità

Prodotta dalla Analog Devices, dispone di queste caratteristiche:

- unità di calcolo veloci e flessibili;
- flusso dei dati da e per le unità di calcolo rapido;
- unità di calcolo con precisione e range dinamico elevati;
- doppio generatore d'indirizzi;
- esecuzione efficiente del codice e dei salti;
- supporto dei linguaggi ad alto livello.

Di seguito vengono analizzati velocemente i vari punti.

5.1.1 Unità di calcolo

I DSP della famiglia sono in grado di eseguire tutte le istruzioni in un singolo ciclo di clock, grazie ad una pipeline a tre stadi (fetch, decode e execute). Dispongono di un vasto set di operazioni aritmetiche: oltre alle operazioni tradizionali di addizione, sottrazione e moltiplicazione dispongono di addizione/sottrazione combinata, approssimazione di $1/x$, approssimazione di $1/\sqrt{x}$, *min*, *max*, *clip*, *shift* e *rotate*. Gestiscono numeri fixed point a 32 bit, floating point single precision a 32 bit e floating point a precisione estesa a 40 bit, tutti secondo lo standard IEEE. Supportano le eccezioni, che possono venire segnalate tramite interrupt software oppure per mezzo di specifici bit dei registri di stato. Purtroppo, come vedremo, non tutto lo standard IEEE è implementato.

5.1.2 Flusso dei dati

La famiglia è caratterizzata da una architettura di tipo Harvard, possiede un blocco registri composto da 16 registri di uso generale a 40 bit per contenere numeri fixed point o floating point, indifferentemente. Esiste un altro blocco di 16 registri secondari, usati durante i context switch. Ogni blocco di registri dispone di 10 porte di accesso che consentono, ad ogni ciclo di clock, l'esecuzione di tutte queste operazioni contemporaneamente:

- il blocco di registri può leggere o scrivere due operandi;
- la ALU può ricevere due operandi;
- il moltiplicatore può ricevere due operandi;
- la ALU e il moltiplicatore possono elaborare due risultati (tre se la ALU esegue una addizione/sottrazione combinata).

Il formato di ogni istruzione permette di svolgere più trasferimenti dati e operazioni aritmetiche in modo parallelo nella stessa istruzione. Ad esempio, in una singola istruzione si può codificare un prodotto, una addizione, una sottrazione ed un salto condizionati.

5.1.3 Precisione e range dinamico

Durante le operazioni floating point i processori eseguono calcoli a precisione estesa a 40 bit, per limitare l'errore di troncamento. I calcoli fixed point a 32 bit producono risultati memorizzati in un accumulatore a 80 bit, in modo da poter eseguire calcoli a 32 bit reali e ridurre il rumore dovuto agli arrotondamenti quando vengono eseguite sequenze di calcoli del tipo multiply & accumulate.

5.1.4 Doppio generatore d'indirizzi

Le istruzioni macchina generali accettano come operandi registri, valori numerici immediati e indirizzamenti in memoria diretti.

L'indirizzamento indiretto viene invece effettuato mediante l'uso di speciali registri puntatori presenti in due *DAG* (data address generator). Su ognuno di questi registri è possibile eseguire una pre-modifica senza aggiornamento del registro oppure una post-modifica con aggiornamento del registro. Tramite i DAG è anche possibile implementare in hardware un'indirizzamento circolare o bit-reversed, senza vincoli sulla posizione o sulla dimensione dell'area di memoria che viene indirizzata. Ogni DAG ha un DAG secondario che viene utilizzato durante i context switch.

5.1.5 Esecuzione del codice

I DSP supportano salti ridardati e non ritardati, condizionati e non condizionati. Sono disponibili istruzioni specifiche per gestire i cicli ed il loro uso non comporta alcun over-head nel tempo d'esecuzione; inoltre l'ingresso e l'uscita dai cicli è rapida (impiega un solo ciclo di clock). I cicli sono nidificabili (fino a 6 livelli) e interrompibili.

5.1.6 Linguaggi ad alto livello

L'architettura della famiglia ha diverse caratteristiche utili per facilitare l'implementazione di compilatori per linguaggi ad alto livello e sistemi operativi:

- registri dati general-purpose;
- dati a 32 bit nativi;
- ampio spazio d'indirizzamento;
- indirizzamento con eventuale pre o post modifica;
- gestione senza vincoli dei buffer circolari;
- stack per il programma, i loop e gli interrupt integrati nel chip.

In particolare l'architettura è progettata per poter implementare facilmente le *Numerical C extensions*, frutto del lavoro del comitato ANSI NCEG (*Numerical C Extensions Group*), facente capo alla commissione ANSI X3J11, che ha definito e mantiene lo standard del linguaggio C. Si tratta di alcune estensioni al linguaggio standard e di un set di routine richiamabili da linguaggio C per gestire i vettori e le matrici di dati impiegati per il calcolo numerico e l'elaborazione dei segnali.

5.2 Programmi a corredo

Nel kit vengono forniti questi programmi:

- compilatore **g21k**;
- assemblatore **asm21k**;
- linker **ld21k**;
- librarian **lib21k**;
- simulatore **wsim_ez**;

- loader **ldr21k**;
- utility **dh21k**.

5.2.1 Il compilatore g21k

Si tratta di un port del gcc, *GNU C compiler*; comprende le Numerical C extensions per la manipolazione di array, matrici e tipi di dati complessi. Ulteriori estensioni al linguaggio permettono la gestione di puntatori circolari e array a dimensione dinamica. Viene fornita una *C Runtime Library* che include funzioni DSP specifiche e particolarmente ottimizzate per questa famiglia di dispositivi.

Il compilatore implementa via software un *runtime stack* per variabili locali (identificate dalla keyword opzionale *auto* del C), parametri di funzioni, dati temporanei e indirizzi di ritorno da subfunction, in modo da garantire un livello di nidificazione arbitrario. Infatti il DSP mette a disposizione, a livello hardware, uno stack in grado di memorizzare solo indirizzi di ritorno e che ha una profondità di soli 30 elementi. Per maggiori informazioni a riguardo consultare lo USER'S MANUAL.

Il compilatore presenta anche alcune restrizioni sul suo funzionamento, che sono state raccolte alle pagine 2-15, 2-16, 2-17 e 2-18 del C TOOLS MANUAL. Alcuni parametri importanti che possono essere passati al compilatore sono elencati di seguito:

- **-O** e **-O2** servono per attivare l'ottimizzazione del codice da parte del compilatore. Il compilatore effettuerà il riordino e compattamento delle istruzioni assembly al fine di ridurre il tempo d'esecuzione e lo spazio di memoria occupato dal programma. **-O** attiva l'ottimizzazione meno approfondita; **-O2** richiede l'ottimizzazione massima, a scapito del tempo di compilazione.
- **-S** ferma il compilatore dopo la fase di compilazione, senza assemblare i file compilati; in questo caso l'output è costituito dai file assembly ottenuti dalla traduzione dei sorgenti C.
- **-g** include nel programma eseguibile delle *debug informations*, per rendere possibile il debugging del programma *a livello di codice C*. Questa opzione è incompatibile con lo switch **-O2**.

5.2.2 Il simulatore

Riceve in ingresso i file eseguibili ottenuti dalla fase di compilazione e linking con le liberie e ne simula l'esecuzione all'interno del DSP in un gradevole ambiente grafico. Possiede queste caratteristiche:

- simulazione dell'esecuzione delle istruzioni del programma a livello di singoli cicli di clock, in modo single step o multi step;
- riconoscimento di operazioni illegali, come accessi in aree di memoria inesistenti;
- supporto di breakpoint multipli;
- simulazione dei dispositivi hardware di I/O come porte seriali, canali DMA e di link, con possibilità di utilizzare file di stimolo;
- simulazione degli interrupt interni ed esterni;
- disassemblatore con possibilità di assemblare istruzioni on line;
- visualizzazione e possibilità di modifica dei registri e delle locazioni di memoria;
- possibilità di visualizzare in modo avanzato il contenuto della memoria.

Attraverso CBUG, un modulo interno al simulatore e derivato dal GNU debugger, è possibile eseguire il debugging a livello di codice C, ma solo se i sorgenti del programma sono stati compilati con l'opzione **-g**.

5.2.3 L'utility dh21k

Tramite il programma **DspHost** è possibile accedere dal DSP alle risorse del PC cui è collegato tramite l'interfaccia seriale. E' possibile leggere la tastiera, scrivere sullo schermo e trasferire file da e su disco. Per comunicare, il programma che gira sul DSP chiama le funzioni dichiarate in **conio.h**, **direct.h**, **fcntl.h**, **io.h**, **process.h**, **stdio.h**, **sys/stat.h** e **sys/types.h**, che mimano ed estendono quelle di I/O definite dallo standard C. Per altre informazioni consultare il capitolo 7 del REFERENCE MANUAL, DSPHOST REFERENCE.

5.3 Limitazioni del sistema EZ-KIT LITE

5.3.1 Compilatore C

Gli switch del compilatore **-O**, **-ansi**, **-fno-builtin**, **-fno-short-double**, **-pedantic** e **-wfloat-convert** descritti in questo elaborato non vengono supportati dalla versione del compilatore distribuita con lo EZ-KIT LITE.

5.3.2 Architecture file

Ogni sistema basato su questa famiglia di DSP può avere una propria configurazione hardware per quanto riguarda la posizione dei blocchi di memoria, le loro ampiezze e le rispettive dimensioni.

Per caratterizzare meglio ciascun sistema è necessario scrivere, per ogni tipo di sistema, un *architecture file* che specifica come sono allocate le sue risorse di memoria e gli indirizzi delle porte di I/O mappate in memoria. In questo modo il compilatore, l'assemblatore e il linker possono operare efficacemente su un ampio range di configurazioni diverse e il programmatore può sfruttare riferimenti simbolici invece che soli indirizzi fisici.

Purtroppo il kit di valutazione EZ-KIT LITE ha un architecture file predefinito che non si può cambiare, chiamato **ezkit.ach**, il cui contenuto è mostrato di seguito. Per altre informazioni al riguardo consultare ASSEMBLER TOOLS & SIMULATOR MANUAL.

```
!-------------------------------------------------------------------
.SYSTEM         SHARC_EZKIT_Lite;
!
! This architecture file is required for used with the SHARC EZ-KIT
! Lite development software. It is structured for use with the C
! compiler but also can be used with assembly code.
!
! When compiling with g21k the "-nomem" switch should be used to
! disable unnecessary runtime initialization.
!
! This architecture file allocates:
! Internal 133 words of 48-bit run-time header in memory block 0
!           16 words of 48-bit initialization code in memory block 0
!          619 words of 48-bit kernel code in memory block 0
!         7424 words of 48-bit C code space in memory block 0
!           4K words of 32-bit PM C data space in memory block 0
!
!           8K words of 32-bit C DM data space in memory block 1
!           4K words of 32-bit C heap space in memory block 1
!         3712 words of 32-bit C stack space in memory block 1
!          384 words of 32-bit kernel data in memory block 1

.PROCESSOR = ADSP21061;

!   ------------------------------------------------------------
!   Internal memory Block 0
!   ------------------------------------------------------------
```

```
.SEGMENT /RAM/BEGIN=0x20000/END=0x20084/PM/WIDTH=48        seg_rth;
.SEGMENT /RAM/BEGIN=0x20085/END=0x20094/PM/WIDTH=48        seg_init;
.SEGMENT /RAM/BEGIN=0x20095/END=0x202ff/PM/WIDTH=48        seg_knlc;
.SEGMENT /RAM/BEGIN=0x20300/END=0x21fff/PM/WIDTH=48        seg_pmco;
.SEGMENT /RAM/BEGIN=0x23000/END=0x23fff/PM/WIDTH=32        seg_pmda;

!   ------------------------------------------------------------
!   Internal memory Block 1
!   ------------------------------------------------------------
.SEGMENT /RAM/BEGIN=0x24000/END=0x25fff/DM/WIDTH=32        seg_dmda;
.SEGMENT /RAM/BEGIN=0x26000/END=0x26fff/DM/WIDTH=32 /cheap seg_heap;
.SEGMENT /RAM/BEGIN=0x27000/END=0x27e7f/DM/WIDTH=32        seg_stak;
.SEGMENT /RAM/BEGIN=0x27e80/END=0x27fff/DM/WIDTH=32        seg_knld;

.ENDSYS;
```

6 Il supporto IEEE

Sebbene i dispositivi siano in grado di elaborare numeri secondo i formati definiti dallo standard, alcune loro caratteristiche funzionali non li rendono perfettamente conformi.

6.1 Generazione delle eccezioni

Le uniche eccezioni che vengono gestite sono overflow, underflow e invalid operation. In particolare:

- non viene implementato alcun metodo (eccezione o flag) per segnalare che il risultato di un'operazione è inesatto (loss of precision);
- operazioni con operandi del tipo NaN producono un'eccezione invalid operation (se abilitata) e ritornano sempre un risultato del tipo NaN;
- numeri denormalizzati utilizzati come operandi vengono arrotondati a 0 prima dell'esecuzione dell'operazione, senza che venga generata alcuna underflow exception;
- qualsiasi risultato di un'operazione aritmetica che appare denormalizzato o in underflow viene arrotondato a 0 e viene prodotta una underflow exception.

L'eccezione invalid operation viene generata quando:

- uno degli operandi è NaN;
- si cerca di sommare infinità con segni diversi;
- si cerca di sottrarre infinità con segni uguali;
- durante una conversione da floating point a fixed point, quando il risultato genera overflow o infinità e non è attivo il *saturation mode* (bit ALUSAT del registro MODE1), che in questo caso arrotonderebbe il numero al massimo fixed point rappresentabile (0x7FFFFFFF oppure 0x80000000, in funzione del segno), senza generare eccezioni.

Ciascuna eccezione produce un diverso interrupt mascherabile, che può essere disattivato settando il relativo bit nel registro IMASK (*interrupt mask register*):

- il bit 25 (FLTOI, *floating point overflow interrupt*) controlla la generazione dell'overflow interrupt;

- il bit 26 (FLTUI, *floating point underflow interrupt*) abilita la produzione dell'underflow interrupt;
- il bit 27 (FLTII, *floating point invalid interrupt*) governa la generazione dell'invalid operation interrupt.

6.2 Gestione delle eccezioni

Ogni volta che si verifica un'eccezione viene dapprima attivato il relativo bit di segnalazione nei registri ASTAT (*arithmetic status register*) e STKY (*sticky status register*) e successivamente viene generato un interrupt mascherabile.

Questi dispositivi mettono a disposizione tre modi per gestire le eccezioni:

- Tramite interrupt: l'eccezione viene gestita immediatamente in una routine apposita. E' il metodo da preferire se è importante correggere tutte le eccezioni appena accadono.
- Tramite il registro ASTAT: i flag che riportano lo stato delle eccezioni vengono modificati automaticamente quando l'operazione è stata eseguita e vanno testati subito dopo. E' è il metodo da usare per monitorare l'esecuzione di specifiche operazioni.
- Tramite il registro STKY: i flag di questo registro vengono settati automaticamente dopo l'esecuzione delle operazioni ma non vengono mai cancellati. Ci si può limitare a verificare i bit di questo registro dopo una serie di operazioni, se la gestione delle eccezioni non è critica. Prima di iniziare la sequenza di operazioni i bit devono essere azzerati esplicitamente.

E' importante sottolineare che, a causa della pipeline a tre stadi, le routine di gestione delle eccezioni vengono mandate in esecuzione esattamente due istruzioni dopo che l'eccezione è avvenuta. Le due istruzioni nel mezzo vengono eseguite comunque.

6.3 Modi di arrotondamento

I modi di arrotondamento supportati sono solo l'arrotondamento verso lo 0 e l'arrotondamento al più vicino e al pari. Gli arrotondamenti verso $-\infty$ e verso $+\infty$ non sono stati implementati. Il modo attivo viene controllato dal bit TRUNC (*truncate bit*) del registro MODE1:

- se 0 viene selezionato il modo arrotondamento al più vicino e al pari;
- se 1 viene selezionato il modo arrotondamento verso lo 0.

Osservazione: i massimi numeri rappresentabili distano entrambi solo 1 LSB dai valori che rappresentano $-\infty$ e $+\infty$, quindi se nel caso di arrotondamento al più vicino e al pari viene generato un risultato che è nel mezzo di uno di questi gap, il valore verrà arrotondato rispettivamente a $-\infty$ oppure a $+\infty$.

6.4 Modi single precision e extended precision

I numeri floating point possono essere interpretati, calcolati e forniti come risultato sia in single precision (32 bit) che in extended precision (40 bit). Il modo attivo viene controllato dal bit RND32 (*round to 32 bit*) del registro MODE1:

- se 0 viene selezionato il modo extended precision;
- se 1 viene selezionato il modo single precision.

Nel modo single precision gli 8 bit meno significativi degli operandi di ogni istruzione floating point vengono forzati a zero prima di eseguire l'operazione, troncando le rispettive mantisse a 23 bit (non includendo il bit implicito). La mantissa del risultato dell'operazione viene arrotondata a 23 bit e i bit meno significativi vengono posti a zero prima di porre il risultato nell'operando di destinazione. Nel modo extended precision gli operandi e il risultato vengono mantenuti tutti a 40 bit.

7 Le istruzioni floating point

In questa sezione vengono analizzate le istruzioni macchina per la manipolazione dei floating point messe a disposizione dalla famiglia di DSP in esame. Prima di procedere si rammenta che:

- tutte le istruzioni vengono eseguite in un singolo ciclo di clock;
- i flag AZ, AU, AN, AV, AC, AS e AI sono generati dalla ALU;
- i flag MN, MV, MU e MI sono generati dal moltiplicatore;
- i flag SV, SZ e SS sono generati dallo shifter;
- tutti i flag precedenti sono contenuti nel registro ASTAT;
- i flag generati dalla ALU e dal moltiplicatore sono disponibili anche in versione sticky nel registro STKY;
- la dimensione degli operandi viene controllata dal bit RND32 del registro MODE1;
- il tipo di arrotondamento viene controllato dal bit TRUNC del registro MODE1.

7.1 Operazioni matematiche

7.1.1 Fn=Fx+Fy

Somma gli operandi nei registri Fx e Fy e pone il risultato normalizzato nel registro Fn. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;
- nel caso si verifichi un overflow dopo l'arrotondamento al termine dell'operazione, il risultato viene posto al minimo normalizzato negativo o al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $\pm\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato oppure 0, altrimenti è 0;
- AU è 1 se il risultato è denormalizzato, altrimenti è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando almeno uno degli operandi è *NaN* oppure quando gli operandi sono infinità con segni opposti, altrimenti è 0.

7.1.2 Fn=Fx-Fy

Sottrae gli operandi nei registri Fx e Fy e pone il risultato normalizzato nel registro Fn. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;
- nel caso si verifichi un overflow dopo l'arrotondamento al termine dell'operazione, il risultato viene posto al minimo normalizzato negativo o al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $\pm\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato oppure 0, altrimenti è 0;
- AU è 1 se il risultato è denormalizzato, altrimenti è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando almeno uno degli operandi è *NaN* oppure quando gli operandi sono infinità con segni uguali, altrimenti è 0.

7.1.3 Fn=ABS(Fx+Fy)

Calcola il valore assoluto della somma degli operandi nei registri Fx e Fy e pone il risultato normalizzato nel registro Fn. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;
- nel caso si verifichi un overflow dopo l'arrotondamento al termine dell'operazione, il risultato viene posto al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $+\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato oppure 0, altrimenti è 0;
- AU è 1 se il risultato è denormalizzato, altrimenti è 0;
- AN è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando almeno uno degli operandi è *NaN* oppure quando gli operandi sono infinità con segni opposti, altrimenti è 0.

7.1.4 Fn=ABS(Fx-Fy)

Calcola il valore assoluto della differenza degli operandi nei registri Fx e Fy e pone il risultato normalizzato nel registro Fn. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;

- nel caso si verifichi un overflow dopo l'arrotondamento al termine dell'operazione, il risultato viene posto al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $+\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato oppure 0, altrimenti è 0;
- AU è 1 se il risultato è denormalizzato, altrimenti è 0;
- AN è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando almeno uno degli operandi è *NaN* oppure quando gli operandi sono infinità con segni uguali, altrimenti è 0.

7.1.5 Fn=(Fx+Fy)/2

Somma gli operandi nei registri Fx e Fy, divide per 2 e pone il risultato normalizzato nel registro Fn. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;
- nel caso si verifichi un overflow dopo l'arrotondamento al termine dell'operazione, il risultato viene posto al minimo normalizzato negativo o al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $\pm\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato oppure 0, altrimenti è 0;
- AU è 1 se il risultato è denormalizzato, altrimenti è 0;

- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando almeno uno degli operandi è *NaN* oppure quando gli operandi sono infinità con segni opposti, altrimenti è 0.

7.1.6 Fn=MIN(Fx,Fy)

Pone nel registro Fn il minore tra gli operandi nei registri Fx e Fy. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- l'operazione MIN$(+0,-0)$ ritorna -0.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 se almeno uno degli operandi è *NaN*, altrimenti è 0.

7.1.7 Fn=MAX(Fx,Fy)

Pone nel registro Fn il maggiore tra gli operandi nei registri Fx e Fy. Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- l'operazione MAX($+0$,-0) ritorna $+0$.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 se almeno uno degli operandi è *NaN*, altrimenti è 0.

7.1.8 Fn=CLIP Fx BY Fy

Il valore di ritorno è:

$$Fn = \begin{cases} Fx & \text{se } |Fx| < |Fy| \\ |Fy| & \text{se } |Fx| \geq |Fy| \text{ e } Fx > 0 \\ -|Fy| & \text{se } |Fx| \geq |Fy| \text{ e } Fx < 0 \end{cases}$$

Caratteristiche:

- operandi di tipo *NaN* producono un risultato *NaN*, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;

- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 se almeno uno degli operandi è NaN, altrimenti è 0.

7.1.9 Fn=Fx*Fy

Moltiplica gli operandi nei registri Fx e Fy, ponendo il risultato nel registro Fn. Caratteristiche:

- operandi di tipo NaN producono un risultato NaN, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;
- nel caso si verifichi un overflow dopo l'arrotondamento al termine dell'operazione, il risultato viene posto al minimo normalizzato negativo o al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $\pm\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- MN è 1 se il risultato è negativo, altrimenti è 0;
- MV è 1 se il risultato è in overflow, altrimenti è 0;
- MU è 1 se il risultato è in underflow, altrimenti è 0;
- MI è 1 quando almeno uno degli operandi è NaN oppure quando un operando è $\pm\infty$ e l'altro è ± 0, altrimenti è 0.

7.1.10 Fn=-Fx

Cambia il segno dell'operando nel registro Fx e mette il risultato nel registro Fn. Caratteristiche:

- un operando di tipo NaN produce un risultato NaN, con tutti i bit forzati a 1;
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 se l'operando è NaN, altrimenti è 0.

7.1.11 Fn=ABS Fx

Trova il valore assoluto dell'operando nel registro Fx, azzerando il bit di segno, e pone il risultato nel registro Fn. Caratteristiche:

- un operando di tipo NaN produce un risultato NaN, con tutti i bit forzati a 1;
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 0;

- AV è 0;
- AC è 0;
- AS è 1 se l'operando è negativo, altrimenti è 0;
- AI è 1 se l'operando è NaN, altrimenti è 0.

7.1.12 Fn=PASS Fx

Copia il contenuto dell'operando nel registro Fx nel registro Fn. Caratteristiche:

- un operando di tipo NaN produce un risultato NaN, con tutti i bit forzati a 1;
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 se l'operando è NaN, altrimenti è 0.

7.1.13 Fn=Fx COPYSIGN Fy

Prende l'operando Fx, ne altera il segno per renderlo concorde a quello dell'operando Fy e scrive il risultato in Fn. Fx e Fy non vengono alterati. Caratteristiche:

- operandi di tipo NaN producono un risultato NaN, con tutti i bit forzati a 1;
- operandi denormalizzati vengono arrotondati a ± 0 prima di eseguire l'operazione.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 se almeno un operando è NaN, altrimenti è 0.

7.2 Istruzioni di conversione

7.2.1 Fn=RND Fx

Arrotonda l'operando extended precision nel registro Fx verso la single precision secondo il modo di arrotondamento corrente e pone il risultato a 32 bit nel registro Fn, allineandolo a sinistra. Gli 8 bit meno significativi vengono posti a 0. Caratteristiche:

- un operando di tipo NaN produce un risultato NaN, con tutti i bit forzati a 1;
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione;
- nel caso si verifichi un overflow dopo l'arrotondamento, il risultato viene posto al minimo normalizzato negativo o al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $\pm\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 se l'operando è NaN, altrimenti è 0.

7.2.2 Rn=TRUNC Fx, Rn=TRUNC Fx BY Ry

Converte l'operando nel registro Fx in un fixed point a 32 bit con segno usando l'arrotondamento verso lo 0 (indipendentemente dallo stato del bit TRUNC del registro MODE1) e pone il risultato in Rn. Se il fattore di scala Ry viene specificato, il suo valore viene aggiunto all'esponente di Fx prima di eseguire la conversione. Caratteristiche:

- un operando di tipo *NaN* produce un risultato floating point *NaN*, con tutti i bit forzati a 1;
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione;
- se il saturation mode è attivo, overflow positivi e $+\infty$ ritornano il massimo intero positivo (0x7FFFFFFF) mentre overflow negativi e $-\infty$ ritornano il minimo intero negativo (0x80000000);
- se il saturation mode è disattivato, un operando $\pm\infty$ o un risultato in overflow provocano il ritorno di un risultato con tutti i bit forzati a 1;
- un risultato in underflow viene arrotondato a 0.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 1 se il risultato prima di essere approssimato è denormalizzato, altrimenti è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se la conversione ha fatto scorrere la mantissa a sinistra oppure se l'operando era $\pm\infty$, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando l'operando è *NaN* oppure quando il saturation mode è disattivato e l'operando è $\pm\infty$ o il risultato è in overflow, altrimenti è 0.

7.2.3 Rn=FIX Fx, Rn=FIX Fx BY Ry

Converte l'operando nel registro Fx in un fixed point a 32 bit con segno usando il modo di arrotondamento corrente e pone il risultato in Rn. Se il fattore di scala Ry viene specificato, il suo valore viene aggiunto all'esponente di Fx prima di eseguire la conversione. Caratteristiche:

- un operando di tipo *NaN* produce un risultato floating point *NaN*, con tutti i bit forzati a 1;
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione;
- se il saturation mode è attivo, overflow positivi e $+\infty$ ritornano il massimo intero positivo (0x7FFFFFFF) mentre overflow negativi e $-\infty$ ritornano il minimo intero negativo (0x80000000);
- se il saturation mode è disattivato, un operando $\pm\infty$ o un risultato in overflow provocano il ritorno di un risultato con tutti i bit forzati a 1;
- un risultato in underflow positivo viene arrotondato a 0;
- un risultato in underflow negativo viene arrotondato a 0 se l'arrotondamento è al più vicino e al pari, oppure al fixed point -1 se l'arrotondamento è verso lo 0;
- se il bit TRUNC del registro MODE1 è 1, l'istruzione tronca la mantissa verso $-\infty$;
- se il bit TRUNC è 0, l'operazione arrotonda il numero all'intero più vicino.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 1 se il risultato prima di essere approssimato è denormale, altrimenti è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se la conversione ha fatto scorrere la mantissa a sinistra oppure se l'operando era $\pm\infty$, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando l'operando è *NaN* oppure quando il saturation mode è disattivato e l'operando è $\pm\infty$ o il risultato è in overflow, altrimenti è 0.

7.2.4 Fn=FLOAT Rx, Fn=FLOAT Rx BY Ry

Converte l'operando fixed point nel registro Rx in un floating point e lo pone nel registro risultato Fn. Se il fattore di scala Ry viene specificato, il suo valore viene aggiunto all'esponente di Fn, dopo aver eseguito la conversione. Caratteristiche:

- l'arrotondamento è al più vicino e al pari oppure verso 0, secondo il valore del bit TRUNC;
- il risultato è in precisione estesa, indipendentemente dal valore del bit di controllo RND32;
- l'eventuale scalatura dell'esponente può causare underflow o overflow del risultato; inoltre:
  - in situazione di underflow viene ritornato ± 0;
  - in situazione di overflow viene ritornato $\pm\infty$ nel caso di arrotondamento al più vicino e al pari, oppure il minimo normalizzato negativo o il massimo normalizzato positivo nel caso di arrotondamento verso lo 0.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato o 0, altrimenti è 0;
- AU è 1 se il risultato dopo l'arrotondamento è denormalizzato, altrimenti è 0;
- AN è 1 se il risultato è negativo;
- AV è 1 se il risultato è in overflow;
- AC è 0;
- AS è 0;
- AI è 0.

7.2.5 Rn=FPACK Fx

Converte l'operando floating point single precision nel registro Fx in uno short floating point a 16 bit e lo pone nel registro risultato Rn. Il formato di destinazione è mostrato nella figura 7 e si compone di una mantissa di 11 bit, di un'esponente di 4 bit e del bit di segno. Il valore viene ritornato nei 16 bit meno significativi del registro Rn.

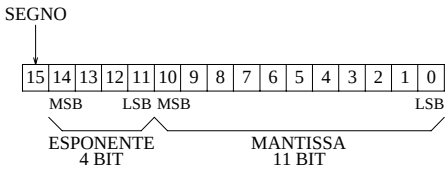

Figura 7: Formato short floating point a 16 bit.

Condizione	Risultato
$exp \leq 110$	il numero viene arrotondato a 0
$110 < exp \leq 120$	l'esponente del risultato viene forzato a 0; la sua mantissa è formata facendo scorrere opportunamente verso destra la mantissa di Fx, bit implicito compreso
$120 < exp \leq 135$	l'esponente del risultato è formato concatenando il MSB e i tre LSB di *exp*, mentre la sua mantissa è ottenuta dagli 11 MSB della mantissa di Fx
$exp > 135$	il numero viene arrotondato al minimo negativo o al massimo positivo rappresentabili, secondo il segno di Fx

Tabella 8: Funzionamento dell'istruzione FPACK.

Importante: il short floating point è l'unico formato in cui i numeri denormalizzati vengono gestiti correttamente dal DSP. Quando è necessario convertire un numero che una volta convertito darebbe un risultato in underflow, l'esponente viene posto a 0 e la mantissa (bit implicito compreso) viene fatta scorrere a destra opportunamente, ottenendo un risultato che è denormalizzato. Questo metodo consente di avere un underflow graduale e aumenta il range dei numeri rappresentabili, a scapito della ridotta precisione consentita dai numeri denormalizzati.

Il funzionamento dell'istruzione viene riassunto nella tabella 8, dove *exp* è il valore dell'esponente unbiased.
Flag alterati:

- SZ è 0;
- SV è 1 in caso di overflow, altrimenti è 0;
- SS è 0.

Condizione	Risultato
$exp = 0$	l'esponente del risultato è $120 - n$, dove n è il numero di zeri più significativi nella mantissa di Rn; la mantissa è ottenuta da quella di Rn facendola scorrere a sinistra finché il primo 1 significativo diventa implicito
$0 < exp \leq 15$	l'esponente del risultato viene ottenuto dai 3 LSB dell'esponente di Rn, preceduti dal suo MSB e da 4 copie del complemento del MSB; la mantissa del risultato è la mantissa di Rn allineata a sinistra con 12 zeri di seguito

Tabella 9: Funzionamento dell'istruzione FUNPACK.

7.2.6 Fx=UNPACK Rn

Converte il numero short floating point nel registro Rn in un floating point single precision e lo pone nel registro risultato Fx. Il funzionamento dell'istruzione viene riassunto nella tabella 9, dove *exp* è il valore dell'esponente unbiased.
Flag alterati:

- SZ è 0;
- SV è 0;
- SS è 0.

7.3 Istruzioni seminumeriche

Gestiscono i floating point a basso livello, operando sui campi esponente e mantissa.

7.3.1 Fn=SCALB Fx BY Ry

Scala l'esponente dell'operando nel registro Fx mediante l'addizione ad esso del numero fixed point con segno contenuto nel registro Ry. Pone il risultato nel registro Fn. Caratteristiche:

- un operando di tipo *NaN* produce un risultato *NaN*, con tutti i bit forzati a 1;

- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione;
- un risultato denormalizzato viene arrotondato a ± 0;
- nel caso si verifichi un overflow al termine dell'operazione, il risultato viene posto al minimo normalizzato negativo o al massimo normalizzato positivo se l'arrotondamento è verso lo 0, oppure a $\pm\infty$ se l'arrotondamento è al più vicino e al pari.

Flag alterati:

- AZ è 1 se il risultato è denormalizzato oppure 0, altrimenti è 0;
- AU è 1 se il risultato è denormalizzato, altrimenti è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 se il risultato è in overflow, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 se l'operando è NaN, altrimenti è 0.

7.3.2 Rn=MANT Fx

Estrae la mantissa dell'operando nel registro Fx, includendo il bit implicito e senza il bit di segno, ponendola nel registro Rn. Poiché il risultato è esattamente di 32 bit non è necessario alcun tipo di arrotondamento. Caratteristiche:

- un operando di tipo NaN o $\pm\infty$ produce un risultato fixed point con tutti i bit forzati a 1 (cioè -1);
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 0;

- AV è 0;
- AC è 0;
- AS è 1 se l'operando è negativo, altrimenti è 0;
- AI è 1 quando l'operando è *NaN* oppure quando è $\pm\infty$, altrimenti è 0.

7.3.3 Rn=LOGB Fx

Converte l'esponente dell'operando nel registro Fx in un fixed point senza bias (sottraendogli 127) e lo pone nel registro Rn. Caratteristiche:

- un operando di tipo *NaN* produce un risultato fixed point con tutti i bit forzati a 1 (cioè -1);
- un operando denormalizzato viene arrotondato a ± 0 prima di eseguire l'operazione;
- se il saturation mode è disabilitato, gli operandi $\pm\infty$ e ± 0 ritornano $+\infty$ e $-\infty$ in formato floating point, rispettivamente;
- se il saturation mode è abilitato, un operando $\pm\infty$ ritorna il massimo intero positivo (0x7FFFFFFF), mentre un operando ± 0 ritorna il minimo intero negativo (0x80000000).

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se il risultato è negativo, altrimenti è 0;
- AV è 1 quando l'operando è $\pm\infty$ oppure 0, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 se l'operando è *NaN*, altrimenti è 0.

7.4 Istruzioni di confronto

7.4.1 COMP(Fx,Fy)

Confronta gli operandi nei registri Fx e Fy. Flag alterati:

- AZ è 1 se Fx = Fy, altrimenti è 0;
- AU è 0;
- AN è 1 se Fx < Fy, altrimenti è 0;
- AV è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando almeno uno degli operandi è *NaN*, altrimenti è 0.

Il registro ASTAT memorizza i risultati degli ultimi 8 confronti nei suoi bit $[31, 24]$, detti CACC (*compare accumulation bits*). Ad ogni confronto il blocco di bit viene shiftato verso destra, sovrascrivendo il bit 24, che viene perso. Il bit 31 di ASTAT viene attivato se e solo se Fx > Fy.

7.5 Operazioni matematiche approssimate

Il DSP dispone di due istruzioni macchina per calcolare in modo approssimato i valori delle funzioni $1/c$ e $1/\sqrt{c}$, con c numero floating point. Il funzionamento di ciascuna istruzione si riassume in due passi:

- l'esponente del risultato viene calcolato esattamente attraverso delle semplici manipolazioni dell'esponente dell'operando;
- la mantissa del risultato viene approssimata leggendo i suoi bit più significativi da una una look-up table indirizzata da una combinazione dei bit meno significativi dell'esponente e di quelli più significativi della mantissa dell'operando.

Il risultato di queste istruzioni è detto *seed* perché, se necessario, può essere usato come valore iniziale dell'algoritmo di *Newton-Raphson*, che consente di aumentare l'accuratezza della mantissa del risultato.

7.5.1 L'algoritmo di Newton-Raphson

Si tratta di un algoritmo iterativo convergente, nel quale il risultato di un ciclo diventa il parametro del ciclo successivo; la proprietà di convergenza assicura che si giunga ad una approssimazione migliore di quella ottenuta dall'iterazione precedente.

L'idea fondamentale è quella di scrivere la funzione che si desidera calcolare, che chiamiamo $x = g(c)$, nei termini di un'altra funzione $f(x)$ tale che, se $\bar{x}$ è il risultato numerico esatto, si abbia $f(\bar{x}) = 0$. L'algoritmo trova, per iterazioni successive, un'approssimazione via via migliore di $\bar{x}$, punto in cui $f(x)$ interseca l'asse orizzontale, attraverso la formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Procedendo iterativamente in questo modo si costruisce una successione $\{x_n\}$ di numeri convergente a $\bar{x}$. Ad ogni iterazione l'algoritmo raddoppia l'accuratezza del risultato ottenuto dal ciclo precedente, dunque il valore iniziale x_0 può essere poco accurato, come quello prodotto dalle istruzioni RECIPS e RSQRTS. Per illustrare con maggiore dettaglio il funzionamento della tecnica si propongono due esempi: il primo calcola il reciproco di un numero, mentre il secondo il reciproco della sua radice quadrata.

Primo esempio

Consideriamo il caso del calcolo della funzione reciproco $x = g(c) = 1/c$, agendo come segue:

$$x = \frac{1}{c} \Longrightarrow \frac{1}{x} - c = f(x) = 0$$

Otteniamo:

$$f(x) = \frac{1}{x} - c$$

$$f'(x) = -\frac{1}{x^2}$$

Il valore $\bar{x}$ che annulla la $f(x)$ è il risultato esatto del reciproco. Si vede graficamente che, se x_n è un punto nell'intorno di $\bar{x}$, il punto successivo x_{n+1} risulta compreso nell'intervallo $]x_n, \bar{x}]$, se $x_n \leq \bar{x}$, oppure nell'intervallo $[\bar{x}, x_n[$, se $x_n > \bar{x}$, quindi è più accurato di quello precedente. Procedendo nei calcoli si ottiene la formula ricorsiva:

$$x_{n+1} = x_n - \frac{\frac{1}{x_n} - c}{-\frac{1}{{x_n}^2}} = x_n(2 - c\,x_n)$$

che può essere codificata in un algorimo per il DSP; x_0 può essere il seed ottenuto dall'istruzione RECIPS.

Secondo esempio

Nel caso si voglia approssimare meglio la funzione $1/\sqrt{c}$, si trova:

$$x = g(c) = \frac{1}{\sqrt{c}} \Longrightarrow \frac{1}{x^2} - c = f(x) = 0$$

Dunque:

$$f(x) = \frac{1}{x^2} - c$$

$$f'(x) = -\frac{2}{x^3}$$

Per concludere, la formula ricorsiva da codificare diventa:

$$x_{n+1} = 0.5\,x_n(3 - c\,{x_n}^2)$$

dove x_0 è il seed ottenuto dall'istruzione RSQRTS.

7.5.2 Fn=RECIPS Fx

Genera una stima del reciproco di un numero floating point; l'operando è il registro Fx e il risultato viene posto nel registro Fn. Definendo l'operando d'ingresso come $v = (-1)^s 2^e (1.f)$, in notazione unbiased, il reciproco approssimato ritornato da RECIPS è l'ultimo membro della:

$$\frac{1}{v} = (-1)^s 2^{-e-1} \frac{2}{(1.f)} \simeq (-1)^s 2^{-e-1} \mathrm{LUT}[f_{[MSB,MSB-6]}]$$

Descrizione del funzionamento:

- l'esponente unbiased del risultato viene calcolato esattamente cambiando il segno di e e sottraendogli 1;
- i MSB della mantissa del risultato (bit implicito escluso) vengono letti da una apposita tabella LUT, posta in ROM, indirizzata dal numero ottenuto dai 7 MSB di f (bit implicito escluso).

La mantissa del risultato è accurata fino all'ottavo bit più significativo, dunque l'errore commesso sulla mantissa è

$$|\mathrm{LUT}[f_{[MSB,MSB-6]}] - f| < 2^{-8}$$

Considerando che l'esponente occupa 8 bit, il risultato viene rappresentato con 16 bit effettivi. Caratteristiche dell'istruzione:

- un operando di tipo *NaN* produce un risultato *NaN*, con tutti i bit forzati a 1;
- il segno di Fn è il segno di Fx;
- un'operando ± 0 provoca il ritorno di $\pm\infty$ e l'attivazione del flag AV di overflow;
- se l'esponente unbiased di Fx è $e > 125$ il risultato è ± 0, a seconda del segno di Fx.

Flag alterati:

- AZ è 1 se il risultato è 0, altrimenti è 0;
- AU è 0;
- AN è 1 se l'operando è negativo, altrimenti è 0;
- AV è 1 se l'operando è 0, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 se l'operando è *NaN*, altrimenti è 0.

La stima dell'inverso può essere usata come valore d'ingresso per l'algoritmo di Newton-Raphson, per estendere la precisione della mantissa del risultato. Ad ogni iterazione raddoppia l'accuratezza del risultato ottenuto dal ciclo precedente. Usando RECIPS, l'accuratezza della mantissa di partenza è di 8 bit; dopo un'iterazione aumenta a 16 bit e dopo due iterazioni diventa di 32 bit. L'algoritmo da implementare è il seguente:

$$x[n+1] = x[n]\,(2.0 - v\,x[n])$$

dove v è il numero di cui si vuole trovare il reciproco e $x[0]$ è il seed dato dalla istruzione RECIPS.

Una possibile implementazione della funzione di divisione tra due floating point che sfrutta la tecnica di Newton-Raphson è data di seguito. L'algoritmo esegue 2 iterazioni e il risultato è accurato entro ± 1 LSB della mantissa, in formato single precision. Impiega 8 cicli di clock.

```
/* Input:                        */
/*      F0=n (numeratore)        */
/*      F12=d (denominatore)     */
/*      F11=2.0                  */
/* Output:                       */
/*      F0=n/d                   */
/* Registri alterati:            */
/*      F0,F6,F7                 */

F6=RECIPS F12;  /* F6=x[0] */

F7=F6*F12;      /* F7=d*x[0] */
F7=F11-F7;      /* F7=2.0-d*x[0] */
F6=F6*F7;       /* F6=x[0]*(2.0-d*x[0])=x[1] */

F7=F6*F12;      /* F7=d*x[1] */
F7=F11-F7;      /* F7=2.0-d*x[1] */
F6=F6*F7;       /* F6=x[1]*(2.0-d*x[1])=x[2] */

F0=F0*F6;       /* F0=n*x[2] */
```

Facendo un'uso attento delle istruzioni parallele è possibile implementare un algoritmo che impiegando solo 8 cicli di clock esegue 3 iterazioni, con un risultato accurato fino al LSB della mantissa, indipendentemente dal fatto che sia attivo il modo single precision o extended precision.

```
/* Input:                        */
/*      F0=n (numeratore)        */
/*      F12=d (denominatore)     */
/*      F11=2.0                  */
/* Output:                       */
/*      F0=n/d                   */
/* Registri alterati:            */
/*      F0,F7,F12                */

F0=RECIPS F12, F7=F0;   /* F0=x[0] */
```

```
                        /* F7=n */
F12=F0*F12;             /* F12=d*x[0] */
F7=F0*F7, F0=F11-F12;   /* F7=n*x[0] */
                        /* F0=2.0-d*x[0] */

F12=F0*F12;             /* F12=d*x[0]*(2.0-d*x[0])=d*x[1] */
F7=F0*F7, F0=F11-F12;   /* F7=n*x[0]*(2.0-d*x[0])=n*x[1] */
                        /* F0=2.0-d*x[1] */
/**********************/
F12=F0*F12;             /* F12=d*x[1]*(2.0-d*x[1])=d*x[2] */
F7=F0*F7, F0=F11-F12;   /* F7=n*x[1]*(2.0-d*x[1])=n*x[2] */
                        /* F0=2.0-d*x[2] */
/**********************/
F0=F0*F7;               /* F0=n*x[2]*(2.0-d*x[2])=n*x[3] */
```

Rimuovendo il blocco segnato si eseguono 2 iterazioni in 5 cicli di clock e il risultato è accurato entro ± 1 LSB della mantissa, in formato single precision.

7.5.3 Fn=RSQRTS Fx

Genera un seed per il reciproco della radice quadrata di un numero floating point; l'operando è il registro Fx e il risultato viene posto nel registro Fn. Definendo come $v = 2^e (1.f)$ l'operando d'ingresso, in notazione unbiased, il valore ritornato da RSQRTS è l'ultimo membro della:

$$\frac{1}{\sqrt{v}} = 2^{-(e/2)-1} \frac{2}{\sqrt{(1.f)}} \simeq 2^{-(e>>1)-1} \mathrm{LUT}[(e_{[0]} \bowtie f_{[MSB,MSB-5]})]$$

Descrizione del funzionamento:

- l'esponente unbiased del risultato viene calcolato cambiando il segno di e, dopo averlo shiftato a destra di un bit, e sottraendo 1;
- i MSB della mantissa della stima (bit implicito escluso) vengono letti da una apposita tabella LUT, posta ROM, indirizzata dal numero ottenuto concatenando (simbolo $\bowtie$) il LSB di e e i 6 MSB di f (bit implicito escluso).

Poiché la mantissa del risultato è accurata fino al quarto bit più significativo l'errore commesso sulla mantissa è $< 2^{-4}$. Considerando che l'esponente occupa 8 bit, il risultato viene rappresentato con 12 bit effettivi. Caratteristiche dell'istruzione:

- un operando di tipo *NaN* o negativo diverso da 0 produce un risultato *NaN*, con tutti i bit forzati a 1;
- un'operando ± 0 provoca il ritorno di $+\infty$ e l'attivazione del flag AV di overflow;
- $+\infty$ ritorna $+0$.

Flag alterati:

- AZ è 1 se il risultato è $+0$, altrimenti è 0;
- AU è 0;
- AN è 1 se l'operando è -0, altrimenti è 0;
- AV è 1 se l'operando è 0, altrimenti è 0;
- AC è 0;
- AS è 0;
- AI è 1 quando l'operando è *NaN* oppure negativo diverso da 0, altrimenti è 0.

La stima di $1/\sqrt{x}$ può essere usata come valore d'ingresso l'algoritmo di Newton-Raphson, per estendere la precisione della mantissa del risultato. Ad ogni iterazione raddoppia l'accuratezza del risultato ottenuto dal ciclo precedente. Usando RSQRTS, l'accuratezza della mantissa di partenza è di 4 bit; dopo un'iterazione aumenta a 8 bit e dopo tre iterazioni diventa di 32 bit. L'algoritmo da implementare è il seguente:

$$x[n+1] = 0.5x[n](3 - vx[n]x[n])$$

dove v è il numero di cui si vuole trovare il reciproco della radice e $x[0]$ è il seed dato da RSQRTS.

Una possibile implementazione della funzione di reciproco della radice che sfrutta la tecnica di Newton-Raphson è data di seguito. L'algoritmo esegue 3 iterazioni e il risultato è accurato fino al LSB della mantissa, indipendentemente dal fatto che sia attivo il modo single precision o extended precision. Impiega 13 cicli di clock.

```
/* Input:                        */
/*      F0=n                     */
/*      F1=0.5                   */
/*      F8=3.0                   */
/* Output:                       */
/*      F4=risultato             */
/* Registri alterati:            */
/*      F0,F1,F4,F8,F12          */

F4=RSQRTS F0;           /* F4=x[0] */

F12=F4*F4;              /* F12=x[0]*x[0] */
F12=F12*F0;             /* F12=n*x[0]*x[0] */
F4=F1*F4, F12=F8-F12;   /* F4=0.5*x[0] */
                        /* F12=3.0-n*x[0]*x[0] */
F4=F4*F12;              /* F4=0.5*x[0]*(3.0-n*x[0]*x[0])= */
                        /* x[1] */

F12=F4*F4;              /* F12=x[1]*x[1] */
F12=F12*F0;             /* F12=n*x[1]*x[1] */
F4=F1*F4, F12=F8-F12;   /* F4=0.5*x[1] */
                        /* F12=3.0-n*x[1]*x[1] */
F4=F4*F12;              /* F4=0.5*x[1]*(3.0-n*x[1]*x[1])= */
                        /* x[2] */
/**********************/
F12=F4*F4;              /* F12=x[2]*x[2] */
F12=F12*F0;             /* F12=n*x[2]*x[2] */
F4=F1*F4, F12=F8-F12;   /* F4=0.5*x[2] */
                        /* F12=3.0-n*x[2]*x[2] */
F4=F4*F12;              /* F4=0.5*x[2]*(3.0-n*x[2]*x[2])= */
                        /* x[3] */
/**********************/
```

Rimuovendo il blocco segnato si eseguono 2 iterazioni in 9 cicli di clock e il risultato è accurato entro ± 1 LSB della mantissa, in formato single precision.

7.5.4 Osservazione

L'istruzione RSQRTS è stata introdotta perché a volte si rivela necessario normalizzare i dati e spesso il fattore di normalizzazione è la radice quadrata di una certa

quantità. Ad esempio, nelle operazioni che coinvolgono vettori, i versori relativi vengono trovati dividendo scalarmente i vettori per la loro lunghezza.

Per trovare $\sqrt{x}$ basta considerare che:

$$\sqrt{x} = \frac{x}{\sqrt{x}}$$

7.6 Operazioni matematiche parallele

Diciamo solo che esistono delle istruzioni combinate, dette istruzioni multifunzione, che permettono di eseguire più istruzioni in parallelo in un singolo ciclo di clock. Le istruzioni multifunzione si suddividono in tre tipi:

- istruzioni addizione/sottrazione combinate;
- istruzioni moltiplicatore e ALU;
- istruzioni moltiplicatore e addizione/sottrazione combinate.

I registri usati dalle istruzioni multifunzione floating point non possono essere scelti arbitrariamente:

- per le istruzioni che riguardano il moltiplicatore un operando deve essere F0, F1, F2 o F3 e l'altro F4, F5, F6 o F7;
- per le istruzioni che riguardano la ALU un operando deve essere F8, F9, F10 o F11 e l'altro F12, F13, F14 o F15;
- i registri risultato possono essere qualsiasi, ma distinti tra loro.

Tipo	Dimensione	Descrizione
int	32 bit	numero fixed point
float	32 bit	numero floating point

Tabella 10: Tipi dati nativi.

8 Linguaggio C

Lo standard ISO 9899-1990, oltre a definire e caratterizzare il linguaggio C, descrive anche come deve essere composta la libreria standard e quale deve essere il suo comportamento a run time. Noi analizzeremo l'implementazione della parte di libreria che tratta direttamente i numeri floating point, che comprende le funzioni e le costanti definite negli header **errno.h**, **float.h**, **math.h** e **signal.h**.

Cominciamo vedendo i tipi di dati e le estensioni al linguaggio che questa particolare implementazione mette a disposizione dell'utente. Di seguito vedremo cosa dice lo standard circa i numeri floating point e la gestione delle loro eccezioni. Per ultimo analizzaremo i singoli file header.

8.1 Tipi base

La famiglia di DSP in esame può elaborare operandi sia a 32 bit (fixed point e floating point) che a 40 bit (solo floating point). Il compilatore non supporta i floating point a 40 bit, dunque tutti i tipi aritmetici del C supportati nativamente dal DSP si riducono a quelli illustrati nella tabella 10. Tutti gli altri tipi dati del C vengono mappati nei tipi **int** e **float** secondo la tabella 11.

Lo switch **-fno-short-double** impone al compilatore di non usare floating point a 32 bit (single precision) per rappresentare i **double**, che è l'impostazione di default, ma piuttosto floating point a 64 bit (double precision). Lo switch **-ansi** sortisce il medesimo effetto. Di default i **double** sono in single precision. Le variabili dichiarate come **long double** vengono sempre rappresentate come **double**.

Le operazioni su numeri a doppia precisione (**double** e **long double**) vengono calcolate tramite emulazione software e si avvantaggiano solo parzialmente dell'unità di calcolo floating point hardware. Dunque risultano molto più lente delle analoghe funzioni che operano su numeri single precision.

Lo switch **-wfloat-convert** produce un messaggio di warning da parte del compilatore ogni volta che esso ha necessità di effettuare una *floating point silent conversion*, cioè una conversione implicita di un numero da single precision a double precision o viceversa, non richiesta esplicitamente dal programmatore tramite un *cast*.

Tipo	Mappato in	Dimensione
long int	int	32 bit
short int	int	32 bit
unsigned int	int	32 bit
unsigned long int	int	32 bit
unsigned short int	int	32 bit
char	int	32 bit
unsigned char	int	32 bit
double	float o double	32 o 64 bit
long double	double	64 bit
complex int	int + int	32+32 bit
complex float	float + float	32+32 bit

Tabella 11: Mappatura dei tipi non nativi.

Gli switch appena discussi non vengono riconosciuti dalla versione del compilatore distribuita con lo EZ-KIT LITE. Dunque i **double** sono sempre in single precision; se si vuole forzare il compilatore a usare variabili in double precision, esse vanno necessariamente dichiarate di tipo **long double**.

8.2 Tipi interi

Lo standard C richiede che nel file **limits.h** vengano specificati i range dei tipi interi della particolare implementazione. Le definizioni sono tipiche di una macchina con registri a 32 bit che riesce a manipolare efficacemente dati a 32 bit. Il file **limits.h** contiene:

```
#define CHAR_BIT        32
#define SCHAR_MIN       -2147483648
#define SCHAR_MAX       2147483647
#define UCHAR_MAX       4294967295u

#define CHAR_MIN        SCHAR_MIN
#define CHAR_MAX        SCHAR_MAX

#define MB_LEN_MAX      1

#define SHRT_MIN        SCHAR_MIN
#define SHRT_MAX        SCHAR_MAX
#define USHRT_MAX       UCHAR_MAX
```

```
#define INT_MIN         SHRT_MIN
#define INT_MAX         SHRT_MAX
#define UINT_MAX        USHRT_MAX

#define LONG_MIN        INT_MIN
#define LONG_MAX        INT_MAX
#define ULONG_MAX       UINT_MAX
```

8.3 Tipi complessi

Il tipo **complex** è stato introdotto dalle Numerical C extensions. Un numero complesso viene rappresentato attraverso due numeri **float** se viene dichiarato come **complex float** o con due **int** se viene dichiarato come **complex int**. In ogni caso la prima variabile in memoria rappresenta la parte reale, la seconda quella immaginaria.

8.4 Estensioni al linguaggio

GNU e Analog Devices hanno aggiunto delle estensioni al linguaggio C standard. Il flag **-ansi** identifica tutte le estensioni GNU al linguaggio standard, facendo produrre al compilatore un messaggio di avvertimento ogni volta che ne incontra una. Lo switch **-pedantic** opera in modo simile a **-ansi**, ma blocca la compilazione in caso si incontri un'estensione al linguaggio.

Per ulteriori informazioni circa le estensioni GNU e Analog Devices consultare il capitolo 5 del C TOOLS MANUAL. Per approfondire le Numerical C extensions vedere il capitolo 6.

8.4.1 Estensioni GNU

Le estensioni GNU supportate dal compilatore **g21k** riguardano:

- *non-constant initializers*;
- *constructor expressions*;
- *labeled elements in initializers*;
- *statements and declarations within expressions*;
- *variable lenght arrays*;
- costrutto **typeof()**;

- costrutto **inline**.

Inoltre la FSF (*Free Software Foundation*) ha inserito nel compilatore gcc il supporto per le estensioni al linguaggio proposte dal comitato ANSI NCEG (Numerical C Extensions Group), già visto precedentemente.

8.4.2 Estensioni Analog Devices

Le estensioni Analog Devices sono principalmente l'introduzione delle due keyword **dm** e **pm** per specificare la residenza in data memory o program memory di una certa variabile o per qualificare il tipo di un puntatore. Lo spazio di memoria data memory è quello di default per tutte le variabili, comprese quelle allocate sullo stack.

8.4.3 Funzioni built-in

Sono un piccolo set di funzioni che il compilatore riconosce e rimpiazza automaticamente *in-line*, cioè direttamente con il loro codice assembly, senza inserire alcuna chiamata a funzione. Il compilatore riconosce **abort**, **abs**, **exit**, **fabs**, **labs**, **memcmp**, **memcpy**, **strcmp**, **strcpy**, **strlen** e **sqrt**.

Lo switch **-fno-builtin** disattiva le funzioni in-line; il compilatore inserirà delle normali chiamate alla libreria runtime.

8.5 Floating point e C

Alcune considerazioni fondamentali.

- Due macchine possono usare la medesima rappresentazione IEEE per i formati floating point, eppure è possibile ottenere risultati diversi dagli stessi calcoli. Questo può dipendere:
  - dal modo in cui le macchine arrotondano i risultati che non possono essere rappresentati esattamente;
  - dal modo in cui le funzioni sono state implementate;
  - dal modo in cui i sistemi eseguono le singole istruzioni macchina.
- Troppe macchine usano ancora oggi un'aritmetica floating point rapida ma scorretta.
- Architetture diverse possono notificare in modo diverso le eccezioni al programma utente; inoltre alcune macchine possono generare solo alcune eccezioni.

Se lo standard C avesse tentato di definire esattamente il comportamento da tenere in tutti questi casi non sarebbe mai stato approvato. Come risultato, lo standard C è molto descrittivo nel campo dell'aritmetica floating point, si sforza di definire la situazione in generale ma dice poco a proposito dell'implementazione.

8.6 Eccezioni e C

Come visto in precedenza, ciascuna operazione matematica può produrre, in generale, una o più eccezioni. Le eccezioni generabili, secondo lo standard IEEE, sono:

- *overflow*;
- *underflow*;
- *loss of precision*;
- *division by 0*;
- *invalid operation*.

Le eccezioni fanno parte delle situazioni d'errore che si possono avere durante l'esecuzione dei programmi e vengono usualmente trattate nel modo più conveniente supportato dalla macchina: se l'architettura dispone di più metodi per gestire le eccezioni, la libreria può mettere a disposizione del programmatore diverse alternative.

Oltre alle singole istruzioni macchina, anche le routine matematiche e quelle di qualsiasi altro tipo possono incorrere in problemi durante l'esecuzione che devono essere segnalati all'utente. Il C standard non è in grado di identificare e gestire tutte queste situazioni di errore in modo completo e univoco.

8.7 Errori e C

Il C standard mette a disposizione due modi per gestire le situazioni di errore:

- tramite codici di errore, includendo l'header **errno.h**;
- tramite segnali, includendo il file **signal.h**.

Il primo metodo si basa sul ritorno, da parte della funzione chiamata, di un codice che indica se si è verificato un problema. Il codice deve essere controllato dal programma utente. E' l'equivalente dei controlli che si effettuano, in linguaggio

assembly, dopo la chiamata di certe routine o l'esecuzione di particolari istruzioni macchina.

Il secondo metodo consiste nell'interruzione del normale flusso del programma per passare subito il controllo a delle funzioni di gestione del problema. Di solito per ciascun tipo di segnale viene innescata una funzione dedicata. La gestione dei segnali è simile, a livello logico, alla gestione degli interrupt che si esegue in linguaggio assembly.

9 errno.h

La presenza dell'header **errno.h** è legata al modo in cui tradizionalmente vengono gestite le condizioni di errore in C.

9.1 Background

Convenzionalmente le funzioni C che danno un'indicazione del verificarsi di un qualsiasi errore durante la loro esecuzione lo fanno attraverso il ritorno di un numero intero che va interpretato in modo booleano. Poiché le funzioni non riferiscono esattamente quale errore è avvenuto si conviene di memorizzare l'error code esatto nella variabile globale **errno**, di tipo **int**.

Un programma che non si preoccupa di analizzare il motivo esatto di un errore può limitarsi a controllare il valore ritornato; in quei rari casi in cui interessano i dettagli si può controllare **errno** per vedere l'ultimo errore memorizzato. Naturalmente bisogna controllare prima che sia troppo tardi: se viene eseguita un'altra chiamata con esito negativo il valore verrà sovrascritto. Inoltre è necessario controllare **errno** solo dopo una chiamata che ha dato una segnalazione d'errore, perché una chiamata valida non ne altera il valore.

Secondo lo standard, **errno** è inizializzato a 0 alla partenza del programma e non deve mai essere posto a 0 da alcuna funzione. Il C descrive tutti gli errori matematici unicamente con due codici di errore, che devono essere obbligatoriamente definiti in **errno.h**:

- **EDOM** per un *domain error*, che raggruppa division by 0 e invalid operation;
- **ERANGE** per un *range error*, che raggruppa overflow e underflow, anche se la gestione di quest'ultimo errore non è obbligatoria.

Lo standard consente all'implementazione di definire altre macro di questo tipo. Una loss of precision è un errore incerto da riferire: come visto precedentemente, quella che è considerata una grave perdita per un programmatore può essere una questione assolutamente irrilevante per un altro. Come risultato nel C standard la loss of precision non viene considerata.

9.2 Implementazione

Lo standard non specifica quali valori debbano assumere i due codici di errore, richiede solo che siano diversi da 0; in questa implementazione essi valgono:

```
#define EDOM    33
#define ERANGE  34
```

10 float.h

In questo file viene messo ogni parametro, dipendente dalla macchina, che può essere d'aiuto al programmatore numerico serio. Vediamo le macro che, secondo lo standard, devono essere definite e il particolare valore che assumono in questa implementazione. I simboli il cui nome inizia con **FLT** documentano il tipo **float**, quelli che iniziano con **DBL** si riferiscono al tipo **double** e quelli inizianti con **LDBL** si riferiscono al tipo **long double**.

Nota: molti di questi parametri sono ricavabili direttamente dalla conoscenza dei formati IEEE.

10.1 FLT_ROUNDS

Indica il modo d'arrotondamento usato dalla macchina. I possibili valori sono:

- **-1** se non è determinabile;
- **0** se è verso lo 0 (troncamento);
- **1** se è al il più vicino;
- **2** se è verso $+\infty$;
- **3** se è verso $-\infty$.

Eventuali altri valori indicano che il tipo di arrotondamento è definito dall'implementazione. In questo contesto vale **4** e infatti un commento nel file **float.h** avvisa che:

- single precision floating point numbers are rounded to nearest;
- double precision floating point numbers are truncated.

Il programmatore può cambiare il modo d'arrodontamento all'interno di eventuali sue routine assembly, con l'obbligo però di ripristinarlo al loro termine.

10.2 FLT_RADIX

Indica la radice della rappresentazione esponenziale, cioè la base in cui è rappresentato l'esponente di tutti i floating point. Vale **2**.

Dimensione double	Macro che viene definita	Switch selettore
single precision	__DOUBLE_ARE_FLOATS__	-fshort-double
double precision	__DOUBLE_ARE_DOUBLES__	-fno-short-double

Tabella 12: Macro per segnalare la dimensione dei double.

10.3 XXX_MANT_DIG

Le macro **FLT_MANT_DIG**, **DBL_MANT_DIG** e **LDBL_MANT_DIG** indicano il numero di cifre, in base **FLT_RADIX**, contenute nella mantissa di un numero **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MANT_DIG    24
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MANT_DIG    FLT_MANT_DIG
#else
#define DBL_MANT_DIG    52
#endif
#define LDBL_MANT_DIG   DBL_MANT_DIG
```

La tabella 12 riassume quali macro sono definite quando i double sono in single precision (situazione di default) e quando sono in double precision, situazione ottenuta attivando lo switch **-fno-short-double** del compilatore.

10.4 FLT_DIG, DBL_DIG, LDBL_DIG

Indicano il numero di cifre decimali di precisione per **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_DIG         6
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_DIG         FLT_DIG
#else
#define DBL_DIG         15
#endif
#define LDBL_DIG        DBL_DIG
```

10.5 XXX_MIN_EXP

Le macro **FLT_MIN_EXP**, **DBL_MIN_EXP** e **LDBL_MIN_EXP** indicano il minimo esponente, in base **FLT_RADIX**, che può avere un numero normalizzato

di tipo **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MIN_EXP     -126
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MIN_EXP     FLT_MIN_EXP
#else
#define DBL_MIN_EXP     -1021
#endif
#define LDBL_MIN_EXP    DBL_MIN_EXP
```

10.6 XXX_MAX_EXP

Le macro **FLT_MAX_EXP**, **DBL_MAX_EXP** e **LDBL_MAX_EXP** indicano il massimo esponente, in base **FLT_RADIX**, che può avere un numero normalizzato di tipo **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MAX_EXP     127
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MAX_EXP     FLT_MAX_EXP
#else
#define DBL_MAX_EXP     1023
#endif
#define LDBL_MAX_EXP    DBL_MAX_EXP
```

10.7 XXX_MIN_10_EXP

Le macro **FLT_MIN_10_EXP**, **DBL_MIN_10_EXP** e **LDBL_MIN_10_EXP** indicano la minima potenza decimale rappresentabile da un numero normalizzato di tipo **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MIN_10_EXP  -37
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MIN_10_EXP  FLT_MIN_10_EXP
#else
#define DBL_MIN_10_EXP  -307
#endif
#define LDBL_MIN_10_EXP DBL_MIN_10_EXP
```

10.8 XXX_MAX_10_EXP

Le macro **FLT_MAX_10_EXP**, **DBL_MAX_10_EXP** e **LDBL_MAX_10_EXP** indicano la massima potenza decimale rappresentabile da un numero normalizzato di tipo **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MAX_10_EXP  38
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MAX_10_EXP  FLT_MAX_10_EXP
#else
#define DBL_MAX_10_EXP  308
#endif
#define LDBL_MAX_10_EXP DBL_MAX_10_EXP
```

10.9 FLT_MIN, DBL_MIN, LDBL_MIN

Indicano il più piccolo numero positivo normalizzato rappresentabile di tipo tipo **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MIN         1.1754943508222875E-38F
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MIN         FLT_MIN
#else
#define DBL_MIN         2.2250738585072014E-308
#endif
#define LDBL_MIN        DBL_MIN
```

Poiché l'intervallo dei numeri negativi è speculare a quello dei numeri positivi, cambiando il segno a queste costanti di ottengono i normalizzati negativi più piccoli (in valore assoluto).

10.10 FLT_MAX, DBL_MAX, LDBL_MAX

Indicano il più grande numero positivo normalizzato rappresentabile di tipo tipo **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_MAX         1.701411834605e+38F
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_MAX         FLT_MAX
```

```
#else
#define DBL_MAX         1.797693134862315708e+308
#endif
#define LDBL_MAX        DBL_MAX
```

Poiché l'intervallo dei numeri negativi è speculare a quello dei numeri positivi, cambiando il segno a queste costanti di ottengono i normalizzati negativi più grandi (in valore assoluto).

10.11 FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON

Indicano il minimo $x > 0.0$ tale che $1.0 + x \neq 1.0$, per i tipi **float**, **double** e **long double**, rispettivamente. Il frammento di codice che ne definisce il valore è il seguente:

```
#define FLT_EPSILON     1.1920928955078125E-07F
#ifdef __DOUBLES_ARE_FLOATS__
#define DBL_EPSILON     FLT_EPSILON
#else
#define DBL_EPSILON     2.2204460492503131e-16
#endif
#define LDBL_EPSILON    DBL_EPSILON
```

11 signal.h

L'header mette a disposizione le facility per gestire i segnali, sotto forma di macro e funzioni.

11.1 Background

Un segnale è una interruzione straordinaria del programma che si verifica durante la sua esecuzione. Esistono due tipi di segnali:

- quelli *sincroni* si hanno a causa di azioni intraprese dal programma stesso, come la divisione per 0;
- quelli *asincroni* avvengono a causa di un'azione esterna al programma, come la pressione di un tasto da parte di un utente.

I segnali si distinguono anche in base al trattamento di default che subiscono da parte del sistema:

- Alla partenza del programma alcuni segnali sono ignorati di default: se l'utente non indica esplicitamente che desidera gestirli, la loro eventuale generazione non altera l'esecuzione del programma, passando del tutto inosservata.
- Gli altri segnali devono essere sempre gestiti dall'utente: i segnali non ignorati che non sono gestiti vengono considerati errori fatali e terminano l'esecuzione del programma quando invocati.

Per rispondere ad un segnale si usa un *signal handler*, che esegue le operazioni appropriate per il segnale ricevuto. Un handler viene solitamente installato dal programma prima che questo inizi l'esecuzione vera e propria. Il signal handler può rimediare al problema, che altrimenti forzebbe il programma a terminare l'esecuzione o a continuare ignorando la condizione d'errore. Tutte le implementazioni conformi del C devono includere almeno gli strumenti essenziali per gestire i segnali, tra cui le funzioni:

- **raise**, per generare un segnale;
- **signal**, per comunicare al sistema quale funzione eseguire in risposta ad un particolare segnale.

L'insieme dei segnali gestibili da un programma varia a seconda dell'implementazione. Lo standard richiede che siano definiti almeno 6 segnali, identificati dalle seguenti macro, che devono espandersi in interi positivi non nulli:

- **SIGABRT**, *abort signal*;
- **SIGFPE** , *floating point exception*;
- **SIGILL** , *illegal instruction*;
- **SIGINT** , *system interrupt*;
- **SIGSEGV**, *segmentation violation*;
- **SIGTERM**, *software termination signal*.

Ogni implementazione è obbligata a rispondere solamente se uno di questi segnali è inviato esplicitamente tramite **raise**. Non è detto che questi segnali vengano generati anche indipendentemente. Questa limitazione è necessaria perché certi ambienti non sono in grado di accorgersi del verificarsi di una di queste condizioni.

11.1.1 int signal(int signo, void (*addr)(int))

Diciamo brevemente che **signal** indica al sistema quale handler eseguire in risposta ad un determinato segnale **signo**, passando all'ambiente l'indirizzo **addr** della funzione handler. Lo standard descrive delle macro che possono essere usate come argomento **addr** di **signal** o che possono essere il suo valore di ritorno:

- **SIG_DFL** indica il gestore di default;
- **SIG_ERR** indica un errore nell'impostazione del segnale;
- **SIG_IGN** indica di ignorare il segnale.

Le macro devono espandersi in interi che non possano mai essere l'indirizzo di una funzione; in questa implementazione abbiamo:

```
#define SIG_DFL ((void (*)(int))0x01)
#define SIG_ERR ((void (*)(int))0x02)
#define SIG_IGN ((void (*)(int))0x03)
```

Secondo lo standard, il prototipo di una routine di gestione dei segnali è sempre del tipo:

```
void my_handler(int signo);
```

Il parametro **signo** contiene il numero del segnale, utile per fare gestire più segnali alla stessa funzione.

11.1.2 int raise(int signo)

Spedisce il segnale **signo** al programma. Se effettuato all'interno di un signal handler, l'invio del segnale può essere ritardato fino all'uscita dalla routine di gestione stessa.

11.2 Implementazione

I 6 segnali standard non vengono supportati in hardware da questo DSP e vengono definiti in **signal.h** solo per motivi di conformità:

```
#define SIGABRT 32
#define SIGFPE  38
#define SIGILL  33
#define SIGINT  34
#define SIGSEGV 35
#define SIGTERM 36
#define SIGALRM 37
```

Possono essere usati come segnali software in un programma, ma poiché vengono scatenati solo tramite una chiamata a **raise**, risulta più efficiente chiamare le relative routine direttamente. La definizione di **SIGALRM** non era richiesta dallo standard.

In questa implementazione vengono mappati come segnali sia i segnali software precedenti, generati dallo stesso programma, che gli interrupt hardware, riassunti da queste definizioni:

```
#define SIG_SOVF        3
#define SIG_TMZ0        4
#define SIG_VIRPTI      5
#define SIG_IRQ2        6
#define SIG_IRQ1        7
#define SIG_IRQ0        8
#define SIG_SPR0I       10
#define SIG_SPR1I       11
#define SIG_SPT0I       12
#define SIG_SPT1I       13
#define SIG_LP2I        14
#define SIG_LP3I        15
#define SIG_EP0I        16
#define SIG_EP1I        17
#define SIG_EP2I        18
```

Interrupt	Numero	Descrizione	Segnale invocato
FLTOI	25	floating point overflow	SIG_FLTO
FLTUI	26	floating point underflow	SIG_FLTU
FLTII	27	floating point invalid	SIG_FLTI

Tabella 13: Eccezioni mappabili in segnali.

```
#define SIG_EP3I        19
#define SIG_LSRQ        20
#define SIG_CB7         21
#define SIG_CB15        22
#define SIG_TMZ         23
#define SIG_FIX         24
#define SIG_FLTO        25
#define SIG_FLTU        26
#define SIG_FLTI        27
#define SIG_USR0        28
#define SIG_USR1        29
#define SIG_USR2        30
#define SIG_USR3        31
```

In questo modo la C runtime library permette di gestire gli interrupt tramite signal handler scritti in C. In particolare, i segnali **SIG_FLTO**, **SIG_FLTU** e **SIG_FLTI** possono essere prodotti dagli interrupt relativi alle eccezioni floating point, secondo la tabella 13. In questa implementazione tutti i segnali sono ignorati di default.

11.3 Osservazioni

Alcune considerazioni conclusive. Per maggiori informazioni consultare lo USER'S MANUAL e il capitolo INTERRUPTS del C TOOLS MANUAL, pagina C-1 e seguenti.

11.3.1 Non persistenza della signal

La **signal** predispone la connessione logica tra segnale e handler solo per una sola invocazione dello stesso, cioè in modo non persistente; pertanto la **signal** dovrebbe essere ripetuta all'interno della funzione handler se si desiderano altre invocazioni.

Le funzioni non standard **interrupt**, **interruptf** e **interrupts**, definite in **signal.h**, permettono di predisporre degli *interrupt dispatcher* persistenti e particolarmente efficienti in termini di velocità.

11.3.2 Interrupt nesting

Nell'ambiente runtime C di questa famiglia di DSP è presente e attivo l'*interrupt nesting*; questo significa che se un interrupt ha mandato in esecuzione un signal handler e arriva un interrupt con priorità maggiore, la routine corrente è sospesa e il controllo viene passato a quella con priorità maggiore. Poiché un handler blocca tutti gli interrupt a priorità minore, è meglio che la sua esecuzione sia particolarmente veloce.

11.3.3 Comunicazione col main program

Poiché gli interrupt possono avvenire in qualunque momento, un handler non può ritornare un valore al programma principale direttamente. Un modo per passare dati al programma è tramite l'uso di variabili globali, accessibili sia dalla funzione handler che dal programma. Le variabili vanno dichiarate con lo specificatore **volatile** del C, in modo che il compilatore sia avvertito che possono cambiare valore in tempi e modi indipendenti dal normale flusso d'esecuzione.

11.3.4 Esempio

Il programma seguente mostra come gestire le eccezioni matematiche (e gli interrupt in genere) tramite i segnali. La macro **MAP** sfrutta un'estensione del compilatore per assegnare direttamente un pattern binario ad una variabile di tipo floating point.

```
#include <float.h>
#include <signal.h>

void my_handler(int signo)
{
    /* gestisci il problema */

    /* ricarica l'handler */
}

#define MAP(pattern) (__extension__ \
    ((union { float __f; int __i; }) {__i:pattern}).__f)
```

```
int main()
{
    float infp = MAP(0x7f800000);   /* infinito positivo */
    float max = FLT_MAX;
    float min = FLT_MIN;
    float one = 1;
    float zero = 0;
    float c;

    /* aggancia i segnali che interessano */

    signal(SIG_FLTI, my_handler);
    signal(SIG_FLTO, my_handler);
    signal(SIG_FLTU, my_handler);

    c = zero * infp; /* provoca invalid operation */

    c = max;
    c = c * 2;       /* provoca overflow */

    c = min;
    c = c / 2;       /* provoca underflow */

    return 0;
}
```

12 math.h

L'header **math.h** dichiara svariate funzioni matematiche e definisce la costante **HUGE_VAL** sotto forma di macro.

12.1 Background

Tutte le funzioni matematiche vanno definite almeno per argomenti di tipo **double**, poiché vige la regola che argomenti di tipo **float** vengono convertiti automaticamente dal compilatore in **double**. Ogni funzione deve essere eseguita come se si trattasse di un'istruzione macchina singola, senza generare alcuna eccezione visibile esternamente.

Lo standard prescrive che le funzioni matematiche rilevino le situazioni di domain error e range error e le indichino al programma attraverso la facility **errno**, come segue:

- **EDOM** si verifica quando si tenta di calcolare una funzione per un parametro che cade al di fuori del suo dominio. Per esempio, se si tenta di calcolare **asin(x)** di un numero **x** fuori dall'intervallo $[-1, 1]$ si incorre in questo tipo di errore. Il valore tradizionalmente ritornato in questa situazione è zero, ma alcune implementazioni possono ritornare un codice *NaN*. Ogni implementazione ha la facoltà di definire degli errori di dominio addizionali, purché questi errori siano coerenti con la definizione matematica della funzione. In un'implementazione che supporti gli infiniti, questo permette di ritornare un domain error se una funzione il cui dominio matematico non include infinito viene chiamata con un parametro infinito.

- **ERANGE** si verifica quando una funzione matematica tenta di costruire un valore troppo grande (overflow) che non può essere rappresentato in quel tipo di dato (**float**, **double** o **long double**). Quando questo succede, la funzione dovrebbe restituire **HUGE_VAL**, con lo stesso segno di quello che sarebbe il risultato corretto.

- Se il valore è troppo piccolo per poter essere rappresentato (underflow), la funzione che lo ha calcolato deve restituire 0. La variabile **errno** può essere settata al valore **ERANGE**, a discrezione dell'implementazione.

12.1.1 La macro HUGE_VAL

Rappresenta il più grande valore positivo assegnabile a un **double**. Nelle macchine che dispongono di un codice per rappresentare $+\infty$, la **HUGE_VAL** potrebbe

essere quel codice, quindi in generale non è detto che sia un numero. Una funzione matematica può restituire **HUGE_VAL** o **-HUGE_VAL** per indicare che il risultato è $+\infty$ oppure $-\infty$. Per esempio, un sistema può restituire **HUGE_VAL** per indicare il risultato di **tan(pi/2.0)**, che come noto è $+\infty$.

12.1.2 La funzione ideale

Una funzione matematica ideale:

- Dovrebbe accettare come argomenti tutti i valori compresi nel proprio dominio. Dovrebbe dare un domain error per tutti gli altri, aggiornando **errno** e ritornando un NaN.
- Dovrebbe produrre un risultato finito se il suo argomento è un valore finito.
- Dovrebbe dare un range error, aggiornando **errno**, per tutti i valori troppo grandi o troppo piccoli per poter essere rappresentati. Se il valore è troppo grande dovrebbe restituire **HUGE_VAL** oppure **-HUGE_VAL**, come opportuno. Se il valore è troppo piccolo la funzione dovrebbe restuire 0.
- Dovrebbe dare il risultato più sensato per degli argomenti di tipo NaN, $-\infty$, $+\infty$, **HUGE_VAL** e **-HUGE_VAL**.
- Su una implementazione che supporta codici NaN multipli dovrebbe mantenere codici NaN particolari ovunque possibile.

12.2 Matematica robusta

Di solito soltanto il programma numerico più sofisticato fa attenzione alle macro definite in **float.h**. Spesso i programmatori fanno delle ipotesi implicite sul campo di variazione o sulla dimensione minima o massima di alcuni parametri floating point, limitando la portabilità dei loro programmi.

Abbiamo visto che le tre insidie principali dell'aritmetica floating point sono l'overflow, l'underflow e la loss of precision. Le macro definite in **float.h** possono essere usate per riconoscere queste potenziali situazioni di errore prima che si verifichino. Di seguito vengono presentati alcuni esempi che riguardano il tipo **float**; considerazioni analoghe valgono anche per i **double**.

12.2.1 Riconoscere l'overflow

Per evitare l'overflow è necessario assicurarsi che nessun numero superi, in valore assoluto, la costante **FLT_MAX**. Non serve a niente controllare il risultato finale come segue:

```
y = expf(x);
if (y > FLT_MAX) {
    /* gestisci l'overflow */
}
```

perché quando il test verrà eseguito l'overflow si sarà già verificato. In generale, se il valore ritornato da **expf** è troppo grande per essere rappresentato, la **y** potrebbe contenere: un codice di errore, il valore **FLT_MAX** oppure un numero arbitrario. Un test più sensato potrebbe essere:

```
if (x > logf(FLT_MAX) {
    /* gestisci l'overflow */
} else {
    y = expf(x);
}
```

Si può evitare il calcolo di **logf(FLT_MAX)** attraverso l'uso di una macro, già vista precedentemente:

```
if (x > FLT_MAX_10_EXP) {
    /* gestisci l'overflow */
} else {
    y = powf(10, x);
}
```

12.2.2 Riconoscere l'underflow

Bisogna assicurarsi che nessun valore scenda, in valore assoluto, sotto la costante **FLT_MIN**. Generalmente le conseguenze di un underflow non sono così disastrose come nel caso dell'overflow, ma possono causare problemi lo stesso. Se un processore implementa il gradual overflow secondo quanto definito dallo standard IEEE alcuni dei suoi effetti vengono attenuati.

Quasi tutte le rappresentazioni sostituiscono il valore 0 a un risultato troppo piccolo per potere essere rappresentato; in questi casi si corrono dei guai solo se si divide per un numero che è andato in underflow precedentemente. Per individuare un underflow si può fare questo test:

```
y = expf(x);
if (y < FLT_MIN) {
    /* gestisci l'underflow */
}
```

che non è sciocco come quello precedente su **FLT_MAX**, anche se viene eseguito dopo che il danno è già stato fatto. Oppure:

```
if (x < logf(FLT_MIN) {
    /* gestisci l'underflow */
} else {
    y = expf(x);
}
```

12.2.3 Riconoscere la loss of precision

Può verificarsi quando si sottraggono valori quasi uguali. L'unico modo per salvarsi da tale destino è un'accurata analisi del problema prima di scrivere il programma.

E' tuttavia possibile cautelarsi rispetto ad un'altra forma più sottile di precision loss, che si manifesta quando si somma un numero piccolo a uno grande, in valori assoluti. Nell'addizione possono andare persi dei contributi importanti derivanti dai numeri più piccoli.

Ad esempio, ciò si può verificare durante una quadratura, che è un'integrazione numerica che approssima un'integrazione continua di una funzione. Una tecnica di quadratura consiste nel calcolare l'area sottesa dalla funzione come somma delle aree di una successione di rettangoli che stanno appena al di sotto della curva. Chiaramente, più stretti risultano i rettangoli, tanto meglio la loro somma approssimerà l'integrale vero della funzione. Sfortunatamente questo è vero solo in teoria, perché se le aree dei rettangoli sono abbastanza piccole parte del loro contributo, o tutto, andrà perso nella somma.

12.2.4 Osservazioni

A volte si devono elaborare numeri che non sono rappresentabili sotto forma di **float** perché sono troppo elevati, troppo piccoli o perché il tipo non ha abbastanza risoluzione. Un rimedio ovvio è quello di passare al tipo **double**, che migliora le prestazioni precedenti. Sfortunatamente questo richiede un tempo di calcolo maggiore sulle macchine che non implementano i **double** nativamente, come quella in esame.

Per quanto riguarda il riconoscimento e la gestione di domain error e di range error si può procedere in due modi, in generale:

- Si può controllare sempre la validità del parametro prima di chiamare la funzione, come illustrato di seguito:

```
if (x < -1 || x > 1) {
    /* x non è compresa nell'intervallo [-1,1] */
    /* gestisci l'errore */
}
else {
    y = asinf(x); /* procedi tranquillamente */
}
```

- Si può controllare lo stato di **errno**, dopo che la funzione è stata chiamata, nel modo seguente:

```
errno = 0; /* azzera */
y = asinf(x);
if (errno == EDOM) {
    /* x non era compresa nell'intervallo [-1,1] */
    /* gestisci l'errore */
}
```

12.3 Implementazione

Per ragioni di velocità di esecuzione la **math.h** dichiara, oltre ai prototipi per le funzioni obbligatorie che lavorano su **double**, anche i prototipi di altrettante funzioni equivalenti che operano su **float**. Queste ultime risultano significativamente più veloci.

Seguendo i dettami dello standard, la variabile **errno** e le macro **EDOM** e **ERANGE** vengono usati per indicare le situazioni d'errore. Nell'implementazione il valore di **HUGE_VAL** coincide con quello di **DBL_MAX**, ed è definito in **math.h** come segue:

```
#define HUGE_VAL 1.797693134862315708e+308
```

12.3.1 La macro __DOUBLES_ARE_FLOATS__

Se la macro **__DOUBLES_ARE_FLOATS__** è definita, le chiamate di tipo **double** vengono mappate in chiamate di tipo **float**, mediante le seguenti definizioni, scritte in **math.h**:

```
#define acos    acosf
#define asin    asinf
#define atan    atanf
#define atan2   atan2f
```

```
#define cos     cosf
#define sin     sinf
#define tan     tanf
#define cot     cotf
#define cosh    coshf
#define sinh    sinhf
#define tanh    tanhf
#define exp     expf
#define frexp   frexpf
#define ldexp   ldexpf
#define log     logf
#define log10   log10f
#define modf    modff
#define pow     powf
#define sqrt    sqrtf
#define rsqrt   rsqrtf
#define ceil    ceilf
#define fabs    fabsf
#define floor   floorf
#define fmod    fmodf
#define atoff   atof
```

12.3.2 Strana limitazione

La versione del compilatore distribuita con lo EZ-KIT LITE chiama sempre le versioni **float** delle funzioni, indipendentemente dai tipi dichiarati nel prototipo delle funzioni e dal fatto che la macro **__DOUBLES_ARE_FLOATS__** sia definita o meno.

Tutte le volte che si invoca una funzione passando come parametro un double precision il compilatore esegue una conversione implicita del numero verso la single precision. Un parametro è in double precision se è di tipo **long double** oppure se è di tipo **double** e il **double** è in double precision. Se il risultato della chiamata deve essere posto in una variabile double precision il compilatore esegue anche una conversione implicita da single precision a double precision.

Questo comportamento appare piuttosto strano perché fra i sorgenti della run-time library compaiono anche le implementazioni delle funzioni matematiche in double precision. Sono stati individuati due modi per poter usare le funzioni in versione double precision:

1. Chiamare le funzioni indirettamente tramite puntatore a funzione, come illustra l'esempio seguente.

```
long double acos(long double);
long double x, y;

int main()
{
    long double (*p)(long double) = acos;
    y = p(x); /* calcola l'arcocoseno */
    return 0;
}
```

Questo metodo ha successo solo con le funzioni **acos**, **asin**, **atan**, **atan2**, **tan**, **cot**, **cosh**, **sinh**, **tanh**, **exp**, **frexp**, **ldexp**, **log**, **log10**, **modf**, **pow**, **rsqrt**, **ceil**, **floor** e **fmod**. Se si tenta questo approccio con le funzioni rimanenti (**cos**, **sin**, **sqrt** e **fabs**) si ottiene un messaggio di errore da parte del compilatore del tipo:

```
warning: conflicting types for built-in function 'cos'
```

che ne provoca l'immediata terminazione. Per potere usare queste ultime quattro funzioni in versione double precision è indispensabile procedere col secondo metodo.

2. Prendere le funzioni che interessano, cambiarne il nome, assemblarle e linkarle direttamente col proprio codice.

12.3.3 Il metodo di analisi

Il procedimento adottato per analizzare ogni funzione dichiarata in **math.h** è stato il seguente:

- Analisi della sua documentazione, fornita nel C RUNTIME LIBRARY MANUAL.
- Confronto della documentazione con quanto viene richiesto dallo standard, con annotazione dei punti di non conformità.
- Controllo dell'implementazione, per verificarne la conformità sia rispetto alla documentazione che allo standard. In particolare, si è verificato se la gestione degli errori è stata compiuta correttamente.
- Misura del tempo di esecuzione.

Per misurare i tempi di esecuzione si è fatto uso del *cycle counter*, una facility messa a disposizione dal simulatore. Si tratta di un contatore a 64 bit il cui contenuto viene incrementato in corrispondenza di ogni ciclo di clock. La finestra del cycle counter si raggiunge dal menu aprendo **Core|Counters|Cycle Count**.

Poiché le funzioni fanno uso di diversi salti condizionati, il loro tempo d'esecuzione dipende abbastanza dall'ordine di grandezza del parametro di ingresso. Pertanto si è ritenuto opportuno riportare diversi tempi di esecuzione, per alcuni tra i parametri più significativi. Per alcune routine è stato possibile risalire alle condizioni esatte che determinano il numero di cicli di clock necessari.

I cicli riportati si riferiscono al puro tempo di chiamata, esecuzione e ritorno e non considerano il numero di istruzioni necessarie per passare i parametri e per memorizzare il risultato della chiamata, che è variabile. Nell'esempio seguente vengono illustrati alcuni tipi di chiamata con il relativo codice generato.

```
#include <math.h>

float x, y;
float *px, *py;

int main()
{
    y = cos(1.0);   /* f4=0x3f800000;          */
                    /* cjump (pc, _cosf) (DB); */
                    /* dm(i7,m7)=r2;           */
                    /* dm(i7,m7)=pc;           */
                    /* dm(_y)=f0;              */

    y = cos(x);     /* f4=dm(_x);              */
                    /* cjump (pc, _cosf) (DB); */
                    /* dm(i7,m7)=r2;           */
                    /* dm(i7,m7)=pc;           */
                    /* dm(_y)=f0;              */

    *py = cos(*px); /* i4=dm(_px);             */
                    /* i0=dm(_py);             */
                    /* f4=dm(i4,m5);           */
                    /* cjump (pc, _cosf) (DB); */
                    /* dm(i7,m7)=r2;           */
                    /* dm(i7,m7)=pc;           */
                    /* dm(i0,m5)=f0;           */
    return 0;
```

}

Le misure eseguite riguardano il tempo di esecuzione dei salti ritardati `cjump` e delle due istruzioni nella coda dei salti ritardati (la pipeline ha tre livelli di profondità). Queste due istruzioni servono per predisporre lo stack prima di eseguire la chiamata a funzione; la prima memorizza il frame pointer, mentre la seconda memorizza l'indirizzo a cui la funzione chiamata deve ritornare. Per maggiori informazioni riguardo all'organizzazione dello stack e alle convenzioni di chiamata di questa implementazione consultare il capitolo C RUNTIME ENVIRONMENT del C TOOLS MANUAL.

Per ottenere i tempi di esecuzione reali bisognerebbe aggiungere dai 2 ai 6 cicli di clock, a seconda di come vengono ricavati i parametri e di come viene memorizzato il risultato.

Si sono ricavati i tempi di esecuzione di tutte le funzioni in single precision e di tutte quelle in double precision che sono richiamabili indirettamente tramite puntatore, come visto precedentemente. Le funzioni standard chiamano spesso delle routine interne, descritte più avanti.

12.3.4 acos, acosf

Sintassi:

double acos(double x);

float acosf(float x);

Implementazione:

Le **acos** e **asin** sono implementate in **asin.asm** e condividono gran parte del codice; **acosf** e **asinf** sono implementate in **asinf.asm** e condividono gran parte del codice.

Descrizione:

Calcolano il valore principale dell'arco coseno di **x**. Il parametro deve essere nell'intervallo $[-1,-1]$; il risultato, espresso in radianti, è nel range $[0,\pi]$.

Tempi di esecuzione:

La **acosf** impiega:

- 70 cicli se $|\mathbf{x}| < 0.5$;
- 106 cicli se $|\mathbf{x}| >= 0.5$;
- 36 cicli se $\mathbf{x} \notin [-1,1]$.

La **acos** impiega:

- 87 cicli se $\mathbf{x} = 0$;
- 112 cicli se $|\mathbf{x}| = 1E-6$;
- 112 cicli se $|\mathbf{x}| = 1E-3$;
- 115 cicli se $|\mathbf{x}| = 1$;
- 65 cicli se $|\mathbf{x}| = 10$;
- 65 cicli se $|\mathbf{x}| = 1E2$;
- 65 cicli se $|\mathbf{x}| = 1E3$;
- 65 cicli se $|\mathbf{x}| = 1E4$;
- 65 cicli se $|\mathbf{x}| = 1E5$.

Note: Viene prodotto un domain error se l'argomento non è contenuto nell'intervallo $[-1,-1]$. In questa situazione **errno** viene settata a **EDOM** e viene ritornato 0, anche se quest'ultima azione non è richiesta dallo standard.

Aderenza allo standard C:
Ok

12.3.5 asin, asinf

Sintassi:

double asin(double x);

float asinf(float x);

Implementazione:
Le **asin** e **acos** sono implementate in **asin.asm** e condividono gran parte del codice; **asinf** e **acosf** sono implementate in **asinf.asm** e condividono gran parte del codice.

Descrizione:
Calcolano il valore principale dell'arco seno di **x**. Il parametro deve essere nell'intervallo $[-1,-1]$; il risultato, espresso in radianti, è nel range $[-\pi/2, \pi/2]$.

Tempi di esecuzione:
La **asinf** impiega:

- 66 cicli se $|\mathbf{x}| < 0.5$;
- 100 cicli se $|\mathbf{x}| >= 0.5$;
- 35 cicli se $\mathbf{x} \notin [-1,1]$.

La **asin** impiega:

- 72 cicli se $\mathbf{x} = 0$;
- 108 cicli se $|\mathbf{x}| = 1E-6$;
- 108 cicli se $|\mathbf{x}| = 1E-3$;
- 122 cicli se $|\mathbf{x}| = 1$;
- 64 cicli se $|\mathbf{x}| = 10$;
- 64 cicli se $|\mathbf{x}| = 1E2$;
- 64 cicli se $|\mathbf{x}| = 1E3$;
- 64 cicli se $|\mathbf{x}| = 1E4$;
- 64 cicli se $|\mathbf{x}| = 1E5$.

Note: Viene prodotto un domain error se l'argomento non è contenuto nell'intervallo $[-1,-1]$. In questa situazione **errno** viene settata a **EDOM** e viene ritornato 0, anche se quest'ultima azione non è richesta dallo standard.

Aderenza allo standard C:
Ok

12.3.6 atan, atanf

Sintassi:

double atan(double x);

float atanf(float x);

Implementazione:
Le **atan** e **atan2** sono implementate in **atan.asm** e condividono gran parte del codice; **atanf** e **atan2f** sono implementate in **atanf.asm** e condividono gran parte del codice.

Descrizione:
Calcolano il valore principale dell'arco tangente di **x**. Il risultato, espresso in radianti, è nel range $[-\pi/2, \pi/2]$.

Tempi di esecuzione:
La **atanf** impiega:

- 52 cicli se $\mathbf{x} = 0$;
- 52 cicli se $|\mathbf{x}| = 1E-6$;
- 72 cicli se $|\mathbf{x}| = 1E-3$;

- 92 cicli se $|\mathbf{x}| = 1$;
- 88 cicli se $|\mathbf{x}| = 10$;
- 88 cicli se $|\mathbf{x}| = 1E2$;
- 88 cicli se $|\mathbf{x}| = 1E3$;
- 68 cicli se $|\mathbf{x}| = 1E4$;
- 68 cicli se $|\mathbf{x}| = 1E5$.

La **atan** impiega:

- 88 cicli se $\mathbf{x} = 0$;
- 99 cicli se $|\mathbf{x}| = 1E-6$;
- 119 cicli se $|\mathbf{x}| = 1E-3$;
- 139 cicli se $|\mathbf{x}| = 1$;
- 135 cicli se $|\mathbf{x}| = 10$;
- 135 cicli se $|\mathbf{x}| = 1E2$;
- 135 cicli se $|\mathbf{x}| = 1E3$;
- 115 cicli se $|\mathbf{x}| = 1E4$;
- 115 cicli se $|\mathbf{x}| = 1E5$.

Note: Non sono possibili errori di dominio perché questo si estende a tutti i reali.

Aderenza allo standard C:
Ok

12.3.7 atan2, atan2f

Sintassi:

```
double atan2(double x, double y);
float atan2f(float x, float y);
```

Implementazione:
Le **atan2** e **atan** sono implementate in **atan.asm** e condividono gran parte del codice; **atan2f** e **atanf** sono implementate in **atanf.asm** e condividono gran parte del codice.

Descrizione:
Calcolano il valore principale dell'arco tangente del valore **y/x**. Il segno dei due argomenti viene preso in considerazione per determinare il quadrante del risultato. Il risultato, espresso in radianti, è nel range $[-\pi, \pi]$.

Se $\mathbf{x} \neq 0$ e $\mathbf{y} = 0$ il risultato è $-\pi/2$ oppure $\pi/2$, a seconda del segno di **x**.

Tempi di esecuzione:

La **atan2f** impiega:

- 24 cicli se $\mathbf{x} = 0$ e $\mathbf{y} = 0$;
- 41 cicli se $|\mathbf{x}| = 0$ e $|\mathbf{y}| = 1$;
- 76 cicli se $|\mathbf{x}| = 1E-6$ e $|\mathbf{y}| = 1$;
- 95 cicli se $|\mathbf{x}| = 1E-3$ e $|\mathbf{y}| = 1$;
- 116 cicli se $|\mathbf{x}| = 1$ e $|\mathbf{y}| = 1$;
- 112 cicli se $|\mathbf{x}| = 10$ e $|\mathbf{y}| = 1$;
- 112 cicli se $|\mathbf{x}| = 1E2$ e $|\mathbf{y}| = 1$;
- 112 cicli se $|\mathbf{x}| = 1E3$ e $|\mathbf{y}| = 1$;
- 92 cicli se $|\mathbf{x}| = 1E4$ e $|\mathbf{y}| = 1$;
- 92 cicli se $|\mathbf{x}| = 1E5$ e $|\mathbf{y}| = 1$.

La **atan2** impiega:

- 72 cicli se $\mathbf{x} = 0$ e $\mathbf{y} = 0$;
- 89 cicli se $|\mathbf{x}| = 0$ e $|\mathbf{y}| = 1$;
- 137 cicli se $|\mathbf{x}| = 1E-6$ e $|\mathbf{y}| = 1$;
- 157 cicli se $|\mathbf{x}| = 1E-3$ e $|\mathbf{y}| = 1$;
- 177 cicli se $|\mathbf{x}| = 1$ e $|\mathbf{y}| = 1$;
- 173 cicli se $|\mathbf{x}| = 10$ e $|\mathbf{y}| = 1$;
- 173 cicli se $|\mathbf{x}| = 1E2$ e $|\mathbf{y}| = 1$;
- 173 cicli se $|\mathbf{x}| = 1E3$ e $|\mathbf{y}| = 1$;
- 153 cicli se $|\mathbf{x}| = 1E4$ e $|\mathbf{y}| = 1$;
- 153 cicli se $|\mathbf{x}| = 1E5$ e $|\mathbf{y}| = 1$.

Note: Viene prodotto un domain error se **x** e **y** sono entrambe nulle. In questo caso viene ritornato 0, ma **errno** non viene settata a **EDOM** come dovrebbe. La pagina L-8 del C RUNTIME LIBRARY MANUAL riporta un'indicazione errata.

Aderenza allo standard C:

No

12.3.8 cos, cosf

Sintassi:

double cos(double x);

float cosf(float x);

Implementazione:

Le **cos** e **sin** sono implementate in **sin.asm** e condividono gran parte del codice; **cosf** e **sinf** sono implementate in **asinf.asm** e condividono gran parte del codice.

Descrizione:

Calcolano il coseno di **x**. Il parametro viene interpretato in radianti; il risultato è nell'intervallo $[-1,-1]$.

Tempi di esecuzione:

La **cosf** impiega sempre 56 cicli.

Note: Non viene ritornato alcun errore. Anche se **cos** e **cosf** non incorrono in alcun errore di dominio, l'accuratezza del loro risultato decresce significativamente per argomenti molto grandi.

Aderenza allo standard C:

Ok

12.3.9 sin, sinf

Sintassi:

double sin(double x);

float sinf(float x);

Implementazione:

Le **sin** e **cos** sono implementate in **sin.asm** e condividono gran parte del codice; **sinf** e **cosf** sono implementate in **asinf.asm** e condividono gran parte del codice.

Descrizione:

Calcolano il seno di **x**. Il parametro viene interpretato in radianti; il risultato è nell'intervallo $[-1,-1]$.

Tempi di esecuzione:

La **sinf** impiega:

- 40 cicli se $\mathbf{x} = 0$;
- 55 cicli se $|\mathbf{x}| = 1E-6$;

- 55 cicli se $|\mathbf{x}| = 1E-3$;
- 55 cicli se $|\mathbf{x}| = 1$;
- 55 cicli se $|\mathbf{x}| = 10$;
- 55 cicli se $|\mathbf{x}| = 1E2$;
- 55 cicli se $|\mathbf{x}| = 1E3$;
- 55 cicli se $|\mathbf{x}| = 1E4$;
- 55 cicli se $|\mathbf{x}| = 1E5$.

Note: Non viene ritornato alcun errore. Anche se **sin** e **sinf** non incorrono in alcun errore di dominio, l'accuratezza del loro risultato decresce significativamente per argomenti molto grandi.

Aderenza allo standard C:
Ok

12.3.10 tan, tanf

Sintassi:

double tan(double x);

float tanf(float x);

Implementazione:
Le **tan** e **cot** sono implementate in **tan.asm** e condividono gran parte del codice; **tanf** e **cotf** sono implementate in **tanf.asm** e condividono gran parte del codice.

Descrizione:
Calcolano la tangente di **x**. Il parametro viene interpretato in radianti.

Tempi di esecuzione:
La **tanf** impiega:

- 46 cicli se $\mathbf{x} = 0$;
- 58 cicli se $|\mathbf{x}| = 1E-6$;
- 58 cicli se $|\mathbf{x}| = 1E-3$;
- 58 cicli se $|\mathbf{x}| = 1$;
- 46 cicli se $|\mathbf{x}| = \pi/2$;
- 46 cicli se $|\mathbf{x}| = \pi$;
- 46 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;

- 46 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 58 cicli se $|\mathbf{x}| = 10$;
- 58 cicli se $|\mathbf{x}| = 1E2$;
- 58 cicli se $|\mathbf{x}| = 1E3$;
- 58 cicli se $|\mathbf{x}| = 1E4$;
- 58 cicli se $|\mathbf{x}| = 1E5$.

La **tan** impiega:

- 74 cicli se $\mathbf{x} = 0$;
- 97 cicli se $|\mathbf{x}| = 1E-6$;
- 97 cicli se $|\mathbf{x}| = 1E-3$;
- 97 cicli se $|\mathbf{x}| = 1$;
- 85 cicli se $|\mathbf{x}| = \pi/2$;
- 85 cicli se $|\mathbf{x}| = \pi$;
- 85 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 85 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 97 cicli se $|\mathbf{x}| = 10$;
- 97 cicli se $|\mathbf{x}| = 1E2$;
- 97 cicli se $|\mathbf{x}| = 1E3$;
- 97 cicli se $|\mathbf{x}| = 1E4$;
- 97 cicli se $|\mathbf{x}| = 1E5$.

Note: Non viene ritornato alcun errore. Anche se **tan** e **tanf** non incorrono in alcun errore di dominio, l'accuratezza del loro risultato decresce significativamente per argomenti molto grandi; inoltre bisogna fare attenzione quando il loro argomento è prossimo ad un multiplo dispari di $\pi/2$.

Aderenza allo standard C:
Ok

12.3.11 cot, cotf

Sintassi:

```
double cot(double x);
float cotf(float x);
```

Implementazione:

Le **cot** e **tan** sono implementate in **tan.asm** e condividono gran parte del codice; **cotf** e **tanf** sono implementate in **tanf.asm** e condividono gran parte del codice.

Descrizione:

Calcolano la cotangente di **x**. Il parametro viene interpretato in radianti.

Tempi di esecuzione:

cotf impiega:

- 48 cicli se $\mathbf{x} = 0$;
- 60 cicli se $|\mathbf{x}| = 1E-6$;
- 60 cicli se $|\mathbf{x}| = 1E-3$;
- 60 cicli se $|\mathbf{x}| = 1$;
- 48 cicli se $|\mathbf{x}| = \pi/2$;
- 48 cicli se $|\mathbf{x}| = \pi$;
- 48 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 48 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 60 cicli se $|\mathbf{x}| = 10$;
- 60 cicli se $|\mathbf{x}| = 1E2$;
- 60 cicli se $|\mathbf{x}| = 1E3$;
- 60 cicli se $|\mathbf{x}| = 1E4$;
- 60 cicli se $|\mathbf{x}| = 1E5$.

cot impiega:

- 87 cicli se $\mathbf{x} = 0$;
- 99 cicli se $|\mathbf{x}| = 1E-6$;
- 99 cicli se $|\mathbf{x}| = 1E-3$;
- 99 cicli se $|\mathbf{x}| = 1$;
- 87 cicli se $|\mathbf{x}| = \pi/2$;
- 87 cicli se $|\mathbf{x}| = \pi$;
- 87 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 87 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 99 cicli se $|\mathbf{x}| = 10$;

- 99 cicli se $|\mathbf{x}| = 1E2$;
- 99 cicli se $|\mathbf{x}| = 1E3$;
- 99 cicli se $|\mathbf{x}| = 1E4$;
- 99 cicli se $|\mathbf{x}| = 1E5$.

Note: Non viene ritornato alcun errore. Anche se **cot** e **cotf** non incorrono in alcun errore di dominio, l'accuratezza del loro risultato decresce significativamente per argomenti molto grandi; inoltre bisogna fare attenzione quando il loro argomento è prossimo ad un multiplo pari di $\pi/2$.

Aderenza allo standard C:
Ok

12.3.12 cosh, coshf

Sintassi:
double cosh(double x);

float coshf(float x);

Implementazione:
Le **cosh** e **sinh** sono implementate in **sinh.asm** e condividono gran parte del codice; **coshf** e **sinhf** sono implementate in **sinhf.asm** e condividono gran parte del codice.

Descrizione:
Calcolano il coseno iperbolico di **x**.

Tempi di esecuzione:
coshf impiega:

- 60 cicli se $\mathbf{x} = 0$;
- 81 cicli se $|\mathbf{x}| = 1E-6$;
- 81 cicli se $|\mathbf{x}| = 1E-3$;
- 81 cicli se $|\mathbf{x}| = 1$;
- 81 cicli se $|\mathbf{x}| = \pi/2$;
- 81 cicli se $|\mathbf{x}| = \pi$;
- 81 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 81 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 81 cicli se $|\mathbf{x}| = 10$;
- 81 cicli se $|\mathbf{x}| = 1E2$;

- 27 cicli se $|\mathbf{x}| = 1E3$;
- 48 cicli se $|\mathbf{x}| = 1E4$;
- 64 cicli se $|\mathbf{x}| = 1E5$.

cosh impiega:

- 95 cicli se $\mathbf{x} = 0$;
- 116 cicli se $|\mathbf{x}| = 1E-6$;
- 116 cicli se $|\mathbf{x}| = 1E-3$;
- 116 cicli se $|\mathbf{x}| = 1$;
- 116 cicli se $|\mathbf{x}| = \pi/2$;
- 116 cicli se $|\mathbf{x}| = \pi$;
- 116 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 116 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 116 cicli se $|\mathbf{x}| = 10$;
- 116 cicli se $|\mathbf{x}| = 1E2$;
- 51 cicli se $|\mathbf{x}| = 1E3$;
- 51 cicli se $|\mathbf{x}| = 1E4$;
- 51 cicli se $|\mathbf{x}| = 1E5$.

Note: La descrizione delle funzioni, a pagina L-29, da un'indicazione errata sul valore massimo che può assumere **x**, almeno per quanto riguarda la **coshf**.

Secondo lo standard un range error può verificarsi se **x** è un numero positivo molto elevato. Analizzando la **coshf** con il simulatore si trova che essa ritorna un *NaN* del tipo 0xFFFFFFFF nel caso si abbia $|\mathbf{x}| \gtrsim 88$. Però non si verifica alcun range error, dunque il controllo va fatto manualmente. Si può procedere controllando il parametro prima di chiamare la funzione, oppure controllando il risultato dopo averla chiamata. Nel secondo caso va usato un trucco come il seguente, che ovviamente risulta essere non portabile:

```
union
{
    float f;
    int i;
} y;
```

```
y.f = coshf(x);
if (y.i == 0xFFFFFFFF) {
    /* gestisci il range error */
}
```

Aderenza allo standard C:

Ok

12.3.13 sinh, sinhf

Sintassi:

double sinh(double x);

float sinhf(float x);

Implementazione:

Le **sinh** e **cosh** sono implementate in **sinh.asm** e condividono gran parte del codice; **sinhf** e **coshf** sono implementate in **sinhf.asm** e condividono gran parte del codice.

Descrizione:

Calcolano il seno iperbolico di **x**.

Tempi di esecuzione:

sinhf impiega:

- 21 cicli se $\mathbf{x} = 0$;
- 21 cicli se $|\mathbf{x}| = 1E-6$;
- 40 cicli se $|\mathbf{x}| = 1E-3$;
- 40 cicli se $|\mathbf{x}| = 1$;
- 64 cicli se $|\mathbf{x}| = \pi/2$;
- 64 cicli se $|\mathbf{x}| = \pi$;
- 64 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 64 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 64 cicli se $|\mathbf{x}| = 10$;
- 64 cicli se $|\mathbf{x}| = 1E2$;
- 64 cicli se $|\mathbf{x}| = 1E3$;
- 64 cicli se $|\mathbf{x}| = 1E4$;
- 64 cicli se $|\mathbf{x}| = 1E5$.

sinh impiega:

- 46 cicli se $\mathbf{x} = 0$;
- 57 cicli se $|\mathbf{x}| = 1E-6$;
- 76 cicli se $|\mathbf{x}| = 1E-3$;
- 76 cicli se $|\mathbf{x}| = 1$;
- 120 cicli se $|\mathbf{x}| = \pi/2$;
- 120 cicli se $|\mathbf{x}| = \pi$;
- 120 cicli se $|\mathbf{x}| = \pi \cdot 3/2$;
- 120 cicli se $|\mathbf{x}| = \pi \cdot 2$;
- 120 cicli se $|\mathbf{x}| = 10$;
- 120 cicli se $|\mathbf{x}| = 1E2$;
- 55 cicli se $|\mathbf{x}| = 1E3$;
- 55 cicli se $|\mathbf{x}| = 1E4$;
- 55 cicli se $|\mathbf{x}| = 1E5$.

Note: Analizzando la **sinhf** con il simulatore si trova che essa ritorna **x** quando $\mathbf{x} < 0$ e 0 quando $\mathbf{x} \gtrsim 1.0009906$. La funzione è stata evidentemente implementata male.

Aderenza allo standard C:
No

12.3.14 tanh, tanhf

Sintassi:
double tanh(double x);

float tanhf(float x);

Implementazione:
La **tanh** è implementata in **tanh.asm**; la **tanhf** è implementata in **tanhf.asm**.

Descrizione:
Calcolano la tangente iperbolica di **x.**

Tempi di esecuzione:
tanhf impiega:

- 25 cicli se $\mathbf{x} = 0$;
- 25 cicli se $|\mathbf{x}| = 1E-6$;

- 43 cicli se $|\mathbf{x}| = 1E-3$;
- 84 cicli se $|\mathbf{x}| = 1$;
- 22 cicli se $|\mathbf{x}| = 10$;
- 22 cicli se $|\mathbf{x}| = 1E2$;
- 22 cicli se $|\mathbf{x}| = 1E3$;
- 22 cicli se $|\mathbf{x}| = 1E4$;
- 22 cicli se $|\mathbf{x}| = 1E5$.

tanh impiega:

- 53 cicli se $\mathbf{x} = 0$;
- 64 cicli se $|\mathbf{x}| = 1E-6$;
- 82 cicli se $|\mathbf{x}| = 1E-3$;
- 123 cicli se $|\mathbf{x}| = 1$;
- 61 cicli se $|\mathbf{x}| = 10$;
- 61 cicli se $|\mathbf{x}| = 1E2$;
- 61 cicli se $|\mathbf{x}| = 1E3$;
- 61 cicli se $|\mathbf{x}| = 1E4$;
- 61 cicli se $|\mathbf{x}| = 1E5$.

Note: Non sono possibili errori di dominio perché questo si estende a tutti i reali. Analizzando **tanh** e **tanhf** si trova che:

- se $|\mathbf{x}| \gtrsim 9.011$ viene ritornato ± 1, a seconda del segno di **x**;
- se $|\mathbf{x}| > \log 3/2 \simeq 0.549$ viene ritornato $1 - \frac{2}{\exp(2\mathbf{x})+1}$;
- se $\mathbf{x} \lesssim 2.441E-4$ viene ritornato **x**.

Aderenza allo standard C:

Ok

12.3.15 exp, expf

Sintassi:

double exp(double x);

float expf(float x);

Implementazione:

La **exp** è implementata in **exp.asm** e la **expf** è implementata in **expf.asm**.

Descrizione:

Calcolano l'esponenziale naturale di **x**.

Tempi di esecuzione:

expf impiega:

- 23 cicli se $\mathbf{x} = 0$;
- 44 cicli se $|\mathbf{x}| = 1E-6$;
- 44 cicli se $|\mathbf{x}| = 1E-3$;
- 44 cicli se $|\mathbf{x}| = 1$;
- 44 cicli se $|\mathbf{x}| = 10$;
- 44 cicli se $|\mathbf{x}| = 1E2$;
- 19 cicli se $|\mathbf{x}| = 1E3$;
- 19 cicli se $|\mathbf{x}| = 1E4$.

exp impiega:

- 48 cicli se $\mathbf{x} = 0$;
- 2058 cicli se $|\mathbf{x}| = 1E-6$;
- 2058 cicli se $|\mathbf{x}| = 1E-3$;
- 2171 cicli se $|\mathbf{x}| = 1$;
- 2171 cicli se $|\mathbf{x}| = 10$;
- 2171 cicli se $|\mathbf{x}| = 1E2$;
- 52 cicli se $|\mathbf{x}| = 1E3$;
- 52 cicli se $|\mathbf{x}| = 1E4$.

Note: Secondo lo standard un range error può verificarsi se **x** è un numero positivo molto elevato, ma ciò non accade in questa implementazione. Analizzando **exp.asm** si trova che:

- viene ritornato un *NaN*, 0x7FEFFFFFFFFFFFFF, se vale che $\mathbf{x} > 709$;
- viene ritornato il più piccolo numero floating point positivo (cioè 0x0010000000000000, 2.2E-308), se vale che $\mathbf{x} < -725$;
- viene ritornato 1 se $|\mathbf{x}| < 5.55E-17$.

Analizzando **expf.asm** si trova che:

- viene ritornato un numero denormalizzato (0x00002710, in decimale 1.401298E-41) se $\mathbf{x} > 127$;
- viene ritornato il più piccolo numero positivo (0x00800000, cioè 1.175494E-38) se $\mathbf{x} < -88$;
- viene ritornato 1 se $|\mathbf{x}| < 1E-9$.

Il ritorno del denormalizzato è sintomo di una cattiva implementazione della funzione per due motivi. Il primo è che il DSP non supporta i denormalizzati; il secondo è che, anche se li supportasse, era più ovvio ritornare un numero positivo grande, $+\infty$ oppure un NaN. Il frammento di codice della **expf** incriminato è:

```
output_too_large:
    JUMP (PC, restore_state) (DB);
    F0=10000;           /*Set some error here*/
    FETCH_RETURN
```

Probabilmente l'intenzione degli implementatori era di ritornare il floating point 10000.0 in caso $\mathbf{x} > 127$. Però l'assemblatore crede, in modo molto discutibile, che si debba mettere l'intero 10000 (cioè 0x00002710) nel registro R0 e produce di conseguenza un un codice macchina per il trasferimento di 0x00002710 in R0. I programmatori dovevano scrivere:

```
    F0=10000.0;
```

Per aggiustare la situazione senza modificare la libreria si potrebbe chiamare sempre la seguente funzione, invece di invocare direttamente la **expf**:

```
float my_expf(float x)
{
    union
    {
        float f;
        int i;
    } y;

    y.f = expf(x);
    if (y.i == 0x00002710) {
        y.f = FLT_MAX;
```

```
    }
    return y.f;
}
```

Aderenza allo standard C:
Parzialmente

12.3.16 frexp, frexpf

Sintassi:
double frexp(double x, int *exp);

float frexpf(float x, int *exp);

Implementazione:
La **frexp** è implementata sia in **frexp.asm** che in **frexpp.asm**; la **frexpf** è implementata sia in **frexpf.asm** che in **frexpfp.asm**.

Descrizione:
Spezzano il numero floating point **x** in una parte frazionaria e in una potenza intera di 2. Il numero che viene ritornato è la parte frazionaria, con valore assoluto compreso nell'intervallo $[1/2, 1[$, oppure nullo se $\mathbf{x} = 0$. L'esponente della potenza viene memorizzato all'indirizzo passato in **exp**; il valore è forzato a zero se $\mathbf{x} = 0$.

Tempi di esecuzione:
La **frexpf** impiega:

- 19 cicli se **x** è un NaN;
- 16 cicli se **x** è $\pm\infty$;
- 16 cicli altrimenti.

La **frexp** impiega:

- 24 cicli se $\mathbf{x} \neq 0$;
- ∞ cicli se $\mathbf{x} = 0$.

Note: Le funzioni in **frexp.asm** e **frexpf.asm** vengono chiamate quando **exp** punta in data memory, mentre vengono invocate le implementazioni in **frexpp.asm** e **frexpfp.asm** se **exp** punta in program memory. Le differenze tra le versioni riguardano solo il modo in cui viene memorizzato l'esponente in **exp**. Le **ldexp** e **ldexpf** eseguono il ruolo opposto delle **frexp** e **frexpf**.

Utilizzando il simulatore si nota che invocando **frexpf** viene chiamata una funzione che non coincide con quella implementata in **frexpf.asm**. In particolare la funzione reale controlla anche se **x** è un *NaN*; in caso affermativo setta **errno** con **EDOM** e ritorna un *NaN* di codice 0xFFFFFFFF. In caso **x** sia $\pm\infty$ viene ritornato **x**. Il manuale, a pagina L-42, non menziona questi comportamenti.

Aderenza allo standard C:

La **frexpf** ok. La **frexp** no, perché può entrare in un ciclo infinito.

12.3.17 ldexp, ldexpf

Sintassi:

double ldexp(double x, int exp);

float ldexpf(float x, int exp);

Implementazione:

La **ldexp** è implementata in **ldexp.asm**, mentre la **ldexpf** è implementata in **ldexpf.asm**.

Descrizione:

Restituiscono il numero $\mathbf{x} \cdot 2^{\mathbf{exp}}$.

Tempi di esecuzione:

La **ldexpf** impiega sempre 9 cicli. La **ldexp** impiega:

- 19 cicli se l'esponente biased del risultato è ≤ 2047;
- 22 cicli altrimenti.

Note: Le **frexp** e **frexpf** eseguono il compito opposto delle **ldexp** e **ldexpf**. La **ldexp** ritorna **HUGE_VAL** senza settare **errno**, se l'esponente biased del numero risultante è maggiore di 2047.

La **ldexpf** invece non effettua alcun controllo sul range dell'esponente. Si risolve in una semplice istruzione `SCALB`, dunque ritorna **x** se $\mathbf{x} = \pm\infty$ oppure un *NaN* di codice 0xFFFFFFFF se **x** è un *NaN*. Le indicazioni date sul manuale, a pagina L-67, non sono corrette.

Aderenza allo standard C:

Ok

12.3.18 log, logf

Sintassi:

double log(double x);

float logf(float x);

Implementazione:

Le **log** e **log10** sono implementate in **logs.asm** e condividono gran parte del codice; **logf** e **log10f** sono implementate in **logsf.asm** e condividono gran parte del codice.

Descrizione:

Calcolano il logaritmo naturale di **x**. Secondo lo standard un errore di dominio si verifica se $\mathbf{x} < 0$, mentre un range error può verificarsi se $\mathbf{x} = 0$ o se **x** è un numero positivo molto piccolo.

Tempi di esecuzione:

La **logf** impiega:

- 28 cicli se $\mathbf{x} < 0$;
- 27 cicli se $\mathbf{x} = 0$;
- 61 cicli se $\mathbf{x} = 1E-6$;
- 61 cicli se $\mathbf{x} = 1E-3$;
- 61 cicli se $\mathbf{x} = 1$;
- 61 cicli se $\mathbf{x} = 10$;
- 61 cicli se $\mathbf{x} = 1E2$;
- 61 cicli se $\mathbf{x} = 1E3$;
- 61 cicli se $\mathbf{x} = 1E4$;
- 61 cicli se $\mathbf{x} = 1E5$.

La **log** impiega:

- 52 cicli se $\mathbf{x} < 0$;
- 51 cicli se $\mathbf{x} = 0$;
- 3320 cicli se $\mathbf{x} = 1E-6$;
- 3320 cicli se $\mathbf{x} = 1E-3$;
- 2532 cicli se $\mathbf{x} = 1$;
- 3320 cicli se $\mathbf{x} = 10$;
- 3365 cicli se $\mathbf{x} = 1E2$;

- 3365 cicli se $\mathbf{x} = 1E3$;
- 3320 cicli se $\mathbf{x} = 1E4$;
- 3365 cicli se $\mathbf{x} = 1E5$.

Note: Analizzando **log** e **logf** si trova che se $\mathbf{x} \leq 0$ viene ritornato 0 e **errno** viene settata a **EDOM**.

Aderenza allo standard C:
Ok

12.3.19 log10, log10f

Sintassi:

double log10(double x);

float log10f(float x);

Implementazione:
Le **log10** e **log** sono implementate in **logs.asm** e condividono gran parte del codice; **log10f** e **logf** sono implementate in **logsf.asm** e condividono gran parte del codice.

Descrizione:
Calcolano il logaritmo decimale di **x**. Secondo lo standard un errore di dominio si verifica se $\mathbf{x} < 0$, mentre un range error può verificarsi se $\mathbf{x} = 0$ o se **x** è un numero positivo molto piccolo.

Tempi di esecuzione:
La **log10f** impiega:

- 28 cicli se $\mathbf{x} < 0$;
- 27 cicli se $\mathbf{x} = 0$;
- 61 cicli se $\mathbf{x} = 1E-6$;
- 61 cicli se $\mathbf{x} = 1E-3$;
- 61 cicli se $\mathbf{x} = 1$;
- 61 cicli se $\mathbf{x} = 10$;
- 61 cicli se $\mathbf{x} = 1E2$;
- 61 cicli se $\mathbf{x} = 1E3$;
- 61 cicli se $\mathbf{x} = 1E4$;
- 61 cicli se $\mathbf{x} = 1E5$.

La **log10** impiega:

- 52 cicli se $\mathbf{x} < 0$;
- 51 cicli se $\mathbf{x} = 0$;
- 3378 cicli se $\mathbf{x} = 1E-6$;
- 3378 cicli se $\mathbf{x} = 1E-3$;
- 2544 cicli se $\mathbf{x} = 1$;
- 3378 cicli se $\mathbf{x} = 10$;
- 3423 cicli se $\mathbf{x} = 1E2$;
- 3423 cicli se $\mathbf{x} = 1E3$;
- 3378 cicli se $\mathbf{x} = 1E4$;
- 3423 cicli se $\mathbf{x} = 1E5$.

Note: Analizzando **log10** e **log10f** si trova che se $\mathbf{x} \leq 0$ viene ritornato 0 e **errno** viene settata a **EDOM**.

Aderenza allo standard C:
Ok

12.3.20 modf, modff

Sintassi:

double modf(double x, double *nptr);

float modff(float x, float *nptr);

Implementazione:
La **modf** è implementata sia in **modf.asm** che in **modfp.asm**; la **modff** è implementata sia in **modff.asm** che in **modffp.asm**.

Descrizione:
Spezzano l'argomento **x** in due parti: una intera n e una frazionaria f, in modo tale che si abbia $|f| < 1.0$ e $f + n = \mathbf{x}$. Sia f che n avranno lo stesso segno di **x**. Il valore ritornato è f, mentre n viene memorizzata all'indirizzo **nptr**.

Tempi di esecuzione:
La **modff** impiega sempre 18 cicli. La **modf** impiega:

- 111 cicli se $\mathbf{x} = -1$;
- 31 cicli se $\mathbf{x} = 0$;
- 31 cicli se $\mathbf{x} = 1E-6$;
- 31 cicli se $\mathbf{x} = 1E-3$;

- 111 cicli se $\mathbf{x} \geq 1$.

Note: Le **modf** e **modff** non generano alcun errore. Le funzioni in **modf.asm** e **modff.asm** vengono chiamate quando **nptr** punta in data memory, mentre vengono invocate le implementazioni in **modfp.asm** e **modffp.asm** se **nptr** punta in program memory. Le differenze tra le versioni riguardano solo il modo in cui viene memorizzato il valore in **nptr**. Le funzioni **modf** e **modff** non devono essere confuse con **fmod** e **fmodf**, che calcolano il resto della divisione di un numero floating point per un altro.

Aderenza allo standard C:
Ok

12.3.21 pow, powf

Sintassi:

double pow(double x, double y);

float powf(float x, float y);

Implementazione:
La **pow** è implementata in **pow.asm**, mentre la **powf** è implementata in **powf.asm**.

Descrizione:
Calcolano $\mathbf{x}^{\mathbf{y}}$. Il valore di ritorno è:

- $\mathbf{x}^{\mathbf{y}}$ se $\mathbf{x} > 0$ oppure se $\mathbf{x} < 0$ e **y** è un intero;
- 1 se $\mathbf{x} \neq 0$ e $\mathbf{y} = 0$;
- 0 se $\mathbf{x} = 0$ e $\mathbf{y} \leq 0$ oppure se $\mathbf{x} < 0$ e **y** non è un intero.

Tempi di esecuzione:
La **powf** impiega:

- 149 cicli se $\mathbf{x} > 0$;
- 162 cicli se $\mathbf{x} < 0$ e **y** è un intero;
- 149 cicli se $\mathbf{x} \neq 0$ e $\mathbf{y} = 0$;
- 42 cicli se $\mathbf{x} = 0$ e $\mathbf{y} \leq 0$;
- 45 cicli se $\mathbf{x} < 0$ e **y** non è un intero.

Note: In caso il risultato non possa essere rappresentato, perché **x** è negativa e **y** non è un numero intero esatto, oppure perché $\mathbf{x} = 0$ e $\mathbf{y} \leq 0$ si ha un domain error: viene ritornato 0 e **errno** viene settata a **EDOM**.

In caso di overflow o underflow si ha un range error: viene ritornato 0 e **errno** viene settata a **ERANGE**.

La documentazione a pagina L-91 è incompleta.

Aderenza allo standard C:
Ok

12.3.22 sqrt, sqrtf

Sintassi:
double sqrt(double x);

float sqrtf(float x);

Implementazione:
La **sqrt** è implementata in **sqrt.asm**, mentre la **sqrtf** è implementata in **sqrtf.asm**.

Descrizione:
Calcolano la radice quadrata di **x**. Un domain error si verifica se $\mathbf{x} < 0$.

Tempi di esecuzione:
La **sqrtf** impiega:

- 15 cicli se $\mathbf{x} < 0$;
- 12 cicli se $\mathbf{x} = 0$;
- 31 cicli se $\mathbf{x} > 0$.

Note: Se $\mathbf{x} < 0$ si genera un domain error: **errno** viene settata a **EDOM** e viene ritornato 0.

La funzione sfrutta l'algoritmo di Newton-Raphson, con seed ottenuto usando l'istruzione macchina `RSQRTS`.

Aderenza allo standard C:
Ok

12.3.23 rsqrt, rsqrtf

Sintassi:
double rsqrt(double x);

float rsqrtf(float x);

Implementazione:
La **rsqrt** è implementata in **rsqrt.asm**, mentre la **rsqrtf** è implementata in **rsqrtf.asm**.

Descrizione:

Calcolano il reciproco della radice quadrata di **x**. Un domain error si verifica se $\mathbf{x} < 0$.

Tempi di esecuzione:

La **rsqrtf** impiega:

- 15 cicli se $\mathbf{x} < 0$;
- 14 cicli se $\mathbf{x} = 0$;
- 29 cicli se $\mathbf{x} > 0$.

La **rsqrt** impiega:

- 62 cicli se $\mathbf{x} = -1$;
- 35 cicli se $\mathbf{x} = 0$;
- 62 cicli se $\mathbf{x} \geq 1E-6$.

Note: Se $\mathbf{x} < 0$ si genera un domain error: **errno** viene settata a **EDOM** e viene ritornato 0. Se $\mathbf{x} = 0$ viene ritornato 0 senza generare alcun errore.

La funzione sfrutta l'algoritmo di Newton-Raphson, con seed ottenuto usando l'istruzione macchina `RSQRTS`.

Questa funzione non era richiesta dallo standard C e non viene menzionata nel C RUNTIME LIBRARY MANUAL.

Aderenza allo standard C:

Non applicabile

12.3.24 ceil, ceilf

Sintassi:

double ceil(double x);

float ceilf(float x);

Implementazione:

La **ceil** è implementata in **ceil.asm**, mentre la **ceilf** è implementata in **ceilf.asm**.

Descrizione:

Calcolano e ritornano il minimo numero intero, espresso sotto forma di floating point, che non sia minore di **x**. Eseguono l'equivalente di un arrotondamento verso $+\infty$.

Tempi di esecuzione:

La **ceilf** impiega:

- 18 cicli se **x** è un numero intero;
- 21 cicli se **x** non è un numero intero.

La **ceil** impiega:

- 32 cicli se $\mathbf{x} = -1$;
- 32 cicli se $\mathbf{x} = 0$;
- 211 cicli se $\mathbf{x} = 1E-6$;
- 211 cicli se $\mathbf{x} = 1E-3$;
- 109 cicli se $\mathbf{x} \geq 1$.

Note: Le funzioni non generano alcun tipo di errore.

Aderenza allo standard C:
Ok

12.3.25 floor, floorf

Sintassi:
double floor(double x);

float floorf(float x);

Implementazione:
La **floor** è implementata in **floor.asm**, mentre la **floorf** è implementata in **floorf.asm**.

Descrizione:
Calcolano e ritornano il massimo numero intero, espresso sotto forma di floating point, che non sia maggiore di **x**. Eseguono l'equivalente di un arrotondamento verso $-\infty$.

Tempi di esecuzione:
La **floorf** impiega:

- 18 cicli se **x** è un numero intero;
- 20 cicli se **x** non è un numero intero.

La **floor** impiega:

- 112 cicli se $\mathbf{x} = -1$;
- 35 cicli se $\mathbf{x} \geq 0$.

Note: Le funzioni non generano alcun tipo di errore.

Aderenza allo standard C:
Ok

12.3.26 fabs, fabsf

Sintassi:

double fabs(double x);

float fabsf(float x);

Implementazione:

La **fabs** è implementata in **fabs.asm**, mentre la **fabsf** è implementata in **fabsf.asm**.

Descrizione:

Calcolano e ritornano il valore assoluto di **x**.

Tempi di esecuzione:

La funzione **fabsf** fa parte dell'insieme di funzioni built-in, quindi viene solitamente espansa inline dal compilatore. Il compilatore inserisce l'istruzione macchina `ABS`, che impiega un unico ciclo di clock.

Note: Le funzioni non generano alcun tipo di errore.

Aderenza allo standard C:

Ok

12.3.27 fmod, fmodf

Sintassi:

double fmod(double x, double y);

float fmodf(float x, float y);

Implementazione:

La **fmod** è implementata in **fmod.asm**, mentre la **fmodf** è implementata in **fmodf.asm**.

Descrizione:

Calcolano il resto floating point dell'operazione $\mathbf{x}/\mathbf{y}$. Si tratta del corrispondente floating point del resto intero che si ha in una divisione intera tra due interi. In particolare le funzioni restituiscono un numero reale f tale che:

- f ha lo stesso segno di **x**;
- $|f| < |\mathbf{y}|$;
- esiste un intero k tale che $k * \mathbf{y} + f = \mathbf{x}$.

Tempi di esecuzione:

La **fmodf** impiega:

- 14 cicli se $\mathbf{y} = 0$;
- 16 cicli se $\mathbf{x} = 0$ e $\mathbf{y} \neq 0$;
- 45 cicli altrimenti.

Note: Secondo lo standard se $\mathbf{y} = 0$ l'implementazione può generare un domain error oppure ritornare 0. L'implementazione in esame ritorna 0; non viene generato alcun tipo di errore.

Queste funzioni si possono usare per eseguire la riduzione di una variabile ad un subrange durante i calcoli che coinvolgono funzioni periodiche. Sono più accurate e sicure della sottrazione diretta di un multiplo intero dell'intervallo. Le funzioni **fmod** e **fmodf** non devono essere confuse con le **modf** e **modff**, che estraggono le parti frazionaria ed intera da un numero floating point.

Aderenza allo standard C:
Ok

12.4 Osservazioni

12.4.1 ceil, floor e modf

Queste funzioni consentono di manipolare in diversi modi la parte frazionaria di un numero floating point. Usare queste funzioni è molto più sicuro che passare attraverso conversioni in interi perché esse possono gestire valori floating point arbitrari senza causare overflow. La **floor** arrotonda verso $-\infty$, mentre **ceil** arrotonda verso $+\infty$. Per arrotondare all'intero più vicino un valore floating point **x** arbitrario si può scrivere:

```
y = (x < 0) ? ceil(x - 0.5) : floor(x + 0.5)
```

12.4.2 cosh, sinh e tanh

Bisognerebbe invocare sempre queste funzioni piuttosto che calcolarle indirettamente tramite la loro definizione in termini di esponenziali, poiché il risultato che si ottiene è più accurato.

12.4.3 Altre funzioni non standard

La Analog Devices ha aggiunto nella runtime library altre funzioni e definizioni, oltre a quelle obbligatorie richieste dallo standard C, che vengono dichiarate in **asm_sprt.h**, **macros.h**, **matrix.h**, **filters.h**, **trans.h**, **21020.h**, **21060.h**, **complex.h**, **dspc.h**, **stats.h**, **comm.h**, **def21020.h** e **def21060.h** Per approfondimenti, consultare lo C RUNTIME LIBRARY MANUAL.

12.5 Le routine interne

Sono state implementate diverse routine interne di supporto, utilizzate sia dalle funzioni matematiche vere e proprie che dal compilatore durante le operazioni floating point che non sono riducibili ad una singola istruzione macchina. Non tutte queste routine possono essere chiamate direttamente da codice C. Di seguito vengono elencate tutte, file per file, con una breve descrizione. Più avanti si analizzeranno le routine base usate per svolgere le quattro operazioni matematiche elementari in double precision e le routine di conversione tra single precision e double precision.

12.5.1 mth_sprt.asm

Nel file **mth_sprt.asm** sono implementate molte delle funzioni fondamentali che operano sui floating point. Nei rispettivi commenti vengono indicati i registri d'ingresso, quelli d'uscita, quelli alterati e il tempo di esecuzione. Le routine implementano parzialmente la gestione di NaN, infinità e denormalizzati. In questo file sono contenute:

- **___float_divide**, che esegue una divisione in single precision, ritornando il quoziente;
- **___lib_ldtof** e **___lib_dtof**, che coindicono; il loro compito è eseguire la conversione dai tipi **double** e **long double** verso il tipo **float**, rispettivamente;
- **___lib_ftold** e **___lib_ftod**, che coincidono; il loro compito è eseguire la conversione dal tipo **float** verso i tipi **double** e **long double**, rispettivamente;
- **___lib_dmult** che esegue un prodotto in double precision;
- **___lib_dadd** e **___lib_dsub**, che sono quasi coincidenti; eseguono somma e differenza in double precision, rispettivamente;
- **___lib_ldtoi** e **___lib_dtoi**, che coincidono; il loro compito è eseguire la conversione dai tipi **double** e **long double** verso il tipo **int**, rispettivamente;
- **___lib_ldtoui** e **___lib_dtoui**, che coincidono; eseguono la conversione dai tipi **double** e **long double** verso il tipo **unsigned int**, rispettivamente;
- **___lib_itold** e **___lib_itod**, che coincidono; il loro compito è eseguire la conversion dal tipo **int** verso i tipi **double** e **long double**, rispettivamente;
- **___lib_ldtox** e **___lib_dtox**, che coincidono; eseguono la conversione dai tipi **double** e **long double** verso il tipo fixed point, rispettivamente;

- **___lib_xtold** e **___lib_xtod**, che coincidono; il loro compito è eseguire la conversione dal tipo fixed point verso i tipi **double** e **long double**, rispettivamente.

12.5.2 subf.asm

Nel file **subf.asm** sono contenute le funzioni **___cmpdf**, **___subdf3**, **___ldsub** e **___dsub**, che coincidono e chiamano **___lib_dsub**; sono usate dal compilatore per i confronti e le sottrazioni tra **double** e **long double**.

12.5.3 flt_sprt.asm

Nel file **flt_sprt.asm** sono contenute:

- **___truncdfsf2**, **___ldtof** e **___dtof**, che coincidono e chiamano la routine **___lib_dtof**;
- **___divsf3** e **___fdiv**, che sono coincidenti e chiamano la **___float_divide**;
- **___floatunssisf2**, **___utof** e **___ultof**, che coincidono; eseguono la conversione dai tipi **unsigned** e **unsigned long** verso il tipo **float**.

12.5.4 dbl_sprt.asm

Nel file **dbl_sprt.asm** sono contenute:

- **___muldf3**, **___ldmult** e **___dmult**, che coincidono e chiamano la routine **___lib_dmult**;
- **___divdf3**, **___lddiv** e **___ddiv**, che coincidono; il loro compito è eseguire la divisione tra numeri in double precision;
- **___adddf3**, **___ldadd** e **___dadd**, che coincidono e chiamano la routine **___lib_dadd**;
- **___extendsfdf2**, **___ftold** e **___ftod**, che coincidono e chiamano la routine **___lib_ftod**;
- **___xtold** e **___xtod**, che coincidono e chiamano **___lib_xtod**;
- **___floatsidf**, **___itold** e **___itod**, che coincidono e chiamano **___lib_itod**.

12.5.5 fxd_sprt.asm

Nel file **fxd_sprt.asm** sono contenute:

- **___xdiv**, che chiama **___float_divide**;
- **___ldtox** e **___dtox**, che coincidono e chiamano **___lib_dtox**.

13 Analisi di alcune routine base

Osservazione importante: le versioni assemblate delle routine, presenti nella runtime library, sono leggermente diverse dalle versioni fornite sotto forma di codice sorgente. Quelle disponibili nei sorgenti eseguono anche alcune verifiche aggiuntive sulla categoria degli operandi (normalizzati, denormalizzati, *NaN* oppure infinità) e possono dare luogo a ulteriori diramazioni nel flusso d'esecuzione degli algoritmi.

Per evidenziare bene le differenze si è fatto un lavoro di reverse engineering sul codice assemblato. Nei sorgenti mostrati in seguito sono state inserite alcune speciali righe di commento, per identificare la provenienza dei vari blocchi di codice, in accordo con queste regole:

- `/-----/` indica che le righe successive fanno parte solo della routine assemblata;
- `/+++++/` indica che le righe successive fanno parte solo della routine sorgente;
- `/=====/` indica che le righe successive fanno parte di entrambe.

Nei flow chart presentati nelle pagine seguenti i simboli in colore verde sono relativi a codice e diramazioni presenti solo nelle routine assemblate, mentre i simboli in colore rosso si riferiscono solamente alle routine sorgenti.

13.1 ___float_divide

Esegue una divisione floating point single precision, usando il metodo di arrotondamento corrente del processore (selezionato dal bit TRUNC del registro MODE1). Utilizza l'istruzione `RECIPS` come seed per un algoritmo di Newton-Raphson a due iterazioni, già visto precedentemente.

```
/* Calling Parameters               */
/*     F12 contains the Denominator */
/*     F11 contains the value 2.0   */
/*     F7  contains the Numerator   */
/* Return Registers                 */
/*     F7  contains the Quotient    */
/* Altered Registers                */
/*     F0, F7, F12                  */
/* Cycle Count                      */
/*     8 cycles                     */
```

```
___float_divide:
    F0=RECIPS F12;          /* Get 4 bit seed R0=1/D */
    F12=F0*F12;             /* D' = D*R0 */
    F7=F0*F7, F0=F11-F12; /* F0=R1=2-D', F7=N*R0 */
    F12=F0*F12;             /* F12=D'=D'*R1 */
    F7=F0*F7, F0=F11-F12; /* F7=N*R0*R1, F0=R2=2-D' */
    RTS (DB), F12=F0*F12; /* F12=D'=D'*R2 */
    F7=F0*F7, F0=F11-F12; /* F7=N*R0*R1*R2, F0=R3=2-D' */
    F7=F0*F7;               /* F12=N*R0*R1*R2*R3 */
```

13.1.1 Note

Si tratta di una routine interna non chiamabile direttamente da codice C. Nel file **flt_sprt.asm** è stata definita la seguente funzione, chiamabile da codice C con prototipo **float __fdiv(float N, float D)**, che chiama a sua volta la **___float_divide**.

```
___divsf3:
___fdiv:
    put(R7);
    put(R11);
    R7=PASS R4, R2 = MODE1;
    BIT CLR MODE1 65536;    /* set to 40-bit mode */
    CALL (PC, ___float_divide) (DB);
    R12=PASS R8, FETCH_RETURN
    F11=2.0;
restore_state:
    F0=RND F7, MODE1 = R2;
    get(R11,1);
    get(R7,2);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
```

13.2 ___lib_ldtof, ___lib_dtof

Esegue una conversione floating point da double precision a single precision utilizzando il modo di arrotondamento corrente del processore.

```
/* Calling Parameters                              */
/*     R0  contains upper word (|s|EXP|mantissa)  */
```

```
/*     R1  contains lower word (LSBs of mantissa) */
/* Return Registers                               */
/*     F0  contains floating point result         */
/* Altered Registers                              */
/*     F0, F1, F2, F3                             */

___lib_ldtof:
___lib_dtof:
/+++++/
    R3=FEXT R0 BY 20:11(SE); /* Look for NaN,INF,Denorm */
    IF SZ R0=R0-R0,R1=M5;    /* Flush Denorm */
    R3=R3+1;                 /* zero=>not finite */
    R3=FEXT R0 BY 0:20;      /* Get msb mantissa */
    IF EQ JUMP (PC,ret_SP_INFNAN)(DB),R3=R3 OR R1;
/=====/
    F2=1.0e0;                /* Need a one to start */
/-----/
    R3=FEXT R0 BY 0:20;
/=====/
    R2=R2 OR LSHIFT R3 BY 3; /* Shift into float mantissa */
    R3=FEXT R1 BY 9:23;      /* Extract lower mantissa to R3 */
    R1=-43;                  /* Relative exponent */
    F3=FLOAT R3 BY R1;       /* Place into float format */
    F1=F2+F3;                /* Determine mantissa of result */
    R2=FEXT R0 BY 20:11;     /* Extract biased exponent */
    R3=D_EXP_BIAS;           /* Get bias of double exponent */
/-----/
    RTS (DB), F1=F1 COPYSIGN F0;
/+++++/
    R0=PASS R0;              /* What's sign of DP? */
    IF LT R1=BSET R1 BY 31;  /* Copy sign of result */
    RTS (DB);
/=====/
    R2=R2-R3;                /* Get unbiased exponent */
    F0=SCALB F1 BY R2;       /* Set exponent of result */
/+++++/
ret_SP_INFNAN:
    R1=0x7F800000;           /* ALU NE=>NaN */
    IF NE R1=R1+1;           /* Mantissa non-zero?=>NaN */
    RTS(DB),R0=PASS R0;      /* Sign of input */
    IF LT R1=BSET R1 BY 31;  /* Copied to output */
```

```
R0=R1;
```

13.2.1 Funzionamento

La figura 8 riassume il flusso dell'algoritmo; la fase di elaborazione svolge questi compiti (la coppia [R0:R1] contiene l'operando in double precision):

1. inizializza `F2=1.0`;
2. tramite le istruzioni `R3=FEXT R0 BY 0:20` e `R2=R2 OR LSHIFT R3 BY 3` estrae i 20 bit MSB della mantissa dell'operando, che stanno nel suo MSW alle posizioni [19,0], e li pone in F2 allineandoli al MSB della sua mantissa;
3. le istruzioni `R3=FEXT R1 BY 9:23` e `F3=FLOAT R3 BY -43` estraggono i successivi 23 bit dalla mantissa dell'operando, che stanno nel suo LSW alle posizioni [31:9], e convertono il numero intero che si ottiene in un floating point, utilizzando il coefficiente di scalatura -43 per tenere conto della posizione della virgola;
4. il numero `F1=F2+F3` è un single precision che contiene la mantissa esatta del risultato; restano da sistemare il segno e l'esponente;
5. `F1=F1 COPYSIGN F0` oppure la coppia di istruzioni `RO=PASS R0` e `IF LT R1=BSET R1 BY 31` modificano il segno di F1 per renderlo concorde con quello dell'operando;
6. `F2=FEXT R0 BY 20:11` estrae l'esponente biased dall'operando; l'istruzione `R2=R2-D_EXP_BIAS` sottrae il bias e `F0=SCALB F1 BY R2` scala l'esponente di F1 della quantità indicata da R2 e pone il risultato in F0.

13.2.2 Note

Questa è una routine interna non chiamabile direttamente da codice C. Nel file **flt_sprt.asm** è stata definita la seguente funzione, chiamabile da codice C con prototipi **float __ldtof(long double x)** e **float __dtof(double x)**, che chiama a sua volta la **___lib_ldtof**.

```
___truncdfsf2:
___ldtof:
___dtof:
    put(R3);
    reads(R4,1);
    reads(R8,2);
```

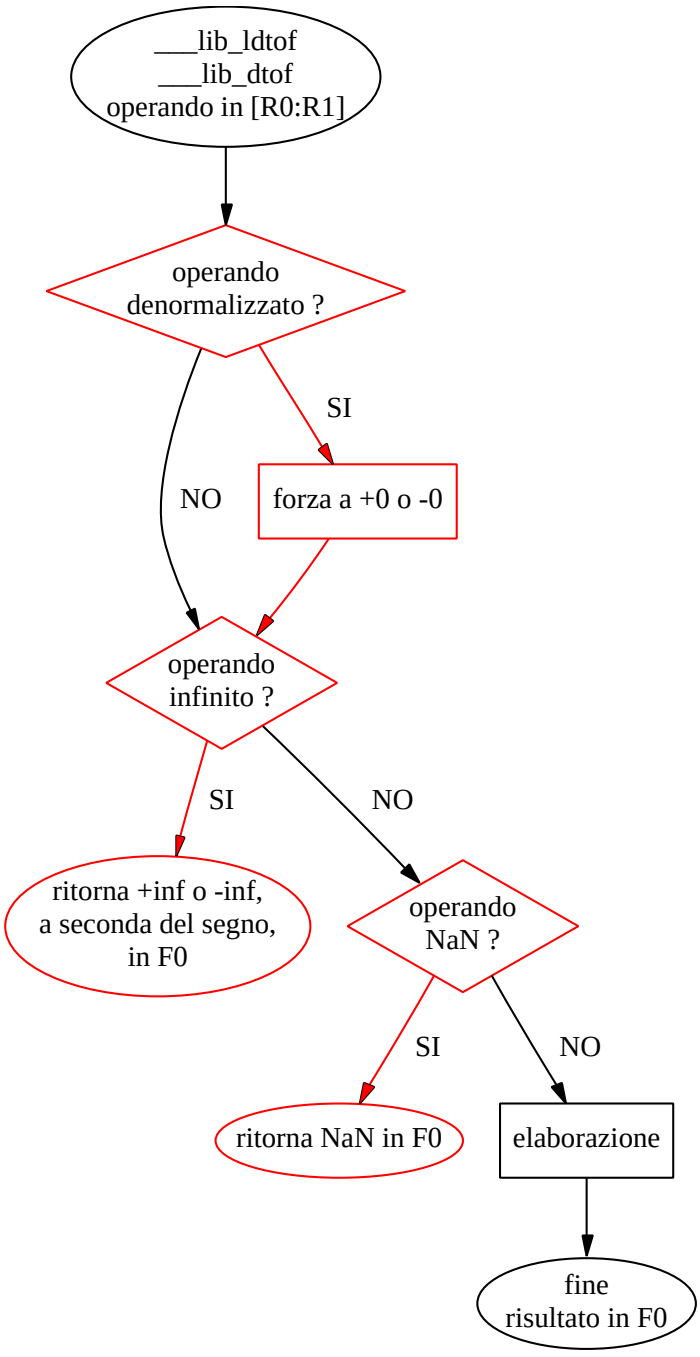

Figura 8: Conversione double precision verso single precision.

```
    CALL (PC, ___lib_dtof) (DB);
    R0=PASS R4;
    R1=PASS R8, FETCH_RETURN
    get(R3,1);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
```

13.3 ___lib_ftold, ___lib_ftod

Esegue una conversione floating point da single precision a double precision.

```
/* Calling Parameters                                        */
/*     F0 contains floating point value                      */
/* Return Registers                                          */
/*     R0 contains MSW of double value (|s|EXP|mantissa)     */
/*     R1 contains LSW of double value (lower mantissa)      */
/* Altered Registers                                         */
/*     F0, F1, F2, F3                                        */

___lib_ftold:
___lib_ftod:
/+++++/
    R3=FEXT R0 BY 23:8(SE);   /* Check for ZERO, INF or NaN */
    IF SZ JUMP (PC,ret_DP_zero)(DB); /* It's 0 or denorm */
    R3=R3+1;                  /* Check for INF or NaN */
    R1=FEXT R0 BY 0:23;       /* Zero=>INF */
    IF EQ JUMP (PC,ret_DP_INFNAN)(DB);
/=====/
    R3=LOGB F0;               /* Get exponent of float */
    R2=-R3;                   /* Prepare to zero exponent */
    F0=SCALB F0 BY R2, R1=R0; /* Zero exponent */
    IF EQ RTS (DB);     /* Input == 0, so output=0 */
    R2=D_EXP_BIAS;      /* Get bias for a double */
    R3=R2+R3;           /* Compute biased exp for double */
    R1=LSHIFT R3 BY 20; /* Position exponent bits */
/+++++/
    R0=PASS R0;
    IF LT R1=BSET R1 BY 31; /* Place sign bit */
/-----/
    F1=F1 COPYSIGN F0
```

```
/=====/
    F0=ABS F0;          /* Clear sign bit */
    R2=FEXT R0 BY 3:20; /* Place upper mantissa bits */
    R1=R1 OR R2;        /* Place into double word */
    R2=0x7FFFFFF8;      /* Mask out everything but mant */
    R2=R0 AND R2;       /* Lower mantissa bits only */
    R3=51;              /* Relative exponent of lower mant */
    RTS (DB), F0=F0-F2; /* Compute remainder */
    R1=FIX F0 BY R3, R0=R1; /* Place lower mantissa */
    R1=LSHIFT R1 BY 1;  /* Upshift lower mant to unsigned */
/+++++/
ret_DP_INFNAN:
    RTS (DB),R0=PASS R0;    /* Enter with R1 0=>INF, !0=>NaN */
    R0=0x7FF00000;          /* Max exponent */
    IF LT R0=BSET R0 BY 31; /* restore sign */
ret_DP_zero:
    RTS (DB),R1=R1-R1;      /* Return a signed DP zero */
    R2=PASS R0, R0=R1;      /* Make +0 */
    IF LT R0=BSET R0 BY 31; /* want -0 instead */
```

13.3.1 Funzionamento

La figura 9 riassume il flusso dell'algoritmo; la fase di elaborazione svolge questi compiti (F0 contiene l'operando in single precision):

1. `R3=LOGB F0` estrae l'esponente unbiased dell'operando;
2. `F0=SCALB F0 BY -R3, R1=R0` scala l'esponente, rendendo F0 un numero con esponente nullo;
3. se F0 è 0.0 l'algoritmo termina restituendo [F0:F1] = 0.0;
4. `R3=R3+D_EXP_BIAS` calcola l'esponente biased per il risultato in double precision;
5. `R1=LSHIFT R3 BY 20` posiziona l'esponente nella nella MSW del risultato;
6. `F1=F1 COPYSIGN F0` oppure la coppia di istruzioni `R0=PASS R0` e `IF LT R1=BSET R1 BY 31` modificano il segno di F1 per renderlo concorde a quello dell'operando;
7. `R2=FEXT R0 BY 3:20` e `R1=R1 OR R2` estraggono la parte della mantissa che deve andare nel MSW del risultato, la fondono con l'esponente e mettono tutto nel MSW del risultato;

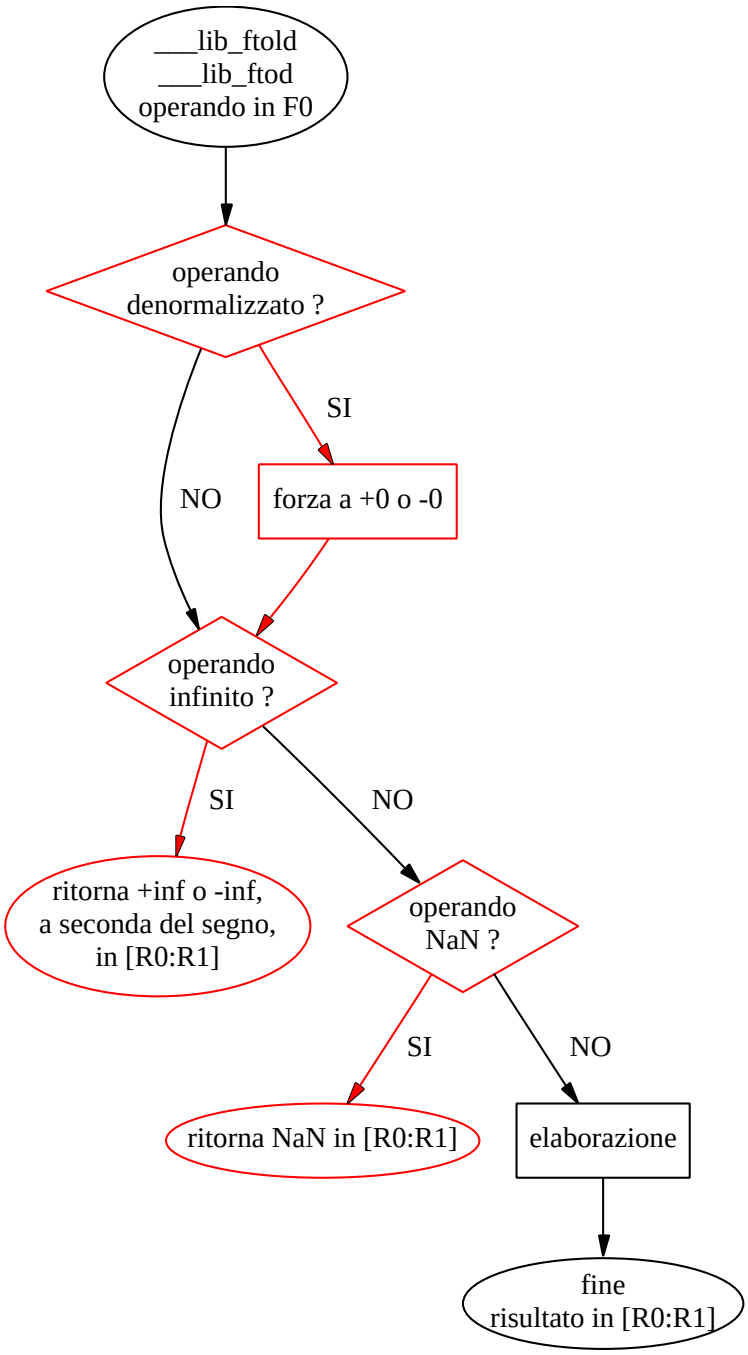

Figura 9: Conversione single precision verso double precision.

8. `R2=R0 AND 0x7FFFFFF8` conserva solo i 20 MSB della mantissa e l'esponente dell'operando;

9. `F0=F0-F2` calcola la parte di mantissa mancante dal risultato, che andrà nella sua LSW;

10. `R1=FIX F0 BY 51, R0=R1` e `R1=LSHIFT R1 BY 1` preparano la parte di mantissa che deve andare nel LSW del risultato e la pongono in R1;

11. il risultato in double precision è in [R0:R1].

13.3.2 Note

Questa è una routine interna non chiamabile direttamente da codice C. Nel file **dbl_sprt.asm** è stata definita la seguente funzione, chiamabile da codice C con prototipi **long double __ftold(float x)** e **double __ftod(float x)**, che chiama a sua volta la **___lib_ftod**.

```
___extendsfdf2:
___ftold:
___ftod:
    CALL (PC, ___lib_ftod) (DB);
    R0=PASS R4, put(R3);
    FETCH_RETURN
    get(R3,1);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
```

13.4 ___divdf3, ___lddiv, ___ddiv

Esegue una divisione tra numeri in double precision, producendo un risultato in double precision. Si tratta di una routine chiamabile direttamente da codice C, con i prototipi **long double __lddiv(long double num, long double denom)** e **double __ddiv(double num, double denom)**, quest'ultimo da usarsi solo se il tipo **double** è in double precision.

```
restore_state2:
    get(R15,1);
    get(R14,2);
    get(R13,3);
    get(R11,4);
```

```
    get(R10,5);
    get(R9,6);
    get(R7,7);
    get(R6,8);
    get(R5,9);
    get(R2,11);
    FETCH_RETURN
    MR0F=R2, get(R2,12);
    MR1F=R2, get(R2,13);
    MR2F=R2, get(R3,10);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
___divdf3:
___lddiv:
___ddiv:
    R2=MR2F;
    R2=MR1F,put(R2);
    R2=MR0F,put(R2);
    put(R2);
    put(R3);
    put(R5);
    put(R6);
    put(R7);
    put(R9);
    put(R10);
    put(R11);
    put(R13);
    put(R14);
    put(R15);
    reads(R4,1);
    reads(R8,2);
    reads(R12,3);
    R7=PASS R4, R11=R8;       /* Hold these values for later */
    R14=PASS R12, reads(R15,4); /* Read lower word of denom */
    R10=D_EXP_BIAS;           /* Take recip of 0 exp number */
    IF LT R10=BSET R10 BY 31; /* Copy SIGN of input value */
    R12=FEXT R14 BY 20:11;    /* Extract biased exponent */
    R0=FEXT R14 BY 0:20;      /* Copy mantissa */
    CALL (PC, ___lib_dtof) (DB); /* Convert to flt for recip */
    R0=R0 OR LSHIFT R10 BY 20;   /* Place sign+exponent */
```

```
R1=R15;                      /* Get LSBs of mantissa */
R10=D_EXP_BIAS;
CALL (PC, ___lib_ftod) (DB); /* Convert back to dbl */
F0=RECIPS F0;         /* Get flt recip */
R12=R12-R10;          /* Compute unbiased exponent */
R10=FEXT R0 BY 20:11; /* Extract exponent */
R10=R10-R12;
IF LT JUMP (PC, underflow);
R9=D_MAX_BIASED_EXP;
COMP(R10,R9);
IF GT JUMP (PC, overflow);
R12=FEXT R0 BY 0:20;          /* Copy mantissa */
R0=PASS R0;
IF LT R12=BSET R12 BY 31;     /* Copy sign */
CALL (PC, ___lib_dmult) (DB); /* Compute D'=D*R0 */
R12=R12 OR LSHIFT R10 BY 20;
R13=R1;
R8=D_TWO_MSW;                 /* Prepare for divide */
R9=D_TWO_LSW;                 /* by loading a 2.0 */
CALL (PC, ___lib_dmult) (DB); /* Compute N*R0 */
R10=PASS R0, R14=R7;          /* Read MSW of num */
R11=PASS R1, R15=R11;         /* Read LSW of num */
R14=PASS R8, R15=R9;          /* Load 2.0 */
CALL (PC, ___lib_dsub) (DB);  /* Compute R1=2-D' */
R6=PASS R0, R7=R1;            /* Save N*R0 */
R12=PASS R10, R13=R11;        /* Load D' */
CALL (PC, ___lib_dmult) (DB); /* Compute N*R0*R1 */
R14=PASS R6, R15=R7;          /* Load N*R0 */
R12=PASS R0, R13=R1;          /* Load R1 */
CALL (PC, ___lib_dmult) (DB); /* Compute D'=D'*R1 */
R6=PASS R0, R7=R1;            /* Save N*R0*R1 */
R14=PASS R10, R15=R11;        /* Load D' */
R14=PASS R8, R15=R9;          /* Load 2.0 */
CALL (PC, ___lib_dsub) (DB);  /* Compute R2=2-D' */
R10=PASS R0, R11=R1;          /* Save D' */
R12=PASS R0, R13=R1;          /* Move D' */
CALL (PC, ___lib_dmult) (DB); /* Compute N*R0*R1*R2 */
R14=PASS R6, R15=R7;          /* Load N*R0*R1 */
R12=PASS R0, R13=R1;          /* Load R2 */
CALL (PC, ___lib_dmult) (DB); /* Compute D'=D'*R2 */
R6=PASS R0, R7=R1;            /* Save N*R0*R1*R2 */
```

```
    R14=PASS R10, R15=R11;        /* Load D' */
    R14=PASS R8, R15=R9;          /* Load 2.0 */
    CALL (PC, ___lib_dsub) (DB);  /* Compute R3=2-D' */
    R10=PASS R0, R11=R1;          /* Save D' */
    R12=PASS R0, R13=R1;          /* Move D' */
    CALL (PC, ___lib_dmult) (DB); /* Compute N*R0*R1*R2*R3 */
    R14=PASS R6, R15=R7;          /* Load N*R0*R1*R2 */
    R12=PASS R0, R13=R1;          /* Load R3 */
    CALL (PC, ___lib_dmult) (DB); /* Compute D'=D'*R3 */
    R6=PASS R0, R7=R1;            /* Save N*R0*R1*R2*R3 */
    R14=PASS R10, R15=R11;        /* Load D' */
    R14=PASS R8, R15=R9;          /* Load 2.0 */
    CALL (PC, ___lib_dsub) (DB);  /* Compute R4=2-D' */
    R10=PASS R0, R11=R1;          /* Save D' */
    R12=PASS R0, R13=R1;          /* Move D' */
    CALL (PC, ___lib_dmult) (DB); /* Comp. N*R0*R1*R2*R3*R4 */
    R14=PASS R6, R15=R7;          /* Load N*R0*R1*R2*R3 */
    R12=PASS R0, R13=R1;          /* Load R4 */
    CALL (PC, ___lib_dmult) (DB); /* Compute D'=D'*R4 */
    R6=PASS R0, R7=R1;            /* Save N*R0*R1*R2*R3*R4 */
    R14=PASS R10, R15=R11;        /* Load D' */
    R14=PASS R8, R15=R9;          /* Load 2.0 */
    CALL (PC, ___lib_dsub) (DB);  /* Compute R5=2-D' */
    R10=PASS R0, R11=R1;          /* Save D' */
    R12=PASS R0, R13=R1;          /* Move D' */
    CALL (PC, ___lib_dmult) (DB); /* C. N*R0*R1*R2*R3*R4*R5 */
    R14=PASS R6, R15=R7;          /* Load N*R0*R1*R2*R3*R4 */
    R12=PASS R0, R13=R1;          /* Load R5 */
    /*Should be 96-bits accurate*/
overflow:       /* All jump to restore_state2 */
underflow:      /* Also set HUGEVAL */
    JUMP (PC, restore_state2);  /* Also set 0 */
```

13.4.1 Funzionamento

La figura 10 riassume il flusso dell'algoritmo. La fase di calcolo del seed esegue questi passi:

1. le istruzioni

   ```
   R10=D_EXP_BIAS;
   ```

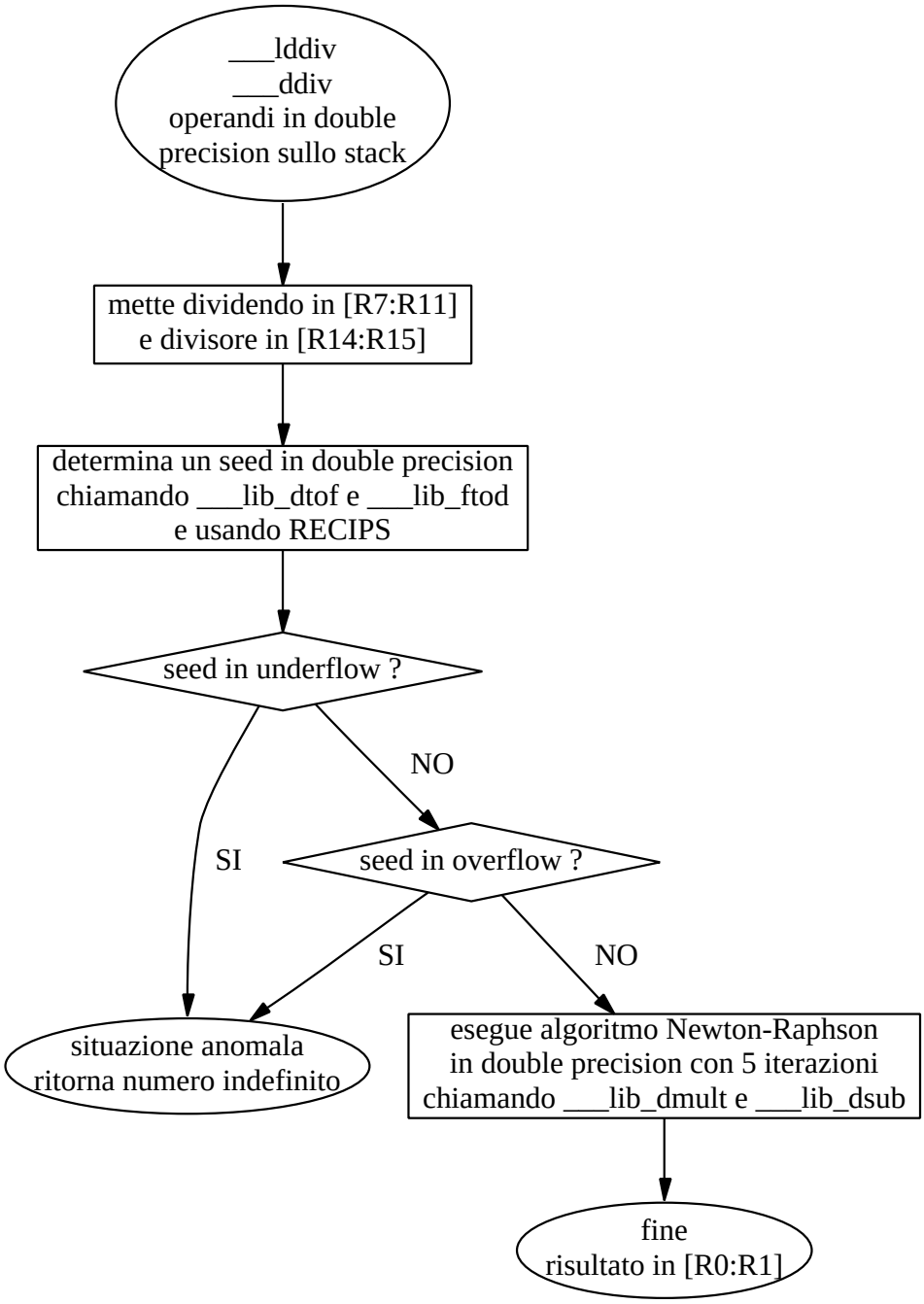

Figura 10: Divisione in double precision.

```
IF LT R10=BSET R10 BY 31;
R0=FEXT R14 BY 0:20;
R0=R0 OR LSHIFT R10 BY 20;
R1=R15;
```

costruiscono un numero double precision in [R0:R1], indentico al divisore [R14:R15] a meno del valore dell'esponente unbiased, che viene fissato a 0;

2. una chiamata a `___lib_dtof` converte l'operando in [R0:R1] in un single precision e mette il risultato in F0;

3. l'istruzione `F0=RECIPS F0` produce in F0 un'approssimazione del reciproco di F0;

4. una chiamata a `___lib_ftod` converte l'approssimazione in un double precision che viene restituito in [R0:R1]; questo numero è il seed a meno del valore dell'esponente, che viene calcolato esattamente più avanti;

5. le istruzioni `R12=FEXT R14 BY 20:11` e `R12=R12-D_EXP_BIAS` producono in R2 l'esponente unbiased del divisore;

6. `R10=FEXT R0 BY 20:11` calcola l'esponente biased di [R0:R1];

7. l'istruzione `R10=R10-R12` fornisce in R10 l'esponente biased corretto per il seed;

8. le istruzioni

```
R12=FEXT R0 BY 0:20;
R0=PASS R0;
IF LT R12=BSET R12 BY 31;
R12=R12 OR LSHIFT R10 BY 20;
R13=R1;
```

costruiscono in [R12:R13] il seed esatto; [R7:R11] contiene il dividendo, mentre [R14:R15] contiene il divisore;

9. le istruzioni

```
IF LT JUMP (PC, underflow);
COMP(R10,D_MAX_BIASED_EXP);
IF GT JUMP (PC, overflow);
```

controllano eventuali underflow o overflow del seed, nel qual caso la routine ritorna in modo anomalo con un valore indefinito; i commenti che indicano che viene ritornato **HUGE_VAL** in caso di overflow o 0 in caso di underflow non rispecchiano la reale implementazione.

La fase successiva consiste in un algoritmo tipo Newton-Raphson in double precision con 5 iterazioni. L'algoritmo è leggermente diverso da quello visto nella descrizione dell'istruzione `RECIPS` e usato nella **___float_divide**. Calcolare il quoziente $Q = N/D$ è equivalente a calcolare il valore

$$Q = \frac{N \cdot R_0 \cdot R_1 \cdots R_n}{D \cdot R_0 \cdot R_1 \cdots R_n}$$

Se, fissato n, si scelgono i coefficienti $R_0 \ldots R_n$ in modo tale che il denominatore approssimi abbastanza bene il numero 1 allora il quoziente si può approssimare come:

$$Q \simeq N \cdot R_0 \cdot R_1 \cdots R_n$$

Il procedimento è illustrato in dettaglio nel testo HWANG KAI AND FAYE A. BRIGGS, COMPUTER ARCHITECTURE & PARALLEL PROCESSING, MCGRAW-HILL, 1984. Le regole per il calcolo dei coefficienti sono:

$$R_i = \begin{cases} \texttt{RECIPS}(D) & \text{se } i = 0 \\ 2 - D_{i-1} & \text{se } i = 1 \ldots n \end{cases}$$

$$D_i = \begin{cases} D \cdot R_0 & \text{se } i = 0 \\ D_{i-1} \cdot R_i & \text{se } i = 1 \ldots n \end{cases}$$

Il numero n di iterazioni determina direttamente l'accuratezza del risultato. In questa implementazione vale 5 e l'algoritmo esegue questi passi:

1. calcola $R_0 = \texttt{RECIPS}(D)$;
2. calcola $D_0 = D \cdot R_0$;
3. calcola e salva $N \cdot R_0$;
4. calcola $R_1 = 2 - D_0$;
5. calcola e salva $N \cdot R_0 \cdot R_1$;
6. calcola $D_1 = D_0 \cdot R_1$;

7. calcola $R_2 = 2 - D_1$;
8. calcola e salva $N \cdot R_0 \cdot R_1 \cdot R_2$;
9. calcola $D_2 = D_1 \cdot R_2$;
10. calcola $R_3 = 2 - D_2$;
11. calcola e salva $N \cdot R_0 \cdot R_1 \cdot R_2 \cdot R_3$;
12. calcola $D_3 = D_2 \cdot R_3$;
13. calcola $R_4 = 2 - D_3$;
14. calcola e salva $N \cdot R_0 \cdot R_1 \cdot R_2 \cdot R_3 \cdot R_4$;
15. calcola $D_4 = D_3 \cdot R_4$;
16. calcola $R_5 = 2 - D_4$;
17. calcola $Q = N \cdot R_0 \cdot R_1 \cdot R_2 \cdot R_3 \cdot R_4 \cdot R_5$.

Il risultato viene costruito in modo incrementale, attraverso il prodotto del numeratore con i coefficienti $R_0, \ldots R_5$, man mano che questi vengono ricavati.

13.5 ___lib_dadd, ___lib_dsub

Esegue una addizione o una sottrazione in double precision, producendo un risultato in double precision.

```
/* Calling Parameters                    */
/*     R12 = |s|EXP|mantissa| of X      */
/*     R13 = mantissa of X               */
/*     R14 = |s|EXP|mantissa| of Y      */
/*     R15 = mantissa of Y               */
/* Return registers                      */
/*     R2 = |s|EXP|mantissa of (X op Y) */
/*     R1 = mantissa of (X op Y)         */
/* Altered registers                     */
/*     R0, R1, R2, R3, R4, R5            */

/+++++/
ADD_INFNAN0:
    R2=FEXT R12 BY 0:20; /* abnormal exit code */
```

```
    R2=R2 OR R13;        /* R0:R1 are INF|NaN */
    IF NE RTS;           /* NaN! */

    /* We've got INF + ??? */

    R2=FEXT R14 BY 20:11; /* Is other not finite */
    RTS,COMP(R2,R5);      /* other finite=>return R12:R13 */
    IF EQ R0=R14;         /* not finite */
    IF EQ R1=R15;         /* => use R14:R15 */
/=====/
___lib_dsub:
    R12=BTGL R12 BY 31;   /* Invert sign of X */
___lib_dadd:
    R3=FEXT R12 BY 20:11;     /* Get biased exponent of X */
/+++++/
    IF SZ R12=R12-R12,R13=M5; /* flush any denorm */
    R5=0x7FF;                 /* Max exponent */
    COMP(R3,R5),R1=R13;
    IF EQ JUMP(PC,ADD_INFNAN0),R0=PASS R12;
/=====/
    R2=FEXT R14 BY 20:11;     /* Get biased exp of Y */
/+++++/
    IF SZ R14=R14-R14,R15=M5; /* Flush any denorm */
    COMP(R2,R5),R1=R15;       /* Look for INF|NaN */
    IF EQ RTS, R0=PASS R14;   /* Other wasn't, so use this */
/=====/
    R1=R2-R3;                    /* Test for greater value */
    IF GE JUMP (PC, shift_x_op); /* Y bigger, so shift X */
swap_ops:
    R14=PASS R12, R12=R14; /* Swap MSWs */
    R15=PASS R13, R13=R15; /* Swap LSWs */
    R1=-R1;
shift_x_op:
    R1=-R1;              /* Invert difference for shift */
    R3=LSHIFT R13 BY R1; /* Downshift X mantissa by diff */
    R4=FEXT R12 BY 0:20; /* Extract MSBs of X mantissa */
    R4=BSET R4 BY 20;    /* Make implict 1 explicit */
    R2=LSHIFT R4 BY R1;  /* Downshift X MS mantissa by diff */
    R0=32;             /* Need to OR in LSBs of MS mantissa */
    R1=R1+R0;          /* Compute new shift value */
    R3=R3 OR LSHIFT R4 BY R1; /* Place upper bits */
```

```
    R12=PASS R12, R4=dm_0;    /* Test for sign of X */
    IF GE JUMP (PC, extract_y_mant); /* X is >= 0, figure Y */
invert_x:
    R3=-R3;           /* Invert LSW of shifted X mant */
    NOP;              /* CHIP ANOMALY: Late carry */
    R2=R4-R2+CI-1;    /* Invert MSW of shifted X mant */
extract_y_mant:
    R0=FEXT R14 BY 0:20;       /* Extract MSBs of mantissa */
    R0=BSET R0 BY 20;          /* Make implicit 1 explicit! */
    R14=PASS R14, R1=R15;      /* Test for sign of Y */
    IF GE JUMP (PC, add_core); /* Y is positive */
invert_y:
    R1=-R1;           /* Invert LSW of Y mantissa */
    NOP;              /* CHIP ANOMALY: Late carry */
    R0=R4-R0+CI-1;    /* Invert MSW of Y mantissa */
add_core:
    R1=R1+R3, R5=dm_0; /* Add LSWs, Sign of result */
    NOP;              /* CHIP ANOMALY: Late carry */
    R0=R0+R2+CI;      /* Add MSWs */
normalize:
    IF GE JUMP (PC, check_MSW);
invert_result:
    R5=0x80000000;    /* Sign of result */
    R1=-R1;           /* Invert LSW */
    NOP;              /* CHIP ANOMALY: Late carry */
    R0=R4-R0+CI-1;    /* Invert MSW */
check_MSW:
    IF EQ JUMP (PC, small_result); /* All zeros in MSW */
    R2=EXP R0;          /* Check for redundant bits */
    R2=-R2;             /* Invert for shift */
    R0=LSHIFT R0 BY R2; /* Move into position */
    R4=LSHIFT R1 BY R2; /* Move LSBs for later */
    R3=32;              /* Shift to lower word */
    R3=R2-R3;
    R0=R0 OR LSHIFT R1 BY R3;
    R1=FEXT R14 BY 20:11;     /* Extract exponent */
    R4=R1-R2, R1=R4;          /* Add normalizing amount */
    R2=10;                    /* Scale for 11.21 number */
    R4=R4+R2;                 /* Adjust final exponent */
    R0=BCLR R0 BY 30;         /* Make explicit 1 implicit! */
    R2=LSHIFT R0 BY -10;      /* Place MSW mantissa */
```

```
    R1=LSHIFT R1 BY -10;      /* Place LSW mantissa */
    R1=R1 OR LSHIFT R0 BY 22; /* Place MS of LSW mantissa */
place_exp_sign:
    RTS (DB);
    R2=R2 OR LSHIFT R4 BY 20; /* Place exponent */
/-----/
    F2=F2 COPYSIGN F5;
/+++++/
    R2=R2 OR R5;              /* Place sign bit */
/=====/
small_result:
    R1=LSHIFT R1 BY -1;       /* Shift unsigned value down */
    R2=R2-R2, R4=R0;          /* Set both R2 and R4 to zero */
    R1=PASS R1;               /* Test for zero result */
    IF EQ JUMP (PC, place_exp_sign); /* 0 result, set sign */
    R2=EXP R1;                /* Test for redundant bits */
    R2=-R2;                   /* Invert for shift */
    R0=R0 OR LSHIFT R1 BY R2; /* Normalize word */
    R1=FEXT R14 BY 20:11;     /* Extract exponent */
    R4=R1-R2, R1=R4;          /* Add normalizing amount */
    R2=-21;                   /* Add scale for small number */
    R4=R4+R2;                 /* Adjust final exponent */
    JUMP (PC, place_exp_sign) (DB);  /* Complete result */
    R0=BCLR R0 BY 30;         /* Make explicit 1 implicit! */
    R2=LSHIFT R0 BY -10;      /* Place MSW mantissa */
```

13.5.1 Funzionamento

Le sottrazioni vengono ricondotte ad addizioni, come illustrato nella figura 11, che riassume anche i controlli effettuati sugli operandi prima di entrare nell'algoritmo di addizione vero e proprio.

In figura 12 è mostrata la seconda parte dell'algoritmo, che si occupa di determinare la mantissa in double precision del risultato. Ecco i passi principali:

1. se necessario si scambiano gli operandi floating point per fare in modo che Y sia il più grande, in valore assoluto;

2. da entrambi gli operandi si estraggono le mantisse a 52 bit;

3. si aggiungono alle mantisse i bit impliciti, ottenendo due numeri interi [R0:R1] e [R2:R3] di 53 bit ciascuno;

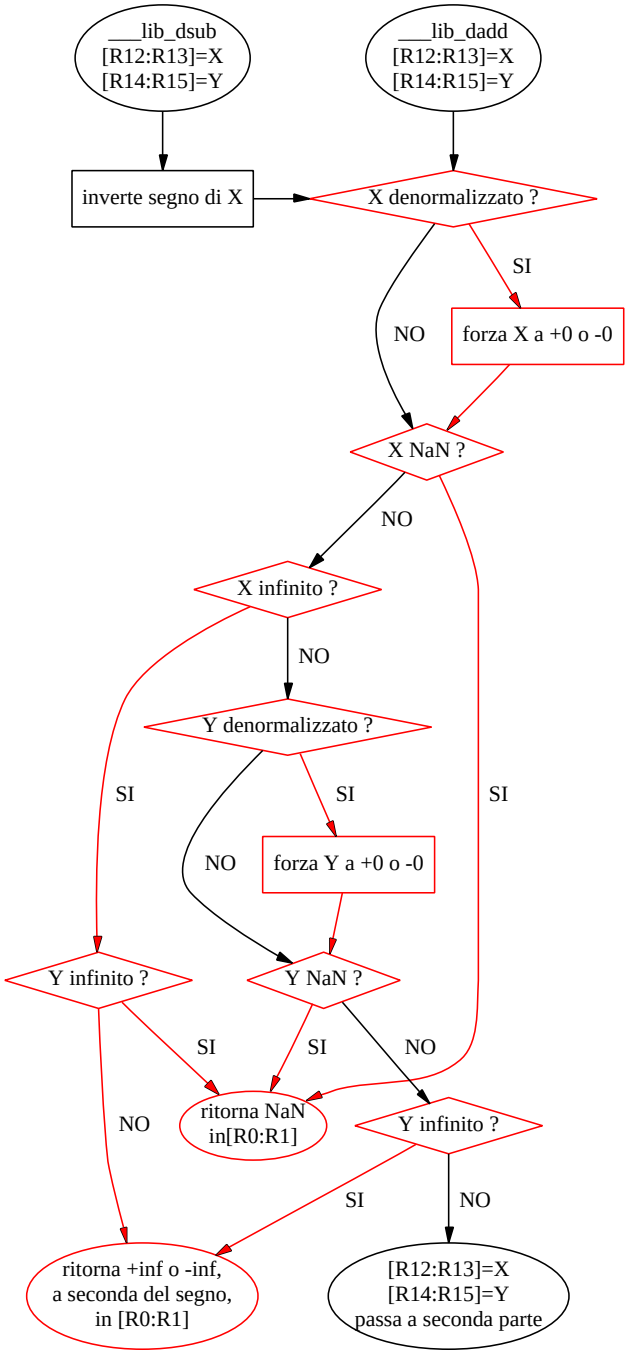

Figura 11: Prima parte addizione in double precision.

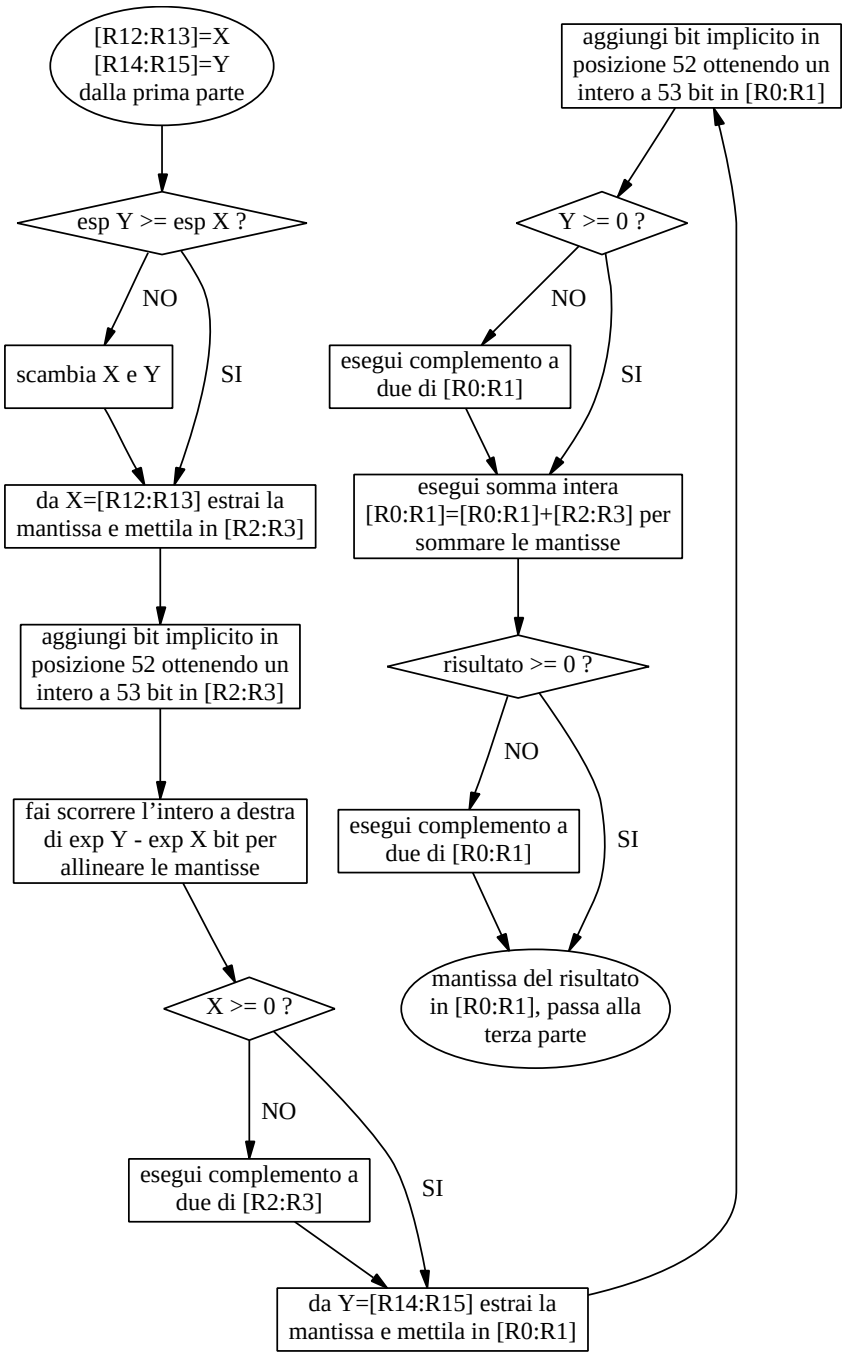

Figura 12: Seconda parte addizione in double precision.

4. si fa scorrere verso destra la mantissa [R2:R3], che corrisponde all'operando che era più piccolo in valore assoluto, allineandola in questo modo con la mantissa proveniente dall'altro operando;

5. se necessario, attraverso un complemento a due si cambiano i segni dei due interi per fare in modo che rispecchino i segni dei numeri floating point da cui provengono;

6. si sommano;

7. se il risultato è negativo se ne cambia il segno e si tiene traccia di questa situazione tramite `R5=0x80000000`, per poter cambiare il segno del risultato floating point alla fine dell'algoritmo.

In figura 13 è mostrata la terza parte dell'algoritmo, che si occupa di normalizzare la mantissa, calcolare l'esponente e costruire il risultato in double precision. Di seguito sono riportati i passi fondamentali:

1. il flusso d'esecuzione si divide in due: se il MSW del risultato dell'addizione (cioè R0) è nullo si salta alla label `small_result:`;

2. in entrambi i casi si calcola il numero di bit non significativi presenti in R0 oppure in R1 tramite l'istruzione `R2=EXP R0` oppure `R2=EXP R1`; si tratta del numero di bit nulli consecutivi che si hanno partendo dal MSB del registro;

3. questo numero viene utilizzato per calcolare di quanti bit vanno fatti scorrere verso sinistra i registri precedenti allo scopo di normalizzare le mantisse, allineando i vari bit alle posizioni dove, nel formato double precision, vanno i bit della mantissa;

4. successivamente si eliminano i bit impliciti, resi espliciti nella seconda parte dell'algoritmo;

5. si calcola l'esponente del risultato, tenendo conto del valore dell'esponente di Y e del numero di shift che sono stati eseguiti per normalizzare la mantissa;

6. l'ultimo passo consiste nel settare il segno del risultato tramite l'istruzione `F2=F2 COPYSIGN F5` oppure `R2=R2 OR R5`, in modo concorde al segno del risultato della somma tra mantisse eseguita nella seconda parte dell'algoritmo.

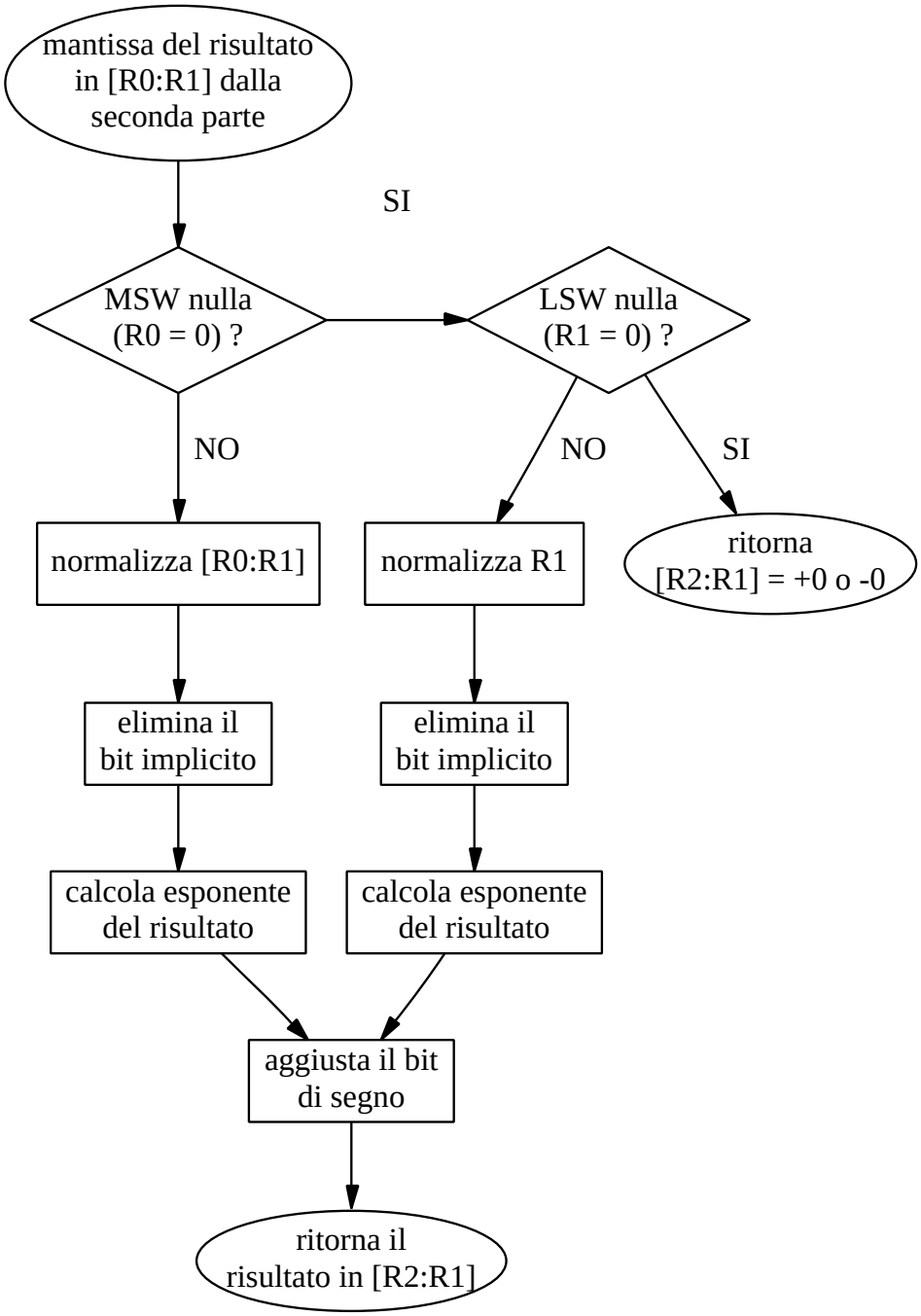

Figura 13: Terza parte addizione in double precision.

13.5.2 Note

Questa è una routine interna non chiamabile direttamente da codice C. Nel file **dbl_sprt.asm** è stata definita la seguente funzione, chiamabile da codice C con prototipi **long double __ldadd(long double x, long double y)** e **double __dadd(double x, double y)**, che chiama a sua volta la **___lib_dadd**.

```
___adddf3:
___ldadd:
___dadd:
    put(R3);
    put(R5);
    put(R6);
    put(R7);
    put(R9);
    put(R10);
    put(R11);
    put(R13);
    put(R14);
    put(R15);
    reads(R14,1);
    reads(R15,2);
    CALL (PC, ___lib_dadd) (DB);
    reads(R12,3);
    reads(R13,4); /* Read Y LSW */
    get(R15,1);
    get(R14,2);
    get(R13,3);
    get(R11,4);
    get(R10,5);
    get(R9,6);
    get(R7,7);
    get(R6,8);
    get(R5,9);
    FETCH_RETURN
    get(R3,10);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
```

Nel file **subf.asm** è stata definita la seguente funzione, chiamabile da codice C con prototipi **long double __ldsub(long double x, long double y)** e **double**

__dsub(double x, double y), che chiama a sua volta la **___lib_dsub**.

```
___cmpdf:
___subdf3:
___ldsub:
___dsub:
    put(R3);
    put(R5);
    put(R13);
    put(R14);
    put(R15);
    reads(R14,1);
    reads(R15,2);
    CALL (PC, ___lib_dsub) (DB); /* Compute X-Y */
    reads(R12,3);
    reads(R13,4); /* Read Y LSW */
restore_state:
    FETCH_RETURN
    get(R15,1);
    get(R14,2);
    get(R13,3);
    get(R5,4);
    get(R3,5);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
```

13.6 ___lib_dmult

Esegue una moltiplicazione tra numeri in double precision, producendo un risultato in double precision.

```
/* Calling Parameters                */
/*     R12 = |s|EXP|mantissa| of X   */
/*     R13 = mantissa of X           */
/*     R14 = |s|EXP|mantissa| of Y   */
/*     R15 = mantissa of Y           */
/* Return registers                  */
/*     R0 = |s|EXP|mantissa of (X*Y) */
/*     R1 = mantissa of (X*Y)        */
/* Altered registers                 */
```

```
/*     R0, R1, R2, R3                */
/*     MRF                           */

___lib_dmult:
/-----/
    R3=BCLR R12 BY 31;
    R3=R3 OR R13;
    IF EQ JUMP (PC,zero_result); /* primo operando nullo */
    R3=BCLR R14 BY 31;
    R3=R3 OR R15;
    IF EQ JUMP (PC,zero_result); /* secondo operando nullo */
/=====/
    R0=LSHIFT R15 BY 11;         /* Move LSW up, zero lower */
    R1=LSHIFT R13 BY 11;
    MRF=R0*R1 (UUF);
    R2=MR1F;
    MR0F=R2;
    R2=MR2F;
    MR1F=R2;
    R2=LSHIFT R15 by -21;        /* Move LSW into place*/
    R3=LSHIFT R13 BY -21;
    R2=R2 OR LSHIFT R14 BY 11;
    R3=R3 OR LSHIFT R12 BY 11;   /* Move MSW into place 1.31 */
    R2=BSET R2 BY 31;            /* Set hidden bit */
    R3=BSET R3 BY 31;
    MRF=MRF+R0*R3 (UUF);
    MRF=MRF+R1*R2 (UUF);
    R0=MR1F;
    MR0F=R0;
    R0=MR2F;
    MR1F=R0;
    MRF=MRF+R2*R3 (UUF);
    R0=0x00000001;       /* We need to do rounding here */
    R1=0x00000100;       /* so we are going to add 1/2LSB */
    MRF=MRF+R0*R1 (UUF); /* to the number we just created */
/-----/
    R0=LSHIFT R14 BY -20;
    R1=LSHIFT R12 BY -20;
    R0=BCLR R0 BY 11;
    R1=BCLR R1 BY 11;
/+++++/
```

```
    R0=FEXT R14 BY 20:11;   /* Extract exponents */
    IF SZ JUMP(PC,ret_DP_zero),R0=R12 XOR R14;
    R1=FEXT R12 BY 20:11;
    IF SZ JUMP(PC,ret_DP_zero),R0=R12 XOR R14;
    R3=0x7FF;               /* Max exponent value */
    COMP(R0,R3);
    IF EQ JUMP(PC,MPY_DP_INFNAN0);
    COMP(R1,R3);
    IF EQ JUMP(PC,MPY_DP_INFNAN1)(DB);
/=====/
    R0=R0+R1;               /* Add exponents */
    R1=D_EXP_BIAS;          /* Remove bias of exponent */
    R0=R0-R1;
    IF MV JUMP (PC, large_prod)(DB); /* Product overflow */

/*
okay, now a little thinking...   we've got a number which
should be 1<=x<2.  If it is, then we just clear it's most
significant bit and shift it 11 bits to the right.  This is
equivalent to shifting 11 bits and then clearing bit 20.  Now
if the number isn't 1<=x<2, we know that the number x<1.
We've got to figure out how many bits we need to shift it to
the left in order to get it 1<=x<2.  If we use the exp
function on it, it returns the opposite of (the number of
leading zeroes -1), call this "y".  So, if we shift the number
(-y)+1 to the left, then we'll be "normalized".  Thus, if we
shift 11 to the right (=-11 to the left) then (-y)+1 to the
left, we've ended up shifting -11-y+1 to the left.

So, watch this... */

    R1=MR1F;            /* Get MSW of product */
    R3=-11;             /* Base shift if no overflow */
    R1=PASS R1, R2=M5;  /* is MSb of x set */
    IF GE R2=EXP R1;    /* if not then we'll need to adjust */
    IF GE R2=R2-1;      /* to get y-1 =(-y)+1 */
    R3=R3-R2;           /* Compute actual shift value */
    R1=LSHIFT R1 BY R3; /* Shift MSW into place */
    R1=BCLR R1 BY 20;   /* Remove hidden bit */
    R0=R0+R2;       /* Adjust exp for non-normalized number */
    R0=R0+1;        /* Increment exponent */
```

```
    R1=R1 OR LSHIFT R0 BY 20;   /* Move exponent into place */
    R0=R14 XOR R12;             /* Compute sign bit */
    BTST R0 BY 31;              /* Test sign bit */
    IF NOT SZ R1=BSET R1 BY 31; /* Set sign bit of result */
    R0=MR0F;
    R1=LSHIFT R0 BY R3, R0=R1;  /* Shift LSW */
    R2=32;
    RTS (DB), R3=R3+R2;
    R2=MR1F;
    R1=R1 OR LSHIFT R2 BY R3;
/+++++/
MPY_DP_INFNAN0:
    R2=FEXT R14 BY 0:20;   /* Check mantissa */
    R2=R2 OR R15,R0=R14;   /* 0=>INF */
    IF NE RTS, R1=PASS R2; /* Return NaN */
    COMP(R0,R1);           /* One is INF, is other INF|NaN? */
    IF NE JUMP(PC,ret_DP_INF),R2=R14 XOR R12; /* figure sign */

    /* If we fall thru, either both are INF, or 2nd is
     * NaN.  Just return properly signed 2nd */

MPY_DP_INFNAN1:
    R0=BCLR R12 BY 31;      /* clear sign */
    RTS(DB), R1=PASS R13;
    R3=R12 XOR R14;
    IF LT R0=BSET R0 BY 31; /* INF or NaN */
ret_DP_INF:
    RTS (DB),R1=R1-R1;      /* Signed INF */
    R2=PASS R2,R0=R14;
    IF LT R0=BSET R0 BY 31;
/=====/
    /* WE NEED SOMETHING HERE, BUT WHAT? */
large_prod:
    RTS;
/-----/
zero_result:
    RTS (DB), R1=R1-R1;
    R3=R12 XOR R14, R0=R1;
    IF LT R0=BSET R0 BY 31;
/+++++/
ret_DP_zero:
```

```
RTS (DB),R1=R1-R1;      /* Return a signed DP zero */
R2=PASS R0, R0=R1;      /*Make +0 */
IF LT R0=BSET R0 BY 31; /*want -0 instead*/
```

13.6.1 Funzionamento

La figura 14 riporta la prima parte dell'algoritmo; analizziamo il funzionamento di ciascuno dei tre blocchi rettangolari.

1. Il primo blocco rappresenta le istruzioni

   ```
   R0=LSHIFT R15 BY 11;
   R2=LSHIFT R15 BY -21;
   R2=R2 OR LSHIFT R14 BY 11;
   R2=BSET R2 BY 31;

   R1=LSHIFT R13 BY 11;
   R3=LSHIFT R13 BY -21;
   R3=R3 OR LSHIFT R12 BY 11;
   R3=BSET R3 BY 31;
   ```

   che congiuntamente estraggono le due mantisse dagli operandi in double precision X=[R12:R13] e Y=[R14:R15] e le pongono in [R2:R0] e [R3:R1], allineandole a sinistra e rendendo esplicito il bit implicito; la situazione è illustrata dalla figura 15.

2. Il secondo blocco rappresenta le istruzioni

   ```
   MRF=R0*R1(UUF); /* esegue prodotto tra le LSW */

   R2=MR1F;        /* scala */
   MR0F=R2;
   R2=MR2F;
   MR1F=R2;

   MRF=MRF+R0*R3 (UUF); /* prodotti misti tra LSW e MSW */
   MRF=MRF+R1*R2 (UUF);

   R0=MR1F;        /* scala */
   MR0F=R0;
   R0=MF2F;
   MR1F=R0;
   ```

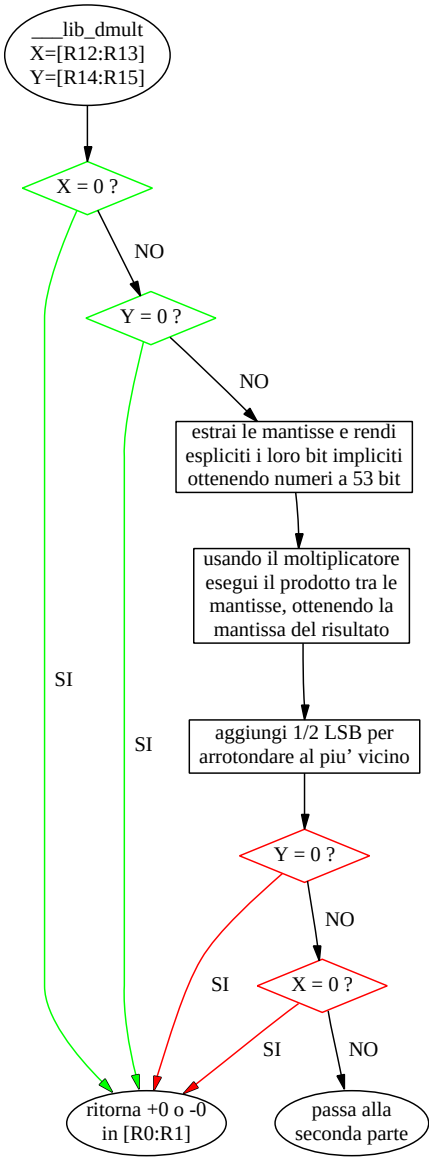

Figura 14: Prima parte moltiplicazione in double precision.

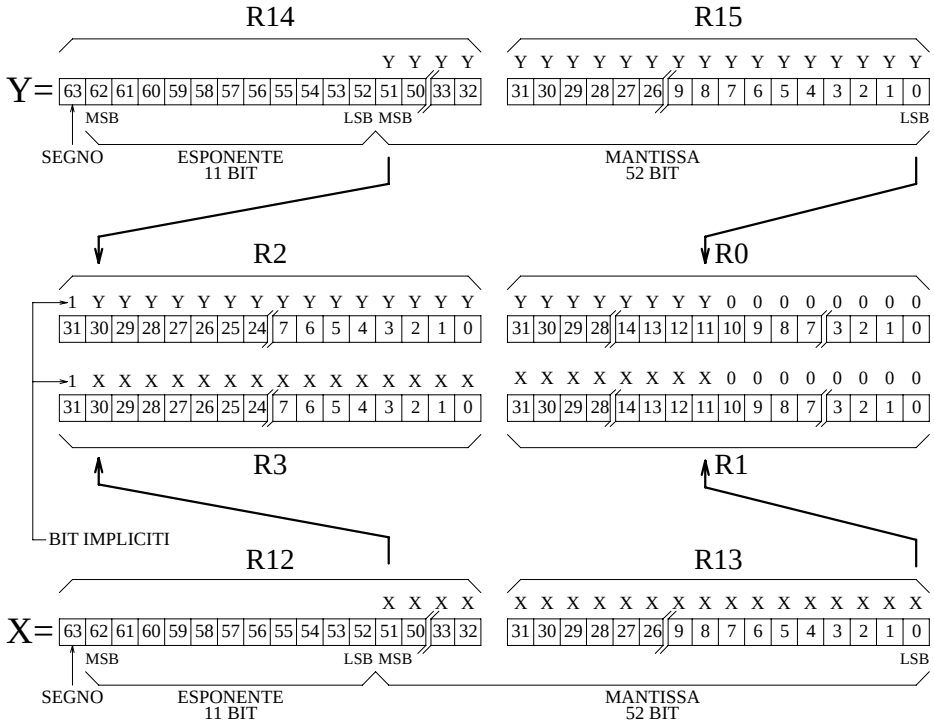

Figura 15: Estrazione delle mantisse.

```
MRF=MRF+R2*R3 (UUF); /* prodotto tra le due MSW */
```

che realizzano il prodotto tra le due mantisse utilizzando il moltipicatore, spezzandole in più parti; si tratta di un prodotto frazionario (indicato dal modificatore `(UUF)` delle istruzioni) poiché le mantisse sono quantità frazionarie.

3. Il terzo blocco è composto dalle istruzioni

```
R0=0x00000001;
R1=0x00000100;
MRF=MRF+R0*R1 (UUF);
```

che aggiungono 1/2 LSB alla somma delle mantisse, per arrotondarla al più vicino.

La figura 16 illustra la seconda parte dell'algoritmo. Analizziamo il funzionamento dei tre blocchi rettangolari.

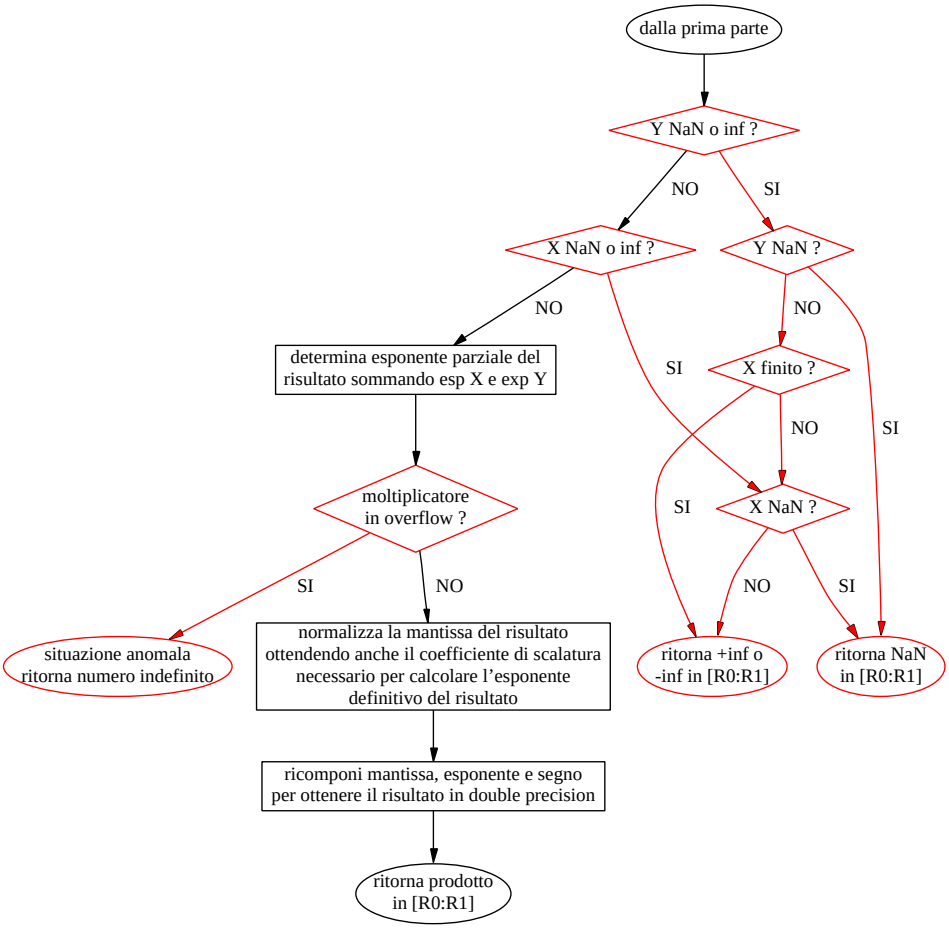

Figura 16: Seconda parte moltiplicazione in double precision.

1. Il primo blocco determina il valore dell'esponente del risultato; si tratta di un valore parziale perché la mantissa non è stata ancora normalizzata e questa operazione può richiedere un aggiustamento dell'esponente.

```
R0=FEXT R14 BY 20:11;
R1=FEXT R12 BY 20:11;
R0=R0+R1
```

2. Il secondo blocco si occupa di normalizzare la mantissa, cioè shiftarla verso sinistra quanto basta per fare in modo che il suo primo bit significativo venga a coincidere con il suo MSB; questo fornisce anche il coefficiente di scalatura da aggiungere all'esponente parziale per ottenere l'esponente definitivo del risultato.

3. L'ultimo blocco ricompone mantissa, esponente e segno per formare il risultato in double precision.

13.6.2 Note

Questa è una routine interna non chiamabile direttamente da codice C. Nel file **dbl_sprt.asm** è stata definita la seguente funzione, chiamabile da codice C con prototipi **long double __ldmult(long double x, long double y)** e **double __dmult(double x, double y)**, che chiama la **___lib_dmult**.

```
___muldf3:
___ldmult:
___dmult:
    R2=MR2F;
    R2=MR1F,put(R2);
    R2=MR0F,put(R2);
    put(R2);
    put(R3);
    put(R5);
    put(R6);
    put(R7);
    put(R9);
    put(R10);
    put(R11);
    put(R13);
    put(R14);
    put(R15);
    reads(R14,1);
```

```
    reads(R15,2);
    CALL (PC, ___lib_dmult) (DB);
    reads(R12,3);
    reads(R13,4); /* Read Y mantissa */
restore_state2:
    get(R15,1);
    get(R14,2);
    get(R13,3);
    get(R11,4);
    get(R10,5);
    get(R9,6);
    get(R7,7);
    get(R6,8);
    get(R5,9);
    get(R2,11);
    FETCH_RETURN
    MR0F=R2, get(R2,12);
    MR1F=R2, get(R2,13);
    MR2F=R2, get(R3,10);
    RETURN (DB);
    RESTORE_STACK
    RESTORE_FRAME
```

13.7 Tempi di esecuzione

La tabella 14 riporta i tempi di esecuzione delle routine analizzate precedentemente. Il compilatore chiama automaticamente le funzioni **___ldtof** e **___ftold** quando deve eseguire una conversione implicita tra tipi floating point distinti. Chiama **___lddiv**, **___ldadd**, **___ldsub** e **___ldmult** quando deve eseguire delle operazioni elementari su numeri in double precision. Per quanto riguarda la divisione tra single precision, il compilatore inserisce sempre la funzione **___float_divide** in-line piuttosto che generare una chiamata a **___fdiv**.

Nome funzione	Tempo di esecuzione
___float_divide	8
___fdiv	25
___ldtof	26
___ftold	28
___lddiv	1030
___ldadd	84
___ldsub	78
___ldmult	98

Tabella 14: Tempi di esecuzione.

14 Bibliografia

Per la famiglia di DSP:

- *ADSP-2106X EZ-KIT Lite reference manual*, Analog Devices;
- *ADSP-21000 Family application handbook*, volume 1, Analog Devices;
- *ADSP-21000 Family assembler tools & simulator manual*, Analog Devices;
- *ADSP-21000 Family C runtime manual*, Analog Devices;
- *ADSP-21000 Family C tool manual*, Analog Devices;
- *ADSP-2106X SHARC user's manual*, Analog Devices.

Per la rappresentazione floating point:

- *Computer organization*, V. Carl Hamacher, McGraw-Hill;
- *Struttura e progetto dei calcolatori*, D. A. Patterson, J. L. Hennessy, Zanichelli;
- *Il microprocessore 80386/80387*, Stephen P. Morse, Eric J. Isaacson, Douglas J. Albert, Tecniche Nuove;
- *Dispense del corso di Elettronica Dei Sistemi Digitali*, Prof.ssa Alessandra Flammini, CLUB.

Per il linguaggio C:

- *The C Programming Language*, standard ISO 9899-1990;

- *C, a reference manual*, Samuel P. Harbison, Guy L. Steele, Prentice Hall;
- *The standard C library*, P.J. Plauger, Prentice Hall.

Inoltre:

- *The TMS320C4X user's guide*, Texas Instruments;
- *The scientist and engineer's guide to digital signal processing*, Steven W. Smith, California Technical Publishing;
- *Software Manual for the Elementary Functions*, William J. Cody, William M. Waite, Prentice Hall.