UNVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell' Informazione

Tesi di laurea Triennale in Ingegneria Elettronica

STUDIO SUGLI ALGORITMI PER BLOCCHI CIRCUITALI ARITMETICI E DIVISORI

Relatore: **Prof. Andrea Cester**

Laureando: **Luca Boscolo Galazzo**

Matricola: **563691**

Anno Accademico 2010/2011

Indice

1 Introduzione 3

1.1 Standard IEEE 754 4

1.2 Divisione matematica 6

2 Divisione a cifre ricorrenti 7

2.1 Norestoring Division e Restoring Division 7

2.2 Conventional Restoring 9

2.2.1 Esempio 10

2.2.2 Esempio 12

2.3 Divisore SRT 15

2.3.1 Esempio 16

2.4 Divisione High Radix 18

3 Divisione con Funzioni Iterative 21

3.1 Newton – Raphson 21

3.1.1 Esempio 24

3.2 Goldschmidt 25

4 Very High Radix 27

5 Algoritmi con Look-up Table 28

5.1 Direct Approximations 29

5.1.1 Esempio 30

5.2 Linear Approximations 32

6 Algoritmi a latenza variabile 33

7 Analisi prestazioni 34

8 Conclusioni 37

9 Biografia 38

1 Introduzione

Negli ultimi anni l'elettronica digitale è diventata una parte comune a molti strumenti elettronici.
Uno dei requisiti per rispondere all'aumento dell'esigenze del mercato di questi dispositivi elettronici è la velocità.
Le moderne applicazioni comprendono diverse operazioni in virgola mobile come addizione, moltiplicazione divisione e radice quadrata.
Per rispondere a queste esigenze si è implementato nell' hardware di alcuni dispositivi, per alte prestazioni, nuove funzioni in floating point.
In FPU recenti, l'accento è stato posto sulla progettazione di sommatori sempre più veloci e moltiplicatori, facendo meno attenzione ai divisori.
Una percezione comune della divisione è che, essendo un'operazione non usata di frequente, la sua attuazione non sia necessaria per migliorare le prestazione generali.
Tuttavia, è stato dimostrato che ignorare la sua implementazione può portare a riduzioni delle prestazioni per molte applicazioni.
Mentre la sua implementazione via software risulta molto svantaggiosa.
Tuttavia, lo spazio di progettazione degli algoritmi e le implementazioni sono numerose a causa del gran numero di parametri coinvolti.
Algoritmi di divisione possono essere suddivisi in cinque classi: *cifre ricorrenti* (digit recurrence), *funzioni iterative* (functional iteration), *radix molto alto* (very high radix), *tabelle di ricerca* (table look-up) e la *latenza variabile* (variable latency.).
La base di queste classi è la evidente differenza nelle operazioni di hardware utilizzati per la loro implementazioni, come la moltiplicazione, sottrazione e tabella di look-up.
Molti algoritmi di divisione pratico non sono forme pure di una classe particolare, ma piuttosto sono combinazioni di più tipi.
Ad esempio, un algoritmo di alte prestazioni può utilizzare una tabella di look-up per ottenere un approssimazione iniziale per il reciproco, utilizzare un algoritmo di iterazione funzionale a convergere quadraticamente al quoziente, e completa nel tempo variabile che utilizza una tecnica di latenza variabile.
Si farà ora una descrizione di questi cinque algoritmi su cui si basano queste tipologie e sulla loro implementazione.
Si eviterà di parlare di altri gruppi di divisori in quanto, come detto in precedenza, sono il risultato di incroci di questi cinque particolari tipi base.
Non andrò molto nel dettaglio sulle implementazioni, in quanto non basterebbe una tesi per parlare di questo.
Le considerazioni che si andranno fare riguarderanno la velocità, il consumo, e l'area di silicio richiesta per implementarla.
Varie tecniche sono state proposte per aumentare ulteriormente le prestazioni di divisione, tra cui gestione temporanea di semplice dello stadio low-radix, la sovrapposizione di una fase con un altro stadio, e prescaling degli operandi di ingresso.

1.1 Standard IEEE 754

Lo standard IEEE 754 per il calcolo in virgola mobile, è lo standard più diffuso nel campo del calcolo automatico.
Questo standard definisce il formato per la rappresentazione dei numeri in virgola mobile, compreso ±0 e i numeri denormalizzati, gli infiniti e i NaN, ed un set di operazioni effettuabili su questi.
Inoltre si danno quattro metodi di arrotondamento e ne descrive cinque eccezioni.
Un numero in virgola mobile, secondo lo standard IEEE è rappresentato su parole di 32 bit per numeri a precisione singola, 64 bit per numeri a precisione doppia, 128 bit per numeri a precisione quadrupla
Tali numeri sono divisi in tre parti:
- un bit per il segno *s*
- un campo detto esponente e
- un campo detto mantissa *m*

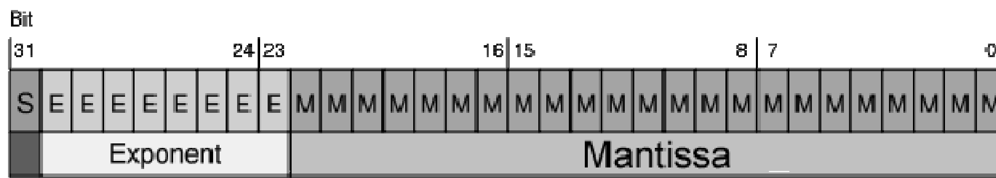

Fig.1 Rappresentazione numeri in flothing point a singola precisione

Il valore contenuto in questi campi è calcolabile come:

$$(-1)^s \cdot 2^e \cdot m$$

Il valore *s* vale 0 per i numeri positivi e 1 per i numeri negativi.
Il campo *e* dedicato all'esponente rappresenta un numero in forma intera, ha dimensione di 8 bit per i numeri in precisione singola, 11 per numeri in precisione doppia, 15 per i numeri a precisione quadrupla.
Il campo m rappresenta una sequenza di cifre dopo la virgola.
Tutte le mantisse sono normalizzate in modo che il numero prima della virgola sia 1, per cui per un dato *m* il valore matematico corrispondente è M=1,*n*.
In pratica, la mantissa è costituita dal numero binario **1**, seguito dalla virgola e dalla parte intera del numero rappresentato, in forma binaria.
In questo modo la mantissa risulta così compresa tra 1 e 2. Quando un numero è normalizzato, come risulta dal suo esponente, il primo bit della mantissa, pari a 1, viene omesso per convenienza.
Altre caratteristiche dello standard IEEE 754 sono i simboli speciali per rappresentare eventi non usuali.

Per esempio, a causa di una divisione per zero si permette di porre una sequenza di bit pari a – inf o +inf; l'esponente più grande è utilizzato rappresentare tale (inf).
Si stabilisce simboli per operazioni non valide 0/0 oppure la sottrazione di infinito da infinito. Questo simbolo è NaN (Not a Number).

ESPONENTE	MANTISSA	OGGETTO RAPPRESENTATO
0	0	0
0	Non zero	± numero denormalizzato
1-254	qualsiasi cosa	± numero in virgola mobile
255	0	± infinito
255	Non zero	NaN

Codifica IEEE 754 dei numeri in virgola mobile in singola precisione

Addizione/sottrazione in virgola mobile

Per effettuare l'addizione e la sottrazione tra operandi in virgola mobile si devono dapprima confrontare gli esponenti: se sono diversi è necessario portare preliminarmente il più piccolo allo stesso valore del più grande spostando verso sinistra il punto di radice di un numero opportuno di posizioni.
A questo punto l'operazione si può effettuare direttamente sulle mantisse.
Se necessario la mantissa del risultato va poi normalizzata.

Moltiplicazione in virgola mobile

Per calcolare l'esponente del prodotto si sommano gli esponenti degli operandi.
Si esegue la moltiplicazione delle mantisse.
Si verificare se il prodotto è normalizzato.
A questo punto si può controllare se si è verificato un overflow o un underflow.
E' necessario arrotondare il numero.
Il segno del prodotto dipende dal segno degli operandi di partenza. Se questi hanno ambedue lo stesso segno, il segno è positivo, altrimenti è negativo.

1.1 Divisione matematica

Prima di cominciare ad esaminare in dettaglio questo ed altri algoritmi voglio ricordare la semplice divisione (matematica).
Essa è definita come l'operazione inversa della moltiplicazione
Se

$$a \cdot b = c$$

dove b è diverso da zero, allora

$$a = c \div b$$

Nell'espressione sopra, a rappresenta il quoziente (quoto nel caso di divisione senza resto), b il divisore (cioè la quantità che divide) e c il dividendo (cioè la quantità da dividere).
Da notare, la divisone per b=0 non è definita.
Se l' operazione di divisione ammette resto:

Dividendo = Quoziente * Divisore + Resto

dove ovviamente la quantità chiamata resto è minore del divisore

2 Divisione a cifre ricorrenti

2.1 Norestoring Division e Restoring Division

L' algoritmo a cifre ricorrenti utilizza il metodo sottrattivo per calcolare una cifra del quoziente per ogni iterazione.
Gli operandi di ingresso si presume che siano rappresentati in un formato normalizzato in virgola mobile con n bit.
Il formato più comune nei computer moderni è lo standard IEEE 754 per il calcolo binario in virgola mobile.
Il primo algoritmo della divisione che storicamente è stato introdotto è stato questo.
Si basa su un algoritmo della divisione simile a quello che ci è stato insegnato per fare la divisione a mano.
Il principio di funzionamento di questo principio può essere ricondotto alla seguente formula iterativa:

$$R_{j+1} = r \cdot R_j - q_{j+1} \cdot D$$

dove

R_{j+1} è il resto al passo j-esimo
r è il numero di cifre dell'alfabeto
q_{j+1} è la cifra j-esima del quoziente $q_j = \{0,1\}$
D è il divisore

Ora verifichiamo se la formula iterativa produce il quoziente richiesto su un numero fissato m di cifre.
Si ha

$j = 0$, $\quad R_1 = r \cdot R_0 - q_1 \cdot D$

$j = 1$, $\quad R_2 = r \cdot R_1 - q_2 \cdot D = r[r \cdot R_0 - q_1 \cdot D] - q_2 \cdot D =$
$\quad = r^2 \cdot R_0 - [r.q_1 + r.\cdot q_2] \cdot D$

…….

$$j = m \ , \qquad R_m = r^m \cdot R_0 - \left[r^{m-1} \cdot q_1 + r^{m-2} \cdot q_2 + ... + r \cdot q_{m-1} + q_m \right] \cdot D$$

quindi

$$\frac{R_0}{D} = r^{-1} \cdot q_1 + r^{-2} \cdot q_2 + ... + r^{-(m-1)} \cdot q_{m-1} + r^{-m} \cdot q_m + \frac{R_m \cdot r^{-m}}{D}$$

dove sono facilmente riconoscibili il quoziente Q e il resto R

$$Q = \sum_{j=1}^{m} q_j \cdot r^{-j}$$

$$R = r^{-m} \cdot R_m$$

Sulla base di questo criterio, si differenziano molti algoritmi.
Alcuni di essi consentono di avere dei resti parziali R_j negativi ed altri no.
Per quelli che non consentono di avere un resto parziale negativo (ovvero resto parziale solo positivo) è necessario eseguire un'operazione aggiuntiva di ripristino e questi vengono chiamati *restoring division*.
Invece gli algoritmi a cifre ricorrenti che consentono di avere un resto parziale positivo o negativo sono detti *norestoring division* per poterli distinguere dagli altri.
Questi ultimi per poter avere resti parziali positivi e negativi necessitano però una conversione in complemento a 2 degli operandi per poter rappresentare i dati correttamente.
Il più semplice criterio che deriva dall'algoritmo restoring division è la *convetional restoring division*.

2.2 Conventional Restoring

In questo metodo si pone $r = 2$ nell'equazione ricorsiva

$$R_{j+1} = r \cdot R_j - q_{j+1} \cdot D$$

ottenendo in questo modo la formula iterativa diventa

$$R_{j+1} = 2 \cdot R_j - q_{j+1} \cdot D$$

La regola per scegliere la cifra del quoziente è:

$$q_{j+1} = \begin{cases} 0 & \text{se} \quad 2 \cdot R_j < D \\ 1 & \text{se} \quad 2 \cdot R_j \geq D \end{cases}$$

Operativamente si suppone $q_{j+1} = 1$ ad ogni iterazione e si calcola il resto stimato mediante una sottrazione

$$R^*{}_{j+1} = 2 \cdot R_j - q_{j+1} \cdot D$$

Se il resto è positivo va bene

$$R^*{}_{j+1} = R_{j+1}$$

Altrimenti si dovrà ripristinare il valore del resto

$$R_{j+1} = R^*{}_{j+1} + D = R_j$$

Infatti, come detto precedentemente, è per questo motivo che l'algoritmo di divisione si chiama restoring division.
La regola per la decisione della cifra del quoziente e quindi del resto parziale è rappresentata graficamente in figura.

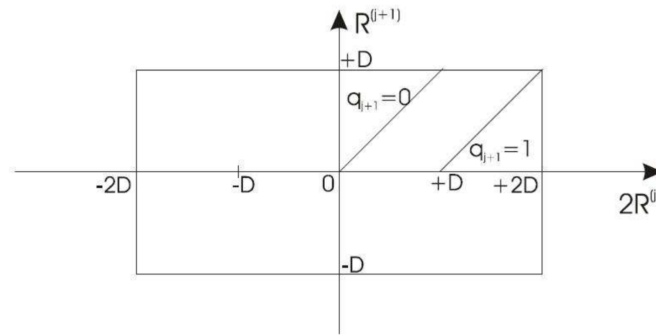

Fig. 2: grafico per la determinazione del quoziente.

Questo algoritmo ha il vantaggio di non rappresentare i dati, dividendo e divisore in complemento a due.
Però richiede a volte (mediamente una volta su due) il ripristino del resto parziale, quindi servirà della circuiteria aggiuntiva per eseguire una somma.

2.2.1 Esempio

Facciamo un esempio di divisione con questo tipo di metodo
Dati
$A = 0.0101_2 = 0.3125_{10}$
$B = 0.1100_2 = 0.75_{10}$
Calcoliamo il loro rapporto $\frac{A}{B}$ su un range di n = 4 bit

R_o	0 . 0 1 0 1	
$2R_o$	0 . 1 0 1 0	
$R_1^*=2R_o\text{-}D$	1 . 0 0 1 1	negativo, $R_1 = 2R_0$, $q_1 = 0$
R_1	0 . 1 0 1 0	
$2R_1$	1 . 0 1 0 0	
$R_2^*=2R_1\text{-}D$	0 . 1 0 0 0	positivo, $R_2 = R_2^*$, $q_2 = 1$
R_2	0 . 1 0 0 0	

$2R_2$	1 . 0 0 0 0	
R_3^*=2R_2-D	0 . 0 1 0 0	positivo, $R_3 = R_3^*$, $q_3 = 1$
R_3	0 . 0 1 0 0	
$2R_3$	0 . 1 0 0 0	
R_4^*=2R_3-D	1 . 010 0 1	negativo, $R_4 = 2R_3$, $q_4 = 0$

Ottenendo

$$\frac{A}{B} = 0.0110 + \frac{2^{-4} \cdot 0.1000}{0.1100}$$

$$Q = 0.0110_2 = 0{,}375_{10} \qquad R = 0.00001000_2 = 0.03125_{10}$$

Che risulta proprio essere

$$A = B \cdot Q + R$$

Per ottenere un risparmio di tempo nell'eseguire la divisione si sono introdotte delle modifiche, ora si accetta anche resti parziali negativi.
Per fare questo è stata modificata la regola per l'assegnazione del quoziente.
In questo modo se nel precedente caso era necessario fare un ripristino ora si accetta un 1 come quoziente ma al resto parziale prossimo si deve assegnare un valore -1 per poter compensare.
In questo modo, il nuovo alfabeto è costituito dalle cifre per il quoziente

$$q_j = \{-1,1\}$$

mentre di norma si tiene inalterato l'alfabeto per il resto.
Con questo nuovo alfabeto, la rappresentazione di un numero N è cambiata per esempio N=3 è ottenuto come

1 -1 1 (MSB a sinistra) 4 - 2 + 1 = 3

Il criterio usato per la scelta di q_{j+1} e R_{j+1} deriva dalla condizione

$$\left|R_{j+1}\right| < D$$

Esso è

$$R_{j+1} = \begin{cases} 2 \cdot R_j - D & \text{se} \quad 2 \cdot R_j > 0 \\ 2 \cdot R_j + D & \text{se} \quad 2 \cdot R_j < 0 \end{cases}$$

$$q_{j+1} = \begin{cases} 1 & \text{se} \quad 0 < 2 \cdot R_j < 2 \cdot D \\ -1 & \text{se} \quad -2 \cdot D < 2 \cdot R_j < 0 \end{cases}$$

Se poi $2 \cdot R_{j+1} = 0$, il resto è nullo ed allora l'operazione è terminata.

Invece, nel caso in cui l'ultimo resto R_n è negativo si deve sommare a questo il divisore modificare il quoziente di conseguenza sottraendo una quantità pari a 2^{-n} (pari ad un ulp).

Graficamente la determinazione del resto può essere ottenuta dalla figura sottostante.

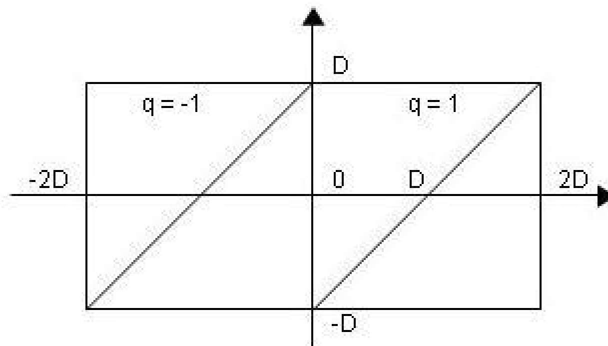

Fig. 3: grafico per la determinazione del quoziente

.

2.2.2 Esempio

Calcoliamo, ora, la divisione di due numeri A e B con questo nuovo metodo.

$A = 0.0101_2 = 0.3125_{10}$

$B = 0.1100_2 = 0.75_{10}$

Calcoliamo il loro rapporto $\frac{A}{B}$ su un range di n = 4 bit

R_o	0 . 0 1 0 1	
$2R_o$	0 . 1 0 1 0	positivo, $q_1 = 1$
$R_1=2R_o$-D	1 . 0 0 1 1	
R_1	1. 0 0 1 1	
$2R_1$	1 . 0 1 0 1	negativo, $q_2 = -1$
$R_2=2R_1$-D	0 . 1 0 0 0	
R_2	0 . 1 0 0 0	
$2R_2$	1 . 0 0 0 0	positivo, $q_3 = 1$
$R_3=2R_2$-D	0 . 0 1 0 0	
R_3	0 . 0 1 0 0	
$2R_3$	0 . 1 0 0 0	negativo, $q_4 = 1$
$R_4=2R_3$-D	1 . 0 1 0 1	

Il resto finale R4 è negativo bisogna allora sommargli D, facendo risultare R4 =2R3

$$R_4 = 0.1000$$
$$R = 2^{-4} * R_4 = 0.00001_2$$

Invece il quoziente risulta:

$$Q = (0\ 1 - 1\ 11) - ulp = 0.0110_2$$

Il vantaggio di questa tecnica è molto semplice, permette di esegue solo una somma o una sottrazione non essendoci nessun ripristino da eseguire (tranne nel caso in cui il resto finale sia negativo). Ciò permette di avere un minor tempo nel calcolo dell’operazione.

2.3 Divisore SRT

I divisori basati sull'algoritmo norestoring furono ulteriormente modificati da Sweney, Robertson e Tocher, ciascuno in maniera indipendente, per migliorare le prestazioni. Dalle lori iniziali risale il nome di questo divisore, SRT che si basa appunto sulla loro tecnica.
In questo tipo di divisore si utilizzano diversamente dai precedenti tre simboli

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

Necessita, a differenza degli altri algoritmi, la normalizzazione degli operandi, dividendo e divisore, in modo che sia

$$\frac{1}{2} < |D| < 1$$
$$\frac{1}{2} < |2 \cdot R_0| < 1$$

La regola per trovare R_{j+1} e q_{j+1} sono invece in questo caso

$$R_{j+1} = \begin{cases} 2 \cdot R_j + D & \text{se} \quad 2 \cdot R_j \le -\frac{1}{2} \\ 2 \cdot R_j & \text{se} \quad -\frac{1}{2} < 2 \cdot R_j < \frac{1}{2} \\ 2 \cdot R_j - D & \text{se} \quad \frac{1}{2} \le 2 \cdot R_j \end{cases}$$

$$q_{j+1} = \begin{cases} -1 & \text{se} \quad 2 \cdot R_j \le -\frac{1}{2} \\ 0 & \text{se} \quad -\frac{1}{2} < 2 \cdot R_j < \frac{1}{2} \\ 1 & \text{se} \quad \frac{1}{2} \le 2 \cdot R_j \end{cases}$$

Queste regole per la decisione del resto parziale e del quoziente sono molto più semplici da eseguire a livello hardware.
Il tempo richiesto per eseguire un confronto è inferiore a quella di una somma/sottrazione.
Inoltre la sua implementazione è più facile e occupa area minore.
Anche l'introduzione del simbolo q=0 risulta una scelta avvincente, nel caso che $q_{j+1}=0$ è necessario soltanto eseguire un confronto e una traslazione senza il bisogno di eseguire somme o sottrazioni.

2.3.1 Esempio

Calcoliamo il rapporto $\frac{A}{B}$ su un range di n = 4 bit usando l'algoritmo SRT

$A = 0.0101_2 = 0.3125_{10}$
$B = 0.1100_2 = 0.75_{10}$

R_o	0 . 0 1 0 1	
$2R_o$	0 . 1 0 1 0	$2R_o > ½$, $q_1 = 1$
$R_1=2R_o$-D	1 . 0 0 1 1	
R_1	1. 0 0 1 1	
$2R_1$	1 . 0 1 0 1	$½ < 2R_1 < ½$, $q_2 = 0$
$R_2=2R_1$	1 . 0 1 0 1	
R_2	1 . 0 1 0 1	
$2R_2$	1 . 1 0 0 1	$2R_2 \le - ½$, $q_3 = -1$
$R_3=2R_2$-D	0 . 0 1 0 0	
R_3	0 . 0 1 0 0	
$2R_3$	0 . 1 0 0 0	$2R_3 > ½$, $q_4 =1$
$R_4=2R_3$-D	1 . 010 0 1	

Il resto finale R4 è negativo, bisogna allora sommargli D, facendo quindi risultare $R_4 = 2R_3$ ed aggiungere un ulp al quoziente.

$$R_4 = 0.1000$$
$$R = 2^{-4} \cdot R_4 = 0.00001_2$$
$$Q = (0\ 1\ 0\ \text{-1}\ 1) - ulp = 0.0110_2$$

Molti altri ricercatori, hanno creato nuovi algoritmi basandosi sul metodo della divisione delle cifre ricorrenti.
In tutti questi, si evidenzia la loro semplicità ed economicità dell' hardware richiesto.
Bene o male, l'implementazione di tutti questi algoritmi richiede sempre un moltiplicatore (per la traslazione) un sommatore/sottrattore (full-adder, carry save adder), alcuni multiplexer come blocchi decisionali ed alcuni registri per contenere i risultati.

2.4 Divisione High Radix

Dopo l’avvento dei divisori SRT si è capito che, per migliorare la velocità, era necessario trovare più cifre del quoziente per volta.
Dalla generica formula ricorsiva di un algoritmo di divisione a cifre ricorrenti

$$R_{j+1} = r \cdot R_j - q_{j+1} \cdot D$$

prendendo un, valore di *r* diverso da 2, si ottengono nuovi algoritmi di divisione.
I nomi che assumono questi algoritmi si basano proprio dal valore che assume r.
Per esempio se r vale 4, 8, 10 i nomi degli algoritmi saranno rispettivamente *radix 4*, *radix 8*, *radix 10*, (in generale *radix r*).
Con il termine high radix si intendono tutti questi algoritmi, successori degli SRT, i quali hanno un valore di *r* (radice) maggiore di 2.
In ogni iterazione, una cifra del quoziente è determinato dalla funzione di selezione quoziente cifre:

$$q_{j+1} = SEL(rP_j, divisore)$$

Questa funzione è una funzione molto cruciale, è da essa che dipende la velocità generale del divisore.
Un’implementazione di questo tipo di algoritmo, può essere suddiviso in semplici funzioni:

- Determinare la cifra successiva del quoziente q_{j+1}, attraverso una funzione che fa uso di una look-up table.
- Generare il prodotto $q_{j+1} \cdot divisore$
- Sottrarre $q_{j+1} \cdot divisore$ dal resto parziale shiftato $r \cdot R_j$

Ognuno di questi componenti, contribuisce alle prestazioni dell'algoritmo.
Il tempo di esecuzione e il corrispondente costo, può variare notevolmente, a seconda dei parametri di questi blocchi.

Come visto in precedenza, per calcolare il quoziente con r = 2 si esegue uno shift a sinistra di un bit del resto parziale per ciclo, per poi fare un confronto.
Ora, con *r* diverso da 2 la cosa diventa più complessa, moltiplicare il resto parziale per r non equivale più ad uno shift di un bit.
Per esempio, ottenere il prodotto $r \cdot R_j$ in un radix 3 è più complesso, non lo si può ottenere con una semplice traslazione.
E’ proprio per questo motivo che, per esempio, radix 3, radix 5, radix 10 non sono molto usati.

Si è piuttosto implementato radix di potenze di 2 (come: 4, 8, 16, 128, 256) dove la moltiplicazione si risolve con una semplice traslazione a sinistra di $\log_2 r$ bit.
Inoltre, per questi nuovi algoritmi, calcolare la cifra del quoziente è molto più complesso.
Supponendo una data precisione del quoziente, il numero di iterazioni necessarie per calcolare il quoziente è ridotta di un fattore f quando il radix è aumentato da r a r^f.
Per esempio un radix 4 calcola 2 bit del quoziente ad ogni iterazione.
Aumentando l'algoritmo si calcola 4 bit ad ogni iterazione, dimezzando quindi la latenza.
Questa riduzione non avviene gratis, la selezione della cifra del quoziente diventa più complicata.
Mentre il numero di cicli per calcolare la divisione è diminuito per l'aumento del radix, è aumentata però la durata di ogni ciclo.
Di conseguenza , il tempo totale richiesto per il calcolo del quoziente di n bit non si è dimezzato come sperato.

I possibili valori che possono assume le cifre non sono più -1 , 0, +1 come nello SRT ma sono molte di più, a seconda dell'algoritmo radix- r in questione.
Il metodo più semplice per scegliere il quoziente è quello di assumere i primi r valori permessi:

$$q_j \in D_a = \{0,\ 1, ..., r, r-1\}$$

ma questo non è il migliore.
Prendendo invece delle cifre consentite simmetriche, si hanno degli incrementi di prestazioni:

$$q_j \in D_a = \{-a, -a+1, ..., -1, 0, 1, ..., a-1, a\}$$

con $a \geq \frac{r}{2}$.

Per questo tipo di divisore a cifre ricorrenti, il grafico per la selezione del quoziente è riportata sottostante.

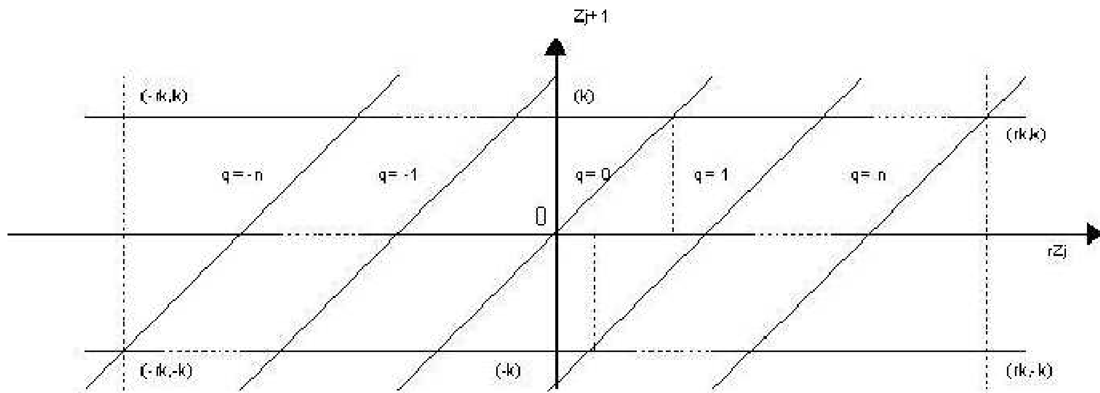

Fig. 3, grafico per la determinazione del quoziente per r > 2

Come si può vedere dalla figura 3, per un determinato resto parziale è possibile scegliere tra vari valori di bit quoziente.
In questo modo, il progettista ha molti gradi di libertà, così può decidere quale regola adoperare per scegliere uno tra i vari bit di quoziente possibili.

3 Divisione con Funzioni Iterative

Un'altra classe di algoritmi usati per compiere una divisione sono quelli che si basano sul calcolo di una funzione iterativa (functional iteration).
Essi usano sviluppi di funzioni come le serie di Taylor, McLaurin, polinomi interpolatori per il calcolo di una funzione che approssima il valore del reciproco del divisore.
Alcuni metodi che meritano di essere discussi sono il metodo Newton-Raphson e il metodo Goldshmitd.

3.1 Newton – Raphson

Il problema del calcolo del rapporto di due numeri interi o frazionali può essere scomposto in un altro modo.
Sia *a* il dividendo e *b* il divisore, il calcolo del rapporto è così suddiviso

- si procede calcolando il reciproco di *b*
- si esegue il prodotto di 1/b con *a*

A prima vista, sembra una cosa che non porti da nessuna parte visto che per eseguire una divisione ne dobbiamo fare un'altra 1/b, sembra che si sia solamente spostato il problema da un'altra parte.
In realtà non è così.
Il reciproco di *b* si può ottenere studiando lo zero della funzione

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

ovvero cercando la soluzione di questa funzione, che è unica

$$x^* = \frac{1}{b}$$

A tal fine si usa il metodo delle tangenti, con questa funzione iterativa

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

Il metodo delle tangenti, detto anche metodo di Newton, fu introdotto da Newton per approssimare lo zero di una funzione, che poi non è altro che l'espansione di Taylor, arrestata al primo ordine.

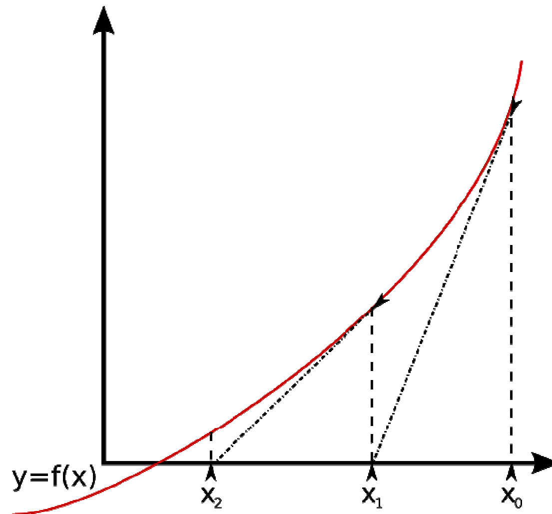

Fig. 4, Grafico delle tangenti alla funzione f(x usato per il calcolo del reciproco del divisore

Questa funzione iterativa conduce alla soluzione desiderata con un numero limitato di iterazioni, essendo f(x) strettamente monotona e con soluzione unica.
Questo metodo assicura la convergenza asintoticamente quadratica, come si dimostra ora.

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

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

L'errore relativo ε_n al passo *n* della procedura iterativa vale

$$\varepsilon_n = \left| \frac{x_n - \frac{1}{b}}{\frac{1}{b}} \right| = |1 - bx_n|$$

mentre l'errore al passo *n+1* è

$$\varepsilon_{n+1} = |1 - bx_{n+1}| = |1 - bx_n(2 - bx_n)| = \left|(1 - bx_n)^2\right| = \varepsilon_n^{\ 2}$$

dove è chiaro che la convergenza al valore reciproco di *b* è quadratica.

Questo significa che, ad ogni iterazione il numero di cifre significative, correttamente generate raddoppia ad ogni ciclo.
Anche se questo algoritmo è più efficace rispetto ad uno che calcola una cifra alla volta ha qualche difetto.
Per essere implementato deve essere necessario l'uso di due moltiplicatori (contemporaneamente) e di un sottrattore, per cui risulta più costoso e ingombrante rispetto ad altri.

3.1.1 Esempio

Facciamo un esempio per vedere se questo funzione iterativa permette il calcolo del reciproco.

Sia $b = 0.0000101_2 \approx 0.0390625_{10}$

Calcoliamo il suo reciproco con questo metodo

N	xn	f(xn)	f'(xn)	xn+1	dx
0	1,00000	0,96094	-1,00000	1,96094	-0,96094
1	1,96094	1,92188	-0,26006	3,77167	-1,81073
2	3,77167	3,73261	-0,07030	6,98765	-3,21599
3	6,98765	6,94859	-0,02048	12,06799	-5,08034
4	12,06799	12,02893	-0,00687	18,44706	-6,37907
5	18,44706	18,40800	-0,00294	23,60139	-5,15432
6	23,60139	23,56232	-0,00180	25,44397	-1,84258
7	25,44397	25,40490	-0,00154	25,59905	-0,15508
8	25,59905	25,55999	-0,00153	25,60000	-0,00095
9	25,60000	25,56094	-0,00153	25,60000	0,00000
10	25,60000	25,56094	-0,00153	25,60000	0,00000

Facciamo partire la funzione iterativa dal valore x_0=1.

Il reciproco di b ($\frac{1}{b} = 25.6$) è approssimato con una buona precisione dopo soli 7 cicli.

Nella realtà, anche se l'approssimazione è molto precisa è necessario eseguire molti più iterazioni del dovuto per poter fare un' arrotondamento del LSB del divisore come previsto dal IEEE 745.

Essendo che il valore di a è contenuto nella mantissa di un numero in virgola mobile, allora il suo valore sarà $1 < a < 2$. Un qualsiasi errore di approssimazione sarebbe quindi amplificato dalla moltiplicazione, causando così un errato calcolo della divisione.

3.2 Goldschmidt

Un altro algoritmo che usa funzioni iterative è quello di Golshmidt.
Esso si basa sulle espansioni di serie.
Consideriamo l'espressione familiare della serie di Taylor come g(y) nel punto P.

$$g(y) = g(p) + (y-p)g^{'}(p) + \frac{(y-p)^2}{2!}g^{''}(p) + ... + \frac{(y-p)n}{n!}g^{(n)}(p) + ...$$

Nel caso della divisione è necessario trovare l'espressione del reciproco del divisore tale che

$$q = \frac{a}{b} = a \cdot g(y)$$

dove g(y) può essere calcolato con un metodo iterativo efficiente.
Un semplice approccio potrebbe essere scegliere g(y) = 1/y con p = 1 e quindi valutare la serie.
Tuttavia è più semplice calcolare prendendo $g(y) = \frac{1}{1+y}$ con p=0 , che è per l'appunto la serie di MacLaurin.
Quindi la serie risultante è:

$$g(y) = \frac{1}{1+y} = 1 - y + y^2 - y^3 + y^4 - ......$$

che nella forma fattorizzata, si trasforma

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

Per avere $g(y) = \frac{1}{b}$ si deve fare la sostituzione $y = b - 1$, e b è normalizzato nell'intervallo $0.5 \leq b < 1$ per avere $-0.5 \leq y < 0$; in questo modo viene assicurata una più rapida convergenza della serie.
Per calcolare la divisione $\frac{a}{b}$ con questa serie, si procede imponendo le seguenti condizioni iniziali

$$N_0 = a$$
$$D_0 = b$$
$$R_0 = 1 - y = 2 - b$$

alla formula iterativa

$$N_{i+1} = N_i \cdot R_i$$
$$D_{i+1} = D_i \cdot R_i$$
$$R_{i+1} = 2 - D_i$$

In tal modo al generico passo i-esimo si ha

$$N_i = a \cdot \left[(1-y) \cdot \left(1+y^2\right) \cdot \left(1+y^4\right) \cdot ..... \cdot \left(1+y^{2i}\right) \right]$$

$$D_i = \left(1 - y^{2i}\right)$$

$$q_i = \frac{N_i}{D_i} = \frac{a \cdot \left[(1-y) \cdot \left(1+y^2\right) \cdot \left(1+y^4\right) \cdot ..... \cdot \left(1+y^{2i}\right) \right]}{\left(1-y^{2i}\right)}$$

Si può vedere che, mentre il denominatore di questa funzione tende ad 1, il numeratore converge al valore q cercato, all'aumentare del numero di passi *i* eseguiti.

Anche se non si è parlato in maniera esaustiva degli algoritmi basati con funzioni iterative si possono trarre delle conclusioni.
In entrambi gli algoritmi Newton-Raphson e Goldshmitd a livello matematico sono equivalenti, comprendono tutte e due moltiplicazioni e un complemento a 2.
La differenza tra i due risiede nell'ordine in cui sono eseguite le moltiplicazioni.
Nel Newton-Raphson, le due moltiplicazioni sono dipendenti, il risultato della prima moltiplicazione viene incanalato nel secondo moltiplicatore.
Nel Goldshmitd invece le due moltiplicazioni, quelle del numeratore e denominatore, sono indipendenti.
Come risultato è che un'implementazione di espansioni di serie può avvantaggiarsi di un moltiplicatore a pipeline per ottenere vantaggi in tempi di latenza.
L'algoritmo di Goldshmitd ha però lo svantaggio che non avviene nessuna autocorrezione dell'errore nel calcolo del quoziente, per garantire la stessa precisione del primo algoritmo bisogna utilizzare dei moltiplicatori con maggiori bit.

4 Very High Radix

Gli algoritmi a cifre ricorrenti sono applicabili solamente per divisione a bassi radix.
Con l'aumento del radix, l'hardware per selezionare le cifre del quoziente e il processo di moltiplicazione diventano sempre più complessi incrementando la durata di un ciclo , l'area o entrambi.
Per realizzare divisori con radix molto alto con un accettabile tempo di ciclo, area, e mezzi per arrotondamento preciso, si usano una variante degli algoritmi a cifra ricorrente, con un più semplice hardware di selezione cifra di quoziente.
Il termine very high radix si applica grosso modo ai divisori che danno più di 10 bit di quoziente ad ogni iterazione.
Gli algoritmi very high radix sono simili ai high radix dato che usano moltiplicazioni per divisioni multiple e tabelle di ricerca per ottenere una approssimazione iniziale del reciproco.
Differiscono nel numero e tipo di operazioni usate in ogni iterazioni usate in ogni iterazione e nella tecnica usata per la selezione della cifra del quoziente.
Rientrano sotto questa categoria l'algoritmo di divisione proposto da Wong and Flynn nel 1992, noto con il nome *Accurate Quotient Approximations*.
Anche il divisore implementato nel coprocessore del Cyrix 83D87 rientra nella categoria a radix alti.
Esso utilizza un algoritmo simile a quello Accurate Quotient Approximations, noto come *Short Reciprocal*.
Vista la loro complessità sia dell'algoritmo su cui si basa, sia del circuito di implementazione, non approfondisco ulteriormente la trattazione.

5 Algoritmi con Look-up Table

Una *look-up table* è una struttura dati, tipicamente array o registri usata per sostituire operazioni di calcolo da eseguire in tempo reale con una semplice operazione di lettura. Esso permette attraverso una combinazione dei dati in ingresso di poter trovare il risultato in un modo molto più veloce (c'è solo il tempo di lettura) rispetto all'eseguire l'operazione in tempo reale.
Una tabella di associazione, o tabella dati, è una struttura che permette di associare ad ogni ammissibile combinazione di dati in ingresso una corrispondente (non necessariamente univoca) configurazione di dati in uscita.
Il termine inglese utilizzato per descriverle, look-up table, sottintende l'operazione di consultazione, il quale permette di associare i dati in uscita a una determinata combinazione dei dati in ingresso.
Anche in un'operazione di divisione può essere usata una tabella di associazione per ridurre la durata dell'operazione.
Quando non è richiesta molta precisione nel determinare il quoziente si può usare solamente la look-up table senza l'uso di altri algoritmi per raffinare il risultato.
Altre volte, quando è richiesta più precisione, si usa la look-up table per ottenere una approssimazione iniziale degli ingressi.
L'utilizzo della look-up table, è il più diffuso modo per approssimare gli operandi della divisione.
Per ottenere queste approssimazioni degli ingressi si usano due modi *direct approximations* e *linear approximations*.
Gli algoritmi che usano la look-up table per l'approssimazione degli ingressi per migliorare le prestazioni sono principalmente gli Very High Radix e i Functional Iterator.
Tipicamente la tabella è implementata con una ROM o una PLA.
Un vantaggio di tabelle look-up è che sono veloci, dal momento che nessun calcolo aritmetico deve essere eseguito.
Lo svantaggio della look-up table sta nel fatto chele dimensioni crescono esponenzialmente con ogni bit di precisione aggiunto.
Di conseguenza, esiste un compromesso tra la precisione della tabella e le sue dimensioni.

5.1 Direct approximations

In questo tipo di approssimazione la tabella conterrà il reciproco dell'operando.
Per indicizzare una tabella si presumo che gli operandi siano normalizzati come previsto dallo standard IEEE, $1 < b < 2$.
Con questo tipo di normalizzazione, si tronca l'operando *b* (il divisore) ai primi k bit e li si utilizzano per indicizzare la tabella.
La dimensione totale di della tabella sarà 2^Km bit, dove k è il numero di bit dell'operando troncato, e *m* è il numero di bit per rappresentare il reciproco.
Per cui l'operando di ingresso alla tabella avrà k bit e il suo reciproco che viene letto dalla tabella e andrà in uscita avrà m bit.
L'operando troncato sarà rappresentato come $1.b'_1\,b'_2\ldots b'_k$ ed invece l'approssimazione del reciproco contenuta nella tabella avrà la forma $0.b'_1\,b'_2\ldots b'_m$.
L'unica eccezione a questo è quando l'ingresso operando è esattamente uno, nel qual caso l'uscita del reciproco dovrebbe anche essere esattamente uno. In questo caso, servirà un hardware separato per rilevare questo caso.
Tutti i valori di ingresso hanno un 1 al primo bit quindi è inutile indicizzare nella tabella questo bit, lo consideriamo come valore implicito.
Allo stesso modo, avendo tutti i valori di output il primo bit a 0 non verranno inseriti nella tabella.
Un comune metodo di progettare la look-up table è di attuare un approssimazione costante a tratti della funzione reciproca.
In questo caso, l'approssimazione per ogni valore inserito si ottiene prendendo il reciproco del punto medio tra $1.b'_1\,b'_2\ldots b'_k$ e il suo successore, in questo modo il punto medio è pari a $1.b'_1\,b'_2\ldots b'_k1$.
Il reciproco del punto medio è arrotondato con l'aggiunta di $2^{-(m+2)}$ e quindi troncando il risultato di m + 1 bit, produce un risultato nella forma $1.b'_1\,b'_2\ldots b'_m$
Così, l'approssimazione di ogni reciproco si trova per ciascuna *i* voce nella tabella sarà la quantità:

$$R_i = \frac{\left\lfloor 2^{m+1} \cdot \left(\frac{1}{c + 2^{-(k+1)}} + 2^{-(m+2)} \right) \right\rfloor}{2^{m+1}}$$

dove $c = 1.b'_1\,b'_2\ldots b'_k$

Das Sarma e Matula hanno dimostrato che questo metodo per la generazione della tabella dei reciproci minimizza l'errore massimo relativo nel risultato finale.
L'errore massimo relativo nel reciproco ε_r stimata per k bit di ingresso e k bit di uscita nel calcolo del reciproco è:

$$|e_r| = \left| R_0 - \frac{1}{b} \right| \leq 1.5 \cdot 2^{-(k+1)}$$

e quindi la tabella garantisce una precisione minima di k + 0,415 bit.
E 'anche dimostrato che, con m = k + g, dove g è il numero di bit di guardia di uscita, il massimo errore relativo è delimitata da

$$|e_r| = 2^{-(k+1)} \cdot \left(1 + \tfrac{1}{2^{g+1}}\right)$$

Quindi, la precisione con k bit di ingresso e (k + g) bit di uscita nella tabella dei reciproci con k $\geq$ 2 e g $\geq$ 0, è di almeno $k + 1 - \log_2\left(1 + \frac{1}{2^{g+1}}\right)$ k+1 .
Di conseguenza, le tabelle generate con uno, due e tre bit di guardia in uscita garantiscono una precisione di almeno k + 0,678 bit, k + 0,830 bit, e k + 0,912 bit, rispettivamente.

5.1.1 Esempio

Calcolo del reciproco con il metodo tabellare.
La memoria accetta ingressi a 4 bit e restituisce l'inverso a 5 bit.
Sono dati il valore di b in n=6 bit

$$\begin{aligned}
b_1 &= 1.001000_2 \cong 1.125_{10} \\
b_2 &= 1.001001_2 \cong 1.141_{10} \\
b_3 &= 1.001010_2 \cong 1.156_{10} \\
b_4 &= 1.001100_2 \cong 1.187_{10} \\
b_5 &= 1.001101_2 \cong 1.203_{10} \\
b_6 &= 1.001110_2 \cong 1.219_{10}
\end{aligned}$$

Bisogna troncare i valori di b ai 5 bit

$$b'_1 = 1.0010$$
$$b'_2 = 1.0010$$
$$b'_3 = 1.0010$$
$$b'_4 = 1.0011$$
$$b'_5 = 1.0011$$
$$b'_6 = 1.0011$$

Ora questi valori, escluso il primo bit, saranno i valori dell'indirizzo in memoria dove sarà contenuto il reciproco.
I valori di b_1 b_2 b_3 come b_4 b_5 b_6 avranno valore reciproco approssimato uguale.
I valori b_1 b_2 b_3 avranno lo stesso valore approssimato del reciproco del numero $b' = 1.00101$.
Allo stesso modo, b_4 b_5 b_6 saranno approssimati al reciproco di $b'' = 1.00111$.
Con l'utilizzo della formula

$$R_i = \frac{\left[2^{m+1} \cdot \left(\frac{1}{c + 2^{-(k+1)}} + 2^{-(m+2)} \right) \right]}{2^{m+1}}$$

si trova che i reciproci b_1 b_2 b_3 sono approssimati al valore decimale 0.8726_{10} mentre b_4 b_5 b_6 a 0.828_{10}.

5.2 Linear Approximations

Invece di utilizzare una approssimazione costante del reciproco, è possibile utilizzare una approssimazione lineare o polinomiale.
Una approssimazione polinomiale è espressa in forma di una serie troncata

$$P(b) = C_0 + C_1 \cdot b + C_2 \cdot b^2 + C_3 \cdot b^3 + ...$$

Con una approssimazione lineare del primo ordine, i coefficienti C_0 e C_1 sono memorizzati in un look-up table, e una moltiplicazione e una addizione sono necessari.
Come esempio, una funzione lineare scegliamo

$$P(b) = -C_1 \cdot b + C_0$$

al fine di approssimare 1/b.

I due coefficienti di C_0 e C_1 vengono letti da due tabelle look-up, ciascuno utilizzando i k bit più significativi di b per indicizzare le tabelle.
L'errore totale della approssimazione lineare ε_{la} con solo k bit di troncamento e m bit di storing per ciascuna tabella è

$$|e_{la}| < 2^{-(2k+3)} + 2^{-m}$$

Per esempio, con m = 2k + 3 l'errore commesso è $|e_{la}| < 2^{-(2k+2)}$.
La dimensione totale richiesto per tabelle è $2k \cdot m \cdot 2$ bit, e poi $m \cdot m$ bit sono necessari per l'unità accumulazione moltiplicazione.
In generale, le approssimazioni lineari garantiscono più precisione rispetto a quelle ad approssimazioni costante, ma richiedono ulteriori operazioni che possono incidere sul ritardo totale.

6 Algoritmi a latenza variabile

A differenza degli algoritmi fin qui esaminati, ne esistono degli altri, come indica il nome stesso, che richiedono un numero di cicli non ben noto a priori.
In questi algoritmi si cerca di migliorare il tempo medio di esecuzione dell'operazione.
Gli svantaggi di questi algoritmi sono la complessità nella logica per la gestione del clock per la determinazione dell'evento "risultato pronto" ed inoltre la progettazione di un dispositivo a latenza variabile che ha impatto anche sull'architettura che fa uso di questo dispositivo.
Infatti sono necessari alcuni segnali ausiliari per indicare quando l'operazione di divisione si è conclusa.

7 Analisi prestazioni

Analizziamo ora, le prestazioni degli algoritmi di divisione visti fin ora.
Nella divisione SRT, per ridurre la latenza di divisone, si possono calcolare più bit ad ogni ciclo.
Tuttavia, aumentando in modo diretto il radix aumenta notevolmente il tempo di ciclo e la complessità di costruzione della divisione.
Due principali mezzi a disposizione per ridurre il tempo di un ciclo sono aumentare i gradi di ridondanza nel selezionare la cifra del quoziente e usare l'operazione di prescaling.
Questi due metodi possono essere combinati per una riduzione ancora maggiore.
Per esempio, nella divisione radix-4 con l'operazione di prescaling, e una sovra-ridondanza delle cifre di quoziente insieme si possono ridurre il numero richiesto di bit di resto parziale per la selezione del quoziente da sei a quattro cicli.
Scegliendo la massima ridondanza per radix-2, la codifica del resto parziale riduce il numero dei bit restanti parziali necessari per la selezione quoziente fino a tre cicli.
Tuttavia, per ognuno di questi miglioramenti, sono necessari maggiore area e maggiore complessità per l'implementazione.
A causa dei limiti di tempo del ciclo e l'area, i moderni divisori SRT sono realisticamente limitati a funzionare a meno di 10 bit per ciclo.
Tuttavia, un divisore a cifre ricorrenti rimane un mezzo efficace, di facile implementazione, ed operante in parallelo con il resto di un processore.
I divisori very high radix vengono utilizzati quando si vogliono elaborare più di 10 bit del quoziente per ciclo.
La differenza principale tra gli algoritmi presentati sono il numero e la larghezza di moltiplicatori utilizzati.
Questi ultimi hanno effetti evidenti sulla latenza dell'algoritmo e sulla dimensione dell'area coinvolta.
Sia il divisore Newton-Raphson e quello a funzioni ad espansione di serie, sono mezzi efficaci per attuare più velocemente l'operazione di divisione.
Per entrambi gli algoritmi, il tempo di ciclo è limitato da due moltiplicazioni.
Nell' iterazione Newton-Raphson, queste moltiplicazioni procedono in serie, mentre in sviluppo in serie, queste moltiplicazioni sono in parallelo.
Per ridurre la latenza delle iterazioni può essere utilizzato una accurata approssimazione.
Questo introduce un compromesso tra area del chip aggiuntivo per un look-up table e la latenza del divisore.
Un'alternativa a una tabella di ricerca è l'utilizzo di un array di prodotto parziale.
Invece di richiedere ulteriore area, tale attuazione potrebbe aumentare il tempo di ciclo attraverso il moltiplicatore.

Il vantaggio principale della divisione per iterazione funzionale è la convergenza quadratica per il quoziente.
Nell' iterazione funzionale non è facile fornire un resto finale.
Di conseguenza, è difficile un corretto arrotondamento per le implementazioni di iterazione funzionale.
Con una implementazione SRT quando è richiesta una latenza inferiore si utilizza una iterazione.
Essi, fornisce un modo per raggiungere latenze inferiori, senza incidere sul serio il tempo di ciclo del processore e senza una grande quantità di hardware aggiuntivo.
Una sintesi degli algoritmi di queste classi è riportata nella tabella 1.

Algoritmo	**Tempo di iterazione**	**Latenza(ciclo)**	**Approsimazione area**
SRT	Qsel table + (multiple form + subtraction)	$\left(\frac{n}{r}\right) + scale$	Qsel table + CSA
Newton - Raphson	2 serial moltiplication	$\left(2\left\lceil \log_2 \frac{n}{i} \right\rceil + 1\right) t_{mul} + 1$	1 multiplier + table + control
espansione in serie	1 moltiplication	$\left(\left\lceil \log_2 \frac{n}{i} \right\rceil + 1\right) t_{mul} + 2,\ \text{se}\ t_{mul} > 1$ $2\left\lceil \log_2 \frac{n}{i} \right\rceil + 3,\quad \text{se}\ t_{mul} = 1$	1 multiplier + table + control
accurate quotient approx	1 moltiplication	$\left(\left(\frac{n}{i}\right) + 1\right) t_{mul}$	3 multiplier + table + control

In questa tabella, n è il numero di bit in operandi di ingresso, i è il numero di bit di precisione da una prima approssimazione, e t_{mul} è la latenza del blocco moltiplicatore / accumulatore per ciclo.
Nessuna delle latenze include il tempo supplementare necessario per l'arrotondamento o normalizzazione.
La tabella 2 fornisce una valutazione approssimativa degli effetti di algoritmo, lunghezza di operando, la larghezza di prima approssimazione, e la latenza moltiplicatore sulla latenza divisione.

Algoritmo	**Radix**	**Latenza(ciclo)**
SRT	4	27
	8	18
	16	14
	256	7

Algoritmo	Pipeline	Approssimazione iniziale	Latenza (cicli)		
			$t_{mul} = 1$	$t_{mul} = 2$	$t_{mul} = 3$
Newton - Raphson	entrambi entrambi	i = 8 i = 16	8 6	15 11	22 16
espansione in serie	no no si si	i = 8 i = 16 i = 8 i = 16	9 7 9 7	17 13 10 8	25 19 14 11
accurate quotient approx	entrambi entrambi	i = 8 i = 16	8 5	16 10	24 15

Tutti gli operandi sono mantissa IEEE in doppia precisione, con n = 53.
Si presume che la tabella look-up per approssimazioni iniziali richiedono un ciclo.
Le latenze SRT sono separate dagli altri algoritmi, in quanto non dipendono dal moltiplicatore di latenza, ma sono solo in funzione della radice dell' algoritmo per la fine di questa tabella.
Per gli algoritmi di divisione che si basano su moltiplicazioni, le latenze sono indicate per moltiplicatori che hanno latenze di uno, due e tre cicli.
Inoltre, le latenze sono indicate per pipeline così come moltiplicatori non pipeline.
Una unità pipeline può iniziare un calcolo ogni nuovo ciclo, mentre una unità senza pipeline può iniziare solo dopo che il calcolo precedente è stato completato.
Dalla tabella 2, la versione avanzata dell' algoritmo accurate quotient approx fornisce la più bassa latenza.
Tuttavia, il requisito di area di questa implementazione è enorme, in quanto richiede almeno 736K di look-up table e tre moltiplicatori.
Per le implementazioni realistiche, con $t_{mul} = 2$ o $t_{mul} = 3$ e i = 8, la più bassa latenza è raggiunta attraverso una implementazione sviluppo in serie.
Tuttavia, tutte le implementazioni di a base moltiplicazione sono molto vicine nelle prestazioni.
Questa analisi dimostra l'estrema dipendenza della latenza divisione sulla latenza del moltiplicatore.
Un fattore di tre differenze di latenza del moltiplicatore può peggiorare quasi un fattore di tre la latenza di divisione per molte delle implementazioni.
E 'difficile per una implementazione SRT, avere un rendimento migliore di quelli a implementazione con moltiplicazione a causa dell' inaffidabilità di radix elevati.
L'uso di tecniche di latenza variabile è in grado di fornire ulteriori mezzi per migliorare le prestazioni dei sistemi che si basano sulla moltiplicazione, senza la difficoltà di arrotondamento che è intrinseca nella implementazione a iterazione funzionale.

8 Conclusioni

In questo testo abbiamo analizzato le cinque classi di divisore.
Tali divisioni in classi è dovuta alle differenti operazioni fondamentali utilizzate nelle implementazioni hardware.
La classe più semplice e più comune di divisione che si trova nella maggior parte dei processori moderni è il divisore SRT .
Le recenti implementazioni commerciali di algoritmi SRT hanno incluso radix-2, radix-4 e radix-8.
Queste implementazioni sono state scelte, in parte, perché operano in parallelo con il resto dell' hardware in virgola mobile e non creano nessun conflitto con le altre unità.
Inoltre, con radix bassi, è stato possibile poter soddisfare le strette esigenze di tempo di ciclo di processori ad alte prestazioni, senza bisogno di grandi quantità di area.
Lo svantaggio di queste implementazioni SRT è la loro latenza relativamente alta.
Un'alternativa a implementazioni SRT è iterazione funzionale, l' espansione di serie è la forma più comune.
L'analisi presentata dimostra che una serie di espansione prevede un'implementazione con la più bassa latenza per le area.
La latenza è ridotta in questa implementazione attraverso l'uso di un riordino delle operazioni nell' iterazione Newton-Raphson per sfruttare il rendimento a ciclo unico dei moltiplicatori di pipeline.
Al contrario, la stessa l'iterazione di Newton-Raphson, con le sue moltiplicazioni di serie, ha una latenza maggiore.
Tuttavia, se un moltiplicatore pipeline è usato in tutte le iterazioni, più di una operazione di divisione può procedere in parallelo.
Molti algoritmi very high radix sono un mezzo interessante per avere una bassa latenza, fornendo inoltre un resto.
L'unica implementazione commerciale di un algoritmo very high radix è l'unità di Cyrix short-reciprocal.
Questa implementazione fa un uso efficiente di un singolo moltiplicare.
Il progettista potrà usare la classe di divisione più idonea a seconda del problema, scegliendo, per esempio, il divisore più veloce oppure quello che occupa minore area.

9 Biografia

[1] Montuschi P., Ciminiera L., Giustina A., "A Division Architecture Combining Newton-Raphson Approximations and Direct Methods Iterations", in: Conference Record of The Twenty-Sixth Asilomar Conference on Signals, Systems and Computers, Oct 1992

[2] Obermann S.F., Flynn M.J., "Division Algorithms and Implementations" in: , IEEE Trans. Computers , Aug 1997

[3] Oberman, S.F.; Flynn, M.J.; "Design issues in division and other floating-point operations" IEEE Transactions on Computers, 1997

[4] Ercegovac, M.D.; Imbert, L.; Matula, D.W.; Muller, J.-M.; Wei, G. "Improving Goldschmidt Division, Square Root, and Square Root Reciprocal", IEEE Trans. Computers 2000

[5] Rodrigues, M.R.D.; Zurawski, J.H.P.; Gosling, J.B.; "Hardware evaluation of mathematical functions" Computers and Digital Techniques, IEE Proceedings E ,1981

[6] Aggarwal, N.; Asooja, K.; Verma, S.S.; Negi, S.; "An improvement in the restoring division algorithm" , Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on ,2009

[7] S.F. Oberman and M.J. Flynn, "Design Issues in Division and Other Floating-Point Operations," IEEE Trans. Computers, vol. 46, no. 2, pp. 154-161, Feb. 1997.

[8] K.G. Tan, "The Theory and Implementation of High-Radix Division," Proc. Fourth IEEE Symp. Computer Arithmetic, pp. 154-163, June 1978.

[9] "IEEE Standard for Binary Floating Point Arithmetic," ANSI/IEEE Standard 754-1985. New York: IEEE, 1985.

[10] E. Schwarz and M. Flynn, "Hardware Starting Approximation for the Square Root Operation," Proc. 11th Symp. Computer Arithmetic, pp. 103-111, July 1993.

[11] M.D. Ercegovac, T. Lang, and P. Montuschi, "Very High Radix Division with Selection by Rounding and Prescaling," IEEE Trans. Computers, vol. 43, no. 8, pp. 909-918, Aug. 1994.