Prodotto tra matrici

Progetto d'esame di Architettura degli Elaboratori A.A. 2020/2021

(sessione autunnale)

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{e} \quad \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Studente: Alessio Muzi

matricola n°: 299329

SPECIFICA E ALGORITMI

Specifica funzionale:

Il progetto prevede la realizzazione di un segmento di codice in linguaggio ***Assembly*** e l'esecuzione attraverso il simulatore **winMIPS64**.

Il codice eseguirà il prodotto di due matrici quadrate di numeri razionali di dimensioni prefissate, 3x3.

Traduzione della specifica in linguaggio ***C***:

```
for (i = 0; i < 3; i++)
        for(j = 0; j < 3; j++)
          {
            c[i][j] = 0;
            for(k = 0; k < 3; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
```

Algoritmi usati:

L'algoritmo impiegato è il classico algoritmo del prodotto riga per colonna, dove l'elemento `c[i][j]` è uguale alla somma dei prodotti tra gli elementi della riga `i` della prima matrice e gli elementi della colonna `j` della seconda.

Esempio:

```
c[1][1]= a[1][1] * b[1][1] + a[1][2] * b[2][1] + a[1][3] * b[3][1]
```

IMPLEMENTAZIONE

Sezione .data:

```
.data
a:          .double 4.2,7.2,3.4,7.7,4.3,6.9,5.7,9.1,6.1   ; input: matrice A
b:          .double 1.4,3.0,2.1,1.9,2.4,1.6,2.8,1.2,2.1   ; input: matrice B
c:          .double 0,0,0,0,0,0,0,0,0                     ; output: matrice risultato
op:         .word   3                                     ; contatore operazioni (prodotto)
elementi:   .word   9                                     ; numero elementi
counterR:   .word   3                                     ; contatore righe
counterC:   .word   3                                     ; contatore colonne
```

Nella sezione .data troviamo tutti i dati di input, output e le variabili necessarie per il calcolo delle operazioni. Le matrici sono matrici quadrate 3x3 di numeri razionali generati casualmente (Matrice A: $0 < n < 10$; Matrice B: $0 < n < 3$).

Inoltre, le matrici sono rappresentate tramite **linearizzazione**, ovvero sono trasformate in array ottenuti concatenando le righe (o le colonne) della matrice. In particolare, nel programma la matrice A è stata linearizzata per righe, mentre la matrice B per colonne, per rendere più semplici e regolari i calcoli degli indici. Sarebbe stato possibile avere per entrambe le matrici una **linearizzazione** per righe (o per colonne), eseguendo appropriati calcoli degli indici (per esempio, nel caso in cui entrambe le matrici siano state linearizzate per righe, invece che spostare il puntatore della matrice B per `8 byte`, sarebbe stato necessario spostarlo di `24`, in modo da rappresentare correttamente la colonna).

Oltre alle matrici sono presenti 4 contatori:

- `op` (**3**) rappresenta il numero di prodotti eseguiti ad ogni ciclo;

- `elementi` (**9**) rappresenta il numero di elementi totali e serve ad indicare quando il programma ha eseguito tutti i calcoli e deve terminare;

- `counterR` (**3**) e `counterC` (**3**) rappresentano il numero di righe e di colonne e servono ad indicare al programma se la riga/colonna selezionata è quella corretta per il calcolo dell'n-esimo elemento.

Sezione .text:

Nella sezione .text abbiamo il corpo del programma, diviso in varie etichette per renderlo più leggibile e facilitare i costrutti di ripetizione e selezione.

Le etichette utilizzate sono: `start`, `loop`, `controllo`, `righe` / `colonne` / `endif_1` e `2` (`selettore righe/colonne`) ed `end`.

```
start:
    DADDI   r1, r0, a         ; puntatore al primo elemento della matrice A
    DADDI   r2, r0, b         ; puntatore al primo elemento della matrice B
    DADDI   r3, r0, c         ; puntatore al primo elemento della matrice risultato
    LW      r4, op(r0)        ; contatore per i prodotti da eseguire
    LW      r5, elementi(r0)  ; numero degli elementi delle matrici
    LW      r6, counterR(r0)  ; contatore righe
    LW      r7, counterC(r0)  ; contatore colonne
```

`start`: Nell'etichetta `start` sono presenti tutte le operazioni di inizializzazione dei dati di **I/O**. Le prime 3 riguardano i puntatori ai primi elementi delle matrici di input e della matrice risultato, mentre le altre 4 caricano in memoria i 4 contatori utilizzati per il ciclo, la terminazione del programma e la selezione delle righe e delle colonne.

```
loop:
    L.D     f0, 0(r1)         ; lettura a[i]
    L.D     f1, 0(r2)         ; lettura b[i]
    MUL.D   f2, f1, f0        ; prodotto tra a[i] e b[i]
    ADD.D   f3, f3, f2        ; somma dei prodotti parziali
    DADDI   r1, r1, 8         ; spostamento puntatore (righe, matrice A)
    DADDI   r2, r2, 8         ; spostamento puntatore (colonne, matrice B)
    DADDI   r4, r4, -1        ; decremento contatore operazioni da eseguire
    BNEZ    r4, loop          ; continua il loop se ci sono ancora elementi nella riga
```

`loop`: Nell'etichetta `loop` sono presenti le operazioni che ci permettono di calcolare il prodotto tra due matrici. Inizialmente il programma carica in memoria il primo elemento della matrice A e B, successivamente esegue il prodotto dei due numeri, lo addiziona ai risultati parziali ottenuti in precedenza (nel caso sia la prima iterazione, il registro `f3` è vuoto) ed infine sposta il puntatore agli elementi successivi. Alla fine viene eseguito un controllo e se il puntatore `op` non è stato esaurito, ricomincia il ciclo.

```
controllo:
    LW      r4, op(r0)        ; reset contatore operazioni
    S.D     f3, 0(r3)         ; scrittura del risultato in c[i]
    DADDI   r3, r3, 8         ; spostamento puntatore matrice risultato
    DADDI   r5, r5, -1        ; decremento contatore elementi
    DADDI   r6, r6, -1        ; decremento contatore righe
    DADDI   r7, r7, -1        ; decremento contatore colonne
    SUB.D   f3, f3, f3        ; reset del registro delle somme parziali
    BNEZ    r6, righe         ; se il contatore delle righe è =/= 0, torna all'inizio della riga
    LW      r6, counterR(r0)  ; altrimenti, reset contatore righe e
    J       endif_1           ; salta
```

`controllo`: Nell'etichetta `controllo` vengono eseguite le operazioni di reset del contatore `op`, scrittura dei risultati, decremento dei contatori

`counterR` e `counterC` e il reset del registro `f3`, il quale contiene le somme parziali. Alla fine di tutto ciò viene eseguito un controllo, il quale stabilisce se dovra essere necesseria la selezione di altre righe o colonne, o se il programma può continuare con l'elemento successivo. La selezione vera e propria viene eseguita nel gruppo di etichette successivo.

```
;selettore righe/colonne
righe:
    DADDI   r1, r1, -24      ; torna all'inizio della riga

endif_1:
    BEQZ    r7, colonne      ; se il contatore delle colonne è = 0, torna al primo elemento della matrice B
    J       endif_2          ; altrimenti, salta

colonne:
    DADDI   r2, r2, -72      ; torna al primo elemento della matrice B
    LW      r7, counterC(r0) ; reset contatore colonne

endif_2:
    BNEZ    r5, loop         ; se ci sono ancora elementi, ricomincia il ciclo
```

`selettore righe/colonne`: In questo gruppo di etichette vengono effettuati due controlli, uno sul contatore delle righe e uno su quello delle colonne.

Nel caso delle righe (quindi della matrice A) se il contatore delle righe è diverso da `0`, vuol dire che quella riga deve rimanere fissa per almeno un altro ciclo e quindi il puntatore deve tornare indietro di `3` numeri. Nel caso in cui invece il contatore è uguale a `0`, il puntatore si troverà già al primo elemento della riga successiva e quindi si può saltare tutto il processo.

Nel caso delle colonne (quindi della matrice B), quando il contatore è uguale a `0` significa che il puntatore si trova all'ultimo elemento della matrice e va quindi riportato al primo (sottraendo `72 byte`). Nel caso in cui questo non sia vero, il programma può continuare saltando questo processo.

Infine viene eseguito un'ultimo controllo, quello del contatore degli elementi, per stabilire se il programma può terminare o deve eseguire un altro ciclo. Nel caso il contatore sia arrivato a `0`, vuol dire che tutti gli elementi della matrice risultato sono stati calcolati e che non ci sono ulteriori calcoli da eseguire. Altrimenti, si ritorna all'etichetta `loop`.

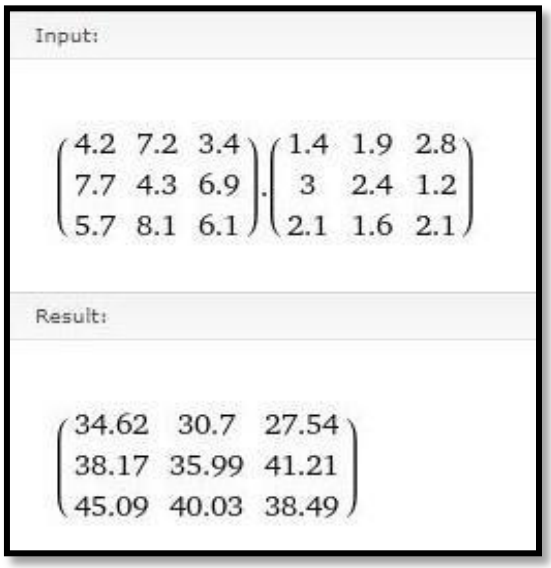
```
end:
    HALT                     ; terminazione programma
```

`end`: Arrivati all'ultima istruzione, il programma può terminare correttamente.

RISULTATI E PERFORMANCE

Risultati:

Risultati previsti per il prodotto delle due matrici prese in considerazione.



Conversioni da esadecimale floating point in decimale dei risultati.

40414f5c28f5c28f = 34.62

403eb33333333334 = 30.7

403b8a3d70a3d70a = 27.54

404315c28f5c28f6 = 38.17

4041feb851eb851e = 35.99

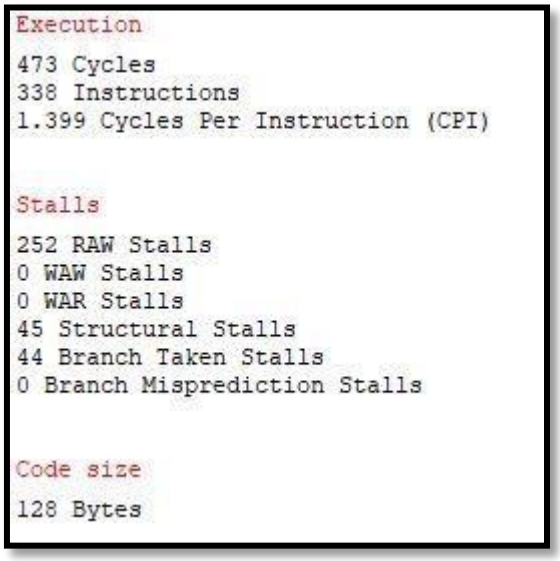
40449ae147ae147b = 41.21

40468b851eb851eb = 45.09

404403d70a3d70a3 = 40.03

40433eb851eb851f = 38.49

Performance:



L'elevato numero di stalli **Read after Write** è dovuto dalla vicinanza tra il prodotto e la somma nell'etichetta `loop`.

OTTIMIZZAZIONE DEL CODICE

Versione 2: Spostando l'operazione `ADD.D` alla fine del loop, il CPI scende a *1.346* e gli stalli **RAW** passano a *135.* (*-18 cicli di clock*)

Versione 3: Spostando l'operazione di spostamento del puntatore della matrice risultato subito prima del salto, il CPI scende a *1.337* e vengono risolti *3* stalli **strutturali.** (*-3 cicli di clock*)

Versione 4: Spostando l'operazione di scrittura del risultato - la quale andava in stallo **RAW** con la somma - due righe più in basso, il CPI scende a *1.284* e gli stalli **RAW** scendono a *117.* (*-18 cicli di clock*)

Versione 5: Spostando l'operazione di moltiplicazione – la quale andava in stallo **RAW** con l'operazione di caricamento in memoria del registro `f1` – una riga più in basso, non otteniamo cambiamenti per quanto riguarda CPI e stalli **RAW**, ma vengono risolti *9* stalli strutturali.

Versione 6: Spostando l'operazione di scrittura del risultato ancora una riga più in basso, otteniamo un ulteriore miglioramento del CPI, che arriva a *1.257*, ed una riduzione di cicli **RAW**, che scendono a 108, ma introduciamo *9* cicli **WAR**. *(-9 cicli di clock*)

Versione 7: Spostando l'operazione di svuotamento del registro delle somme parziali, si riesce a risolvere i cicli **WAR**, ma peggiorando lievemente il CPI, *1.266*. (*+3 cicli di clock*)

Versione 8: Utilizzando tecnice di loop unrolling e register renaming, si può trasformare il loop in modo da calcolare tutta la riga invece che un elemento alla volta. Questo comporta il risparmio di alcune operazioni come il salto alla fine di ogni loop, lo spostamento dei puntatori e una somma. Complessivamente, i cicli di clock vengono ridotti a *319* ma il CPI aumenta a *1.512*, perché eseguiamo meno istruzioni (*211*). Gli stalli **RAW** sono *81*. Inoltre, la lunghezza del codice aumenta leggermente.

Versione 9: Provando invece a srotolare completamente tutte i calcoli, possiamo fare a meno dell'etichetta `controllo` e della `selezione righe/colonne`, poiché selezioneremo le righe e le colonne tramite l'indicazione statica degli indirizzi. Il codice in questo formato esegue 112 istruzioni in *215* cicli di clock (CPI: *1.920*) con *117* stalli **RAW**. Inoltre peggiorano drasticamente la dimensione del codice e la riusabilità.

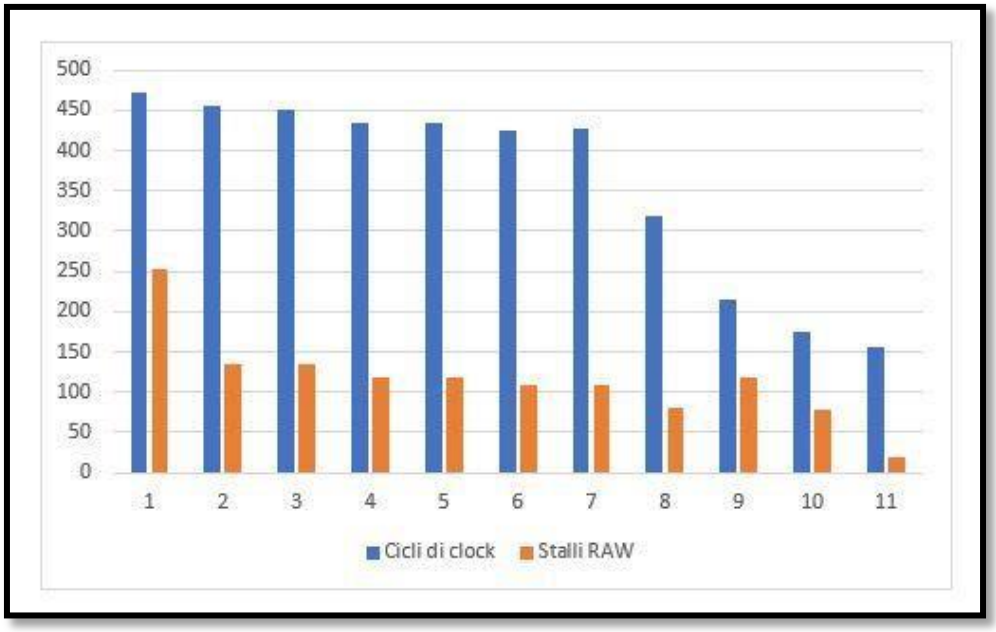
Versione 10: Cercando di separare le operazioni di prodotto, somma e scrittura del risultato con le operazioni di caricamento degli elementi successivi, utilizzando due batterie di registri, si riesce a eliminare molti degli stalli RAW a discapito della leggibilità. Performance: *175* cicli di clock, CPI *1.563*, *77* stalli **RAW**.

Versione 11: Una versione ancora più estremizzata della **versione 10**. Performance: *157* cicli di clock, CPI *1.377*, *20* stalli **RAW**.

Riepilogo performance:

Versioni	Cicli di clock	**CPI**	Stalli RAW	Stalli strutturali	Stalli Branch	Code Size
1	473	1.399	252	36	44	128 byte
2	455	1.346	135	54	55	128 byte
3	452	1.337	135	48	44	128 byte
4	434	1.284	117	48	44	128 byte
5	434	1.284	117	39	44	128 byte
6	425	1.257	108	39	44	128 byte
7	428	1.266	108	45	44	128 byte
8	319	1.512	81	27	26	132 byte
9	215	1.920	117	9	0	448 byte
10	175	1.563	77	33	0	448 byte
11	157	1.377	20	33	0	456 byte

= presenza di stalli WAR



Modifiche del codice:

```
loop:
    L.D     f0, 0(r1)
    L.D     f1, 0(r2)
    MUL.D   f2, f1, f0
    ADD.D   f3, f3, f2
    DADDI   r1, r1, 8
    DADDI   r2, r2, 8
    DADDI   r4, r4, -1
    BNEZ    r4, loop
```

Ver.1

```
loop:
    L.D     f0, 0(r1)
    L.D     f1, 0(r2)
    MUL.D   f2, f1, f0
    DADDI   r1, r1, 8
    DADDI   r2, r2, 8
    DADDI   r4, r4, -1
    ADD.D   f3, f3, f2
    BNEZ    r4, loop
```

Ver.2

```
loop:
    L.D     f0, 0(r1)
    L.D     f1, 0(r2)
    DADDI   r1, r1, 8
    MUL.D   f2, f1, f0
    DADDI   r2, r2, 8
    DADDI   r4, r4, -1
    ADD.D   f3, f3, f2
    BNEZ    r4, loop
```

Ver.5

```
controllo:
    LW      r4, op(r0)
    S.D     f3, 0(r3)
    DADDI   r3, r3, 8
    DADDI   r5, r5, -1
    DADDI   r6, r6, -1
    DADDI   r7, r7, -1
    SUB.D   f3, f3, f3
    BNEZ    r6, righe
    LW      r6, counterR(r0)
    J       endif_1
```

Ver.1

```
controllo:
    LW      r4, op(r0)
    S.D     f3, 0(r3)
    DADDI   r5, r5, -1
    DADDI   r6, r6, -1
    DADDI   r7, r7, -1
    SUB.D   f3, f3, f3
    DADDI   r3, r3, 8
    BNEZ    r6, righe
    LW      r6, counterR(r0)
    J       endif_1
```

Ver.4

```
controllo:
    LW      r4, op(r0)
    DADDI   r5, r5, -1
    DADDI   r6, r6, -1
    S.D     f3, 0(r3)
    DADDI   r7, r7, -1
    SUB.D   f3, f3, f3
    DADDI   r3, r3, 8
    BNEZ    r6, righe
    LW      r6, counterR(r0)
    J       endif_1
```

Ver.5

```
controllo:
    LW      r4, op(r0)
    DADDI   r5, r5, -1
    DADDI   r6, r6, -1
    DADDI   r7, r7, -1
    S.D     f3, 0(r3)
    SUB.D   f3, f3, f3
    DADDI   r3, r3, 8
    BNEZ    r6, righe
    LW      r6, counterR(r0)
    J       endif_1
```

Ver.6

```
controllo:
    LW      r4, op(r0)
    DADDI   r5, r5, -1
    DADDI   r6, r6, -1
    DADDI   r7, r7, -1
    S.D     f3, 0(r3)
    DADDI   r3, r3, 8
    SUB.D   f3, f3, f3
    BNEZ    r6, righe
    LW      r6, counterR(r0)
    J       endif_1
```

Ver.7

```
loop:
    L.D     f0, 0(r1)
    L.D     f1, 0(r2)
    L.D     f2, 8(r1)
    L.D     f3, 8(r2)
    L.D     f4, 16(r1)
    L.D     f5, 16(r2)
    MUL.D   f6, f1, f0
    MUL.D   f7, f2, f3
    MUL.D   f8, f4, f5
    ADD.D   f9, f6, f7
    ADD.D   f10, f9, f8
```

Ver.8

```
loop1:
    L.D     f0, 0(r1)
    L.D     f1, 0(r2)
    L.D     f2, 8(r1)
    L.D     f3, 8(r2)
    L.D     f4, 16(r1)
    L.D     f5, 16(r2)
    MUL.D   f6, f1, f0
    MUL.D   f7, f2, f3
    MUL.D   f8, f4, f5
    ADD.D   f9, f6, f7
    ADD.D   f10, f9, f8
    S.D     f10, 0(r3)
```

Ver.9