Architettura degli Elaboratori – A.A. 2016-2017
Seconda Verifica Intermedia

Scrivere Nome, Cognome, Matricola e Corso (A/B) su tutti i fogli consegnati.
Risultati e calendario degli orali saranno comunicati via WEB appena disponibili.

Si consideri lo pseudo codice:

```
for(i=0; i<128; i++) {
      x[i]=a;
      for(j=0; j<1024; j++) {
            a=c1+y[j]+a;
            c1=c1+b;
            c2=y[j]*y[j];
            x[i] = x[i]+c2;
            y[j] = x[i]
      }
      somma = somma + x[i];
}
```

con x e y vettori di 1024 elementi mentre tutte le altre variabili sono di tipo intero. Per questo codice si richiede di:

- fornire il codice D-RISC risultante dalla compilazione secondo le regole standard
- identificare i degradi delle prestazioni nell'esecuzione su un processore D-RISC pipeline con EU parallela con EU slave che implementa la moltiplicazione fra interi mediante pipeline a 4 stadi
- ottimizzare il codice in modo da ridurre il degrado delle prestazioni e quantificare il guadagno ottenuto
- indicare il working set ed il numero complessivo di fault nel caso di linee di cache da 32 parole (σ=32). Indicare eventuali conseguenze derivanti dall'adozione di una politica di gestione della cache di tipo write-through.

Traccia di soluzione

Compilazione secondo le regole standard

```
      CLEAR Ri
fori: STORE Rx, Ri, Ra          ; x[i] = a;
      CLEAR Rj                  ; necessaria perchè nel for(i...)
forj: LOAD Ry, Rj, Ryj
      ADD Rc1, Ryj, Rtemp
      ADD Ra, Rtemp, Ra         ; a = c1+y[j]+a
      ADD Rc1, Rb, Rc1          ; c1 = c1+b
      MUL Ryj, Ryj, Rc2         ; c2 = y[j]*y[j]
      LOAD Rx, Ri, Rxi
      ADD Rc2, Rxi, Rxi
      STORE Rx, Ri, Rxi         ; x[i] = x[i] + c2
      STORE Ry, Rj, Rxi         ; y[j] = x[i]
      INC Rj
      IF< Rj, Rn1k, forj
conti:ADD Rsomma, Rxi, Rsomma   ; somma = somma + x[i]
      INC Ri
      IF< Ri, Rn128, fori
conti:END
```

Degradi delle prestazioni

```
1.  CLEAR Ri
2.  fori:  STORE Rx, Ri, Ra    ; x[i] = a;
3.  CLEAR Rj                   ; necessaria perchè nel for(i...)
4.  forj:  LOAD Ry, Rj, Ryj
5.  ADD Rc1, Ryj, Rtemp
6.  ADD Ra, Rtemp, Ra          ; a = c1+y[j]+a
7.  ADD Rc1, Rb, Rc1           ; c1 = c1+b
8.  MUL Ryj, Ryj, Rc2          ; c2 = y[j]*y[j]
9.  LOAD Rx, Ri, Rxi
10. ADD Rc2, Rxi, Rxi
11. STORE Rx, Ri, Rxi          ; x[i] = x[i] + c2
12. STORE Ry, Rj, Rxi          ; y[j] = x[i]
13. INC Rj
14. IF< Rj, Rn1k, forj
15. conti: ADD Rsomma, Rxi, Rsomma    ; somma = somma + x[i]
16. INC Ri
17. IF< Ri, Rn128, fori
18. conti: END
```

Dipendenze IU-EU

- 1 sulla 2 (solo alla prima iterazione i)
- 3 sulla 4 (a tutte le 128 “prime iterazioni j”)
- 10 sulla 11, 10 sulla 12,
- 13 sulla 14
- 16 sulla 17

Dipendenze EU-EU

- 8 sulla 10

Degradi legati ai salti

- 14 è un salto solitamente preso (retroazione sulla IM)

- lo stesso vale per 17

Valutazione del degrado (prima iterazione i e j)

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
IM		CLEAR	STORE		CLEAR	LOAD		ADD	ADD	ADD	MUL	LOAD	ADD	STORE						STORE	INC	IF		ADD	LOAD			
IU			CLEAR	STORE	STORE	CLEAR	LOAD	LOAD	ADD	ADD	ADD	MUL	LOAD	ADD	STORE	STORE	STORE	STORE	STORE	STORE	STORE	INC	IF	IF	ADD	LOAD			
DM						STORE			LOAD					LOAD							STORE	STORE						LOAD	
Eumaster				CLEAR			CLEAR			LOAD	ADD	ADD	ADD	MUL	LOAD	ADD	ADD	ADD	ADD				INC					LOAD
Euslave															MUL	MUL	MUL	MUL										

Ci sono due bolle iniziali legate alle istruzioni di inizializzazione **`dei`** registri contatori di iterazione. La dipendenza della **`ADD`** sulla **`STORE`** è influenzata dalla dipendenza EU EU della **`MUL`** sulla **`ADD`** e provoca la bolla più lunga. La dipendenza della **`INC`** sulla **`IF<`** introduce una ulteriore bolla da 1 ciclo e un'ulteriore bolla da 1 è legata al salto di fine ciclo for j (salto preso).

Degradi ai fini del solo calcolo del ciclo interno

Consideriamo solo le dipendenze IU-EU

- 10 sulla 11
- 13 sulla 14

e la dipendenza EU-EU

- 8 sulla 10

Insieme all'effetto del salto di fine ciclo.

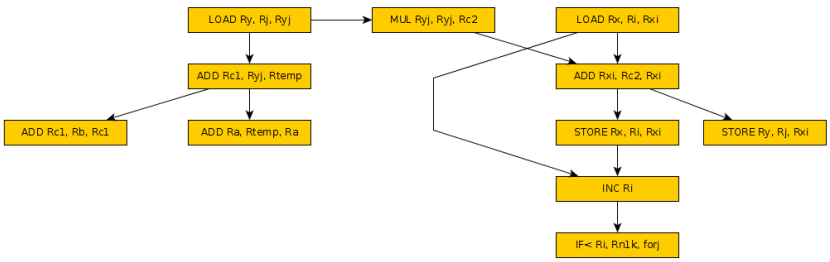
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
IM		LOAD	ADD	ADD	ADD	MUL	LOAD	ADD	STORE						STORE	INC	IF<		ADD	LOAD			
IU			LOAD	ADD	ADD	ADD	MUL	LOAD	ADD	STORE	STORE	STORE	STORE	STORE	STORE	STORE	INC	IF<	IF<	ADD	LOAD			
DM				LOAD					LOAD							STORE	STORE					LOAD	
Eumaster					LOAD	ADD	ADD	ADD	MUL	LOAD	ADD	ADD	ADD	ADD				INC					LOAD
Euslave										MUL	MUL	MUL	MUL										

Otteniamo un tempo di servizio pari a

$$T = \frac{18}{11} t$$

Ottimizzazione del codice

Il grafo data flow è il seguente:



Visto il peso della dipendenza EU-EU conviene anticipare il più possibile la **`MUL`** che può essere effettuata immediatamente dopo la **`LOAD`** che carica Ryj.

```
      CLEAR Ri
fori: STORE Rx, Ri, Ra          ; x[i] = a;
      CLEAR Rj                  ; necessaria perchè nel for(i...)
forj: LOAD Ry, Rj, Ryj, "prefetch+non deallocare"
      MUL Ryj, Ryj, Rc2         ; c2 = y[j]*y[j]
      ADD Rc1, Ryj, Rtemp
      ADD Ra, Rtemp, Ra         ; a = c1+y[j]+a
      ADD Rc1, Rb, Rc1          ; c1 = c1+b
      LOAD Rx, Ri, Rxi, "prefetch"
```

```
       ADD Rc2, Rxi, Rxi
       STORE Rx, Ri, Rxi          ; x[i] = x[i] + c2
       STORE Ry, Rj, Rxi          ; y[j] = x[i]
       INC Rj
       IF< Rj, Rn1k, forj
conti: ADD Rsomma, Rxi, Rsomma    ; somma = somma + x[i]
       INC Ri
       IF< Ri, Rn128, fori
conti: END
```

		0	1	2	3	4	5	6	7	8	9	10
IM		LOAD	MUL	ADD	ADD	ADD	ADD	STORE				
IU			LOAD	MUL	ADD	ADD	ADD	ADD	STORE	STORE	STORE	STORE
DM				LOAD								
Eumaster					LOAD	MUL	ADD	ADD	ADD	ADD	ADD	
Euslave							MUL	MUL	MUL	MUL		

Tuttavia in questo caso notiamo che rimane un effetti consistente sulla **STORE**. Dal grafo data flow si nota che possiamo anticipare la **LOAD Rx, Ri, Rxi** ma anche la **INC Ri**, avendo cura di utilizzare per la **STORE Rx, Ri, Rxi** un registro base anticipato. Possiamo inoltre utilizzare il salto ritardato per eliminare la bolla da salto preso, utilzzando la seconda **STORE** per riempire lo slot.

```
       CLEAR Ri
fori:  STORE Rx, Ri, Ra           ; x[i] = a;
       CLEAR Rj                   ; necessaria perchè nel for(i...)
forj:  LOAD Ry, Rj, Ryj, “prefetch+non deallocare”
       MUL Ryj, Ryj, Rc2          ; c2 = y[j]*y[j]
       ADD Rc1, Ryj, Rtemp
       ADD Ra, Rtemp, Ra          ; a = c1+y[j]+a
       ADD Rc1, Rb, Rc1           ; c1 = c1+b
       LOAD Rx, Ri, Rxi, “prefetch”
       ADD Rc2, Rxi, Rxi
       INC Rj
       STORE Rx, Ri, Rxi          ; x[i] = x[i] + c2
       IF< Rj, Rn1k, forj, delayed
       STORE Ry’, Rj, Rxi         ; y[j] = x[i]
conti: ADD Rsomma, Rxi, Rsomma    ; somma = somma + x[i]
       INC Ri
       IF< Ri, Rn128, fori
conti: END
```

In questo caso riusciamo a ridurre a 1t il degrado delle prestazioni rispetto alle prestazioni ideali:

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IM		LOAD	MUL	ADD	ADD	ADD	LOAD	ADD	INC	STORE		IF<	STORE	LOAD			
IU			LOAD	MUL	ADD	ADD	ADD	LOAD	ADD	INC	STORE	STORE	IF<	STORE	LOAD		
DM				LOAD					LOAD				STORE		STORE	LOAD	
Eumaster					LOAD	MUL	ADD	ADD	ADD	LOAD	ADD	INC					LOAD
Euslave							MUL	MUL	MUL	MUL							

A questo punto abbiamo ottenuto un

$$T = \frac{12}{11}\ t$$

Con un guadagno netto di $\frac{6t}{11}$ rispetto al caso non ottimizzato.

Notare l’utilizzo dei flag sulle **LOAD** dettati dalla composizione del Working Set di cui discutiamo nel prossimo paragrafo.

Ulteriori ottimizzazioni

Possiamo considerare che la x[i] di fatto viene utilizzata nel ciclo for(j...) per accumulare valori che vanno consolidati solo alla fine del for(j...) prima di passare ad una successiva iterazione i (i++). Dunque a livello di codice potremmo rimuovere sia la **`LOAD Rx, Ri, Rxi`** nel ciclo interno, portandola fuori prima di partire con il for(j...) sia la **`STORE Rx, Ri, Rxi`**, portandola fuori dopo l'ultima iterazione del for(j ...). Dunque potremmo avere:

```
       CLEAR Ri
fori:  STORE Rx, Ri, Ra          ; x[i] = a;
       CLEAR Rj                  ; necessaria perchè nel for(i...)
       LOAD Rx, Ri, Rxi, "prefetch" ; estratta dal ciclo interno
forj:  LOAD Ry, Rj, Ryj, "prefetch+non deallocare"
       MUL Ryj, Ryj, Rc2         ; c2 = y[j]*y[j]
       ADD Rc1, Ryj, Rtemp
       ADD Ra, Rtemp, Ra         ; a = c1+y[j]+a
       INC Rj
       ADD Rc1, Rb, Rc1          ; c1 = c1+b
       ADD Rc2, Rxi, Rxi
       IF< Rj, Rn1k, forj, delayed
       STORE Ry', Rj, Rxi        ; y[j] = x[i]
conti: STORE Rx, Ri, Rxi         ; x[i] = x[i] + c2
       ADD Rsomma, Rxi, Rsomma   ; somma = somma + x[i]
       INC Ri
       IF< Ri, Rn128, fori
conti: END
```

che avrebbe portato all'efficienza 1 nell'esecuzione del ciclo più interno (ora da 9 istruzioni):

		0	1	2	3	4	5	6	7	8	9	10	11	12
IM		LOAD	MUL	ADD	ADD	INC	ADD	ADD	IF<	STORE	LOAD			
IU			LOAD	MUL	ADD	ADD	INC	ADD	ADD	IF<	STORE	LOAD		
DM				LOAD								STORE	LOAD	
Eumaster					LOAD	MUL	ADD	ADD	INC	ADD	ADD			LOAD
Euslave							MUL	MUL	MUL	MUL				

Memoria

Il working set è composto da una linea di X (caratterizzato dalla sola proprietà di località) e da tutto Y (caratterizzato da località e riuso) oltre che da tutto il codice (ancora caratterizzato da località e riuso). Dunque servirebbero 1 linea per il codice, una per la parte di X che contiene x[i], 32 linee per Y.
In assenza di pre-fetching, avremmo dunque un numero di fault fisiologici pari a

$$1+ \frac{128}{32} + \frac{1024}{32} = 1+4+32 = 37$$

Utilizzando il flag prefecth sia sulla **`LOAD Rx, Ri, Rxi`** che sulla **`LOAD Ry, Rj, Ryj`** si riduce il numero di fault a 3 (1 per il codice, uno per X e uno per Y) (se e solo se la banda di memoria è sufficiente a leggere asynchronamente la linea successiva per X e Y mentre vengono consumate le linee correnti) .

In caso di utilizzo della politica write-through va controllato che il sottosistema di memoria sopra la cache abbia una banda sufficiente ad eseguire 2 **`STORE`** ogni 13*t*, ma va considerato che ogni 32 iterazioni serve anche leggere un'intera linea di Y (quando i=0). Infine, qualora la capacità della cache disponibile per il nostro programma fosse inferiore alle 10 linee, andrebbero considerati anche i fault non fisiologici per il rimpiazzo delle linee di Y non più presenti.