Architettura degli elaboratori – A.A. 2015-2016
Secondo appello sessione estiva – 5/7/2016

Si riporti su ciascun foglio consegnato Nome, Cognome Numero di matricola e corso (A o B).
I risultati e i calendari degli orali saranno pubblicati sulle pagine web didawiki e/o dei docenti appena disponibili.
Gli orali si svolgeranno immediatamente dopo la pubblicazione dei risultati.

Domanda 1.

Dato un vettore X di N posizioni, l'operazione di convoluzione

a) sostituisce ad ogni posizione *i* diversa dalla prima e dall'ultima la media aritmetica dei valori x_i, x_{i-1} e x_{i+1}
b) calcola la somma di tutte le differenze fra nuovi e vecchi valori nelle diverse posizioni del vettore.

La convoluzione è ripetuta fino a quando la somma delle differenze scende sotto una certa soglia. Lo pseudo codice può essere il seguente:

```
do {
      diff = 0;
      for(int i=1; i<N-2; i++) {
            new = (x[i-1]+x[i]+x[i+1])/3;
            diff = diff + (new-x[i]);
            x[i]=new;
      }
} while(diff > threshold);
```

Si scriva il codice D-RISC che calcola l'operazione di convoluzione su vettori di interi e si valuti il tempo di completamento della sola parte in corsivo nello pseudocodice (una iterazione del **`do-while`**) su due tipi di processore:

a) un processore D-RISC pipeline, scalare, con EU slave che calcola una divisione fra interi in 4t
b) un processore D-RISC pipeline con multithreading simmetrico a due vie e con la stessa unità EU slave del caso precendente. In questo caso si assuma che i due thread eseguando i calcoli della metà inferiore e superiore del vettore, rispettivamente

discutendone le eventuali differenze o similitudini.

Domanda 2.

Si consideri un pipeline di quattro unità firmware U_1, U_2, U_3, U_4, implementate su un unico chip. Ciascuna delle U_i riceve un singolo dato da 32 bit da U_{i-1}, elabora un risultato ancora da 32 bit in k_i cicli di clock ($\sum k_i = K$ e $\forall i: k_i \geq 2\tau$) e manda il risultato a U_{i+1}. Il risultato è calcolato da ciascuna delle U_i mediante k_i microoperazioni $\mu op_1, \ldots, \mu op_{ki}$ e vale che per qualunque $i \in [1,k_i]$

$$W(\mu op_{i-1}) \cap R(\mu op_i) \neq \varnothing$$

U_1 riceve valori in ingresso da un'unità esterna, con un tempo di interarrivo $T_a = 2\tau$, e U_4 manda risultati ad una unità esterna sempre pronta a riceverli. Si calcoli (fornendo tutte le necessarie motivazioni) il tempo di completamento necessario al pipeline per produrre i risultati relativi a *m* ingressi.

Domanda 3.

Dato il microcodice:

1. (RDY=0) nop,
   (=1) (IN+I)%N→ I, I→J, reset RDY, set ACK, 1
2. (segno(M[(I-1)%N] – M[(I+1)%N]) = 0) M[I] → M[J], 0
   (=1) M[J] → M[I], 0

si discutano le caratteristiche della memoria M dettate dalla struttura del micro codice.

Bozza di soluzione

Domanda 1

Il risultato della compilazione dello pseudo codice in D-RISC è il seguente (si fa vedere per completezza la compilazione dell'intero pseudocodice, la parte di interesse è quella che va dalla istruzione 3 alla 15):

```
 1          SUB Rn, #1, Rn2
 2  while:  ADD R0,R0,Rdiff          diff = 0;
 3  for:    SUB Ri, #1, R1
 4          LOAD RbaseA,R1,R1        x[i-1]
 5          LOAD RbaseA, Ri, R2      x[i], serve dopo
 6          ADD R1, R2, R1           x[i-1]+x[i]
 7          ADD Ri, #1, R3
 8          LOAD RbaseA, R3, R3      x[i+1]
 9          ADD R1, R3, R1           x[i-1]+x[i]+x[i+1]
10          DIV R1, #3, R1           new
11          SUB R1, R2, R2           new -x[i]
12          ADD Rdiff, R2, Rdiff     diff = diff + (new - x[i])
13          INC Ri
14          IF< Ri, Rn2, for         fine ciclo for
15          if> Rdiff, Rthreshold, while   fine ciclo while
16          END
```

Simulando l'esecuzione del codice su un processore D-RISC pipeline con EU parallela e EU slave che esegue la divisione fra interi in 4*t*, otteniamo:

	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
IM	SUB	ADD	SUB	LOAD		LOAD	ADD	LOAD			ADD	DIV	SUB	ADD						INC	IF<			SUB		
IU		SUB	ADD	SUB	LOAD	LOAD	LOAD	ADD	LOAD	LOAD	LOAD	ADD	DIV	SUB	ADD	ADD	ADD	ADD	ADD	ADD	INC	IF<	IF<		SUB	
DM							LOAD	LOAD				LOAD														
EUm			SUB	ADD	SUB			LOAD	LOAD	ADD			LOAD	ADD	DIV	SUB	SUB	SUB	SUB	SUB	ADD	INC				SUB
EUdiv																DIV	DIV	DIV	DIV							

Dunque il tempo per il completamento della singola iterazione del ciclo for è 21*t*. Se ne fanno N-2 e pertanto il valore cercato sarà 21(N-2)*t* (ε = 12/21 = 0.57).

Utilizzando un multithreading simmetrico (dunque eseguendo istruzioni di due thread diversi nella stessa "istruzione lunga" a due vie/slot), la situazione non cambia. Avremo le stesse dipendenze (con la stessa distanza fra l'istruzione che induce la dipendenza e l'istruzione su cui la dipendenza è indotta) e dunque il tempo di completamento sarà lo stesso del caso senza multithreading in termini di numero di cicli. Tuttavia, poiché in un ciclo delle varie unità del pipeline si eseguono due istruzioni, il tempo di completamento assoluto sarà dimezzato e pari a 21*((N-2)/2)t.

Diversa sararebbe stata la situazione nel caso avessimo utilizzato un processore D-RISC pipeline con supporto per multithreading interleaving a due vie (nota: questo non era richiesto nel testo, è qui solo per completezza). Le istruzioni da eseguire in ciascuno dei due thread sono le istruzioni 3-14 nel listato, che lavorano su valori diversi di Ri e Rn2. Le istruzioni sono prelevate da IM e mandate alla IU una dal primo thread e una dal secondo:

Thread 1																																		
Thread 2																																		
	-2	-1	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	27	28	29	30	31	32	33
IM	SUB	*SUB*	ADD	*ADD*	SUB	*SUB*	LOAD	*LOAD*	LOAD	*LOAD*	ADD	*ADD*	ADD	*ADD*	LOAD		*LOAD*	ADD	*ADD*	DIV	*DIV*	SUB	*SUB*					ADD	*ADD*	INC	*INC*	IF<	*IF<*	SUB	*SUB*	
IU		SUB	*SUB*	ADD	*ADD*	SUB	*SUB*	LOAD	*LOAD*	LOAD	*LOAD*	ADD	*ADD*	ADD	*ADD*	LOAD	*LOAD*	*LOAD*	ADD	*ADD*	DIV	*DIV*	SUB					*SUB*	ADD	*ADD*	INC	*INC*	IF<	*IF<*	SUB	*SUB*	
DM								LOAD	*LOAD*	LOAD	*LOAD*						LOAD	*LOAD*																		
EUm			SUB	*SUB*	ADD	*ADD*	SUB	*SUB*		LOAD	*LOAD*	LOAD	*LOAD*	ADD	*ADD*	ADD	*ADD*		LOAD	*LOAD*	ADD	*ADD*	DIV	*DIV*	SUB	SUB	SUB	SUB	*SUB*	ADD	*ADD*	INC	*INC*			SUB	*SUB*
EUdiv																								DIV	DIV	DIV	DIV									
																									DIV	*DIV*	*DIV*	*DIV*								

Come si vede due iterazioni (una nel thread "basso" e una nel thread "alto", vengono completate in 31*t* e dunque il calcolo di cui si vuole misurare il tempo di esecuzione impiegherà 31*t* * (N-2)/2 = 15.5(N-2)*t* (ε = (12 / (31/2)) = 24/31 = 0.77) risultando dunque più efficiente del codice che esegue sul processore pipeline senza multithreading. Il

miglioramento delle prestazioni è dovuto al fatto che la decodifica interallacciata delle istruzioni dei due thread "allontana" le istruzioni di un thread che causano le dipendenze dalle istruzioni su cui la dipendenza è indotta, mitigandone gli effetti. Resta la dipendenza EU-EU fra la DIV e la SUB, che di fatto è la maggior sorgente di degrado delle prestazioni in entrambi i casi.

Il codice derivato dal processo di compilazione standard si poteva peraltro ottimizzare in diversi punti (per esempio invertendo le prime due LOAD si eliminava una bolla, l'uso del salto ritardato poteva compensare la bolla di fine ciclo, il calcolo di i+1 e i-1 poteva ugualmente essere anticipato rispetto all'inizio delle relative LOAD, etc.) ma questo non era richiesto nel testo.

Nota bene: la definizione di convoluzione è inesatta ("semplificata" ai fini dell'esercizio), dal momento che tutti gli aggiornamenti degli elementi x_i dovrebbero essere calcolati a partire dal valore corrente di xi, xi-1 e xi+1, mentre nel codice dato riscriviamo subito xi e quindi il cacolo di xi+1 considera il nuovo valore anzichè il vecchio. In una convoluzione vera, si aggiornano i valori in un secondo vettore e prima di cominciare una nuova iterazione del while si scambiano i puntatori dei due vettori utilizzati:

```
do {
        diff = 0;
        for(int i=1; i<N-2; i++) {
                new = (x[i-1]+x[i]+x[i+1])/3;
                diff = diff + (new-x[i]);
                y[i]=new;
        }
        swap_pointers(x[], y[]);
} while(diff > threshold);
```

Domanda 2.

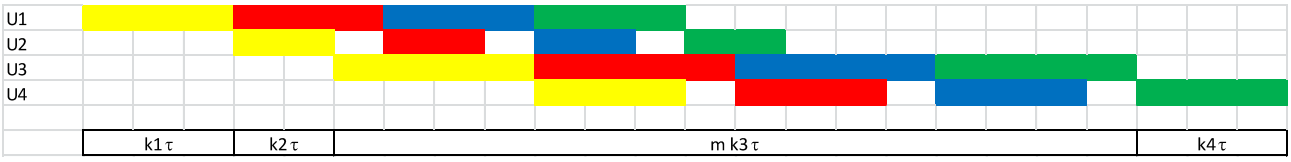
E' logico assumere che le comunicazioni fra gli stadi (e fra la sorgente ed U_1 e fra U_4 e la destinazione) avvengano secondo un protocollo che fa uso di indicatori a transizione di livello. Pertanto possiamo assumere che ciascuno stadio sia controllato da un microcodice tipo:

1. (RDYin=0) nop,
   (=1) μop_1, reset RDYin, set ACKin, 1
2. μop_2, 2
   ...

$k_{(i-1)}$. $\mu op_{k(i-1)}$, k_i

k_i. (ACKout=0) nop,
(=1) μop_{ki}, reset ACKout, set RDYout, 0 // *μop_{ki} scriverà in OUT*

Questo ci permette di concludere che per ciascuno degli stadi il tempo di completamento dell'unica operazione esterna sarà di $k_i \tau$, al netto di eventuali attese relative al RDYin e/o all'ACKout. Supponiamo che $k_1 = 3$, $k_2 = 2$, $k_3 = 4$ e $k_4 = 3$ (valori a caso che soddisfano le condizioni poste nel testo e che hanno massimo e minimo in posizioni arbitrarie) e proviamo a immaginare come potrebbe avvenire l'esecuzione del pipeline mediante un diagramma simile a quelli utilizzati a lezione per lo studio del pipeline. Notiamo innanzitutto che la sorgente esterna dei dati non è un collo di bottiglia per il pipeline, visto che rende disponibili i dati alla velocità minima permessa dal protocollo di comunicazione con indicatori a transizione di livello. Dunque, rappresentato un τ con una casella e l'esecuzione di ognuno dei task con colori diversi:



Possiamo vedere come si utilizzino per eseguire *m* task, *m* $k_m \tau$, con $k_m = \max \{ k_1, k_2, k_3, k_4\}$ cui vanno aggiunti un numero di cicli pari alla sommatoria dei cicli spesi nei rimanenti stadi:

$$k_1 \tau + k_2 \tau + m\, k_3 \tau + k_4 \tau$$

Questa espressione può essere riscritta (scomponendo *(m k3)* in *(m-1)k3 + k3*) come

$$k_1 \tau + k_2 \tau + (m-1)\, k_3 \tau + k_3 \tau + k_4 \tau = k_1 \tau + k_2 \tau + k_3 \tau + k_4 \tau + (m-1)\, k_3 \tau = K \tau + (m-1)\, k_3 \tau$$

e quindi, considerando il generico max k_i

$$\tau\, (K + (m-1)\, (\max k_i))$$

Se avessimo scelto valori diversi per i diversi k_i nulla sarebbe cambiato: il tempo di esecuzione è dominato dal prodotto fra il numero dei task in ingresso e la latenza dello stadio più lento, cui va aggiunto (una volta sola) il peso degli altri stadi.

Domanda 3.

Il microcodice

1. (RDY=0) nop,
   (=1) (IN+I)%N→ I, I→J, reset RDY, set ACK, 1
2. (segno(M[(I-1)%N] – M[(I+1)%N]) = 0) M[I] → M[J], 0
   (=1) M[J] → M[I], 0

utilizza una memoria da N posizioni, dal momento che vengono utilizzati come indirizzi valori modulo N. La memoria è utilizzata per leggere due valori distinti che servono a creare una variabile di condizionamento. Pertanto deve avere (almeno) due ingressi di tipo indirizzo non controllati da commutatori (più in generale, non dipendenti da ingressi dalla parte controllo) uscite da ALU che facciano solo l'operazione richiesta (decremento modulo N nel primo caso e incremento modulo N nel secondo). Nella stessa microistruzione la memoria viene indirizzata per la scrittura con I o J e per la lettura con J o I. Dunque complessivamente, per garantire il funzionamento secondo quando specificato dal micro programma e la condizione di correttezza, occorrerà una memoria con 4 ingressi indirizzo, tre che comandano tre diversi commutatori per la lettura e uno che comanda il selettore di scrittura.