Si consideri un’architettura **D-RISC Pipeline**, con EU parallela **(EU master per le aritmetico logiche corte ed EU slave da 4 stadi per la moltiplicazione e divisione fra interi)**. Il sistema è dotato di **cache** di **primo** livello **associativa su insiemi a 4 vie** con linee (blocchi) da **σ=8 parole**.
Si consideri il seguente frammento di pseudocodice:

```
int x[N], a[N], b[N], z[N];
…
for(int i=0; i<N; i++) {
        x[i] = (a[i]*b[i])+c;
        if(b[i]<N)    // NOTA: si suppone che in ¾ dei casi b[i] risulti minore di N
                {z[b[i]] = x[i]+a[i]; }
                        }
```

Assumiamo che **N=1024**, e che **tutti i vettori ed il codice** risiedano in **cache (on chip con la CPU).**

Si richiede:

1) **di fornire il codice D-RISC risultante dalla compilazione secondo le regole standard;**
2) **di indicare il numero di cache fault in caso di prefetch e senza prefetch;**
3) **l’individuazione di tutte le cause di degrado delle prestazioni;**
4) **di valutare il tempo di completamento ideale e reale del programma (limitatamente al ciclo);**
5) **l’individuazione di possibili ottimizzazioni;**
6) **la valutazione del guadagno di prestazioni ottenuto**.

Bozza di soluzione

Numero di fault

Il **numero** di fault **minimo** (**fisiologici**) è dato dal numero di fault **per il codice** (**2** in questo caso) e da **N/σ fault** per ciascuno dei vettori **a** e **b** **e** per il vettore **z** **(in caso di assenza di prefetch** e usando **write through** per **z)** oppure da **2** fault per il **codice** e **1** per i vettori **a** e **b** (che sono in lettura) **.** Il vettore **x** è prima **scritto** e poi letto (prodotto e consumato internamente), quindi possiamo **trascurare** il **tempo** di **trasferimento** dai livelli superiori della gerarchia di memoria in caso di **fault**. Il vettore **z** è **in sola scrittura** ma, ***non*** essendo scritto **interamente**, il **fault** è uguale (a parte il comando eseguito) ad un **fault di lettura**.

Codice D-RISC

Il programma può essere compilato in D-RISC come segue:

```
1.          CLEAR Ri
2.   loop:  LOAD RbaseA, Ri, Rai      // legge a[i]
3.          LOAD RbaseB, Ri, Rbi      // legge b[i]
4.          MUL Rai, Rbi, Rtemp       // calcola a[i]*b[i] (Rtemp si riuserà)
5.          ADD Rc, Rtemp, Rxi        // calcola a[i]*b[i] + c
6.          STORE Rbasex, Ri, Rxi     // memorizza x[i]
7.          IF>= Rbi, RN, cont        // controlla se b[i] !< N (meno salti presi)
8.          ADD Rxi, Rai, Rtemp       // se è minore calcola a[i]+x[i]
9.          STORE Rbasez, Rbi, Rtemp, non_deallocare // lo memorizza in z[b[i]]
10.  cont:  INC Ri                    // i++
11.         IF< Ri, RN, loop          // se i<N esegui un'altra iterazione
12.         END
```

Cause di degrado delle prestazioni

Abbiamo:

- una dipendenza **EU-EU** fra la **MUL** e la **ADD successiva** (4→5)
- una dipendenza **EU-EU** fra le **ADD** (5→8) *che non causa bolle*
- una dipendenza **IU-EU** fra le **ADD** e le **STORE successive** (5→6 e 8→9)
- una dipendenza **EU-IU** fra la **INC** e la **IF< (10→11)**
- una dipendenza **IU-EU** fra la **seconda LOAD** e la **IF>= (3→7)** *che non causa bolle*

Inoltre, abbiamo **bolle da salto**

- nel **25%** delle **iterazioni** per via della **IF>=** (7) **(con IF< sarebbe stato il 75%)**
- in **tutte** le **iterazioni**, ***tranne l'ultima***, per via della **IF<** (11)

Tempo di completamento

Il tempo di completamento ideale è, come sempre, dato dal numero di istruzioni, cioè **10t** nel nostro caso. Pwer il tempo di completamento reale del programma, consideriamo l'esecuzione di **una iterazione** nei **due casi (b[i]<N o no)**:

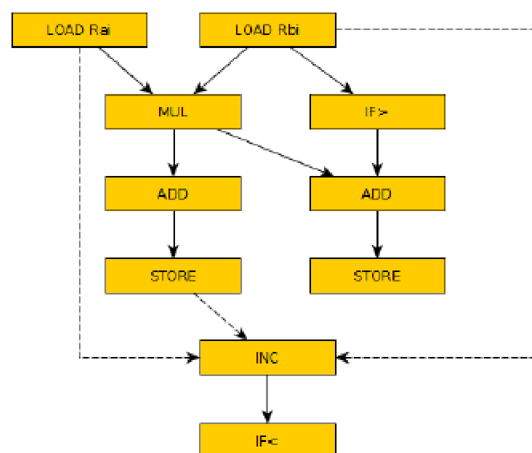
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
IM	LD	LD	MUL	ADD	ST							IF>	ADD	ST	INC		IF<		END	LD			
IU		LD	LD	MUL	ADD		ST					ST	IF>	ADD	ST	ST	INC	IF<	IF<	END	LD		
DM			LD	LD									ST				ST					LD	
Eum				LD	LD	MUL	ADD				ADD				ADD			INC					LD
EU*							MUL	MUL	MUL	MUL													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18				
IM	LD	LD	MUL	ADD	ST							IF>	ADD	INC	IF<		END	LD					
IU		LD	LD	MUL	ADD		ST					ST	IF<	~~ADD~~	INC	IF<	IF<	~~END~~	LD				
DM			LD	LD									ST							LD			
Eum				LD	LD	MUL	ADD				ADD					INC					LD		
EU*							MUL	MUL	MUL	MUL													

Abbiamo **19t** per iterazione con **then** eseguito e **17t** per i **rimanenti casi** dunque per ***ciascuna*** delle ***iterazioni*** spendiamo

$$((3/4)*19+(1/4)*17)t) = 18.5 t$$

Ottimizzazioni

Il grafo data flow (solo per il ciclo) è il seguente:



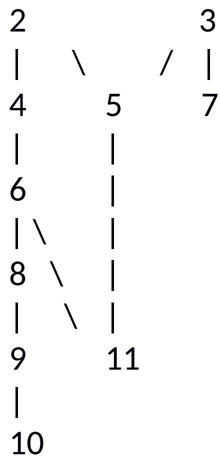
(per errore, è inserito l'arco tra MUL e la ADD di destra, che va tolto, mentre va inserito un arco tra l'ADD di sinistra e quella di destra).

La **prima store (la 6)** può essere chiaramente **posticipata** per **alleviare** il problema della **dipendenza con il calcolo** del valore da memorizzare (**4 e 5**).
La **seconda store (la 9)** invece deve essere **eseguita esclusivamente nel ramo then** e dunque **non** può essere **posticipata**.
La **10** può essere **anticipata purchè** mantenuta **successiva alle load** e **purchè** la **6** **avvenga con il registro base anticipato (chiamato Rbasex-1 nella soluzione).** Questo permette di ridurre l'effetto della dipendenza INC→IF< e, a **seconda** della **posizione** di arrivo della **INC**, di **mitigare** l'effetto di un'**altra dipendenza**. Noi scegliamo di **interporla fra la prima MUL (la 4) e la ADD (la 5) successiva**:

```
      CLEAR Ri                          1     1
loop: LOAD RbaseA, Ri, Rai              2     2
      LOAD RbaseB, Ri, Rbi              3     3
      MUL Rai, Rbi, Rtemp               4     4
      INC Ri                            5     10
      ADD Rc, Rtemp, Rxi                6     5
      IF>= Rbi, RN, cont                7     7
      ADD Rxi, Rai, Rtemp2              8     8
      STORE Rbasez, Rbi, Rtemp2         9     9
cont: IF< Ri, RN, loop, delayed         10    11
      STORE Rbasex-1, Ri, Rxi           11    6
      END
```

dove la colonna di sinistra con i numeri delle istruzioni contiene una nuova numerazione delle stesse, mentre l'ultima colonna (quella a destra) contiene i numeri delle istruzioni del programma iniziale, non ottimizzato.

Il grafo data-flow per questo codice (**oltre** alla precedenza **tra 3 e 4**, non inserita) in termini della nuova numerazione delle istruzioni, è :



La simulazione mostra che **in questo caso** l'esecuzione con il ***then preso*** impiega **14t** e quella ***senza then*** **11t**.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IM	LD	LD	MUL	INC	ADD	IF>		ADD	ST			IF<		ST	LD		
IU		LD	LD	MUL	INC	ADD		IF>	ADD			ST	ST	IF<	ST	LD	
DM			LD	LD										ST		ST	LD
Eum				LD	LD	MUL	INC	ADD			ADD	ADD					
EU*							MUL	MUL	MUL	MUL							

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
IM	LD	LD	MUL	INC	ADD	IF>		ADD	IF<	ST		LD					
IU		LD	LD	MUL	INC	ADD		IF>	ADD	IF<	ST	ST	LD				
DM			LD	LD									ST	LD			
Eum				LD	LD	MUL	INC	ADD			ADD				LD		
EU*							MUL	MUL	MUL	MUL							

Dunque avremo un **tempo medio di completamento per iterazione** pari a

$$((3/4)14+(1/4)11)t = 13.25t.$$

EU out-of-order

La presenza di una unità EUmaster **out-of-order** **non** porta alcun **vantaggio** **nel** **codice** **ottimizzato**. La **seconda ADD non** può comunque essere eseguita **prima di quella che produce Rxi** perchè **legge Rxi**, (**then** **preso**), mentre nel caso di ramo **then** **non** **preso** **non** vi sono **code** di aritmetico logiche sulla **EUmaster**.

Diversa sarebbe stata la situazione **se non avessimo invertito la prima ADD e la INC**, ovvero nel codice

```
CLEAR Ri
loop: LOAD RbaseA, Ri, Rai
LOAD RbaseB, Ri, Rbi
MUL Rai, Rbi, Rtemp
ADD Rc, Rtemp, Rxi
INC Ri
IF>= Rbi, RN, cont
ADD Rxi, Rai, Rtemp2
...
```

In questo caso, la **INC** **verrebbe eseguita sulla EU master** **prima** **del completamento della ADD (in attesa del risultato della MUL).**