AE corso B – A.A. 2019-2020 – 2a prova di verifica intermedia
Tipo **A**

*Riportare nella **prima facciata** in **alto a destra** di ciascuno dei fogli consegnati **Nome**, **Cognome**, numero di **matricola** e **tipo** di compito **(A** in questo caso**)**.*

Si consideri il seguente programma in pseudolinguaggio (supponendo che N<4097):

```
d=5;
For (i=N, i>0, i--)
        {x=d+3;
         d=d*d;
         x=x MOD N;
         if (x>0)&(d>0)
                {v[i]=8*v[i]+d+x}
        }
```

Per tale programma:

1) Si fornisca la compilazione in assembler D-RISC (a cui è aggiunta, oltre alle istruzioni dell'assembler D-RISC descritto nelle note distribuite a lezione, l'istruzione *aritmetica lunga* **MOD** che calcola il resto della divisione intera) utilizzando le regole standard.
2) *a)* Si indichi il numero di parole di ogni linea della cache (σ), e
   b) si fornisca il numero di cache fault (sia con che senza prefetch) per una cache associativa su insiemi di 64K parole, con 512 insiemi di 8 linee ciascuno.
3) Si forniscano le dipendenze tra istruzioni e le prestazioni (tempo di completamento) dell'esecuzione di un ciclo in cui si esegue l'aggiornamento di v[i], su un processore D-RISC pipeline con unità *EU slave pipeline a **2** stadi* per l'esecuzione delle operazioni aritmetiche lunghe, e il tempo di completamento ideale.
4) Si individuino le cause di degrado delle prestazioni dovute a bolle.
5) Si fornisca il codice con eventuali ottimizzazioni per ridurre l'effetto delle bolle, quantificando il guadagno in termini di tempo di servizio; giustificare tale guadagno.

Bozza di soluzione

1) Il programma derivato dalla compilazione in assembler D-RISC secondo le regole standard può essere (a sinistra una numerazione delle istruzioni):

```
          ADDI    RD,#5,RD          //d=5
          ADD     R0,RN,RI          //i=N
1   LOOP: ADDI    RD,#3,RX          //x=d+3
2         MUL     RD,RD,RD          //d=d^2
3         MOD     RX,RN,RX          //x=x MOD N
4         IF<=0   RX,IND            //if x>0
5         IF<=0   RD,IND            //if d>0
6         LOAD    RBv,RI,RvI        //leggi v(i)
7         SHL     RvI,#3,RvI        //calcola 8v(i)
8         ADD     RvI,RD,RTEMP      //calcola 8v(i)+d
9         ADD     RTEMP,RX,RTEMP2   //calcola 8v(i)+d+x
10        STORE   RBv,RI,RTEMP2     //v(i)=8v(i)+d+x
11  IND:  SUBI    RI,#1,RI          //i--
12        IF>0    RI, LOOP          //fine ciclo
13        END
```

2) Cache

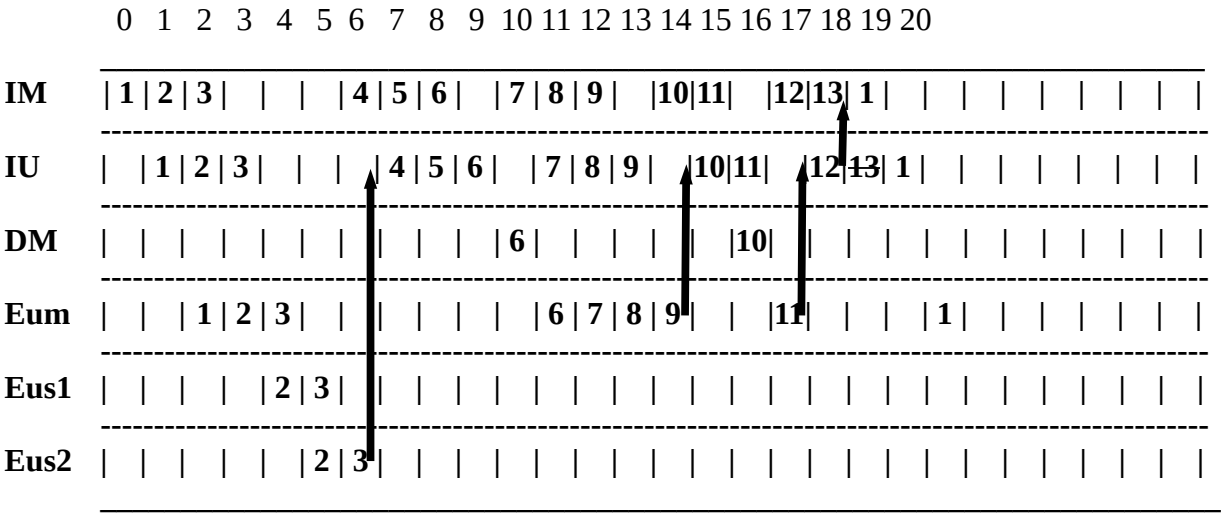
Premessa: la cache è grande a sufficienza da poter contenere sia tutto il codice che il vettore v.

a) $64K/512=2^{16}/2^9=2^7$ parole per ciascun insieme; $2^7/2^3=2^4=16$ parole per ciascuna linea. Quindi, **σ=16**;
b) ***con*** *prefetch*: **1** fault per il codice (14 istruzioni=14 parole che stanno tutte in una linea);
1 fault per v(.) (in lettura – dato che il vettore v entra tutto nella cache, la scrittura avviene alla fine della esecuzione del processo, quando si carica il nuovo processo, e quindi viene effettuata dal sistema operativo e non è a carico di questo processo).
Senza *prefetch*: **1** fault per il codice (14 istruzioni=14 parole che stanno tutte in una linea);
C(N/16) fault per v(.), dove C(y)=intero superiore di y (in lettura – dato che il vettore v entra tutto nella cache, la scrittura avviene alla fine della esecuzione del processo, quando si carica il nuovo processo, e quindi viene effettuata dal sistema operativo e non è a carico di questo processo).

3) Tempo di completamento

Le dipendenze tra le istruzioni sono dovute o al fatto che una istruzione utilizza il risultato di un'altra istruzione (ad esempio, la 3 usa il risultato della 1), oppure al fatto che una istruzione può essere eseguita o meno a seconda dell'esito di un'altra istruzione (che vale tipicamente per le istruzioni in un ramo delle IF, ad esempio, la 6 viene eseguita solo se la 4 e la 5 risultano in un salto non preso).
Nel nostro caso, le **dipendenze** sono: **1→ 3, 2→ 5, 3→4, 4 e 5 su tutte le istruzioni dalla 6 alla 10 (che altrimenti non verrebbero eseguite), 6→ 7, 7→ 8, 8→ 9, 9→ 10, 11→ 12**.

Il tempo di completamento **ideale** del ciclo è uguale al numero di istruzioni del ciclo, cioè **12t**.
Per il tempo di completamento del codice si veda il seguente schema (**si fa l'ipotesi che esista il collegamento diretto tra EUslave2 ed IU, oltre a quello tra EUslave2 ed EUmaster**, altrimenti il tempo aumenterebbe di 1t); la prima riga di EUs rappresenta la EU slave 1 e la seconda riga la EU slave 2. Per maggior chiarezza e semplicità, le istruzioni sono indicate in una unità solo nello slot in cui tale unità termina la loro elaborazione. Ad esempio, IU termina l'elaborazione della istruzione 4 nello slot 7, anche se l'ha ricevuta prima, IM termina l'elaborazione della istruzione 2 (e la passa ad IU che la preleva) nello slot 1. le frecce rappresentano il passaggio di un valore tra l'unità che lo produce e quella che è in attesa per poterlo utilizzare. Ad esempio, la freccia tra EUslave2 ed IU alla fine dello slot 6 rappresenta il passaggio del nuovo valore di Rx ad IU, che aggiorna il valore contenuto nella sua copia di tale registro.



Il tempo di completamento è di **19t**, con una efficienza di **12/19=63.158%** (5t dovuto a bolle e 1t dovuto alla load).

4)Degrado delle prestazioni dovuto a bolle

Le bolle sulla IU sono dovute alle dipendenze: **3→4** (bolla lunga 3t)**, 9→ 10** (bolla lunga 1t)**, 11→ 12** (bolla lunga 1t), oltre alla bolla da salto (lunga 1t) **12→ 1**

5)Ottimizzazione del codice

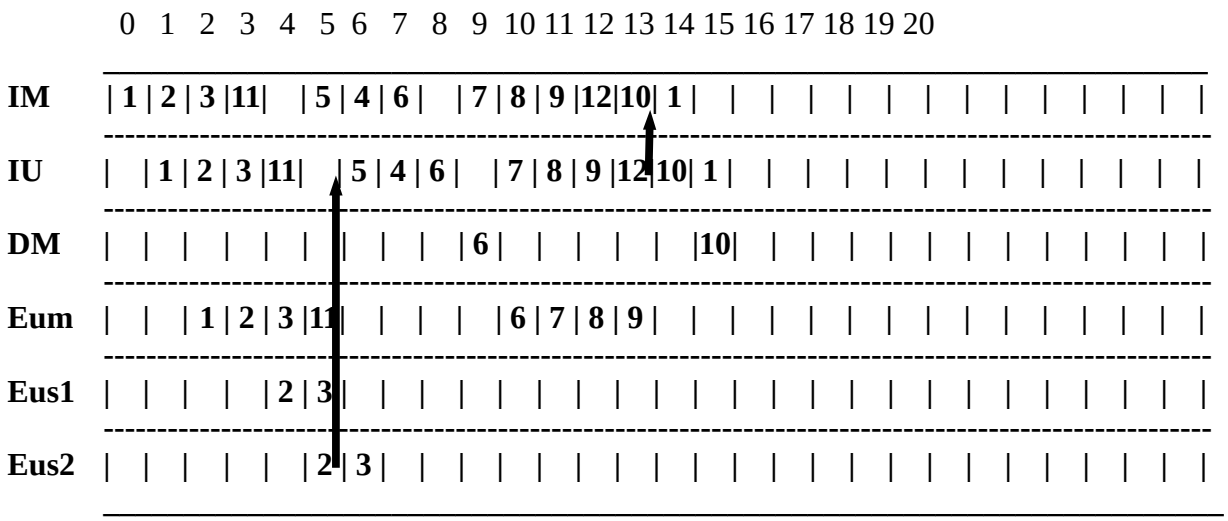
Il codice può essere ottimizzato eseguendo le seguenti modifiche:

- invertire le istruzioni 4 e 5;
- usare RBv *posticipato* (cioè *RBv'= RBv+1*) – il posticipo e non l'anticipo è dovuto al fatto che l'indice I viene decrementato e non incrementato. Questo permette di spostare la 11 dove vogliamo;
- usare la 12 *delayed* e spostare la 10 dopo la 12 **se si vuole ottimizzare il ciclo solo nell'ipotesi descritta al punto 3 del testo, cioè se i salti dei primi due IF non vengono <u>mai</u> presi. Altrimenti, questa modifica non si può fare, e quindi il tempo di completamento risulta più lungo di 1t perché la bolla del salto finale non si può eliminare**. **Una delle possibili vie di mezzo consiste nello spostare la LOAD prima degli IF, e mantenere la modifica su riportata. In questo caso la LOAD deve essere eseguita con la base di v posticipata, se la SUBI viene eseguita prima della LOAD, altrimenti (SUBI dopo di LOAD), si dovranno usare due registri base per v, uno (RBv) con il valore "normale" per la LOAD, e l'altro (Rbv') con il valore posticipato per la STORE. In quest'ultimo caso, la bolla del salto finale viene eliminata (quindi viene eliminata N volte), ma si aggiunge una istruzione inutile (l'esecuzione della LOAD) nel caso che (almeno) uno dei due salti venga preso e quindi si allunga di 1t il tempo nel caso di salti presi, il che può impattare negativamente sul tempo totale di esecuzione del codice.**

Questo porta alla seguente versione del codice:

```
        ADDI    RD,#5,RD          //d=5
        ADD     R0,RN,RI          //i=N
1   LOOP: ADDI  RD,#3,RX          //x=d+3
2       MUL     RD,RD,RD          //d=d^2
3       MOD     RX,RN,RX          //x=x MOD N
11      SUBI    RI,#1,RI          //i--
5       IF<=0   RD, IND           //if d>0
4       IF<=0   RX, IND           //if x>0
6       LOAD    RBv,RI,RvI        //leggi v(i)
7       SHL     RvI,#3,RvI        //calcola 8v(i)
8       ADD     RvI,RD,RTEMP      //calcola 8v(i)+d
9       ADD     RTEMP,RX,RTEMP2   //calcola 8v(i)+d+x
12  IND: IF>0   RI, LOOP, delayed //fine ciclo
10      STORE   RBv,RI,RTEMP2     //v(i)=8v(i)+d+x
13      END
```

da cui abbiamo il seguente schema temporale, con le ipotesi dello schema precedente:



e pertanto il tempo di completamento del programma ottimizzato è di **14t**, con un risparmio di 5t rispetto al codice precedente. Ora risulta una efficienza di **12/14=85,7%.**