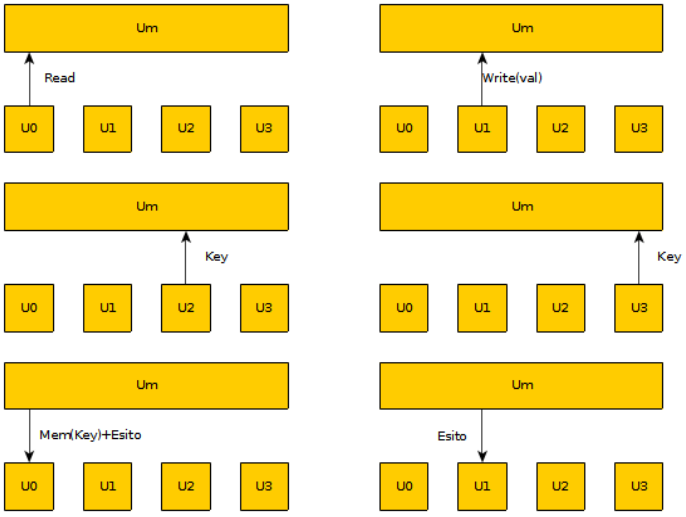
Architettura degli Elaboratori – A.A. 2015-2016
Appello 5 settembre 2016

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

Domanda 1

Una unità fw U_m implementa una mappa associativa <*chiave,valore*> su cui si possono eseguire operazioni di lettura, **read(key),** o scrittura, **write(key,val). read(key)** restituisce il valore associato alla chiave con esito OK se la chiave è presente nella mappa associativa oppure un esito NOK se la chiave non è presente). **write(key,val)** scrive **val** come valore associato a **key**, creando la chiave, se non presente nella mappa). U_m riceve richieste di operazione da due unità U_0 ed U_1. La chiave (e solo la chiave) della relativa operazione viene comunicata da altre due unità, rispettivamente U_2 ed U_3, a completamento dei parametri dell'operazione richiesta da U_0 e U_1. Le richieste avvengono secondo un protocollo a domanda risposta e U_m serve una sola richiesta alla volta, fino al suo completamento. In figura sono riportati due esempi di interazione: lettura da U_0 a sinistra, scrittura da U_1 a destra, il tempo va dall'alto verso il basso. La mappa associativa è memorizzata utilizzando due memorie da *N=1K* posizioni da 8 bit: la prima viene utilizzata per mantenere la chiavi, la seconda per mantenere i valori.



E' noto che non può essere richiesta alcuna operazione per la chiave *key=0*. Calcolare il tempo medio di elaborazione dell'unità, motivandone dettagliatamente la derivazione.

Domanda 2

Si consideri lo pseudo codice:

```
for(i=0; i<N; i++) {
      for(j=0; j<N; j++) {
            x[j] = x[j] + x[j]*x[j];
            sum[i] = sum[i] + x[j];
      }
      somma = somma + sum[i];
}
```

con `x, sum e somma` variabili di tipo intero. Per questo codice si richiede di:

1. fornire il codice D-RISC risultante dalla compilazione secondo le regole standard
2. identificare i degradi delle prestazioni nell'esecuzione su un processore pipeline con EU parallela con EU slave moltiplicatore implementata mediante pipeline a 4 stadi
3. ottimizzare il codice in modo da ridurre il degrado delle prestazioni e quantificare il guadagno ottenuto
4. indicare il working set ed il numero complessivo di fault, motivando la risposta in termini delle dimensioni delle cache.

Bozza di soluzione

Domanda 1

Assumiamo che l'interfaccia di U_m verso U_i abbia due indicatori a transizione di livello: RDY_i in ingresso e ACK_i in uscita. Le interfacce verso U_0 ed U_1 avranno anche un registro OP_i in ingresso (da 1 bit), un registro DATAIN*i* e un registro DATAOUT*i* da 8 bit rispettivamente in ingresso (per i dati da scrivere nella mappa e per il risultato delle letture), oltre ad un registro ESITO*i* in uscita da almento 1 bit (significato: 0=NOK, 1=OK). Le interfacce verso U_2 e U_3 avranno invece esclusivamente un registro $KEYIN_i$ in ingresso da 8 bit.
Poichè non esiste una chiave pari a 0 (8 bit tutti 0) questo valore verrà utilizzato per rappresentare posizioni "libere" nella mappa. Non è necessario inizializzare le memorie a 0 assumendo come al solito che questo sia il valore all'accensione di qualsiasi registro o locazione di memoria.
Assumiamo che le due memorie interne si chiamino K (memoria delle chiavi) e V (memoria dei valori).

Lo pseudocodice delle due operazioni esterne supportate potrebbe essere rapresentato come segue:

```
if(richiesta == lettura da Ui) {
    attendi key da Ui+2
    i=0;
    while(i<N and key[i]!=key)
      i++;
    if(i<N) {
      invia a Ui val[i] OK
    } else {
      invia a Ui NOK
    }
} else {
   attendi key da Ui+2
   i=0; libero = -1;
   while(i<N and key[i]!=key) {
      if(key[i]==0)
      libero = i;
      i++
   }
    if(i==N and libero == -1) {
      invia NOK a Ui
    } else {
      key[libero] = key;
      val[libero]=val;
      invia a Ui OK
    }
}
```

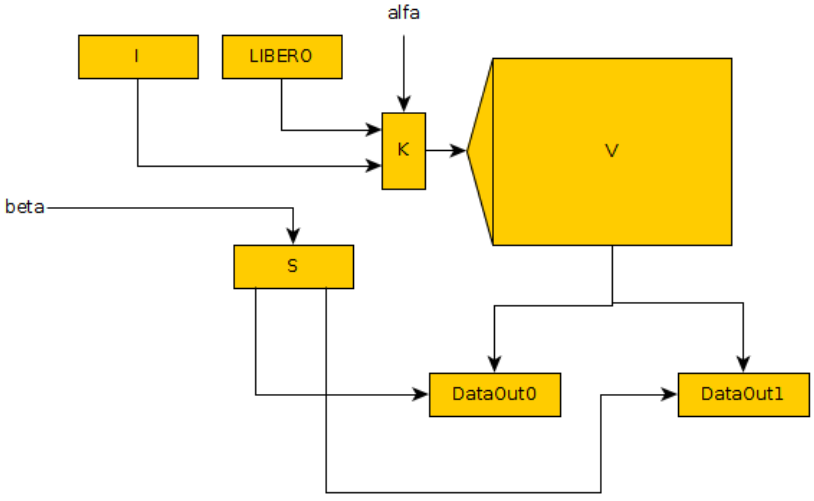
In microcodice:

```
0.      (RDY0,RDY1,RDY2,RDY3,OP0,OP1=00----) nop, 0        // nessuna operazione richiesta
        (=1-0---) nop, 0                                   // op dalla prima unità ma non c'è la chiave pronta
        (=01-0--) nop, 0                                   // op dalla seconda unità ma non c'è la chiave pronta
        // ricerca+lettura dalla U0
        (=1-1-0-) KEY0→ KEY, set ACK2, reset RDY2, 0→ FROM, 0→I, 1
        // ricerca+scrittura dalla U0
        (=1-1-1-) KEY0 → KEY, DATAIN0→ VAL, set ACK2, reset RDY2, 0→ FROM, 0→I, -1 → LIBERO, 2
```

// ricerca e lettura dalla U1
(=01-1-0) KEY1→KEY, set ACK3, reset ACK3, 1→FROM, 0→I, 1
// ricerca e scrittura dalla U1
(=01-1-1) KEY1→KEY, DATAIN1 → VAL, set ACK3, reset RDY3, 1→FROM, 0→I, -1 → LIBERO, 2
// usiamo il controllo residuo per indirizzare verso U0 o U1 la risposta, a seconda del valore di FROM:
1. (zero(K[i]-KEY),I0=10) V[I]→ DATAOUT|$_{FROM}$, "ok" → ESITO|$_{FROM}$, set ACK|$_{FROM}$, reset RDY|$_{FROM}$, 0
(=00) I+1→I, 1
(=01) "nok" → ESITO|$_{FROM}$, set ACK|$_{FROM}$, reset RDY|$_{FROM}$, 0
2. (zero(K[I]-KEY),OR(K[I],I0,LIBERO0=1-0-) VAL->V[I], "ok" → ESITO|$_{FROM}$, set ACK|$_{FROM}$, reset RDY|$_{FROM}$, 0
(=0001) I→ LIBERO, I+1 → I, 2
(=010-) I+1→I, 2
(=01-1) "nok" → ESITO|$_{FROM}$, set ACK|$_{FROM}$, reset RDY|$_{FROM}$, 0
(=01-0) VAL → V[LIBERO], KEY → K[LIBERO], "ok" → ESITO|$_{FROM}$, set ACK|$_{FROM}$, reset RDY|$_{FROM}$, 0

La notazione ACK|$_{FROM}$ indica ACK$_i$ se FROM vale i. L'implementazione utilizza un selettore con FROM come ingresso di controllo, il β come ingresso e i β_i come uscite.

La parte controllo, utilizza 10 variabili di condizionamento, di cui al massimo 6 vengono testate nella stessa microistruzione. Il microprogramma è composto da 3 microistruzioni. Dunque nella tabella di verità delle reti combinatorie della PC avremo un totale di 12 colonne per gli ingressi, di cui al massimo 8 (6 per gli ingressi/variabili di condizionamento e 2 per i bit che rappresentano lo stato) sono significative per ognuna delle linee della tabella di verità. Questo significa un solo livello di porte AND. Vi sono 15 frasi in totale, dunque possiamo dire che al massimo ci saranno 8 1 nelle colonne delle uscite e dei bit del prossimo stato interno (altrimenti codificheremmo gli zeri negando l'uscita). Pertanto avremo anche un unico livello di porte OR. Entrambe le reti della PC avranno dunque un tempo di stabilizzazione pari a $2t_p$.
L'operazione più lunga della parte operativa è l'accesso alla memoria che richiede la stabilizzazione del commutatore sugli indirizzi (scelta fra I e LIBERO) e il t_a per l'accesso alla posizione richiesta. Anche in caso di scrittura in uno dei DATAOUT$_i$, non si ha un aumento del tempo necessario alla stabilizzazione visto che il selettore per il β stabilizza contemporaneamente al commutatore sugli indirizzi della memoria. Dunque il $T\sigma_{PO}$ sarà di t_k + t_a. Per quanto riguarda $T\omega_{PO}$, va notato che la variabile di condizionamento che richiede più tempo per il calcolo impiega un t_a per l'accesso alla memoria (la memoria K deve essere a doppia porta per la condizione di correttezza, quindi niente commutatore sugli ingressi ma due commutatori nella memoria per la lettura contemporanea dagli indirizzi I e LIBERO) e un t_{alu} per il calcolo dei flag,



.
Va notato che la varibile di condizionamento complesso appare nella stessa frase in cui compare la microoperazione "lunga" di scrittura in DATAOUT$_i$ e dunque in questo caso il ciclo di clock calcolato con la formula "a maggiorazione" o analizzando frase per frase il microprogramma è lo stesso:

$\tau = t_a + t_{alu} + \max\{2t_p, 2t_p + t_k + t_a\} + t_p$

Valutiamo ora il tempo medio di elaborazione.
Assumiamo, in mancanza di indicazioni precise:

- che la probabilità di ricevere una richiesta di lettura o scrittura sia 0.5
- che sia in caso di lettura che di scrittura occorra scorrere N/2 posizioni prima di trovare la chiave cercata.

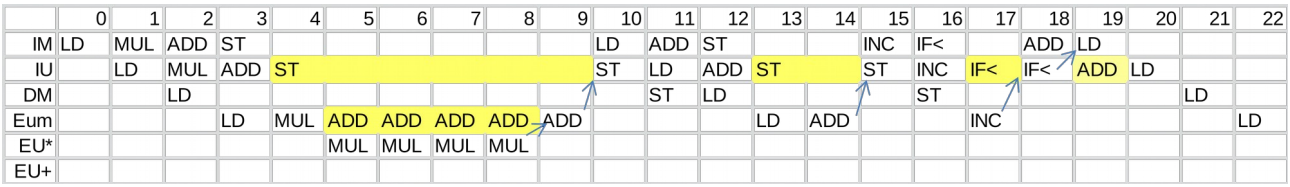
Sotto queste assunzioni, sia la lettura che la scrittura dovranno eseguire 1+N/2 frasi e dunque il tempo medio di elaborazione sarà di (1+N/2) τ.

Domanda 2

Procediamo con la compilazione secondo le regole standard. Consideriamo solo il codice dei corpi dei loop, senza considerare il codice per le inzializzazioni.

```
loop: LOAD Rx, Rj, Rxj  ; caricamento x[j]
      MUL Rxj, Rxj, R1  ; x[j]*x[j]
      ADD R1, Rxj, R2   ; calcolo nuovo valore di x[j]
      STORE Rx, Rj, R2  ; scrittura in memoria per x[j]
      LOAD Rs, Ri, R3   ; caricamento sum[i]
      ADD R3, Rxj, R4   ; calcolo sum[i]+x[j]
      STORE Rs, Ri, R4  ; scrittura in memoria
      INC Rj            ; calcolo di j++
      IF< Rj, RN, loop  ; prossima iterazione for j
      ADD Rso, R4, Rso  ; somma = somma + sum[i]
      INC Ri            ; calcolo di i++
      IF< Ri, RN, loop  ; prossima iterazione for I
      END
```

Simulando l'esecuzione del codice osserviamo che il tempo di completamento della singola iterazione del ciclo interno risulta pari a 19t a fronte di un corpo composto da 9 sole istruzioni.



Il degrado delle prestazioni è legato sia alla dipendenza EU-EU fra moltiplicazione e addizione, che contribuisce ad allungare la dipendenza IU-EU fra la ADD e la STORE, sia alla dipendenza IU-EU fra la seconda ADD e la seconda STORE. Vi è inoltre un problema legato alla dipendenza IU-EU fra INC e IF< e una bolla "da salto" per il ritorno all'inizio della prossima iterazione.

Per il primo problema (dipendenza MUL-ADD) possiamo posticipare la ADD ma soprattutto posticipare la STORE relativa, che è quella che crea i maggiori problemi. Di fatto, possiamo combinare il trattamento dei problemi sollevati dalla dipendenza con l'eliminazione della bolla dovuta al salto utilizzando un IF< delayed (salto ritardato) con la STORE nel delay slot, avendo cura di utilizzare per la STORE un registro base decrementato di 1.

Possiamo anche cercare di anticipare la INC in modo da mitigare o annullare l'effetto della dipendenza INC-IF<.

Infine, osservando che la sum[i] non dipende da j e che il valore può essere memorizzato solo alla fine del ciclo j, possiamo caricare sum[i] prima di entrare nel ciclo j e memorizzarla successivamente all'uscita dal ciclo j, in modo da eliminare o ridurre i problemi generati dalla seconda dipendenza IU-EU.

Possiamo quindi ottimizzare il codice come segue:

```
loop: LOAD Rs, Ri, R3   ; sum[i]
loopj:LOAD Rx, Rj, Rxj  ; x[j]
      MUL Rxj, Rxj, R1  ; x[j]*x[j]
      INC Rj            ; j++
      ADD Rxj, R1, R2   ; x[j] + x[j]*x[j]
      ADD R3, R2, R4    ; sum[i] + x[j]
      IF< Rj, RN, loop1, delayed    ; nuova iterazione
      STORE Rx', Rj, R2 ; delay slot: Rx' = Rx - 1
conti:STORE Rs, Ri, R4  ; sum[i] nuovo
      ADD Rso, R4, Rso  ; somma + sum[i]
      INC Ri            ; i++
      IF< Ri, RN, loop  ; prossima iterazione
cont: END
```

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

Anche considerando che due istruzioni (la LOAD e la STORE per sum[i]) sono state rimosse siamo decisamente al di sopra delle prestazioni del codice standard. In particolare siamo passati da un'efficienza di 9/20 ad una pari a 7/11.
Eventualmente, mantenendo all'interno del ciclo for j la LOAD e la STORE relative a sum[i], la LOAD potrebbe essere utilizzata insieme alla INC Rj per allontanare la ADD dalla MUL. L'anticipo della INC Rj richiede opportuni registri base modificati e risolve anche il problema legato alla dipendenza indotta dalla INC sulla IF< di fine ciclo.

Per capire com'è fatto il working set, e di conseguenza quale sia il numero di fault, osserviamo che:

- il vettore `sum` è acceduto secondo il principio della sola località
- il vettore `x` è acceduto secondo i principi di località e riuso
- il codice è breve ed è acceduto secondo i principi di località e riuso (per via dei cicli)

Dunque il working set conterrà tutto `x`, una pagina di `sum` e le pagine del codice, oltre alla pagina con le variabili scalari (`somma, i, j`, ecc).
Nell'ipotesi che la cache istruzioni abbia almeno 13 (lunghezza del codice) su σ linee, e che quindi il codice possa essere mantenuto permanentemente nella cache IU e che la cache dati possa contanere l'intero array `x` e la linea contenente la posizione corrente di `sum`, oltre le varibili scalari, avremo che il numero di fault (pari al numero di fault fisiologici) sarà di $13/\sigma$ (per il codice) + N/σ (per x) + N/σ (per `sum`). Considerando che gli accessi avvengono sequenzialmente per x (la prima volta) e `sum`, nell'ipotesi che la cache supporti prefecthing i fault possono essere ridotti a 3: uno per il codice, uno per `x` e uno per `sum`. Nel caso in cui la cache non avesse capacità sufficiente a contenere l'intero `x`, i fault per `x` salirebbero ad N^2/σ.