Esercizio 1

Si consideri un array A di interi positivi con $N=2^K$ elementi. L'array è **ordinato**, dal valore più basso (prima posizione) a quello più alto (ultima posizione). Assumendo di avere a disposizione un **processore D-RISC pipeline con EU monolitica e cache di primo livello (sia dati che istruzioni) associativa su insiemi a 4 vie, con linee da 32 parole (sigma = 32) e prefetching,** si calcolino il **numero massimo di fault** nell'esecuzione di una **ricerca binaria** sull'array (all'inizio della ricerca, nessuna pagina dell'array è allocata in cache dati).
Successivamente si fornisca il **tempo di completamento della prima iterazione while della ricerca binaria** nell'ipotesi che l'elemento cercato **non** sia stato trovato.

Lo pseudo codice delle due procedure di ricerca è il seguente:

```
/* ricerca binaria: cerca x cominciando da metà e poi continuando nella metà alta o bassa */
int start = 0;
int stop  = n;
int found = -1;
while(stop - start > 1) {
        int i = start + (stop-start)/2;
         if(A[i] == x) {found = i; break;}
        if(x < A[i]) { stop = i; } else { start = i; }
}
if(A[start] == x) found = start;
if(A[stop] == x)  found = stop;

/* ricerca esaustiva: cerca x controllando le celle una per una fino a che l'array è finito o l'elemento è stato trovato */
int found = -1;
for(int i=0; i<n; i++)
        if(A[i] == x) { found = i; break; }
```

Esercizio 2

Si consideri lo pseudo codice che copia il contenuto di un array X di lunghezza N in un altro array Y di lunghezza N/2 facendo sì che per ogni i in [0, N/2 -1] risulti Y[i] = X[2*i] + X[2*i +1]:

```
for(int i=0; i<N/2; i++)
 { Y[i] = X[2*i] + X[2*i +1]; }
```

Successivamente :

1) si **compili** il codice in assembler D-RISC, secondo le **regole** di compilazione **standard**
2) se ne valutino le **prestazioni** su un **processore pipeline** con **EU** in grado di eseguire **solo** operazioni aritmetico logiche "**corte**" (*no moltiplicazioni e divisioni*), evidenziando eventuali **inefficienze** prestazionali (calcolo su un ciclo)
3) si determini il **working set** del programma e si valuti il **numero di fault** che si verificano durante l'esecuzione (utilizzando il prefetch)
4) si valutino **possibili ottimizzazioni** del codice assembler

Esercizio 1 – bozza di soluzione

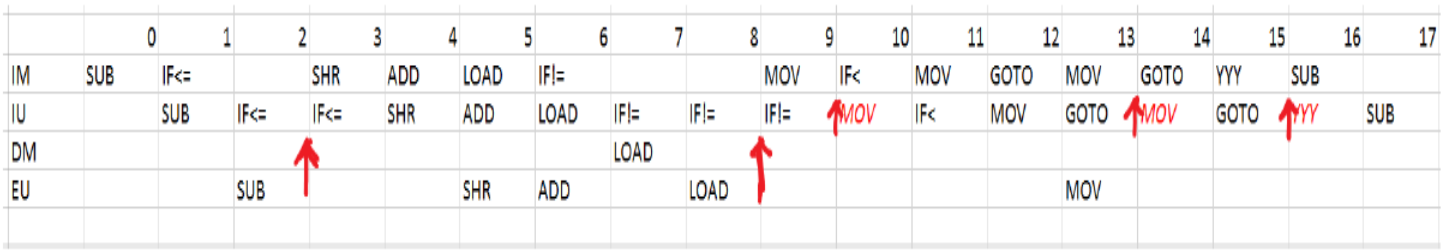
La **ricerca binaria** ha ordine di complessità temporale **logaritmica**. Vengono quindi **controllati al più K elementi**. Gli ultimi elementi trovati possono risiedere nella stessa linea della cache, ma **K** è una approssimazione per **eccesso** del **numero di fault** generati dall'algoritmo. Il fatto che la **cache sia associativa su insiemi non ha impatto**, visto che si **accedono sempre parti diverse** (cioè linee di cache diverse) del vettore fino alle ultime iterazioni, e il **prefetch comporta vantaggi**, solo **quando** il **range** di ricerca diventa **vicino alla dimensione sigma delle linee di cache**.
La ricerca **esaustiva scorre** il vettore per intero (o **fino a quando non viene trovato** l'elemento cercato). Lo **scorrimento** del vettore è pertanto **perfettamente supportato dal prefetching e possiamo approssimare i fault a 1 solo, iniziale**.

Naturalmente, al fine del calcolo del numero totale di fault **vanno contati anche i fault per il codice**, che in questo caso sono **pari a 1, visto che il codice è sicuramente minore di 32 parole** (lunghezza di una linea di cache).

Per fornire il tempo di completamento della prima iterazione, **compiliamo il codice**.

```
while: SUB Rstop, Rstart, Rtemp  // calcolo della condizione
       IF<= Rtemp, #1, end       // salto NON preso nel nostro caso
       SHR Rtemp, #1, Rtemp //  calcolo dell'iterazione (stop-start)/2
       ADD Rstart, Rtemp, Ri
       LOAD RbaseA, Ri, Rai
       IF!= Rai, Rx, cont    // salto preso nel nostro caso
then:  MOV Ri, Rfound              // istruzione NON eseguita nel nostro caso
       GOTO end                    // istruzione NON eseguita nel nostro caso
cont:  IF< Rx, Rai, then2
else2: MOV Ri, Rstart
       GOTO cont2
then2: MOV Rstop, Ri               // istruzione NON eseguita nel nostro caso, per esempio
cont2: GOTO while
end:   YYY                  // inizio della compilazione degli ultimi controlli
```

Simulando vediamo che un'iterazione è completata in **16 t** (assumendo che cerchiamo nella **parte alta** all'iterazione successiva):



le prime due bolle sono da operando (cioè IU-EU), mentre le ultime tre sono da salto.

Esercizio 2 - Bozza di soluzione

Compilazione standard

```
                CLEAR Ri
                SHR Rn, 1, Rnmezzi          //inizializzazione di i
1       LOOP: SHL Ri, 1, Rdisp
2               LOAD Rbasex, Rdisp, R2x     //lettura di X[2*i]
3               ADD Rdisp, 1, Rdisp
4               LOAD Rbasex, Rdisp, R2x1    //lettura di X[2*i +1]
5               ADD R2x, R2x1, Ry           //calcolo di Y[i]
6               STORE Rbasey, Ri, Ry        //memorizzazione di Y[i]
7               INC Ri
8               IF< Ri, Rnmezzi, LOOP       //chiusura del ciclo
                END
```

Valutazione delle prestazioni

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
IM	SHL	LOAD		ADD	LOAD			ADD	STORE			INC	IF<		END	SHL		
IU		SHL	LOAD	LOAD	ADD	LOAD	LOAD	LOAD	ADD	STORE	STORE	STORE	INC	IF<	IF<	END	SHL	
DM					LOAD				LOAD				STORE					
EU			SHL			LOAD	ADD			LOAD	ADD			INC				SHL

Impieghiamo **15t** per eseguire un'iterazione (a regime) che è composta da **8** istruzioni. Efficienza appena superiore al **50%**. La SHL induce una dipendenza sulla prima LOAD, la prima ADD sulla seconda LOAD, la seconda ADD induce dipendenza sulla STORE, la INC sulla IF< e infine abbiamo la bolla da salto preso.

Working set

Entrambi i **vettori** vengono acceduti **sequenzialmente**, **senza** **riutilizzo**. Il working set è dunque formato da **tutto il codice** (acceduto con località e riuso) da **una linea di X e una linea di Y**. Se utilizziamo **prefetch** avremo **un solo fault per X e Y** oltre ai **fault per il codice**. Se $\sigma=8$, abbiamo 2 fault per il codice. **Senza** prefetch, avremo **C(N/σ)** faults sia per X che per Y, dove **C(z)** è la funzione "ceiling" (intero superiore). Se la cache ha 32K parole ed è set-associative con 1K insiemi, allora è **divisa** in 1K insiemi da 32 parole ciascuno. Siccome $\sigma=8$, allora ogni insieme può contenere 4 linee al massimo. Quindi la cache è **divisa** in 1K insiemi, ognuno **diviso** in 4 linee ognuna **divisa** in 8 parole: 32K/1K=32; 32/8=4.

Ottimizzazione del codice

Possiamo eseguire **immediatamente** le due **istruzioni** che calcolano gli **indici per** l'accesso a **X** e **anche** la **INC** Ri. Inoltre, possiamo utilizzare il **delayed branch**, se presente, per **posticipare** la **STORE**, avendo cura di **utilizzare** il **registro base decrementato** per Y **(Rbasey'=Rbasey-1)** per compensare l'incremento di *i* anticipato (a sinistra i numeri delle istruzioni come nel programma precedente):

1 LOOP: SHL Ri, 1, Rdisp
3 ADD Rdisp, 1, Rdisp
7 INC Ri
2 LOAD Rbasex, Rdisp, R2x
4 LOAD Rbasex, Rdisp, R2x1
5 ADD R2x, R2x1, Ry
8 IF< Ri, Rnmezzi, LOOP, delayed
6 STORE Rbasey', Ri, Ry

Con questo codice riusciamo ad eseguire una iterazione in **9t**, che è **quasi** il tempo **ideale (8/9)**. Rimane infatti **solo** la **dipendenza** indotta dalla **ADD sulla STORE**, che, seppure a distanza 2, ha effetto per via delle LOAD nella sequenza di istruzioni che portano alla dipendenza: le LOAD devono passare da tutte e 4 le unità e quindi rallentano la EU.

	0	1	2	3	4	5	6	7	8	9	10	11
IM	SHL	ADD	INC	LOAD	LOAD	ADD	IF<	STORE	SHL			
IU		SHL	ADD	INC	LOAD	LOAD	ADD	IF<	STORE	STORE	SHL	
DM						LOAD	LOAD				STORE	
EU			SHL	ADD	INC		LOAD	LOAD	ADD			SHL