# Architettura degli elaboratori – A.A. 2018—19 – 1° appello – 5/01/19

*Riportare su tutti i fogli consegnati in alto a destra Nome, Cognome, Matricola e Corso di appartenenza (A/B).*
*I risultati saranno resi pubblici via WEB appena disponibili assieme al calendario degli orali.*

## Domanda 1

L'unità firmware B riceve da una unità A interi da 32 bit e ne ricerca la prima occorrenza in una memoria esterna M da 8K parole. L'interazione fra B ed M avviene utilizzando un'interfaccia di memoria come quella del processore D-RISC, a domanda-risposta. La memoria M è ordinata, ovvero per ogni coppia di indirizzi *i* e *j* vale che se *i<j* allora sicuramente *M[i]<=M[j]*. A interagisce con B secondo un protocollo a domanda-risposta. B restituisce ad A un booleano che dice se il numero è stato trovato all'interno della memoria e un intero che indica l'indirizzo dell'intero cercato, in caso sia presente. B usa un algoritmo di ricerca binario, che possiamo descrivere mediante il seguente pseudocodice:

```
start=0; end=8K-1;
while((end-start)>1) {
       mid = (start+end)/2;
       if(v[mid]==target) then "trovato alla posizione mid"; fine ricerca
       if(target > v[mid]) then start = mid
                           else end = mid;
}
if(v[start] == target) then "trovato alla posizione start"; fine ricerca
if(v[end] == target) then "trovato alla posizione end"; fine ricerca
"elemento non trovato"; fine ricerca
```

Si implementi l'unità B e se ne fornisca il ciclo di clock e il tempo medio di elaborazione con le seguenti assunzioni:

- Si assume di trovare il valore cercato dopo 4 iterazioni (per il calcolo del tempo medio)
- Sono a disposizione ALU che fanno operazioni fra interi in $4t_p$
- Il tempo di accesso alla memoria esterna è $t_a = 10t_p$
- Le porte logiche hanno al massimo 8 ingressi

## Domanda 2

Si scriva un programma in assembler D-RISC che implementa una ricerca in un vettore ordinato da 8K posizioni secondo le stesse modalità dell'algoritmo utilizzato per implementare l'unità firmware del precedente esercizio. Si assuma di avere a disposizione un processore D-RISC pipeline con la sola unità EU master che esegue tutte le operazioni aritmetico logiche corte, e che ha un ciclo di clock della stessa durata di quello dell'unità firmware del primo esercizio. Calcolare il tempo di completamento della ricerca assumendo che il dato cercato venga trovato dopo 4 iterazioni.

# Traccia di soluzione

## Domanda 1

Il microcodice dell'unità può essere scritto come segue (INDICE e TROVATO sono i registri utilizzati per comunicare la risposta all'unità A):

1. (RDY=0) nop, 1
   (=1) 0 -> START, 8K-1 -> END, 2
2. (piùdiuno(END-START)=1) SHR((START+END),1) -> MID, SHR((START+END),1) -> IND,
   "read" -> OP, set RDYm, 3
   (=0) START -> IND, "read" -> OP, set RDYm, 4
3. (ACKm,eq(DATAIN,TARGET), gt(DATAIN,TARGET) = 0 - -) nop, 3
   (=11-) MID -> INDICE, 1 -> TROVATO, reset RDY, set ACK, reset ACKm, 1
   (=101) MID -> START, 2
   (=100) MID -> END, 2
4. (ACKm, eq(DATAIN,TARGET)=0-) nop, 4
   (=11) START -> INDICE, 1 -> TROVATO, reset ACKm, set ACK, reset RDY, 1
   (=10) END -> IND, "read" -> OP, set RDYm, reset ACKm, 5
5. (ACKm, eq(DATAIN, TARGET)=0-) nop, 5
   (=11) END -> INDICE, 1 -> TROVATO, reset ACKm, set ACK, reset RDY, 1
   (=10) 0 ->TROVATO, reset ACKm, set ACK, reset RDY, 1

Piùdiuno(x) è una rete combinatoria che calcola OR(X[31:1]) in 2 $t_p$. Eq(X,Y) è una rete che utilizza 32 comparatori in parallelo per confrontare i bit corrispondenti di X e Y e calcola l'OR negato delle loro uscite in 4 $t_p$ (2 per i confrontatori e 2 per l'OR), oppure è realizzata con una ALU (dedicata) di cui prendiamo zero(X-Y), sempre in $4t_p$, viste le specifiche del problema. Gt(X,Y) è calcolato con una ALU che esegue sottrazioni fra interi, eseguendo X-Y e prendendo il negato del bit più significativo del risultato.

Abbiamo 5 stati interni e 5 variabili di condizionamento (di cui al massimo 3 sono testate contemporaneamente). Questo implica che per la $\omega_{PC}$ e la $\sigma_{PC}$ ci sia un solo livello di porte AND. Abbiamo in tutto 14 frasi. Questo implica che per la $\omega_{PC}$ e la $\sigma_{PC}$ ci siano al massimo due livelli di porte OR. Abbiamo quindi $T\omega_{PC} = T\sigma_{PC} = 3t_p$.

$T\omega PO$ è 0 nella 1., $2t_p$ nella 2. e $t_{alu}$ nelle altre microistruzioni.

$T\sigma PO$ è 0 nella 1. $t_{alu}+t_k$ nella 2. (l'operazione di shift destra di una posizione è realizzata collegando i 31 bit più significativi completati a sinistra con uno 0 all'ingresso del commutatore davanti a MID) e $t_k$ nelle altre microistruzioni ($t_{alu}$ e $t_k$ sono i tempi di stabilizzazione del della ALU e del commutatore).

Il ciclo di clock è dunque $\tau = 2t_p + 3t_p + t_{alu} + t_k + \delta = 12t_p$

Per eseguire 4 iterazioni per la ricerca di un elemento si eseguono una volta la 1. e 4 volte la 2. (prima frase) e la 3. (tre volte la terza o la quarta frase e una volta la seconda frase). Per la 1. e la 2. spendiamo un $\tau$, mentre per la 3 spendiamo ($\tau + t_a$) per cui $T = (1 + 4*2)\ \tau + 4\ t_a = 9\tau + 4t_a$. Che fa un totale di $(9*12 + 4 * 10)\ t_p = 148\ t_p$.

## Domanda 2

Il codice assembler può essere compilato (regole standard) come segue:

```
        CLEAR                                                          Rstart
        ADDI R0, #1, Rend
        SHLI Rend, #12, Rend
        SUBI Rend, #1, Rend
while:  SUB Rend, Rstart, Rtemp          // calcola la condizione
        IF<= Rtemp, #1, concludi         // se non vale, esci dal while
        ADD Rstart, Rend, Rtemp          // start + end
        SHR Rtemp, 1, Rmid               // mid = (start+end)/2
        LOAD Rbase, Rmid, Rvmid          // v[mid]
        IF!= Rvmid, Ttarget, nontrovato  // se non trovato prosegui
        MOV Rmid, Rindirizzo             // altrimenti dai risposta
        ADDI R0, #1, Rtrovato
        GOTO fine                        // e ritorna
nontrovato: IF> Rtarget, Rvmid, then     // controlla se maggiore
else:   MOV Rmid, Rend                   // se minore/uguale cerca prima parte
        GOTO cont
then:   MOV Rmid, Rstart                 // altrimenti nella seconda
cont:   GOTO while                       // nuova iterazione
concludi: LOAD Rbase, Rstart, R1         // v[start]
        IF!= R1, Rtarget, cont1          // se non è uguale controlla v[end]
        MOV Rstart, Rtrovato             // se lo è dai risposta
        ADDI R0. #1, Rtrovato
        GOTO fine                        // e ritorna
cont1:  LOAD Rbase, Rend, R2             // v[end]
        IF!= R2, Rtarget, cont2         // se non è uguale devi restituire false
        MOV Rend, Rtrovato               // altrimenti dai risposta
        ADDI R0, #1, Rtrovato
        GOTO fine                        // e ritorna
cont2:  ADDI R0, #-1, Rtrovato           // restituisci falso
fine:   END                        // e ritorna
```

La parte che ci interessa valutare è quella relativa a 4 iterazioni del codice

```
while:  SUB Rend, Rstart, Rtemp          // calcola la condizione
        IF<= Rtemp, #1, concludi         // se non vale, esci dal while
        ADD Rstart, Rend, Rtemp          // start + end
        SHR Rtemp, 1, Rmid               // mid = (start+end)/2
        LOAD Rbase, Rmid, Rvmid          // v[mid]
        IF!= Rvmid, Ttarget, nontrovato  // se non trovato prosegui
        MOV Rmid, Rindirizzo             // altrimenti dai risposta
        ADDI R0, #1, Rtrovato
        GOTO fine                        // e ritorna
nontrovato: IF> Rtarget, Rvmid, then     // controlla se maggiore
else:   MOV Rmid, Rend                   // se minore/uguale cerca prima parte
        GOTO cont
then:   MOV Rmid, Rstart                 // altrimenti nella seconda
cont:   GOTO while                       // nuova iterazione
```

C’è un’ovvia ottimizzazione per evitare una GOTO CONT: GOTO WHILE:

```
while: SUB Rend, Rstart, Rtemp          // calcola la condizione
       IF<= Rtemp, #1, concludi         // se non vale, esci dal while
       ADD Rstart, Rend, Rtemp          // start + end
       SHR Rtemp, 1, Rmid               // mid = (start+end)/2
       LOAD Rbase, Rmid, Rvmid          // v[mid]
       IF!= Rvmid, Ttarget, nontrovato  // se non trovato prosegui
       MOV Rmid, Rindirizzo             // altrimenti dai risposta
       ADDI R0, #1, Rtrovato
       GOTO fine                        // e ritorna
nontrovato: IF> Rtarget, Rvmid, then    // controlla se maggiore
else:  MOV Rmid, Rend                   // se minore/uguale cerca prima parte
       GOTO while
then:  MOV Rmid, Rstart                 // altrimenti nella seconda
cont:  GOTO while                       // nuova iterazione
```

Questo codice presenta dipendenze IU-EU

- Fra la prima SUB e la IF<=
- Fra la SHR e la LOAD
- Fra la LOAD e la IF=

Ovviamente non ci sono dipendenze EU-EU (c’è solo la EU master).

Quando non troviamo l’elemento, l’iterazione del ciclo while può proseguire scegliendo di continuare nella parte inferiore o superiore del vettore. Nei due casi:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IM | SUB | IF<= | | ADD | SHR | LOAD | IF!= | | | | | IF> | MOV | GOTO | | SUB | | |
| IU | | SUB | IF<= | IF<= | ADD | SHR | LOAD | LOAD | IF!= | IF!= | IF!= | | IF> | MOV | GOTO | | SUB | |
| DM | | | | | | | | | LOAD | | | | | | | | | |
| EU | | | SUB | | | ADD | SHR | | | LOAD | | | | | MOV | | | SUB |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IM | SUB | IF<= | | ADD | SHR | LOAD | IF!= | | | | | IF> | MOV | MOV | GOTO | | SUB | | |
| IU | | SUB | IF<= | IF<= | ADD | SHR | LOAD | LOAD | IF!= | IF!= | IF!= | | IF> | MOV | MOV | GOTO | | SUB | |
| DM | | | | | | | | | LOAD | | | | | | | | | | |
| EU | | | SUB | | | ADD | SHR | | | LOAD | | | | | | MOV | | | SUB |

Abbiamo un’iterazione da 15t o 16t. Non conoscendo la posizione e assumendo equiprobabile la prosecuzione della ricerca nella prima o nella seconda metà dell’intervallo il tempo per eseguire un’iterazione sarà dunque di 15.5t.

Quando troviamo l’elemento cercato (4ª iterazione) spendiamo invece:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IM | SUB | IF<= | | ADD | SHR | LOAD | IF!= | | | | MOV | ADDI | GOTO | | |
| IU | | SUB | IF<= | IF<= | ADD | SHR | LOAD | LOAD | IF!= | IF!= | IF!= | MOV | ADDI | GOTO | |
| DM | | | | | | | | | LOAD | | | | | | |
| EU | | | SUB | | | ADD | SHR | | | LOAD | | | MOV | ADDI | |

14t (considerando la bolla per il ritorno dalla procedura). Per le 4 iterazioni avremo quindi

3* 15.5t + 14t = 60.5 t = 121τ.

Se consideriamo che con l'unità firmware avermmo speso (a partità di tempo di accesso in memoria) solo $9\tau$, la differenza è evidente.