# *Architettura degli elaboratori A.A. 2015-2016*

*Primo appello sessione estiva - 13 giugno 2016*

*Scrivere Nome, Cognome, Numero di matricola e Corso di appartenenza (A/B) su tutti i fogli consegnati.*
*I risultati verranno comunicati via WEB appena disponibili, contenstualmente al calendario degli orali per gli ammessi.*

***Domanda 1***

Si consideri il seguente microprogramma:

0. (RDY1,RDY2,OP1,OP2=00--) nop, 0
   (=1-0-) DATAin1 → IN, IND1 → IND, reset RDY1, set ACK1, 1.
   (=01-0) DATAin2 → IN, IND2 → IND, reset RDY2, set ACK2, 2.
   (=--1-1-) IND2 → IND, reset RDY2, set ACK2, 3.
   (=101-) IND1 → IND, reset RDY1, set ACK1, 3.
1. IN + M[IND] → M[IND], 3.
2. IN - M[IND] → M[IND], 3.
3. (ACK3=0) nop, 3.
   (=1) COUNT+1 → COUNT, M[IND] → OUT, set RDY3, reset ACK3, 0.

Si descrivano le operazioni esterne implementate, assumendo che DATAinX, INDx e OP siano registri in ingresso da due unità Ux, rispettivamente da 32, 8 e 1 bit, e che OUT sia un registro in uscita verso una unità U3 da 32 bit. Si calcoli il tempo medio di esecuzione delle operazioni esterne descritte.

Successivamente si illustri un'ottimizzazione del microcodice che riduca la complessità dell'unità firmware e/o il tempo medio di completamento delle operazioni esterne. Si specifichino esattamente quali sono tutti i vantaggi (e/o gli eventuali svantaggi) del microcodice ottimizzato rispetto alla versione originale.

***Domanda 2***

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 e nell'esecuzione di una ricerca esaustiva sull'array (all'inizio della ricerca, nessuna pagina dell'array è allocata in cache dati).

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; }
```

Successivamente si fornisca il tempo di completamento della prima iterazione while della ricerca binaria nell'ipotesi che l'elemento cercato non sia stato trovato.

# *Bozza di soluzione*

**Domanda 1**

L'unità è interposta fra le unità U1 e U2 e l'unità U3. Da U1 e U2 riceve richieste di due tipi di operazioni diverse (registro OPx da 1 bit, OP1 testato solo quando si riceve da U1 (RDY1=1) e OP2 testato solo quando si riceve da U2 (RDY2=1)). Le due operazioni esterne possono essere così descritte:

- OP=0 da U1: DATAin + M[IND] sostituisce M[IND] e viene inviato a U3; si incrementa COUNT
- OP=1 da U1: M[IND] viene inviato a U3; si incrementa COUNT
- OP=0 da U2: DATAin - M[IND] sostituisce M[IND] e viene inviato a U3; si incrementa COUNT
- OP=1 da U2: M[IND] viene inviato a U3; si incrementa COUNT

Per calcolare il tempo medio di esecuzione occorre procedere alla sintesi di PC e PO. Per la PC, notiamo che abbiamo 4 stati (4 micro operazioni diverse) con 9 frasi diverse e un massimo di 4 variabili di condizionamento testate contemporaneamente. Dunque massimo 6 ingressi per il livello AND (2 bit di stato e 4 di variabili di condizionamento) e 9 ingressi per il livello OR (le 9 frasi). Senza ulteriori ottimizzazioni questo porterebbe ad una PC con $T_{\sigma PC}$ e $T_{\omega PC}$ pari a 3 $t_p$. Tuttavia, osservando che possiamo codificare gli 0 a livello OR e successivamente negare l'uscita, il livello OR può essere ridotto ad un unico livello di porte portando i suddetti tempi a $2t_p$.

Per la PO, l'operazione più lunga e senz'altro la micro operazione nella microistruzione 1. (2.), che costa un $t_a$ e un $t_k$+$t_{alu}$ (il selettore di scrittura stabilizza nello stesso periodo e la ALU ha un commutatore in ingresso per scegliere fra IN e COUNT). Per la memoria non è necessario alcun commutatore ne' sugli ingressi (si scrive sempre il risultato della ALU, ne' sugli indirizzi (si opera sempre sulla cella di indirizzo IND). Dunque $T_{\sigma PO}$ sarà in questo caso $t_a$+ $t_k$+$t_{alu}$.
Per la 0. avremo un $T_{\sigma PO}$ pari ad un $t_k$. Per la 4. sarà invece un max{$t_k$, $t_a$}. Tutte le variabili di condizionamento sono semplici, dunque $T_{\omega PO}$ è pari a 0.

Con queste ipotesi il ciclo di clock (calcolato col metodo della maggiorazione, che in questo caso coincide con il metodo esatto perchè $T_{\omega PO}$=0) sarà

$$\tau = 0 + \max\{2t_p, t_a + t_k + t_{alu} + 2t_p\} + t_p = t_a + t_k + t_{alu} + 3t_p$$

Le operazioni esterne richiedono:

operazione 1 (da U1 o U2): 3 $\tau$
operazione 2 (da U1 o U2): 2 $\tau$

dunque qualunque sia la probabilità delle diverse operazioni il tempo medio sarà

$$\frac{(3+2)}{2}\tau = 2.5\,(t_a + t_k + t_{alu} + 3t_p)$$

Il microcodice può essere altamente ottimizzato, considerando le condizioni di Bernstein e notando che non vi sono dipendenze fra le operazioni della 0. e della 1. e della 2. a patto di utilizzare direttamente i registri di ingresso nelle operazioni della 1. e della 2. Dunque si può senz'altro utilizzare un codice:

0. (RDY1,RDY2,OP1,OP2=00--) nop, 0
  (=1-0-) IN + M[IND1]→ M[IND1], IND1→ IND, reset RDY1, set ACK1, 3.
  (=01-0) IN - M[IND2] → M[IND2], IND2 → IND, reset RDY2, set ACK2, 3.
  (=-11-) IND2 → IND, 3.
  (=101-) IND1 → IND, 3.
3. (ACK3=0) nop, 3.
  (=1) COUT+1 → COUNT, M[IND] → OUT, set RDY3, reset ACK3, 0.

Tuttavia possiamo anche notare che la 3. a questo punto può essere eliminata, testando anche ACK3 nelle condizioni della 0. e utilizzando ancora IND1 o IND2 invece che IND per l'indirizzamento della memoria:

0. (RDY1,RDY2,OP,ACK3=00--) nop, 0
  (=1-0-) IN+M[IND1]→ M[IND1], IN+M[IND1]→OUT, COUNT+1→COUNT, reset RDY1, set ACK1, set RDY3, reset ACK3, 0.
  (=01-1) IN-M[IND2]→M[IND2], IN - M[IND2]→OUT, COUNT+1→COUNT, reset RDY2, set ACK2, set RDY3, reset ACK3, 0.
  (=-11-) COUNT+1 → COUNT, M[IND] → OUT, reset RDY1, set ACK1, set RDY3, reset ACK3, 0.
  (=101-) COUNT+1 → COUNT, M[IND] → OUT, reset RDY1, set ACK1, set RDY3, reset ACK3, 0.

In questo caso la PC controllo sparisce, nella PO abbiamo bisogno di 2 ALU da far lavorare in parallelo. Il tempo medio di esecuzione scende ad 1 $\tau$. Essendo una sola rete sequenziale, il ciclo di clock va calcolato come max{Tw, To}. Inoltre, ora viene usato un commutatore sugli ingressi e sugli indirizzi della memoria, per cui il Tw aumenterà di un tk.

Come ultima considerazione, visto che COUNT non viene mai utilizzato per la lettura, possiamo anche pensare di eliminarlo, ottenendo il micro codice:

0. (RDY1,RDY2,OP,ACK3=00--) nop, 0

(=1-0-) IN+M[IND1]→ M[IND1], IN+M[IND1]→OUT, reset RDY1, set ACK1, set RDY3, reset ACK3, 0.
(=01-1) IN-M[IND2]→M[IND2], IN - M[IND2]→OUT, reset RDY2, set ACK2, set RDY3, reset ACK3, 0.
(=-11-) M[IND] → OUT, reset RDY1, set ACK1, set RDY3, reset ACK3, 0.
(=101- ) M[IND] → OUT, reset RDY1, set ACK1, set RDY3, reset ACK3, 0.

**Domanda 2**

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 non comporta vantaggi, se non quanto il range di ricerca non 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.

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

Simulando

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

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