Architettura degli Elaboratori - A.A. 2014-2015 - Primo Appello - 5 giugno 2015

Risultati e calendario degli orali su web appena disponibili

Esercizio 1

Una unità U implementa una "mappa" da N=16 posizioni. Ciascun elemento della mappa è costituito da una chiave k_i e da un valore v_i. Chiavi e valori sono interi positivi e diversi da 0 su 32 bit. L'unità riceve da una unità U_1 richieste di operazione:

- di lettura del valore relativo alla chiave K. Restituisce il valore corrispondente alla chiave, o -1 se la chiave non è presente nella mappa
- di scrittura della coppia <K, V> relativa ad una chiave K non presente nella mappa. Se non ci sono posizioni libere, viene restituito -1
- cancellazione della coppia con chiave K, relativa ad una chiave presente nella mappa (la posizione occupata diventa libera)
- cancellazione dell'intera mappa (tutte le posizioni diventano libere)

L'interfaccia di U-U1 comprende i registri K, VIN e OP in ingresso a U e VOUT in uscita verso U1, oltre agli opportuni e necessari indicatori a transizione di livello.

La mappa è implementata mediante due componenti di memoria standard, da N parole, uno per le chiavi (MK) e uno per i valori (MV).

Si fornisca il microprogramma dell'unità U, si progetti la parte operativa dell'unità U e si indichi il numero di ingressi e di uscite della PC. Si calcoli infine il ciclo di clock della unità sapendo che:

- il ritardo della rete che calcola le uscite (ω_{PC}) (il prossimo stato interno (σ_{PC})) nella parte controllo è $3t_p$ ($3t_p$),
- il tempo di lettura di A è $5t_p$, e quello di scrittura è $3t_p$,
- il ritardo della ALU (operazioni intere) è $5t_p$ e dei commutatori (2 o 4 ingressi) è $2t_p$.

Esercizio 2

Si consideri il seguente frammento pseudo codice:

```
int a[N], b[N], c[N], d[N];
for i=0 to N-1 {
        a[i]=a[i]*b[i]+c[i]+d[i];
        b[i]=b[i]+a[i]+x;
        d[i]=c[i]+d[i];
}
```

Il risultato della compilazione è eseguito su un processore D-RISC pipeline con EU parallela pipeline che esegue somme e sottrazioni intere in *t* e moltiplicazioni e divisioni intere in 4*t*. Le pagine della cache di primo livello sono di 8 parole (σ=8) e la cache opera su domanda con politica *write-through*. La gerarchia di memoria è costituita da cache di primo livello e memoria principale interallacciata.

Si chiede di

a) scrivere il codice prodotto dalla compilazione con le regole standard D-RISC

b) calcolare il numero di fault generati dalla esecuzione del frammento nell'ipotesi che la dimensione della cache sia tale da poter contenere l'intero working set,

c) indicare come sia possibile ridurre questo numero e di quanto

d) ottimizzare il codice e calcolare le prestazioni del codice originale e di quello ottimizzato.

Traccia di soluzione

Esercizio 1

Prima di tutto definiamo l'interfaccia di U con U1, costituita da

- RDY, indicatore a transizione di livelo, KIN da 32 bit, VIN da 32 bit e OP da 2 bit in ingresso. Assumiamo che OP=00 indichi la prima operazione, OP=01 la seconda e così via.
- ACK indicatore a transizione di livello e VOUT da 32 bit in uscita

L'interazione fra U1 e U è a domanda risposta.

Manteniamo il numero delle posizioni occupate in un registro USED da 5 bit. $USED_0$ = 1 sarà a codizione di mappa piena, OR(USED)=0 quella di mappa vuota. Per ogni singola posizione nella mappa, consideriamo la presenza di una chiave pari a (0) come la condizione di "posizione vuota". Assumiamo che all'atto dell'avvio, tutte i registri e le memorie valgano 0 (diversamente sarebbe necessaria un'inizializzazione per USED ed MK).

Il microcodice potrebbe essere scritto come segue (I è un registro da 5 bit, utilizzato per i cicli):

0. */* attesa di un'operazione */*
   (RDY,OP,$USED_0$=0---) NOP, 1
   / operazione di lettura chiave KIN */*
   (=100-) I=0, -1→VOUT, zero(MK[0]-KIN) →FOUND, 1
   / scrittura Kin,Vin */*
   (=1010) I=0, -1→VOUT, VIN→VOUT, not(OR(MK[0])) →FOUND, 2
   (=1011) -1→VOUT, SET ACK, 0 */* non ci sono posizioni libere da scrivere */*
   / cancellazione della coppia con chiave KIN */*
   (=110-) I=0, zero(MK[0]-KIN) →FOUND,3
   / cancellazione dell'intera mappa */*
   (=111-) I=0, 4
1. (I_0,FOUND=00) I+1→I, zero(MK[I+1]-Kin)→FOUND,1 */* non trovato, vai avanti */*
   (=01) I→VOUT,MV[I]→VOUT, set ACK, 0 */* trovato */*
   (=1-) set ACK, -1→VOUT, 0 */* non trovato */*
2. (I_0,FOUND=00) I+1→I, zero(MK[I+1]-Kin)→FOUND,2 */* non trovato, vai avanti */*
   (=01) VIN→MV[I], KIN→MK[I], USED+1→USED, set ACK, 0 */* trovato */*
   (=1-) set ACK, -1→VOUT, 0 */* non trovato */*
3. (I_0,FOUND=00) I+1→I, zero(MK[I+1]-Kin)→FOUND,3 */* non trovato, vai avanti */*
   (=01) 0→MV[I], 0→MK[I], USED-1→USED, VIN→VOUT, set ACK, 0 */* trovato */*
   (=1-) set ACK, -1→VOUT, 0 */* non trovato */*
4. (I_0=0) I+1→I, 0→MK[I], 0→MV[I], 4
   (=1) set ACK, 0→VOUT, 0→USED, 0

Le variabili di condizionamento (uscite PO verso PC) sono 5: RDY, OP, $USED_0$, I_0 e FOUND. Le classi di registri sono nella PO sono:

- USED con commutatore in ingresso (ingressi 0 costante e uscita della ALU). 2 ingressi di controllo dalla PC: β di scrittura β_{USED} e α del commutatore α_{KUSED}
- VOUT con commutatore in ingresso (ingressi 0, -1 e VIN). 3 ingressi dalla ALU (β e α da due bit)
- MK, indirizzata con I, 0 o con l'uscita della ALU. Vi vengono scritti 0 o KIN. Dunque servono due commutatori, uno per gli indirizzi (4 vie, α da 2 bit) e uno per l'ingresso (2 vie,

α da 1 bit). Agli α si aggiunge il β di scrittura per un totale di 4 ingressi di controllo dalla PC

- MV, letta sempre con I e su cui si scrive 0 o VIN, dunque con un β per la scrittura e un α per il commutatore d'ingresso
- FOUND, con ingressi dalla ALU o dall'OR bit a bit dell'uscita MK. Quindi serve un β per la scrittura e un α per il commutatore d'ingresso
- I, con ingressi dalla ALU e costante uguale a 0, ancora serve un β per la scrittura e un α per il commutatore d'ingresso
- RDY e ACK, per cui serve un β ciascuno

Sono necessarie due ALU, visto che nelle microistruzioni troviamo contemporaneamente I+1→I, zero(MK[I+1]-Kin). Una ALU la utilizziamo per le operazioni di incremento e decremento di I e USED. Dunque avrà un commutatore in ingresso (scelta fra I e USED) e la possibilità di eseguire 2 operazioni (+ e -). Servono un α per il commutatore e un α per l'operazione, entrambi da 1 bit. L'altra ALU calcola sempre il flag ZERO della sottrazione fra l'uscita della memoria MK e KIN, dunque non necessita di variabili di controllo.

In totale dunque abbiamo 19 segnali di controllo dalla PC.

Il ciclo di clock lo calcoliamo valutando le sue diverse componenti:

- $T_{\omega PO}$: 0, non vengono usate variabili di condizionamento complesse
- $T_{\sigma PO}$: $t_k + t_m + t_{alu} + t_k$, lettura dalla memoria (commutatore sugli indirizzi e tempo di lettura), ALU e scrittura in un registro con commutatore all'ingresso, micro operazione più lunga fra quelle nelle frasi del microprogramma (zero(MK[I+1]-Kin)→FOUND)
- $T_{\omega PC}$: 3tp, dato
- $T_{\sigma PC}$: 3tp, dato

Dunque $\tau = 0 + \max \{ 3t_p + (2t_p + 5t_p + 5t_p + 2t_p),\ 3t_p\} + t_p = 18\ t_p$

Si noti che la soluzione proposta non fa uso di variabili di condizionamento complesse. Era chiaramente possibile utilizzarle, per esempio testando immediatamente sia l'uguaglianza di una posizione della memoria con la chiave data che l'or della chiave alla stessa posizione della memoria. Questo avrebbe spostato il peso dell'operazione della ALU con ingresso dalla memoria su tutte le microistruzioni.

Esercizio 2

La compilazione in D-RISC secondo le regole standard produrrà il codice assembler:

1. loop: LOAD Ra, Ri, Rai
2. LOAD Rb, Ri, Rbi
3. MUL Rai, Rbi, Rt1
4. LOAD RC, Ri, Rci
5. ADD Rci, Rt1, Rt1
6. LOAD Rd, Ri, Rdi
7. ADD Rdi, Rt1, Rt1
8. STORE Ra, Ri, Rt1
9. ADD Rbi, Rai, Rt2
10. ADD Rt2, Rx, Rt2
11. STORE Rb, Ri, Rt2
12. ADD Rci, Rdi, Rt3
13. STORE Rd, Ri, Rt3
14. INC Ri
15. IF< Ri, Rn, loop

con le dipendenze evidenziate (in nero le EU-EU e in rosso le IU-EU). Il numero di fault del programma può essere valutato analizzando i tipi di accesso in memoria effettuati:

- per tutti gli array abbiamo località ma non riuso (una volta utilizzata la posizione *i-esima*, si passa ad utilizzare la *(i+1)-esima*)
- per il codice abbiamo località e riuso, trattandosi di un ciclo eseguito N volte.

Dunque il working set è cosituito da un blocco di A, B, C, D e dai blocchi che contengono il codice, ed è ragionevole pensare (sono 6 blocchi in tutto) che non vi siano problemi di spazio nella cache di primo livello. Il numero di fault sarà pertanto il numero di fault fisiologici: N/σ fault per ciascuno degli array e 2 fault per il codice. Adottando prefetching, il numero dei fault può essere ridotto a 5 (il solo fault iniziale per gli array e per il codice).

Le dipendenze evidenziate nel codice danno luogo ad una certa inefficienza nell'esecuzione del codice stesso. Le dipendenze IU-EU sono tutte di distanza 1 e l'unica dipendenza EU-EU contribuisce alla prima delle dipendenze IU-EU.

La simulazione nell'esecuzione del codice è dunque:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
IM	1-L	2-L	3-M	4-L	5-A	6-L	7-A	8-S	9-A						10-A	11-S		12-A	13-S		14-I	15-IF		16..	1-L			
IU		1-L	2-L	3-M	4-L	5-A	6-L	7-A	8-S					8-S	9-A	10-A	11-S	11-S	12-A	13-S	13-S	14-I	15-IF	15-IF	16..	1-L		
DM			1-L	2-L		4-L		6-L							8-S				11-S			13-S					1-L	
Eum				1-L	2-L	3-M	4-L	5-A			5-A	6-L	7-A			9-A	10-A			12-A			14-I					1-L
EU*/							3-M	3-M	3-M	3-M																		

ed evidenzia un tempo di servizio pari a T = 24t/15 e quindi un'efficienza di poco superiore al 60%. In questo caso abbiamo assunto che sia IU che EU operino *in-order*. Con una EU *out-of-order* la situazione sarebbe leggermente migliorata, dal momento che la LOAD (6) avrebbe potuto essere eseguita sulla EUmaster al ciclo 8, riducendo di 1 la bolla generata dalla STORE (8).

Dall'analisi data flow del codice, possiamo tuttavia evidenziare che la STORE (8) può essere postposta, non contribuendo di fatto a nessuna delle operazioni successive. In realtà tutte le store possono essere postposte alla fine del ciclo. Utilizzando un ulteriore registro inizializzato alla base del vettore decrementata di 1, possiamo anche utilizzare una delle STORE per riempire il delay slot dell'IF che chiude il ciclo, in modo da eliminare la bolla legata al salto preso. Infine, possiamo anticipare il calcolo del valore da memorizzare in B[i] e C[i] prima delle store, per ridurre ulteriormente l'effetto delle dipendenze sulle STORE. Dunque possiamo considerare il codice seguente:

1. loop: LOAD Ra, Ri, Rai
2. LOAD Rb, Ri, Rbi
3. MUL Rai, Rbi, Rt1
4. LOAD Rc, Ri, Rci
5. LOAD Rd, Ri, Rdi
6. ADD Rbi, Rai, Rt2
7. ADD Rt2, Rx, Rt2
8. ADD Rci, Rt1, Rt1
9. ADD Rdi, Rt1, Rt1
10. ADD Rci, Rdi, Rt3
11. STORE Rb, Ri, Rt2
12. INC Ri
13. STORE Ra', Ri, Rt1
14. IF< Ri, Rn, loop, delayed
15. STORE Rd', Ri, Rt3

Abbiamo anche anticipato la INC Ri, in modo che la dipendenza indotta sulla IF< sia eliminata del tutto. Dunque vediamo con la simulazione:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
IM	1-L	2-L	3-M	4-L	5-L	6-A	7-A	8-A	9-A	10-A	11-S	12-I	13-S	14-IF	15-S	1-L			
IU		1-L	2-L	3-M	4-L	5-L	6-A	7-A	8-A	9-A	10-A	11-S	12-I	13-S	14-IF	15-S	1-L		
DM			1-L	2-L		4-L	5-L						11-S		13-S		15-S	1-L	
Eum				1-L	2-L	3-M	4-L	5-L	6-A	7-A	8-A	9-A	10-A	12-I	13-I				1-L
EU*/							3-M	3-M	3-M	3-M									

che riusciamo ad ottenere un tempo di servizio T = 15t/15 = t e dunque efficienza massima.

Si sarebbe potuto anche notare che la sottoespressione

$$c[i]+d[i]$$

compariva due volte nel codice e dunque poteva essere calcolata una volta sola, risparmiando una delle ADD (la 10. nel codice ottimizzato).