Architettura degli Elaboratori – A.A. 2014-2015

Secondo appello – 26 giugno 2015

Riportare nome, cognome, matricola e corso di appartenenza su tutti i fogli consegnati.
Risultati e calendario degli orali saranno pubblicati su web appena disponibili.

Esercizio 1

Una unità firmware U incorpora una memoria M da N=1M parole e interagisce con altre due unità U_1 ed U_2. Dalle due unità riceve richieste relative alle seguenti operazioni:

- inizializzazione (a zero) della memoria
- assegnazione di un valore VAL alla posizione IND
- sommatoria delle posizioni da INDI a INDJ (compresi) e restituzione del risultato
- restituzione del prodotto dei valori alle posizioni INDI e INDJ

Si progetti l'unità fornendone la lunghezza del ciclo di clock e il tempo medio di elaborazione, assumente che le quattro operazioni vengano richieste con probabilità 0.1, 0.3, 0.3 e 0.3, rispettivamente (si assuma $T_{alu} = 5t_p$ e $T_a = 5t_p$ (in lettura e scrittura) per M).

Esercizio 2

Si fornisca un esempio di codice assembler il cui tempo di completamento risulti diverso quando eseguito su due sistemi con gerarchia di memoria con un unico livello di cache prima della memoria principale. L'unica differenza fra i due sistemi consiste nella cache, associativa su insiemi a due vie nel primo sistema e a quattro vie nel secondo sistema, ma di pari capacità complessiva. Si discutano adeguatamente tutti gli aspetti che determinano il diverso tempo di completamento.

Esercizio 3

Si consideri l'esecuzione della computazione che sui vettori interi di N posizioni A, B e C, calcola C[i] = A[i]*k1 + B[i]/k2 (per ogni i fra 0 e N-1) su un'architettura pipeline con EU che calcola moltiplicazione e divisione intera a 4 stadi. Se ne fornisca una versione ottimizzata e se ne valutino le prestazioni, assumendo che il codice esegue senza generare fault in cache.

BOZZA DI SOLUZIONE

Esercizio 1

Supponiamo l'esistenza di due interfacce con i registri (indicatori a transizione di livello)

- VAL_i, $INDI_i$, $INDJ_i$ (32 bit), OP (2 bit) e RDY_i in ingresso dalla unità U_i (con $i \in [1,2]$)
- RES_i (32 bit) e ACK_i verso la unità U_i

Per ogni registro X_i assumiamo anche che esista una rete combinatoria che calcola X_i a partire da X_1 e X_2 tramite un commutatore il cui α è prodotto tramite controllo un registro di un bit T, settato opportunamente nella prima microistruzione.

Assumendo di dedicare un ciclo all'arbitraggio delle richieste da parte delle due unità U_1 e U_2, il microcodice dell'unità potrebbe essere scritto come segue:

0. (RDY_1,RDY_2,TURNO=00-) nop, 0 // scelta unità da servire : nessuna richiesta
   (=10-) 0→T, 1 // richiesta dalla sola U1
   (=01-) 1→T, 1 // richiesta dalla sola U2
   (=110) 0→T, 1→TURNO, 1 // richiesta da entrambe, turno alla prima
   (=111) 1→T, 0→TURNO, 1 // richiesta da entrambe, turno alla seconda
1. (OP=00) 0→I, 2 // OP = ciclo di azzeramento
   (=01) VAL_t→M[$INDI_t$], reset RDY_t, set ACK_t, 0 // OP = assegnamento
   (=10) 0→SUM, $INDI_t$→I, $INDJ_t$→J, 3 // OP = calcolo somma dell'intervallo
   (=11) M[$INDI_t$]*M[$INDJ_t$] →RES_t, set RDY_t, reset ACK_t, 0 // OP = calcolo del prodotto
2. (I_0=0) 0→M[I], I+1→I,2 // ciclo di azzeramento
   (=1) reset RDY_t, set ACK_t, 0 // alla fine manda il risultato
3. (zero(I-J)=0) SUM+M[I] →SUM, I+1→I, 3 // ciclo di somma
   (=1) reset RDY_t, set ACK_t, SUM+ M[I] →RES_t, 0 // alla fine manda il risultato

Si sarebbe potuto risparmiare il ciclo per l'arbitraggio (la microistruzione 0) testando direttamente anche OP, ma questo avrebbe richiesto una certa cura nella gestione della variabile utilizzata per il controllo residuo.

Per calcolare il ciclo di clock della unità U dobbiamo considerare i vari valori $T_{\omega PC}$, $T_{\sigma PC}$, $T_{\omega PO}$ e $T_{\sigma PC}$.

La parte controllo avrà 2 bit di stato, 7 ingressi (variabili di condizionamento) e un certo numero di uscite (α e β, variabili di controllo verso la PO). Dunque 9 ingressi (il che richiede un doppio livello di porte AND) e un massimo di 256 (2^(9-1)) 1 per ciascuna delle uscite (il che richiede 3 livelli di porte OR), per un totale di $5t_p$ sia per la $T_{\omega PC}$ che per la $T_{\sigma PC}$.

Per la parte operativa abbiamo $5t_p$ per il $T_{\omega PO}$, dovuti alla variabile di condizionamento complessa nella microistruzione 3. La micro operazione più lunga è la M[$INDI_t$]*M[$INDJ_t$] →RES_t che richiede un tempo di stabilizzazione di un commutatore per l'indirizzo della memoria (che deve essere a doppia porta per permettere i due accessi contemporanei), un tempo di accesso in memoria, un t_{alu} ed infine la stabilizzazione del commutatore in ingresso per la scrittura nel registro per un totale di $t_k + t_a + t_{alu} + t_k = 14\ t_p$.

Secondo la formula che esprime la maggiorazione del tempo di ciclo:

$$\tau = T_{\omega PO} + \max\{ T_{\omega PC} + T_{\sigma Po}, T_{\sigma PC}\} + \delta$$

avremo dunque un ciclo di clock pari a $\tau = t_p * (5 + \max\{5 + 14, 5\} + 1) = 25\ t_p$.

Le operazioni richiedono, al netto delle attese per la disponibilità delle richieste:

- OP1, inizializzazione: $2\tau + (M+1)\tau$
- OP2, assegnamento: 2τ
- OP3, calcolo della somma dell'intervallo (di M/2 valori in media): $2\tau+(M/2+1)\tau$
- OP4, moltiplicazione: 2τ

Viste le percentuali il tempo medio di elaborazione sarà dunque la somma pesata di questi valori:

$$T = 1/10\ (2\tau + (M+1)\tau) + 3/10\ (2\tau) + 3/10\ (2\tau+(M/2+1)\tau) + 3/10\ (2\tau)$$
$$= \tau(18+3M/2)/10 \cong\ 3\tau M/20$$

Esercizio 2

Perchè l'esecuzione del codice richieda tempi diversi sulle due diverse architetture, occorre che il working set del programma sia tale da non poter essere allocato nelle due cache in modo da generare lo stesso numero di fault. Per esempio, un codice che accede, correndoli, a 3 vettori per i quali gli indirizzi fisici mappino sullo stesso insieme della cache set associativa, causano un numero di fault pari ai soli fault fisiologici (N/σ ciascuno) nel caso della cache set associativa a quattro vie, mentre causano N fault nel caso di cache set associativa a due vie. Va sottolineato che stiamo parlando di indirizzi fisici, quindi non basta dire qual è il codice Assembler, ma occorre anche indicare (parte di) quelle informazioni della tabella di rilocazione che determinano l'individuazione dello stesso insieme nella cache. Dunque il codice potrebbe essere un codice che opera su vettori tipo:

```
LOOP: LOAD Rba, Ri, R1
      LOAD Rbb, Ri, R2
      ADD R1, R2, R3
      STORE Rbc, Ri, R3
      INC Ri
      IF< Ri, Rn, LOOP
```

quando, detto BLOCCO_CACHE(IND) la parte di indirizzo che identifica il blocco di cache set associativa,

BLOCCO_CACHE(TABRIL(Rba.IPL)|Rba.OFF) = BLOCCO_CACHE(TABRIL(Rbb.IPL)|Rbb.OFF) = BLOCCO_CACHE(TABRIL(Rbc.IPL)|Rbc.OFF)

Esercizio 3

Il codice derivato dallo pseudo codice

```
for(int i=0; i<N; i++) { c[i] = a[i]*k1 + b[i]/k2; }
```

compilato secondo le regole standard è

```
LOOP: LOAD Rba, Ri, R1
      MUL R1, Rk1, R1
      LOAD Rbb, Ri, R2
      DIV R2, Rk2, R2
      ADD R1, R2, R1
      STORE Rbc, Ri, R1
      INC Ri
      IF< Ri, Rn, LOOP
```

per il quale abbiamo dipendenze EU-EU indotte dalla MUL e DIV sulla ADD, e dipendenze IU-EU della ADD sulla STORE e della INC sulla IF. Dalla simulazione vediamo che il tempo di servizio è T = 16t/8 = 2t.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
IM	LOAD	MUL	LOAD	DIV	ADD	STORE	INC							IF<			LOAD			
IU		LOAD	MUL	LOAD	DIV	ADD	STORE						STORE	INC	IF<	IF<		LOAD		
DM			LOAD		LOAD									STORE					LOAD	
EUm				LOAD	MUL	LOAD	DIV	ADD				ADD			INC					LOAD
EU*/						MUL	MUL	MUL	MUL											
								DIV	DIV	DIV	DIV									

Dunque l'efficienza è del 50%. Per diminuire l'effetto delle dipendenze possiamo posticipare la ADD, anticipando la INC, utilizzare delayed branch e inserire la STORE nel delay slot:

```
LOOP: LOAD Rba, Ri, R1
      MUL R1, Rk1, R1
      LOAD Rbb, Ri, R2
      DIV R2, Rk2, R2
      INC Ri
      ADD R1, R2, R1
      IF< Ri, Rn, LOOP, DELAYED
      STORE Rbc', Ri, R1
```

che porta il tempo di servizio a T = 12t/8 = 6t/4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IM	LOAD	MUL	LOAD	DIV	INC	ADD	IF<		STORE				LOAD			
IU		LOAD	MUL	LOAD	DIV	INC	ADD	IF<	IF<	STORE			STORE	LOAD		
DM			LOAD		LOAD										LOAD	
EUm				LOAD	MUL	LOAD	DIV	INC	ADD			ADD				LOAD
EU*/						MUL	MUL	MUL	MUL							
								DIV	DIV	DIV	DIV					

Dunque fra il primo e il secondo codice l'efficienza passa dal 50 al 66%.