Architettura degli elaboratori – A.A. 2019—20 – 1° appello – 9/01/20

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

Un'unità firmware B è collegata ad un'altra unità firmware A ed a una memoria esterna M da 16K parole di 32 bit. B riceve da A un indirizzo di M e preleva da M la parola il cui indirizzo è stato fornito da A, e la invia ad A. Successivamente, preleva da M le parole che occupano posizioni consecutive, a partire da quella successiva indicata sopra, ed invia ad A la generica parola i-esima solamente se l'operazione di lettura è stata eseguita correttamente ed il suo contenuto è maggiore di quello dell'ultima parola inviata ad A. Ad esempio, se A invia a B l'indirizzo della parola che contiene "2" della seguente sequenza di parole 2,3,3,1,5,4,… allora B invierà ad A i valori 2,3,5,… B continua ad operare come su descritto finché non riceve un nuovo indirizzo da A. A quel punto, B ripete la procedura su descritta, a partire dal nuovo indirizzo.

Si implementi l'unità B e se ne fornisca la lunghezza del ciclo di clock, facendo le seguenti assunzioni:

- Sono a disposizione ALU che fanno operazioni fra interi in $5t_p$
- La prima parola prelevata ("2" nell'esempio) è sempre corretta
- Le porte logiche hanno al massimo 8 ingressi

Domanda 2

In riferimento al seguente frammento di pseudocodice:

```
for(k=0;k<N;k++) {
  U[k,k] = A[k,k];
  for(int i=k+1; i<N; i++) {
    L[i,k]=A[i,k]/U[k,k];
    U[k,i] = A[k,i];
  }
}
```

si forniscano:

• compilazione in assembler D-RISC secondo le regole di compilazione standard;

• prestazioni (tempo di servizio) del ciclo interno su un processore D-RISC pipeline con unità EU slave pipeline a 4 stadi per l'esecuzione delle moltiplicazioni e divisioni intere;

• cause di degrado delle prestazioni;

• eventuali ottimizzazioni del ciclo interno (quantificando il guadagno in termini di tempo di servizio).

Schema di soluzione

Domanda 1

Assumiamo che l'interfaccia verso M sia standard, mentre l'interfaccia verso A deve essere composta da due coppie di sincronizzatori a transizione di livello per gestire invii di parola da B verso A e richieste di indirizzi da A verso B in modo autonomo. Assumiamo quindi che questa interfaccia sia costituita da:

- Indicatori a transizione di livello RDYreqA (ingresso), ACKreqA (uscita) e registro INDinA per le richieste da A verso B
- Indicatori a transizione di livello ACKansA (ingresso), RDYansA (uscita) e registro ANSWER, per le risposte da B verso A

Il microcodice contiene tre microistruizioni (stati):

- Uno per l'attesa di richieste da A
- Uno per trattare la prima risposta da M da inviare sicuramente ad A
- Ed uno per trattare le successive riposte da M e/o l'eventuale nuova richiesta da A.

Microcodice

0. (RDYreqA=0) nop, 0.
   (=1) INDinA -> IND, INDinA -> INDIRIZZO, "read"->OP, set RDYoutM, reset RDYreqA, set ACKreqA, 1
1. *// ricezione della prima parola*
   (RDYinM, or(ESITO), ACKansA = 0 - -) nop, 1
   // comunico la prima parola letta alla unità A (non è previsto che possa avere or(ESITO)=1)
   (=101) DATAIN->ANSWER, DATAIN -> ULTIMA, set RDYansA, resetACKansA, reset RDYinM
   // e poi mi appresto a leggerne una seconda
   INDIRIZZO + 1 -> IND, INDIRIZZO+1 -> INDIRIZZO, "read" -> OP, set RDYoutM, 2.
2. *// se non ho richieste o risposte, attendo*
   (RDYreqA, RDYinM, or(ESITO), maggiore(DATAIN,ULTIMA), ACKansA = 0 0 - - -) nop, 2
   // se ho solo una richiesta da A, la servo immediatamente
   (=1 0 - - -) INDinA -> IND, INDinA -> INDIRIZZO, "read"->OP, set RDYoutM, reset RDYreqA, set ACKreqA, 1
   // se ho solo una risposta da M, che verifica la condizione, la invio
   (=0 1 0 1 1) DATAIN->ANSWER, DATAIN->ULTIMA, set RDYansA, reset ACKansA, reset RDYinM,
   // e poi richiedo la parola successiva (assumiamo che +1 sia calcolato modulo 16K)
   INDIRIZZO + 1 -> IND, INDIRIZZO + 1 -> INDIRIZZO, DATAIN -> ULTIMA, "read" -> OP, set RDYoutM, 2.
   // se ho solo una risposta da M, che non verifica la condizione, chiedo la parola successiva
   (=0 1 0 0 -) reset RDYinM, INDIRIZZO + 1 -> IND, INDIRIZZO + 1 -> INDIRIZZO, DATAIN -> ULTIMA, "read" -> OP, set RDYoutM, 2.
   // se ho solo una risposta, con esito non positivo chiedo comunque la prossima
   (=0 1 1 - -) reset RDYinM, INDIRIZZO + 1 -> IND, INDIRIZZO + 1 -> INDIRIZZO, DATAIN -> ULTIMA, "read" -> OP, set RDYoutM, 2.
   // se infine ho sia una nuova richiesta da A che una risposta da M, dò priorità alla nuova richiesta
   (=1 1 - - -) reset RDYinM, INDinA -> IND, INDinA -> INDIRIZZO, "read"->OP, set RDYoutM,
   reset RDYreqA, set ACKreqA, 1

Valutazione delle prestazioni

Sono 3 stati (2 bit per il registro di stato). Ho un massimo di 5 variabili di condizionamento testate contemporaneamente. Livello AND di sigma e omega della PC ha 7 ingressi max => $1t_p$. Il livello OR ha max 5 ingressi (10 frasi, se ho più di 5 "1" codifico gli 0 e nego il risultato) => $1t_p$. In totale quindi 2tp per $omega_{PC}$ e $sigma_{PC}$.

Per la sigmaPO abbiamo solo operazione di trasferimento fra registri e set/reset di sincronizzatori, oltre a una INDIRIZZO + 1 -> INDIRIZZO, che richiede un $t_{alu}+t_k$ (in INDIRIZZO scriviamo o il valore del registro di interfaccia da A o l'uscita della ALU).

La variabile di condizionamento complessa "maggiore(DATAIN,ULTIMA)" richiede anch'essa un t_{alu} e compare in frasi dove la sigmaPO prevede anch'essa un $t_{alu}+t_k$.

Il ciclo di clock sarà quindi pari a:

$$t_{alu} + 2t_p + t_{alu}+t_k+ t_p = 15\ t_p$$

Domanda 2

Codice assembler

Compilando il codice

```
for(k=0;k<N;k++) {
  U[k,k] = A[k,k];
  for(int i=k+1; i<N; i++) {
    L[i,k]=A[i,k]/U[k,k];
    U[k,i] = A[k,i];
  }
}
```

secondo le regole di compilazione standard otteniamo:

```
      CLEAR Rk
Fork: MUL Rk, Rn, Rkk
      ADD Rkk, Rk, Rkk
      LOAD RbaseA, Rkk, R1
      STORE RbaseU, Rkk, R1
      ADD Rk, #1, Ri
Fori: IF>= Ri, Rn, Endk  // necessario per via della inizializzazione
      MUL Ri, Rn, Rik
      ADD Rik, Rk, Rik
      LOAD RbaseA, Rik, R2
      DIV R2, R1, R2
      STORE RbaseL, Rik, R2
      MUL Rk, Rn, Rki
      ADD Rki, Ri, Rki
      LOAD RbaseA, Rki, R3
      STORE RbaseU, Rki, R3
```

```
Endi:  INC Ri
       GOTO Fori
Endk:  INC Rk
       IF< Rk, Rn, FORk
       END
```

Si noti che per la compilazione del ciclo interno non si è potuto utilizzare la versione ottimizzata (controllo della variabile di iterazione alla fine del ciclo) perchè non è garantito che il corpo debba essere eseguito almeno una volta. Infatti, quando k = n-1 si entra con i = n e dunque bisogna terminare immediatamente senze eseguire nessuna iterazione del ciclo interno.

Prestazioni del codice del ciclo interno

Il ciclo più interno è composto dal codice evidenziato. Abbiamo diverse dipendenze (fra parentesi il numero della istruzione):

- Dipendenze EU-EU
  MUL (2) -> ADD (3), MUL (7) -> ADD (8),
- Dipendenze EU-IU
  DIV (5) -> STORE (6), ADD (3) -> LOAD (4), ADD (8) -> LOAD (9), LOAD(9) -> STORE(10)

La simulazione fa vedere come per eseguire ca12 istruzioni occorrano in realtà 31t (efficienza inferiore a 1/3).

	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	28	29	30	31		
IM	IF>=	MUL	ADD	LOAD						DIV	STORE							MUL	ADD	LOAD						STORE			INC	GOTO	XXX	MUL		
IU		IF>=	MUL	ADD	LOAD					LOAD	DIV	STORE						STORE	MUL	ADD	LOAD					LOAD	STORE		STORE	INC	GOTO	XXX	MUL	
DM											LOAD								STORE								LOAD			STORE				
EU m				MUL								LOAD	DIV							MUL	ADD				ADD			LOAD			INC			MUL
EU */1					MUL	ADD		ADD						DIV							MUL													
EU */2						MUL									DIV							MUL												
EU */3							MUL									DIV							MUL											
EU */4								MUL									DIV							MUL										

Ottimizzazioni

Il codice

```
Fori:  IF>= Ri, Rn, Endk  // necessario per via della inizializzazione
       MUL Ri, Rn, Rik
       ADD Rik, Rk, Rik
       LOAD RbaseA, Rik, R2
       DIV R2, R1, R2
       STORE RbaseL, Rik, R2
       MUL Rk, Rn, Rki
       ADD Rki, Ri, Rki
       LOAD RbaseA, Rki, R3
       STORE RbaseU, Rki, R3
Endi:  INC Ri
       GOTO Fori
```

lo possiamo ottimizzare pre calcolando gli offset ed eseguendo in modo posticipato le store, sfruttando anche un salto delayed per il ritorno alla prima istruzione del ciclo:

```
Fori: IF>= Ri, Rn, Endk  // necessario per via della inizializzazione
      MUL Ri, Rn, Rik
      MUL Rk, Rn, Rki
      ADD Rik, Rk, Rik
      ADD Rki, Ri, Rki
      LOAD RbaseA, Rik, R2
      DIV R2, R1, R2
      LOAD RbaseA, Rki, R3
Endi: INC Ri
      STORE RbaseL, Rik, R2
      GOTO Fori, delayed
      STORE RbaseU, Rki, R3
```

Questo porta ad tempo di completamento della singola iterazione pari a soli 19t con un netto miglioramento rispetto alla versione iniziale:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	2
IM	IF>=	MUL	MUL	ADD	ADD	LOAD				DIV	LOAD	INC	STORE	GOTO					STORE	IF>=		
IU		IF>=	MUL	MUL	ADD	ADD	LOAD			LOAD	DIV	LOAD	INC	STORE				STORE	GOTO	STORE	IF>=	
DM											LOAD		LOAD						STORE		STORE	
EU m				MUL	MUL	ADD			ADD	ADD		LOAD	DIV	LOAD	INC							
EU */1					MUL	MUL								DIV								
EU */2						MUL	MUL								DIV							
EU */3							MUL	MUL								DIV						
EU */4								MUL	MUL								DIV					

Le add e le mul per il calcolo del secondo offset ([k,I]) impiegano parte del tempo che prima andava a formare la bolla indotta dalla ADD sulla LOAD. Le store eseguite durante il ritorno ad inizio ciclo danno modo alle load che producono I risultati da memorizzare di terminarne il calcolo.