Architettura degli elaboratori A.A. 2014-2015
Quarto appello – 8 Settembre 2015

Riportare su tutti i fogli Nome, Cognome, Matricole e Corso (A o B).
I risultati saranno resi noti sulle pagine web dei docenti appena disponibili.

Domanda 1

Si consideri un'unità U_C che implementa una cache con metodo di indirizzamento diretto di 4K linee. La cache implementa una politica *write-through*. Ogni linea della cache permette la memorizzazione di un blocco da 8 parole (σ=8). L'unità è interfacciata con il processore P e con una memoria centrale modulare M, interallacciata (4 moduli di memoria da 1G parole ciascuno). Si elenchino le interfacce fra U_C P ed M e si fornisca il microcodice dell'unità U_C, in modo da minimizzare il tempo di servizio. Si consideri la possibilità di utilizzare un'interfaccia fra U_C e M che permetta la lettura/scrittura di 4 a parole alla volta.

Domanda 2

Si consideri il seguente codice:

```
int f(int x) {
  return(x*x + 3*x – 4);
}

integral = 0;

for(int x=a; x<=b; x+=d) {
  integral = (integral + f(x)) * d;
```

e se ne fornisca la compilazione in assembler D-RISC.

Successivamente, si ottimizzi il codice utilizzando anche le tecniche di inlining (espansione del codice di una procedura/funzione al posto della chiamata) e loop unrolling e se ne valutino le prestazioni su un processore D-RISC pipeline, dotato di unità in grado di calcolare moltiplicazione e divisione intere in 4 stadi.

Traccia di soluzione

Domanda 1

Per l'unità consideriamo un'interfaccia di memoria standard da P/MMU (che riferiamo con l'indice 1) e una interfaccia di memoria di ampiezza pari a 4 parole verso M (che riferiamo con l'indice 2). Le operazioni di lettura e scrittura da e per la memoria M utilizzeranno dunque due registri DATAIN2 e DATAOUT2 da 4 parole. Il registro IND2 sarà sempre di 32 bit, ma gli ultimi 2 bit saranno sempre uguali a 00, visto che le operazioni di lettura e scrittura si riferiscono a quattro parole consecutive.

Il microprogramma potrebbe essere scritto come segue:

```
// prima microoperazione: trattiamo il caso semplice di lettura e
// scrittura con cache hit immediatamente. Serve quindi controllare
// subito se il tag cercato è quello che si trova alla linea indirizzata
// dalla parte centrale dell'indirizzo ricevuto dalla MMU:
//   +---------------------------------+---------+--------+
//   |       TAG                       |   BC    |  OFF   |
//   +---------------------------------+---------+--------+
//
0. (RDY1, zero(IND1.TAG-CACHE[IND1.BC].TAG), OP1, ACK2 = 0---)
      // se non ci sono richieste da parte del processore, attendi
      nop, 0

   (=110-) // richiesta di lettura, dato presente in cache (hit lettura)
     CACHE[IND1.BC].PAROLE[IND1.OFF] → DATAIN1,
     0 → ESITO1, reset RDY1, set ACK1, 0

   (=1111) // richiesta di lettura, dato presente in cache (hit scrittura)
     // scrittura nella posizione della cache
     DATAOUT1 → CACHE[IND1.BC].PAROLE[IND.OFF],
     0 → ESITO1, reset RDY1, set ACK1,
     // trattamento write-through: si scrivono le prime (le ultime)
     // 4 parole della linea a seconda se OFF < 4 (OFF>3)
     f(CACHE[IND1.BC].PAROLE,OFF) → DATAOUT2,
     IND1[31..2]..”00” → INDOUT2
     “write” → OP2, set RDY2,
      // quindi si ricomincia a servire la prossima richiesta
      0
   // hit scrittura, ma la memoria principale non ha ancora terminato
   // la precedente richiesta (di scrittura)
   (=1110) nop, 0 // attesa memoria principale

   // cache miss: in questo caso si rimpiazza la linea vittima
   // e poi si serve la richiesta. Servono due letture, una per le
   // prime quattro parole e una per le ultime quattro
   (=10-1)
     // richiesta di lettura delle prime quattro parole
     IND1[31..3]..”000” → INDOUT, “read” → OP2,
     set RDY2, reset ACK2, 1

// attesa delle prime quattro parole
1. (RDY2, or(ESITO2)=0-) nop, 1
    // richiesta delle seconde quattro parole
   (=10) DATAIN2 → CACHE[IND1.BC].PAROLE[0..3],
         IND1.TAG → CACHE[IND1.BC].TAG,
         IND1[31..3]..”100” → INDOUT2, “read” → OP2,
         SET RDY2, reset ACK2, 2
```

// attesa delle seconde quattro parole
2. (RDY2, or(ESITO2)=0-) nop, 2,
   // torna alla prima microistruzione e rianalizza l'op
   // questa volta sarà un cache hit
   (=10) reset ACK2, 0

La funzione f(PAROLE, OFF) è definita in modo da restituire PAROLE[0..3] se OFF < 4 e PAROLE[4..7] altrimenti. In termini di implementazione, l'intera espressione f(CACHE[IND1.BC].PAROLE,OFF) può essere implementata indirizzando e leggendo l'intera linea di otto parole e inviandole ad un commutatore da due ingressi da 4 parole il cui segnale di controllo è dato 0 se OFF<4 e 1 altrimenti.

L'accesso alla memoria interna CACHE avviene sia nella generazione della variabile di condizionamento zero(IND1.TAG-CACHE[IND1.BC].TAG) nella prima microistruzione che in altre istruzioni, con indirizzi diversi. Per soddisfare la condizione di correttezza (PO di Moore) occorre realizzare la memoria come una memoria a doppia porta: il primo indirizzo (che comanda un commutatore di lettura) serve per il calcolo della variabile di condizionamento, il secondo, che comanda sia il commutatore di lettura che quello di scrittura, servirà per tutti gli altri accessi.

Il del tempo di servizio è minimizzato perché gli hit (in lettura e scrittura) sono serviti in un unico ciclo di clock (una microistruzione) e che il trattamento dei miss utilizza altre due microistruzioni per la gestione del fault (necessarie per reperire 8 parole con una banda di memoria da 4 parole a operazione) + una microistruzione per l'esecuzione dell'operazione quando si è sicuri che il contenuto della linea richiesta sia quello corretto.

La cache si troverà ragionevolmente sullo stesso chip del processore, dunque occorre sincerarsi che l'utilizzo di una variabile di condizionamento complesso non "allunghi" il ciclo di clock. In questo caso la valutazione della condizione richiede un tempo pari all'accesso alla memoria (lettura TAG) più un confronto fra due valori da 32 (bit della parola) – 3 (bit di offset) – 10 (bit BC) = 19 bit. Un confrontatore fra valori da 19 bit può essere costruito utilizzando 19 confrontatori da 1 bit (2tp di ritardo) e due livelli di porte and per generare il segnale finale a partire dai 19 bit che rappresentano il risultato dei confrontatori. Dunque il ritardo introdotto sarà di 4tp in tutto.

La prima microistruzione richiede dunque un ciclo di clock di almeno $4t_p$ per la stabilizzazione delle uscite della PO, 2tp per il ritardo di stabilizzazione della rete che calcola le uscite della PC (2 bit di stato e 6 variabili di condizionamento in ingresso) e $1t_a + 1t_k$ per la stabilizzazione dell'operazione più lunga (CACHE[IND1.BC].PAROLE[f(OFF)] → DATAOUT2). Questo non è lontano dal tempo che abbiamo calcolato come tempo di ciclo per il processore D-RISC monolitico.

L'alternativa sarebbe decomporre la 0. in due parti: generazione del risultato del test sui TAG (0.1) e valutazione del risultato del test (0.2):

0.1 (RDY1=0) nop,
   (=1) zero(IND1.TAG-CACHE[IND1.BC].TAG) → Z, 0.2
0.1 (Z, OP1, ACK2= …) // esecuzione delle operazioni richieste

Questo ridurrebbe a 0 il tempo per produrre le variabili di condizionamento, ma lascerebbe invariato quello necessario alla stabilizzazione della funzione di transizione dello stato interno della PO. Il guadagno potrebbe essere al più pari a $4t_p$ per ciclo di clock, a fronte della necessità di due cicli di clock per la realizzazione dell'accesso in caso di cache hit.

Domanda 2

Va compilata sia la funzione che il codice del programma principale.
Compiliamo la procedura utilizzando un meccanismo di passaggio dei parametri (in ingresso e in uscita) per registro: utilizziamo Rx per il valore x in ingresso e Rpres per il risultato in uscita. L'indirizzo di ritorno sarà mantenuto in un registro Rret. Il codice della funzione potrebbe dunque essere:

```
1. fun:  MUL Rx, Rx, Rx2
2.       MUL Rx, #3, R3x
3.       ADD Rx2, R3x, R1
4.       SUB R1, #4, Rpres
5.       GOTO Rret
```

Il codice del programma chiamante sarà invece (assumendo Rfun e Ra inizializzati dal compilatore):

```
6. prog:  ADD R0, R0, Rintegral      // inizializza integrale
7.        ADD Ra, R0, Rx             // valore iniziale in Rx
8. loop:  CALL Rfun, Rret            // chiamata funzione
9.        ADD Rintegral, Rpres, Rintegral   // agg. Parziale
10.       MUL Rintegral   , Rd, Rintegral
11.       ADD Rx, Rd, Rx             // agg. Var iterazione
12.       IF<= Rx, Rb, loop          // controllo fine loop
13.       END
```

C'è una dipendenza EU-EU nel codice: la 1 e la 2 di fun sulla 3. C'è inoltre una dipendenza IU-EU (la 10 sulla 11 nel codice principale) e tre salti presi per iterazione (CALL, GOTO, IF<=).
La simulazione del codice ci darà:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
IM	CALL	ADD	MUL	MUL	ADD	SUB					GOT(	XXX	ADD	MUL	ADD	IF<=		END	CALL	
IU		CALL	~~ADD~~	MUL	MUL	ADD	SUB				SUB	GOT(	~~XXX~~	ADD	MUL	ADD	IF<=	IF<=	~~END~~	CALL
DM																				
Eum					MUL	MUL	ADD				ADD	SUB			ADD	MUL	ADD			
Eui*						MUL	MUL	MUL	MUL								MUL	MUL	MUL	MUL
							MUL	MUL	MUL	MUL										

Il tempo di servizio in questo caso sarà t = 18 t / 10 = 9 t / 5.

Vista la piccola dimensione del codice della funzione, si può utilizzare l'inlining (copia del codice al posto della chiamata di procedura):

```
1. prog:  ADD R0, R0, Rintegral      // inizializza integrale
2.        ADD Ra, R0, Rx             // valore iniziale in Rx
3. loop:  MUL Rx, Rx, Rx2
4.        MUL Rx, #3, R3x
5.        ADD Rx2, R3x, R1
6.        SUB R1, #4, Rpres
7.        ADD Rintegral, Rpres, Rintegral
8.        MUL Rintegral, Rd, Rintegral
9.        ADD Rx, Rd, Rx             // agg. Var iterazione
10.       IF<= Rx, Rb, loop   // controllo fine loop
11.       END
```

Abbiamo rimosso due bolle legate ai salti presi della CALL e GOTO.

Col codice così modificato possiamo ridurre l'effetto della dipendenza IU-EU. La 9. può essere anticipata immediatamente dopo l'ultimo utilizzo di Rx, ovvero immediatamente dopo la 4.

```
1. prog:  ADD R0, R0, Rintegral      // inizializza integrale
2.        ADD Ra, R0, Rx             // valore iniziale in Rx
3. loop:  MUL Rx, Rx, Rx2
```

```
4.        MUL Rx, #3, R3x
5.        ADD Rx, Rd, Rx
6.        ADD Rx2, R3x, R1
7.        SUB R1, #4, Rpres
8.        ADD Rintegral, Rpres, Rintegral
9.        MUL Rpres, Rd, Rpres
10.       IF<= Rx, Rb, loop   // controllo fine loop
11.       END
```

Questo ha anche un effetto sulla dip EU-EU fra le moltiplicazioni e la ADD. La simulazione ci fa vedere come si riesca a scendere a un t = 12 t / 8 = 3 t / 2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
IM	MUL	MUL	ADD	ADD	SUB	ADD				MUL	IF<=	END	MUL						
IU		MUL	MUL	ADD	ADD	SUB				ADD	MUL	IF<=	~~END~~	MUL					
DM																			
Eum			MUL	MUL	ADD	ADD			ADD	SUB	ADD	MUL			MUL				
Eui*				MUL	MUL	MUL	MUL						MUL	MUL	MUL	MUL			
					MUL	MUL	MUL	MUL								MUL	MUL	MUL	MUL

Possiamo mitigare anche l'effetto del salto IF<= effettuando un loop unrolling: eseguiamo più iterazioni consecutivamente, testando ad ogni fine delle iterazioni interne se sia il caso di terminare (e dunque saltando alla fine del loop se e solo se la condizione negata è vera):

```
prog:  ADD R0, R0, Rintegral      // inizializza integrale
       ADD Ra, R0, Rx             // valore iniziale in Rx
loop:  MUL Rx, Rx, Rx2
       MUL Rx, #3, R3x
       ADD Rx, Rd, Rx             // agg. Var iterazione
       ADD Rx2, R3x, R1
       SUB R1, #4, Rpres
       ADD Rintegral, Rpres, Rintegral
       MUL Rintegral, Rd, Rintegral
       IF> Rx, Rb, end            // controllo fine loop
       MUL Rx, Rx, Rx2
       MUL Rx, #3, R3x
       ADD Rx, Rd, Rx             // agg. Var iterazione
       ADD Rx2, R3x, R1
       SUB R1, #4, Rpres
       ADD Rintegral, Rpres, Rintegral
       MUL Rintegral, Rd, Rintegral
       IF<= Rx, Rb, loop          // controllo fine loop
end:   END
```

Infine, possiamo trasformare il codice in modo da eliminare anche il ritardo dovuto all'ultimo salto, utilizzando il delay slot:

```
prog:  ADD R0, R0, Rintegral      // inizializza integrale
       ADD Ra, R0, Rx             // valore iniziale in Rx
loop:  MUL Rx, Rx, Rx2
       MUL Rx, #3, R3x
       ADD Rx, Rd, Rx             // agg. Var iterazione
       ADD Rx2, R3x, R1
       SUB R1, #4, Rpres
       ADD Rintegral, Rpres, Rintegral
       MUL Rintegral, Rd, Rintegral
       IF> Rx, Rb, end            // controllo fine loop
       MUL Rx, Rx, Rx2
       MUL Rx, #3, R3x
       ADD Rx, Rd, Rx             // agg. Var iterazione
       ADD Rx2, R3x, R1
       SUB R1, #4, Rpres
```

```
        ADD Rintegral, Rpres, Rintegral
        IF<= Rx, Rb, loop, delayed        // controllo fine loop
        MUL Rintegral, Rd, Rintegral
end:    END
```

Per questo codice la simulazione ci darà t = 22 t / 16 = 11 t / 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23					
IM	MUL	MUL	ADD	ADD	SUB	ADD				MUL	IF>	MUL	MUL	ADD	ADD	SUB	ADD				IF<=	MUL	MUL	MUL					
IU		MUL	MUL	ADD	ADD	SUB				ADD	MUL	IF>	MUL	MUL	ADD	ADD	SUB				ADD	IF<=	MUL	MUL	MUL				
DM																													
Eum			MUL	MUL	ADD	ADD			ADD	SUB	ADD	MUL		MUL	MUL	ADD	ADD			ADD	SUB	ADD		MUL	MUL	MUL			
Eui*				MUL	MUL	MUL	MUL						MUL	MUL	MUL	MUL									MUL	MUL	MUL	MUL		
					MUL	MUL	MUL	MUL							MUL	MUL	MUL	MUL								MUL	MUL	MUL	MUL	
																MUL	MUL	MUL	MUL							MUL	MUL	MUL	MUL

Se avessimo saputo che le iterazioni erano multiplo di 2, avremmo potuto chiaramente eliminare la IF> intermedia, scendendo ancora un po' con il tempo di servizio (t = 21 t / 16)