ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
www.cslab.ece.ntua.gr

ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΥΠΟΛΟΓΙΣΤΩΝ (5ο εξάμηνο)
ΕΠΑΝΑΛΗΠΤΙΚΗ ΕΞΕΤΑΣΗ (ΦΘΙΝΟΠΩΡΟ 2007)
ΔΙΑΡΚΕΙΑ ΕΞΕΤΑΣΗΣ: 2 ΩΡΕΣ 30 ΛΕΠΤΑ

Οι εξετάσεις θα πραγματοποιηθούν ΧΩΡΙΣ την παρουσία βιβλίων, βοηθημάτων ή άλλου είδους σημειώσεων. Το μόνο που επιτρέπεται να έχετε μαζί σας είναι ένα φύλλο Α4 στο οποίο μπορείτε να έχετε γράψει ό,τι έχετε κρίνει πιο σημαντικό για το μάθημα και θέλετε να το έχετε ως βοήθημά σας. Απαγορεύεται η ανταλλαγή οποιουδήποτε αντικειμένου κατά την ώρα της εξέτασης, ούτε και των φύλλων Α4 που είναι ατομικά.

Καλή Επιτυχία!

Θέμα 1ο

Ο αλγόριθμος φυσαλίδας (bubblesort) είναι ένας απλός αλγόριθμος ταξινόμησης που στηρίζεται σε αλλεπάλληλα περάσματα μιας λίστας αριθμών και την εναλλαγή οποιωνδήποτε δύο διαδοχικών αριθμών που δε βρίσκονται στη σωστή σειρά. Οι επαναλήψεις σταματούν όταν δε χρειάζεται πλέον να γίνει καμία εναλλαγή αριθμών σε ένα πέρασμα, που σημαίνει ότι η λίστα έχει ταξινομηθεί. Η βασική λειτουργία του αλγορίθμου, για την ταξινόμηση ενός πίνακα Α με n στοιχεία, συνοψίζεται στα παρακάτω βήματα:

1. ***i = 0***
2. ***j ← n-1 (δείχνει στο τελευταίο στοιχείο του πίνακα Α)***
3. ***αν A[j] < A[j-1], τότε ενάλλαξε τα δύο στοιχεία***
4. ***j ← j - 1, πήγαινε στο 3 αν j > i***
5. ***αν έγινε εναλλαγή, i ← i +1 και πήγαινε στο 2***

Καλείστε να υλοποιήσετε τον παραπάνω αλγόριθμο σε γλώσσα assembly του MIPS. Υποθέστε ότι ο καταχωρητής $1 περιέχει το πλήθος των στοιχείων του πίνακα Α, ενώ ο καταχωρητής $2 τη διεύθυνση του πρώτου στοιχείου του Α. Επιπλέον, κάθε στοιχείο του Α αποτελείται από 4 bytes.

Για μεγαλύτερη σαφήνεια, χρησιμοποιούμε συμβολικά ονόματα για τους καταχωρητές.

```
# $N: number of elements of A ($1)
# $A: start address of A ($2)
# $end: end address of A plus one
# $current: address of current element (A[j])
# $zero: contains 0 ($0)

      addi $i, $zero, 0              #i=0

Outer:
      addi $j, $N, -1                #j=n-1
      sll $end, $N, 2                #end=n*4
      add $current, $A, $end         #current = A+end
```

```
      addi $swapflag, $zero, 0          #swapflag=0

Inner:
      addi $current, $current, -4       #current=current-4
      lw $tmp1, 0($current)             #tmp1=A[j]
      lw $tmp2, -4($current)            #tmp2=A[j-1]
      slt $cmpflag, $tmp1, $tmp2        #compare A[j],A[j-1]
      beq $cmpflag, $zero, Continue     #if(A[j]>=A[j-1]),we shouldn't swap elements
      sw $tmp2, 0($current)             #A[j]=tmp2
      sw $tmp1, -4($current)            #A[j-1]=tmp1
      addi $swapflag, $zero, 1          #swap occurred

Continue:
      addi $j, $j, -1                   #j=j-1
      slt $cmpflag, $i, $j
      bne $cmpflag, $zero, Inner        #if(i<j), goto Inner
      beq $swapflag, $zero, Exit        #if no swap occurred during this pass, exit
      addi $i, $i, 1                    #i=i+1
      j Outer

Exit:
```

Θέμα 2ο

α) Εξηγείστε συνοπτικά τις ακόλουθες έννοιες: συσχετιστικότητα κρυφής μνήμης, ποινή αστοχίας, χωρική τοπικότητα, χρονική τοπικότητα.

Σελ. 515, 489, 486 από το βιβλίο "Οργάνωση και Σχεδίαση Υπολογιστών" (Patterson, Hennessy).

β) Θεωρήστε το ακόλουθο κομμάτι κώδικα:

```
#define M 8
#define N 4
float a[M+1][N], b[M][N], c[M][N], d[M+2][N];
for (i=0; i<M; i++)
      for (j=0; j<N; j++)
            d[i+2][j] = a[i+1][j] + b[i][j]*c[i][j];
```

Οι πίνακες a,b,c,d περιέχουν στοιχεία κινητής υποδιαστολής μονής ακρίβειας, μεγέθους 4 bytes. Κάνουμε τις εξής υποθέσεις:

- ***Το πρόγραμμα εκτελείται σε έναν επεξεργαστή με μόνο ένα επίπεδο κρυφής μνήμης δεδομένων. Η κρυφή μνήμη είναι συσχέτισης 2 δρόμων (2 way set associative), write-allocate, αποτελείται από 8 blocks δεδομένων, και έχει LRU πολιτική αντικατάστασης. Το μέγεθος του block είναι 32 bytes. Η μικρότερη μονάδα δεδομένων που μπορεί να διευθυνσιοδοτηθεί είναι το 1 byte, ενώ οι διευθύνσεις μνήμης έχουν εύρος 32 bit. Αρχικά, η κρυφή μνήμη δεδομένων είναι άδεια.***
- ***Υποθέτουμε ότι όλες οι μεταβλητές, πλην των στοιχείων των πινάκων, μπορούν να αποθηκευτούν σε καταχωρητές του επεξεργαστή, οπότε οποιαδήποτε αναφορά σε αυτές δεν συνεπάγεται προσπέλαση στην κρυφή μνήμη. Επίσης, ο επεξεργαστής στέλνει προς εκτέλεση τα loads του προγράμματος, με τη σειρά που αυτά εμφανίζονται στο πρόγραμμα.***
- ***Οι πίνακες είναι αποθηκευμένοι στην κύρια μνήμη κατά γραμμές. Για κάθε ένα από τα στοιχεία a[0][0], b[0][0], c[0][0], d[0][0], οι διευθύνσεις του πρώτου byte είναι οι (δίνονται σε 16-δική μορφή): 00004000, 00005000, 00006000, 00007000, αντίστοιχα.***

β.1) Βρείτε ποιες από τις αναφορές στα στοιχεία των πινάκων για όλη την εκτέλεση του παραπάνω κώδικα καταλήγουν σε misses στην cache. Υποδείξτε ποια είναι compulsory, ποια είναι capacity, και ποια conflict. Δώστε τον συνολικό αριθμό των misses.

Καταρχήν, παρατηρούμε ότι σε ένα cache block χωράνε 8 διαδοχικά (κατά γραμμές) στοιχεία του πίνακα. Η cache αποτελείται από 4 sets. Απο τα 32 bits μιας διεύθυνσης, τα 5 least-significant bits αντιστοιχούν στο byte offset (αφού έχουμε 32 bytes ανά cache line). Επιπλέον, αφού έχουμε 4 sets συνολικά, τα 2 επόμενα least-significant bits θα αντιστοιχούν στο index, θα εκφράζουν δηλαδή το set στο οποίο απεικονίζεται το block που περιέχει το byte στο οποίο αναφέρεται η διεύθυνση.
Όπως βλέπουμε επομένως από τις αρχικές διευθύνσεις των πινάκων, όλοι οι πίνακες είναι ευθυγραμμισμένοι ώστε το πρώτο στοιχείο τους να βρίσκεται στην αρχή ενός cache block (byte offset ίσο με το 0), και επιπλέον, τα cache blocks στα οποία ανήκουν θα απεικονίζονται όλα στο set 0 της cache (index ίσο με το 0).
Με άλλα λόγια, για τον πίνακα d για παράδειγμα, τα (διαδοχικά) στοιχεία [d_{00} d_{01} d_{02} d_{03} d_{10} d_{11} d_{12} d_{13}] ανήκουν όλα στο ίδιο block το οποίο θα απεικονιστεί στο set 0, τα στοιχεία [d_{20}...d_{33}] στο set 1, τα στοιχεία [d_{40}...d_{53}] στο set 2, τα στοιχεία [d_{60}...d_{73}] στο set 3, και τα στοιχεία [d_{80}...d_{93}] πάλι από την αρχή στο set 0. Όμοια θα γίνεται η απεικόνιση και στους υπόλοιπους πίνακες.

Για την πρώτη επανάληψη του εσωτερικού loop, οι αναφορές που γίνονται είναι οι ακόλουθες:
a_{10} b_{00} c_{00} d_{20} m m m m (compulsory)
Τα blocks των στοιχείων a_{10}, b_{00}, c_{00} απεικονίζονται όλα στο set 0, ενώ το block του d_{20} απεικονίζεται στο set 1. Μόλις γίνεται η αναφορά στο c_{00}, το μπλοκ που το περιέχει θα αντικαταστήσει το μπλοκ του a_{10} (που είναι το LRU block στο set 0). Έτσι, στις επόμενες επαναλήψεις του εσωτερικού loop, όπου θα γίνονται αναφορές σε στοιχεία των a, b, c τα οποία ανήκουν στα ίδια blocks, θα έχουμε συνεχώς κυκλικές εκτοπίσεις των αντίστοιχων blocks και επομένως conflict misses, όπως φαίνεται παρακάτω. Αντίθετα, τα στοιχεία d_{21}, d_{22}, d_{23} θα βρίσκονται ήδη στην cache αφού έχουν έρθει μαζί με το d_{20} και δεν έχουν εκτοπιστεί από το set 1 όπου βρίσκονται.
a_{11} b_{01} c_{01} d_{21} m m m h (conflict)
a_{12} b_{02} c_{02} d_{22} m m m h (conflict)
a_{13} b_{03} c_{03} d_{23} m m m h (conflict)

Στην επόμενη επανάληψη του εξωτερικού loop, το block του στοιχείου a_{20} θα απεικονιστεί στο set 1 (στον άλλο δρόμο από αυτόν που βρίσκεται το block που περιέχει τα στοιχεία [d_{20}...d_{33}]). Τα blocks που περιέχουν τα στοιχεία των b και c στα οποία γίνονται αναφορές σε αυτή την επανάληψη, βρίσκονται από το προηγούμενο βήμα στους δύο δρόμους του set 0. Έτσι, όλες οι αναφορές για αυτή την επανάληψη του εξωτερικού loop (πλην της πρώτης αναφοράς στο a_{20}) θα είναι hits:
a_{20} b_{10} c_{10} d_{30} m h h h (compulsory)
a_{21} b_{11} c_{11} d_{31} h h h h
a_{22} b_{12} c_{12} d_{32} h h h h
a_{23} b_{13} c_{13} d_{33} h h h h

Στην επόμενη επανάληψη του εξωτερικού loop, το block του στοιχείου a_{30} βρίσκεται ήδη από προηγουμένως στο set 1. Σε αυτό το set όμως θα έρθουν τα blocks που περιέχουν τα στοιχεία των b και c στα οποία γίνονται αναφορές σε αυτή την επανάληψη του εξωτερικού loop. Επομένως, θα έχουμε κυκλικές εκτοπίσεις όπως και στην πρώτη επανάληψη.
a_{30} b_{20} c_{20} d_{40} h m m m (compulsory)
a_{31} b_{21} c_{21} d_{41} m m m h (conflict)

a_{32} b_{22} c_{22} d_{42} m m m h (conflict)
a_{33} b_{23} c_{23} d_{43} m m m h (conflict)

Οι επόμενες επαναλήψεις του εξωτερικού loop, συνεχίζονται όπως η 2η και η 3η επανάληψη. Τις παρουσιάζουμε για λόγους πληρότητας:

a_{40} b_{30} c_{30} d_{50} m h h h (compulsory)
a_{41} b_{31} c_{31} d_{51} h h h h
a_{42} b_{32} c_{32} d_{52} h h h h
a_{43} b_{33} c_{33} d_{53} h h h h

a_{50} b_{40} c_{40} d_{60} h m m m (compulsory)
a_{51} b_{41} c_{41} d_{61} m m m h (conflict)
a_{52} b_{42} c_{42} d_{62} m m m h (conflict)
a_{53} b_{43} c_{43} d_{63} m m m h (conflict)

a_{60} b_{50} c_{50} d_{70} m h h h (compulsory)
a_{61} b_{51} c_{51} d_{71} h h h h
a_{62} b_{52} c_{52} d_{72} h h h h
a_{63} b_{53} c_{53} d_{73} h h h h

a_{70} b_{60} c_{60} d_{80} h m m m (compulsory)
a_{71} b_{61} c_{61} d_{81} m m m h (conflict)
a_{72} b_{62} c_{62} d_{82} m m m h (conflict)
a_{73} b_{63} c_{63} d_{83} m m m h (conflict)

a_{80} b_{70} c_{70} d_{90} m h h h (compulsory)
a_{81} b_{71} c_{71} d_{91} h h h h
a_{82} b_{72} c_{72} d_{92} h h h h
a_{83} b_{73} c_{73} d_{93} h h h h

Συνολικά έχουμε 53 misses. Τα misses που προκύπτουν από αναφορές σε blocks τα οποία φορτώνονται για πρώτη φορά στην cache, είναι compulsory. Όλα τα υπόλοιπα είναι conflict, αφού οφείλονται σε αμοιβαίες εκτοπίσεις blocks εντός του ίδιου set.

β.2) Πώς θα ξαναγράφατε τον κώδικα ώστε να μειωθούν τα misses; Υπολογίστε εκ νέου τον αριθμό των misses.

Για να μειώσουμε τα misses, πρέπει να στοχεύσουμε στα conflict misses, αφού για τα compulsory ούτως ή άλλως δεν μπορούμε να κάνουμε κάποια άμεση τροποποίηση στο loop για να τα αποφύγουμε. Η λογική είναι να ομαδοποιήσουμε τις αναφορές στον κώδικα ώστε να μην έχουμε καταστάσεις όπου 3 blocks των a, b, c ανταγωνίζονται την ίδια στιγμή μεταξύ τους για 2 cache lines του ίδιου set, με αποτέλεσμα το ένα block να διώχνει το άλλο. Επομένως, αυτό που μπορούμε να κάνουμε είναι να σπάσουμε τον κώδικα σε δύο επιμέρους βρόχους (μερικά αθροίσματα):

```
for (i=0; i<M; i++)
      for (j=0; j<N; j++)
            d[i+2][j] = a[i+1][j];
```

```
for (i=0; i<M; i++)
      for (j=0; j<N; j++)
            d[i+2][j] += b[i][j]*c[i][j];
```

Στο πρώτο loop έχουμε συνολικά 9 misses (μόνο τα compulsory του ερωτήματος β.1 τα οποία οφείλονται στις αναφορές στους a και d). Στο δεύτερο loop, τα στοιχεία $d_{20}...d_{93}$ βρίσκονται στην cache από το πρώτο loop, οπότε για τον d δεν έχουμε misses, με εξαίρεση τις 2 τελευταίες επαναλήψεις του εξωτερικού loop, όπου το μπλοκ $[d_{80}...d_{93}]$ δε βρίσκεται πλέον στο set 0 της cache καθώς εκτοπίστηκε στην πρώτη επανάληψη από τα blocks $[b_{00}...b_{13}]$ και $[c_{00}...c_{13}]$. Έτσι έχουμε τα 8 compulsory misses του ερωτήματος β.1 για τις αναφορές στα b και c, καθώς και 1 επιπλέον για τον d. Συνολικά δηλαδή έχουμε 9+8+1=18 misses.

Θέμα 3ο

Υποθέστε ότι έχουμε τα ακόλουθα στάδια σε μια αρχιτεκτονική σωλήνωσης: IF, ID, ALU1, MEM, ALU2, WB. Τα στάδια IF, ID και WB έχουν την ίδια λειτουργικότητα όπως και στην κλασική σωλήνωση 5 σταδίων του MIPS. Στο στάδιο ALU1 γίνεται ο υπολογισμός των τελικών διευθύνσεων μνήμης, για εντολές αναφοράς στη μνήμη, στο στάδιο MEM γίνεται η προσπέλαση στη μνήμη, ενώ στο στάδιο ALU2 γίνεται η εκτέλεση των αριθμητικών πράξεων. Σε αυτή την αρχιτεκτονική, ο ένας από τους τελεστέους προέλευσης (source operands) στις αριθμητικές εντολές μπορεί να είναι αναφορά σε μια θέση μνήμης, π.χ. add $1,$2,100($3). Ο υπολογισμός των διευθύνσεων-στόχων των εντολών διακλάδωσης γίνεται στο στάδιο ALU1, ενώ η εξέταση της συνθήκης για εντολές διακλάδωσης υπό συνθήκη γίνεται στο στάδιο ALU2.

α) Πόσοι αθροιστές απαιτούνται για αυτή την αρχιτεκτονική; Δώστε ένα παράδειγμα ακολουθίας εντολών οι οποίες θα χρησιμοποιούν ταυτόχρονα όλους τους αθροιστές σε κάποιον συγκεκριμένο κύκλο. Παρουσιάστε το διάγραμμα χρονισμού για την ακολουθία αυτή, και υποδείξτε τα στάδια όπου γίνεται η ταυτόχρονη χρήση των αθροιστών.

Τα στάδια που χρησιμοποιούν πράξεις πρόσθεσης είναι 3: το IF (που προσαυξάνει την τρέχουσα τιμή του PC ώστε να δείχνει στην επόμενη κατά σειρά εντολή), το ALU1 (που εκτελεί την πρόσθεση για τον υπολογισμό των τελικών διευθύνσεων, π.χ. 100+$3), και το ALU2 (για εκτέλεση πράξεων από αριθμητικές εντολές, π.χ. add). Απαιτούνται επομένως 3 αθροιστές.

add $1, $1, $2	IF	ID	A1	M	**A2**	WB											
inst1		IF	ID	A1	M	A2	WB										
lw $1, 10($2)			IF	ID	**A1**	M	A2	WB									
inst2				IF	ID	A1	M	A2	WB								
inst3					**IF**	ID	A1	M	A2	WB							

(οι εντολές inst1, inst2, inst3 μπορεί να είναι οποιεσδήποτε εντολές οι οποίες δεν προκαλούν κινδύνους και επομένως stalls στο pipeline)

β) Πόσες ταυτόχρονες προσπελάσεις στο αρχείο καταχωρητών θα πρέπει να υποστηρίζει αυτή η αρχιτεκτονική; Δώστε ένα παράδειγμα ακολουθίας εντολών όπου γίνεται χρήση του μέγιστου αριθμού καταχωρητών σε κάποιον συγκεκριμένο κύκλο. Παρουσιάστε το διάγραμμα χρονισμού

για την ακολουθία αυτή, και υποδείξτε τα στάδια όπου γίνεται η ταυτόχρονη προσπέλαση στο αρχείο καταχωρητών.

Το αρχείο καταχωρητών χρησιμοποιείται σε δύο στάδια, το ID και το WB. Στο ID, διαβάζονται οι source registers μιας εντολής. Για την αρχιτεκτονική που εξετάζουμε, ο μέγιστος αριθμός source registers που μπορούν να προσδιοριστούν σε μια εντολή είναι 2 (π.χ. add $3,$3,0($4)). Στο WB, γράφεται κάποιος target register. Επομένως, πρέπει να υποστηρίζονται το πολύ 3 ταυτόχρονες προσπελάσεις στο αρχείο καταχωρητών.

add $1, $1, $2	IF	ID	A1	M	A2	**WB**										
inst1		IF	ID	A1	M	A2	WB									
inst2			IF	ID	A1	M	A2	WB								
inst3				IF	ID	A1	M	A2	WB							
add $3,$3,0($4)					IF	**ID**	A1	M	A2	WB						

γ) Αν υπήρχε σχήμα προώθησης των αποτελεσμάτων από το στάδιο MEM σε οποιοδήποτε από τα στάδια ALU1, MEM, ALU2, θα μειώνονταν οι καθυστερήσεις; Δικαιολογείστε την απάντησή σας για κάθε ένα από τα στάδια αυτά. Σε όποια περίπτωση πιστεύετε ότι θα υπήρχε μείωση ή αποφυγή των καθυστερήσεων, δώστε και ένα παράδειγμα όπου θα γίνεται εμφανές αυτό, παρουσιάζοντας το αντίστοιχο διάγραμμα χρονισμού.

προώθηση MEM→ALU1

lw $1,0($2)	IF	ID	A1	M	A2	WB										
instr1		IF	ID	A1	M	A2	WB									
lw $3,0($1)			IF	ID	-	-	A1	M	A2	WB						
Με προώθηση:																
lw $1,0($2)	IF	ID	A1	M	A2	WB										
instr1		IF	ID	A1	M	A2	WB									
lw $3,0($1)			IF	ID	A1	M	A2	WB								

Στο παραπάνω παράδειγμα, οι εντολή instr1 είναι τέτοια ώστε να μην δημιουργεί κινδύνους στο pipeline. Χωρίς προώθηση, απαιτούνται 2 stalls πριν η δεύτερη load διαβάσει τα περιεχόμενα του $1. Με το σχήμα προώθησης, στο τέλος του 4ου κύκλου η τιμή που πρόκειται να αποθηκευτεί στον $1

(0($2)) προωθείται κατευθείαν από την έξοδο του MEM στις εισόδους της ALU1, εκεί δηλαδή που πρέπει να χρησιμοποιηθεί. Επομένως δεν περιμένουμε μέχρι να αποθηκευτεί πρώτα η τιμή αυτή στον $1, και έτσι αποφεύγονται οι καθυστερήσεις.

προώθηση MEM→MEM

lw $1,0($2)	IF	ID	A1	M	A2	WB										
sw $1,0($3)		IF	ID	-	-	-	A1	M	A2	WB						
Με προώθηση:																
lw $1,0($2)	IF	ID	A1	M	A2	WB										
sw $1,0($3)		IF	ID	A1	M	A2	WB									

Χωρίς προώθηση, απαιτούνται 3 stalls πριν η store διαβάσει τα περιεχόμενα του $1. Με το σχήμα προώθησης, στο τέλος του 4ου κύκλου, η τιμή που πρόκειται να αποθηκευτεί στον $1 προωθείται από την έξοδο της μονάδας MEM κατευθείαν στην είσοδό της. Επομένως δεν περιμένουμε μέχρι να αποθηκευτεί πρώτα η τιμή αυτή στον $1, και έτσι αποφεύγονται οι καθυστερήσεις.

προώθηση MEM→ALU2:

Εφόσον το στάδιο MEM βρίσκεται πριν το στάδιο ALU2 στο pipeline, δεν έχει νόημα να γίνει κάποιο είδος προώθησης (η ροή τιμών από ένα στάδιο του pipeline σε κάποιο επόμενο γίνεται ούτως ή άλλως, μέσω των ενδιάμεσων καταχωρητών).