ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

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

8 Ιουλίου 2005

ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΟΡΓΑΝΩΣΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Θέματα και Λύσεις Εξετάσεων Ιουλίου 2005

Επώνυμο:	
Όνομα:	
Εξάμηνο:	

Θέμα	Βαθμός
1	
2	
3	

Μην ξεχάσετε να γράψετε το ονοματεπώνυμο σας σε όλες τις κόλλες. Η εξέταση γίνεται με ανοικτά βιβλία και σημειώσεις, χωρίς χρήση Η/Υ ή PDAs. Διάρκεια εξέτασης 2 ½ ώρες. Το διαγώνισμα βαθμολογείται με άριστα 10 μονάδες. Στον τελικό βαθμό προαγωγής του μαθήματος προστίθεται 1 μονάδα (bonus) αν έχετε παραδώσει μία ικανοποιητική λύση στις δύο σειρές που μοιράσθηκαν στο μάθημα (0,5 μονάδες για κάθε σειρά ασκήσεων).

Θέμα 1ο (30%):

Οι *κρυφές μνήμες χωρίς κλειδώματα* (lockup-free caches) και η *πρώιμη ανάκληση δεδομένων* στο υλικό (hardware prefetching) σχεδιάστηκαν για να επιταχύνουν το ρυθμό εκτέλεσης των εντολών στον επεξεργαστή, επικαλύπτοντας το χειρισμό των αστοχιών κρυφής μνήμης (cache misses) με την επεξεργασία άλλων εντολών ή άλλων αστοχιών κρυφής μνήμης. Στο πρόβλημα αυτό υποθέστε ότι η ιεραρχία μνήμης διαθέτει κρυφή μνήμη ενός επιπέδου, με μέγεθος cache line 32 bytes και διάδρομο επικοινωνίας κύριας μνήμης με την κρυφή μνήμη (memory bus) 8 bytes. Υποθέστε, επίσης, ότι ο χρόνος εύρεσης των ζητούμενων δεδομένων στη μνήμη (miss latency) είναι t_{miss} = 2 cycles, και ο χρόνος μεταφοράς δεδομένων στη κρυφή μνήμη (transfer latency) είναι 1 cycle / 8 bytes, δηλαδή για την μεταφορά ολόκληρης της cache line: $t_{transfer}$ = 4 cycles. Τέλος, δίνεται ότι στο σύστημα 1word = 4 bytes.

Εξετάζουμε την εκτέλεση του εξής κώδικα:

```
for (i = 0; i < 1000; i+=4) {
        a += A[i] + B[i];
```

Θεωρούμε ότι οι τιμές των μεταβλητών `4·i`, `a` βρίσκονται στους καταχωρητές `$s4`, `$t0` και οι διευθύνσεις των πινάκων `A` και `B` στους `$s1` και `$s2`. Το περιεχόμενο του καταχωρητή `$s4` είναι αρχικά 0. Ο αντίστοιχος κώδικας σε assembly είναι ο εξής:

```
Loop:   add  $t1, $s1, $s4
        add  $t2, $s2, $s4
        lw   $t3, 0($t1)
```

```
            lw   $t4, 0($t2)
            add  $t0, $t0, $t3
            add  $t0, $t0, $t4
            addi $s4, $s4, 16
            bne  $s4, $s5, Loop
Exit:
```

Στον πίνακα που ακολουθεί φαίνεται η ροή εκτέλεσης του παραπάνω βρόχου, υποθέτοντας ότι έχουμε μοντέλο υλοποίησης single cycle και ότι κανένα από τα στοιχεία των πινάκων A και B δεν βρίσκονται ήδη στην κρυφή μνήμη (συμβολίζουμε με `m` = miss latency, `t` = transfer latency). Θεωρούμε ότι ο σχεδιασμός της κρυφή μνήμης επιτρέπει την προώθηση στον επεξεργαστή της ζητούμενης λέξης αμέσως μόλις αφιχθεί στην κρυφή μνήμη, χωρίς να χρειάζεται να ολοκληρωθεί η μεταφορά ολόκληρης της cache line. Επιπλέον, υποθέτουμε ότι μετά από μία αστοχία, όλες οι επόμενες εντολές πρέπει να περιμένουν έως η ζητούμενη λέξη φτάσει στον επεξεργαστή. Τα στοιχεία των πινάκων είναι ευθυγραμμισμένα στην κρυφή μνήμη, δηλαδή το στοιχείο A[0] καταλαμβάνει την πρώτη θέση μέσα σε κάποια cache line, ομοίως και το στοιχείο B[0].

Τέλος, ο μηχανισμός πρώιμης ανάκλησης δεδομένων, καταγράφει τις ακολουθίες προσπελάσεων στη μνήμη (αστοχίες της κρυφής μνήμης). Αν παρατηρηθούν τρεις συνεχόμενες προσπελάσεις ίδιου βήματος, τότε ενεργοποιείται η πρώιμη ανάκληση των επόμενων θέσεων μνήμης ίδιου βήματος (π.χ. οι αστοχίες δεδομένων που αναφέρονται στις θέσεις μνήμης a+0, a+32, a+64, ενεργοποιούν την πρώιμη ανάκληση των θέσεων μνήμης a+96, a+128, κλπ). Ο μηχανισμός πρώιμης ανάκλησης καλεί στην κρυφή μνήμη ένα δεδομένο (a+96) στον κύκλο που ακολουθεί αμέσως μετά τον πρώτο κύκλο ανάγνωσης από τη μνήμη της τρίτης αστοχίας (a+64). Επιπλέον, προκειμένου να μην κατακλυστεί η κρυφή μνήμη με δεδομένα που δεν χρειάζονται στον επεξεργαστή, κάθε επόμενη πρώιμη ανάκληση (a+128) περιμένει έως ότου κληθεί προς επεξεργασία το δεδομένο που έχει αποθηκευθεί στην κρυφή μνήμη από την προηγούμενη πρώιμη ανάκληση (a+96). Σημειώστε ότι επιτρέπεται η επικάλυψη κύκλων ανάγνωσης από την μνήμη (κύκλων `m`), αλλά ο διάδρομο επικοινωνίας της κύριας μνήμης με την κρυφή μνήμη (memory bus) μπορεί σε κάθε κύκλο να μεταφέρει δεδομένα μίας συγκεκριμένης cache line.

Συμπληρώστε το διάγραμμα χρονισμού του παραπάνω πίνακα για τους πρώτους 100 κύκλους εκτέλεσης. Πόσοι κύκλοι απαιτούνται για την εκτέλεση των 250 επαναλήψεων του `for loop`;

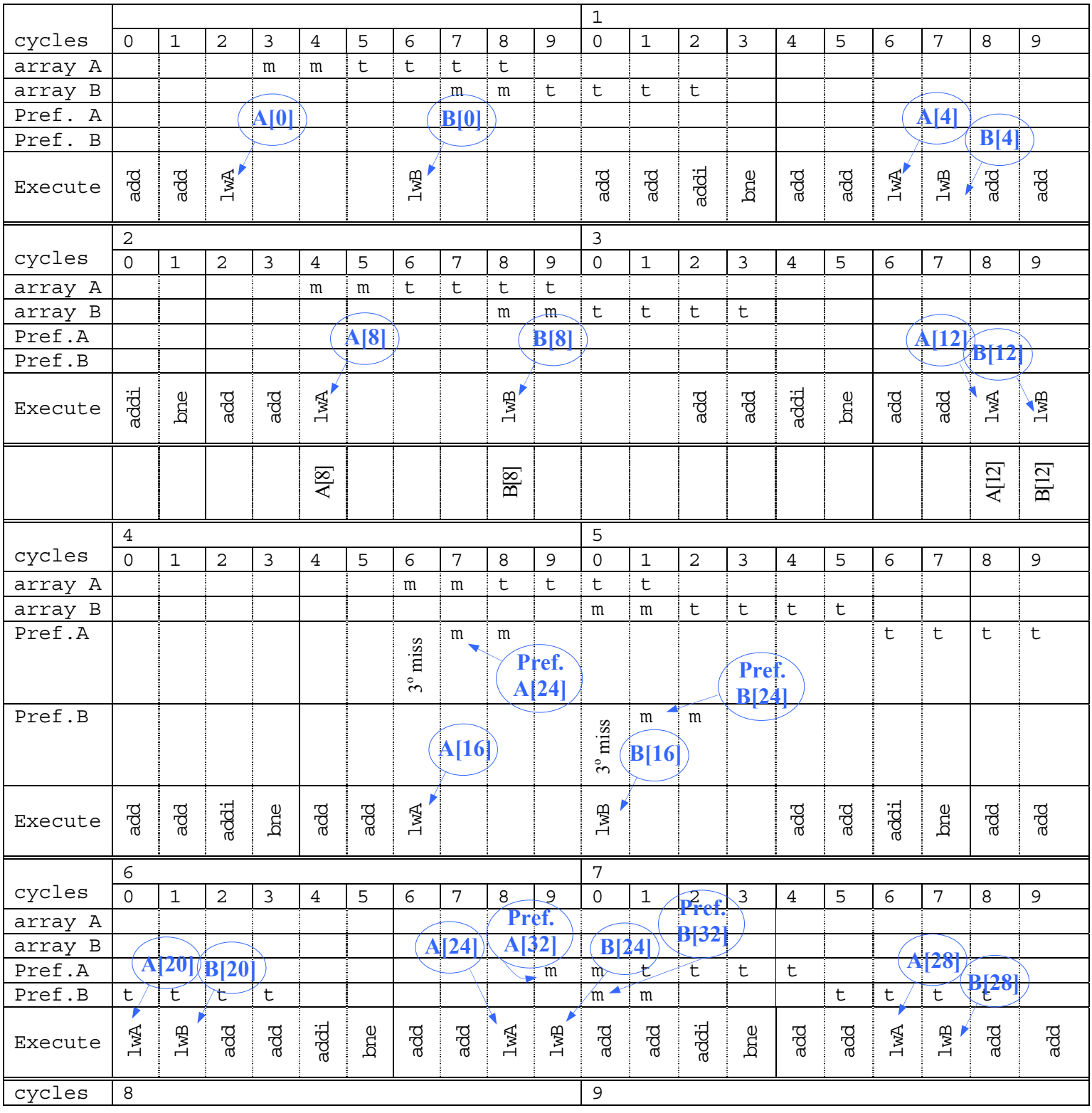
Λύση 1ου Θέματος

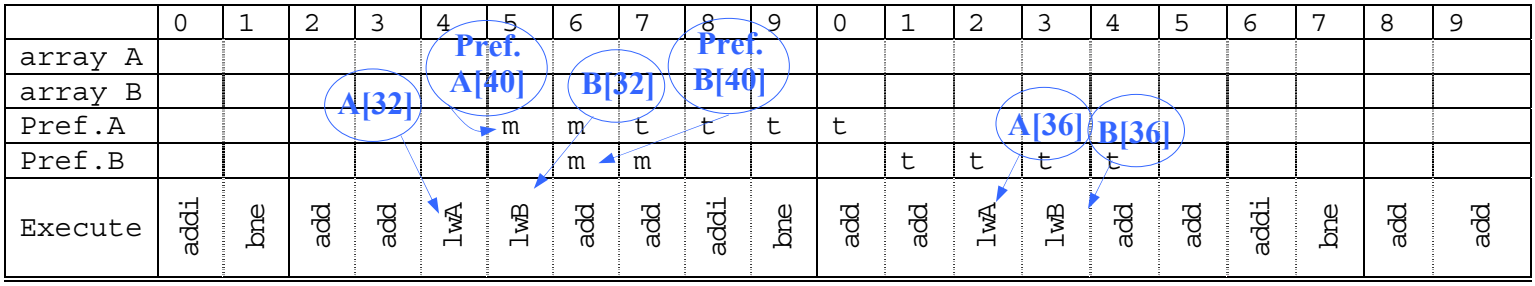
Εφόσον κανένα από τα στοιχεία των πινάκων `A` και `B` δεν βρίσκονται ήδη στην κρυφή μνήμη, κατά την πρώτη αναφορά σε ένα από τα στοιχεία μίας cache line θα πραγματοποιείται cache miss και θα φορτώνεται μία ολόκληρη cache line στην cache. Όπως φαίνεται από το τμήμα του κώδικα που εξετάζουμε, γίνεται προσπέλαση σε στοιχεία μονοδιάστατων πινάκων αποθηκευμένα σειριακά στη μνήμη με βήμα 4. Κάθε cache line περιέχει:

$$\frac{\mu\acute{\varepsilon}\gamma\varepsilon\theta o\varsigma\ cache\ line}{\mu\acute{\varepsilon}\gamma\varepsilon\theta o\varsigma\ \lambda\acute{\varepsilon}\xi\eta\varsigma} = \frac{32bytes}{4bytes/word} = 8words$$

Άρα, κατά την εκτέλεση του κώδικα προσπελαύνονται $\frac{8}{4} = 2$ στοιχεία από κάθε cache line, και επομένως πραγματοποιείται 1 miss (κατά την πρώτη αναφορά σε μία cache line), και στη συνέχεια 1 hit. (Για δική μας διευκόλυνση μπορούμε να θεωρήσουμε ότι δεν πραγματοποιείται σύγκρουση δεδομένων στην cache μεταξύ στοιχείων διαφορετικών πινάκων.) Η ακολουθία αυτή 1miss-1hit, επαναλαμβάνεται 3 φορές, έως ότου ενεργοποιηθεί ο μηχανισμός πρώιμης ανάκλησης.

Κατά τη διάρκεια της 5ης επανάληψης του βρόχου, όπου πραγματοποιείται το 3ο miss, ενεργοποιείται ο μηχανισμός πρώιμης ανάκλησης, αμέσως μετά τον πρώτο κύκλο ανάγνωσης (m) για κάθε έναν από τους δύο πίνακες. Παρατηρούμε ότι τα τρία διαδοχικά misses προκαλούνται από τις αναφορές στα στοιχεία A[0], A[8], A[16]. Δηλαδή έχουν σταθερό βήμα = 8 στοιχεία (32 bytes). Επομένως και ο μηχανισμός πρώιμης ανάκλησης θα φέρνει στην cache τις cache lines που περιέχουν τα επόμενα στοιχεία βήματος 8: A[24], A[32], κλπ. (Η διαδικασία της πρώιμης ανάκλησης λειτουργεί όπως ακριβώς και μία εντολή load με τη διαφορά ότι καλείται από το hardware και προκειμένου να μην δημιουργούνται καθυστερήσεις στην εκτέλεση του προγράμματος έχει μειωμένη προτεραιότητα έναντι των αναγνώσεων από τη μνήμη που προκαλούνται από εντολές του προγράμματος.)





Όπως φαίνεται στον παραπάνω πίνακα, στην περίπτωση μίας επανάληψης του βρόχου όπου οι αναφορές στη μνήμη είναι misses, η διάρκεια εκτέλεσης είναι 14 κύκλοι. Στην περίπτωση που οι αναφορές στη μνήμη είναι hits, η διάρκεια εκτέλεσης είναι 8 κύκλοι. Από την $5^{η}$ επανάληψη και μετά, όλες οι αναφορές στη μνήμη είναι hits, αφού ο μηχανισμός πρώιμης ανάκλησης φέρνει έγκαιρα σε όλες τις περιπτώσεις τα δεδομένα στην cache. Άρα, συνολικά, έχουμε 3 misses και (250 – 3) = 247 hits

Συνολικά: $14 \times 3 + 8 \times 247 = 2018$ κύκλοι εκτέλεσης

Θέμα $2^{ο}$ (35%):

Έστω ότι οι εντολές αναφοράς στη μνήμη (`load, store`) δεν επιτρέπεται να περιέχουν σταθερό όρισμα (offset). Δηλαδή έχουν τη μορφή, π.χ:

```
lw   $s1, ($s2)
```

Για τη λειτουργία: `$s1 = Mem[$s2];`

(i) Πώς επηρεάζει η συγκεκριμένη τροποποίηση του συνόλου εντολών (ISA) την υλοποίηση της αρχιτεκτονικής αγωγού; Θα μπορούσε να μειωθεί το βάθος της σωλήνωσης και πώς; Τι συνέπειες θα είχαν οι όποιες αλλαγές σε throughput και latency;

(ii) Καταγράψτε τις περιπτώσεις όπου ενδέχεται να εμφανισθούν κίνδυνοι δεδομένων καθώς και τις αντίστοιχες αναγκαίες συνθήκες προώθησης δεδομένων (forwarding); Υπενθύμιση: εξαρτήσεις δεδομένων μεταξύ εντολών αριθμητικών/λογικών με αριθμητικές/λογικές καθώς και μεταξύ lw/sw και αριθμητικών/λογικών. Σε ότι αφορά τη δεύτερη περίπτωση εξαρτήσεων, υπάρχει κάποια βελτίωση σε σχέση με την κλασσική σωλήνωση 5 σταδίων του MIPS;

Λύση $2^{ου}$ Θέματος

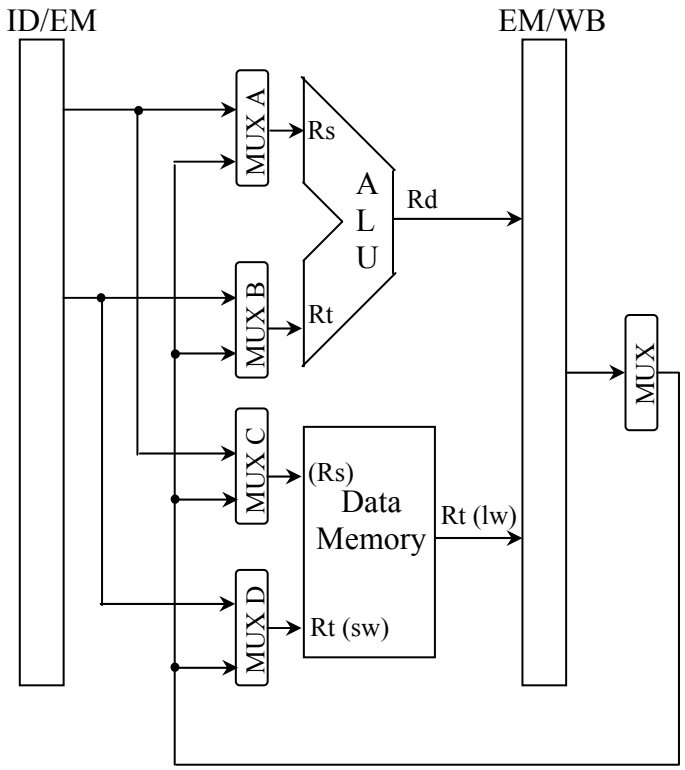
(i) Δεδομένου ότι οι εντολές αναφοράς στη μνήμη δεν χρειάζεται πλέον να πραγματοποιήσουν υπολογισμό της διεύθυνσης ανάγνωσης / εγγραφής, το στάδιο EX της κλασσικής σωλήνωσης 5 σταδίων δεν χρησιμοποιείται για τις εντολές αυτές. Εξάλλου, ως γνωστόν, οι αριθμητικές-λογικές εντολές δεν χρησιμοποιούν το στάδιο MEM. Επομένως, μπορούμε να ενοποιήσουμε τα δύο στάδια σε ένα, (έστω EM), μειώνοντας το βάθος της σωλήνωσης. Αν πρόκειται για αριθμητική-λογική εντολή θα χρησιμοποιείται η ALU, ενώ αν πρόκειται για εντολή αναφοράς στη μνήμη θα προσπελαύνεται η μνήμη. (Οι εντολές άλματος μπορούν να ολοκληρωθούν στα τρία πρώτα στάδια –IF-ID-EM–, όπως στην κλασσική περίπτωση, χρησιμοποιώντας στο $3^{ο}$ στάδιο την ALU.)
Με την ενοποίηση αυτή, μειώνεται το latency, αφού πλέον η πρώτη εντολή βγαίνει από τη σωλήνωση σε 4 κύκλους.

Το throughput στην ιδανική περίπτωση που δεν υπάρχουν καθυστερήσεις παραμένει σταθερό και ίσο με 1 εντολή/κύκλο ρολογιού. Από την άλλη μεριά, αν χρειαζόμαστε εντολές τύπου `lw $s1, α($s2)`, με τη νέα σωλήνωση θα χρειαστούμε πλέον δύο εντολές:

```
addi $s2, $s2, α
lw   $s1, ($s2)
```

Δηλαδή, αυξάνεται ο αριθμός των εντολών του προγράμματος, μειώνοντας την επίδοση. Ωστόσο, όπως θα δούμε στο ερώτημα (ii), η πολυπλοκότητα της σωλήνωσης μειώνεται, μειώνοντας ταυτόχρονα και τους κινδύνους εξαρτήσεων δεδομένων.

(ii) Αναλυτικά, οι κίνδυνοι δεδομένων που η επίλυσή τους απαιτεί προώθηση δεδομένων είναι οι εξής:
(Κατασκευάζουμε ένα απλουστευμένο διάγραμμα μόνο για τα στάδια EM-WB, ώστε να φανούν τα σημεία προώθησης των δεδομένων)



(α) Μεταξύ εντολών αριθμητικών/λογικών:

```
π.χ. add   $s3, $s2, $s1        IF – ID – EM - WB
     add   $s5, $s3, $s4             IF  – ID  – EM - WB

If EM/WB.RegWrite and                  // μία a/l στο στάδιο EM/WB
   ID/EM.ALUop and                     // μία a/l στο στάδιο ID/EM
   (EM/WB.RegRd ≠0)
   ((EM/WB.RegRd = ID/EM.RegRs)        // η επόμενη εντολή a/l χρησιμοποιεί
then forwardA=10                       // τον RegRd της πρώτης

ή  π.χ. add   $s3, $s2, $s1        IF – ID – EM - WB
       add   $s5, $s4, $s3             IF  – ID  – EM - WB
```

```
If EM/WB.RegWrite and                          // μία a/l στο στάδιο EM/WB
    ID/EM.ALUop and                            // μία a/l στο στάδιο ID/EM
    (EM/WB.RegRd ≠0)
    (EM/WB.RegRd = ID/EM.RegRt))               // η επόμενη εντολή a/l χρησιμοποιεί
then forwardB=10                               // τον RegRd της πρώτης
```

(β) Μεταξύ εντολών αριθμητικών/λογικών με εντολές αναφοράς στη μνήμη:

```
   π.χ. add    $s3, $s2, $s1          IF – ID – EM - WB
        lw     $s5, ($s3)                  IF  – ID  – EM - WB
```

```
If EM/WB.RegWrite and                          // μία a/l στο στάδιο EM/WB
    (ID/EM.MemWrite or ID/EM.MemRead) and      // μία sw/lw στο στάδιο ID/EM
    (EM/WB.RegRd ≠0)
    ((EM/WB.RegRd = ID/EM.RegRs)               // η επόμενη εντολή lw/sw χρησιμοποιεί
then forwardC=10                               // τον RegRd της πρώτης για υπολογισμό
                                                  διεύθυνσης
```

και μόνο για εντολές αποθήκευσης:

```
   π.χ. add    $s3, $s2, $s1          IF – ID – EM - WB
        sw     $s3, ($s4)                  IF  – ID  – EM - WB
```

```
If EM/WB.RegWrite and                          // μία a/l στο στάδιο EM/WB
    ID/EM.MemWrite and                         // μία sw στο στάδιο ID/EM
    (EM/WB.RegRd ≠0)
    (EM/WB.RegRd = ID/EM.RegRt))               // η επόμενη εντολή sw χρησιμοποιεί
then forwardD=10                               // τον RegRd της πρώτης ως περιεχόμενο
                                                  προς αποθήκευση στη μνήμη
```

(γ) Μεταξύ εντολών αναφοράς στη μνήμη με αριθμητικές/λογικές εντολές:

```
      π.χ. lw   $s2, ($s4)          IF – ID – EM - WB
           add  $s3, $s2, $s1             IF –  ID  – EM - WB
```

```
If EM/WB.MemRead and                           // μία lw στο στάδιο EM/WB
    ID/EM.ALUop and                            // μία a/l στο στάδιο ID/EM
    (EM/WB.RegRt ≠0)
    ((EM/WB.RegRt = ID/EM.RegRs)               // η επόμενη εντολή a/l χρησιμοποιεί
then forwardA=10                               // τον RegRd της πρώτης
```

```
και   π.χ. lw   $s2, ($s4)          IF – ID – EM - WB
           add  $s3, $s1, $s2             IF –  ID  – EM - WB
```

```
If EM/WB.MemRead and                           // μία lw στο στάδιο EM/WB
    ID/EM.ALUop and                            // μία a/l στο στάδιο ID/EM
    (EM/WB.RegRt ≠0)
    ((EM/WB.RegRt = ID/EM.RegRt)               // η επόμενη εντολή a/l χρησιμοποιεί
then forwardB=10                               // τον RegRd της πρώτης
```

Στην περίπτωση αυτή υπάρχει σαφής βελτίωση σε σχέση με την κλασσική αρχιτεκτονική σωλήνωσης 5 σταδίων. Στην αρχιτεκτονική 5 σταδίων μετά από μία εντολή lw που ακολουθείται από εντολή που χρησιμοποιεί το δεδομένο που φορτώθηκε από τη μνήμη, απαιτείται stall.

Θέμα 3^{ο} (35%):

Θεωρούμε της εξής ακολουθία εντολών. Αρχικά, ισχύει: `$t4` = 16 + `$r1`;

```
loop:     lw   $t1, 0($r1)    #1
          lw   $t2, 0($r2)    #2
          lw   $t3, 0($r3)    #3
          add  $t1, $t1, $t2  #4
          add  $t1, $t1, $t3  #5
          sll  $t1, $t1, 2    #6
          sw   $t1, 0($r4)    #7
          addi $r1, $r1, 4    #8
          addi $r2, $r2, 4    #9
          addi $r3, $r3, 4    #10
          bne  $r1, $t4, loop #11
exit:
```

i) Υποθέτουμε ότι διαθέτουμε έναν επεξεργαστή με αρχιτεκτονική σωλήνωσης 5 σταδίων, όπου υπάρχουν όλα τα δυνατά σχήματα προώθησης. Δείξτε το διάγραμμα εκτέλεσης των επιμέρους φάσεων για κάθε εντολή (κατά την πρώτη σειρά επαναλήψεων), στην περίπτωση που οι εντολές εκτελούνται με τη σειρά που υπαγορεύει ο κώδικας της ρουτίνας. Σημειώστε όλες τις αναγκαίες προωθήσεις δεδομένων. Πόσους κύκλους χρειάζεται για να ολοκληρωθεί η εκτέλεση της ρουτίνας;

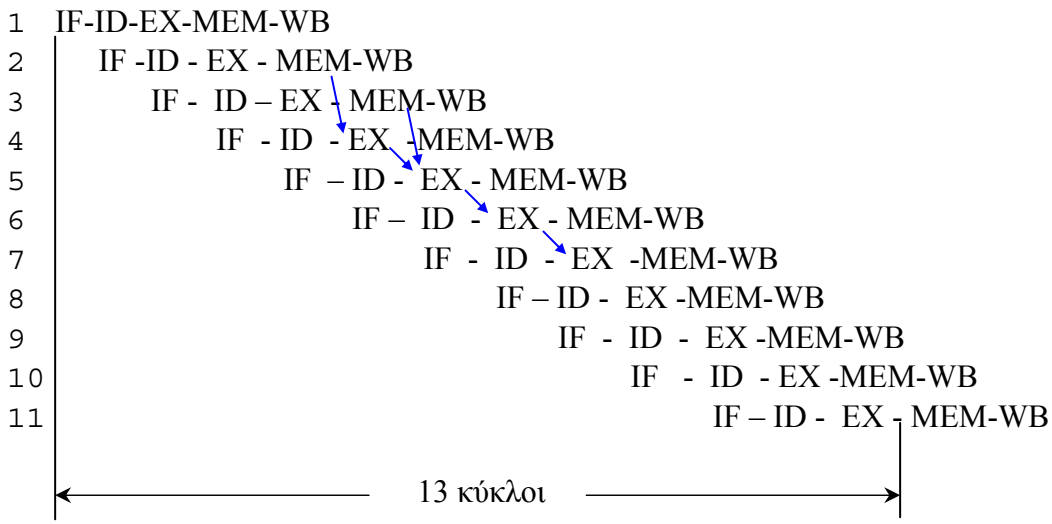
ii) Ο παραλληλισμός επιπέδου εντολής (instruction level parallelism - ILP) αναφέρεται στην έλλειψη εξαρτήσεων μεταξύ των εντολών ενός σειριακού προγράμματος και την εκμετάλλευση του γεγονότος για την παράλληλη εκτέλεσή τους, διατηρώντας την ορθότητα του τελικού αποτελέσματος. Για το σκοπό αυτό, στο εξής (για τα ερωτήματα α και β) υποθέτουμε ότι διαθέτουμε έναν επεξεργαστή με αρχιτεκτονική σωλήνωσης 5 σταδίων, πλάτους 2 σωληνώσεων. Επιπλέον, υπάρχουν όλα τα δυνατά σχήματα προώθησης.

α) Δείξτε το διάγραμμα εκτέλεσης των επιμέρους φάσεων για κάθε εντολή (κατά την πρώτη σειρά επαναλήψεων), στην περίπτωση που οι εντολές εκτελούνται με τη σειρά (in-order: αν οι διαδοχικές εντολές έχουν μεταξύ τους κάποιο είδος εξάρτησης δεδομένων, εισάγεται στη σωλήνωση μία μόνο εντολή/ κύκλο ρολογιού). Σημειώστε όλες τις αναγκαίες προωθήσεις δεδομένων. Πόσους κύκλους χρειάζεται για να ολοκληρωθεί η εκτέλεση της ρουτίνας;

β) Αν οι εντολές μπορούν να εκτελεστούν εκτός σειράς (out-of-order: αν οι διαδοχικές εντολές έχουν μεταξύ τους κάποιο είδος εξάρτησης δεδομένων, αναζητούνται ανεξάρτητες εντολές μεταξύ των επόμενων εντολών με στόχο να εισάγονται στη σωλήνωση 2 εντολές/ κύκλο ρολογιού), πόσους κύκλους κερδίζουμε; Κατασκευάστε το αντίστοιχο διάγραμμα εκτέλεσης και σημειώστε όλες τις αναγκαίες προωθήσεις δεδομένων.

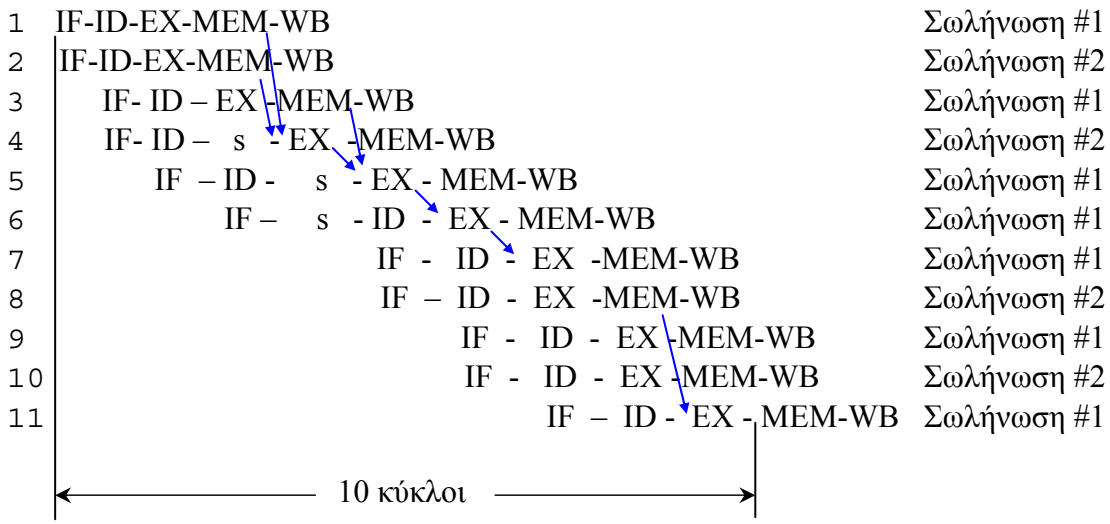
Λύση 3ου Θέματος

i) Σύμφωνα με την αρίθμηση των εντολών που προστέθηκε στην εκφώνηση, η εκτέλεση των εντολών στη σωλήνωση και η προώθηση δεδομένων σε αυτήν έχει ως εξής:



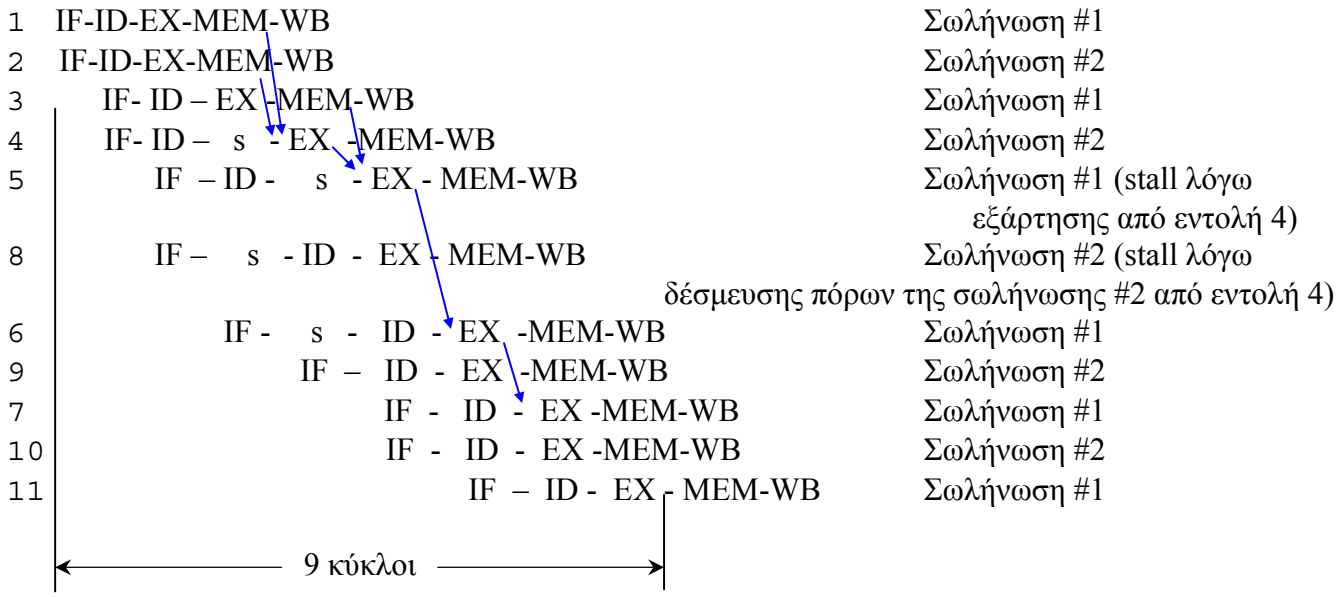
Ο βρόχος θα επαναληφθεί $\frac{16}{4} = 4$ φορές. Οπότε το σύνολο των κύκλων που απαιτούνται για την ολοκλήρωση της ρουτίνας είναι: 4×13 + 2 = 54.

ii-α) Σε κάθε κύκλο, οι εντολές εισέρχονται 2-2 στη σωλήνωση, εκτός αν υπάρχει εξάρτηση δεδομένων του ζεύγους:



Το σύνολο των κύκλων που απαιτούνται για την ολοκλήρωση της ρουτίνας είναι: 4×10 + 2 = 42.

ii-β) Σε κάθε κύκλο, οι εντολές εισέρχονται 2-2 στη σωλήνωση. Αν υπάρχει εξάρτηση δεδομένων σε ένα ζευγάρι εντολών, τότε αναζητούμε, μεταξύ των εντολών που ακολουθούν, μία ανεξάρτητη εντολή:



Το σύνολο των κύκλων που απαιτούνται για την ολοκλήρωση της ρουτίνας είναι: 4×9 + 2 = 38.