ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

www.cslab.ece.ntua.gr

ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ακ. έτος 2006-07

Άσκηση 1

Εξετάζουμε την εκτέλεση του εξής κώδικα:

```
Repeat:   ld r2,100(r3)
          sub r2,r2,r5
          st r2,100(r3)
          sub r3,r3,r6
          sub r1,r1,r7
          bnez r1,Repeat
Exit:
```

Υποθέτουμε ότι έχουμε αρχιτεκτονική σωλήνωσης (pipelining) 5 σταδίων (IF ID EX MEM WB). Έστω ότι η αρχική τιμή του r1 είναι 500 και του r7 είναι 5, και ότι όλες οι αναφορές στη μνήμη ικανοποιούνται από την κρυφή μνήμη σε 1 κύκλο (δεν υπάρχουν δηλαδή αστοχίες).

α) Αρχικά, υποθέτουμε ότι η αρχιτεκτονική σωλήνωσης δε διαθέτει σχήμα προώθησης (forwarding). Επίσης, η εγγραφή σε κάποιον καταχωρητή γίνεται στο πρώτο μισό ενός κύκλου, ενώ η ανάγνωση από τον ίδιον καταχωρητή στο δεύτερο μισό του ίδιου κύκλου. Επιπλέον, η απόφαση για μια διακλάδωση λαμβάνεται στο στάδιο MEM, και για να γίνει η διακλάδωση πρέπει να "καθαριστεί" (flush) το pipeline.

Για την 1η επανάληψη του παραπάνω βρόχου, μέχρι και το load της 2ης επανάληψης, χρησιμοποιείστε ένα διάγραμμα χρονισμού όπως αυτό που παρουσιάζεται στη συνέχεια, για να δείξετε τα διάφορα στάδια του pipeline από τα οποία διέρχονται οι εντολές σε αυτό το διάστημα εκτέλεσης. Υποδείξτε και εξηγείστε τους πιθανούς κινδύνους (hazards) που μπορούν να προκύψουν κατά την εκτέλεση, καθώς και τον τρόπο με τον οποίον αυτοί αντιμετωπίζονται.

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***...***
Εντολή 1	IF	ID	EX	MEM	WB		
Εντολή 2		IF	ID	EX	MEM	WB	
Εντολή 3			...	...	...	...	...
...							

Πόσοι κύκλοι απαιτούνται συνολικά για να ολοκληρωθεί ο παραπάνω βρόχος (για όλες τις επαναλήψεις του, όχι μόνο για την 1η);

Ο βρόχος θα εκτελεστεί για 500/5=100 επαναλήψεις.
Το διάγραμμα χρονισμού του pipeline για τη χρονική διάρκεια που ζητείται είναι το ακόλουθο:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***
ld r2,100(r3)	IF	ID	EX	MEM	WB															
sub r2,r2,r5		IF	ID	-	-	EX	MEM	WB												
st r2,100(r3)			IF	-	-	ID	-	-	EX	MEM	WB									
sub r3,r3,r6						IF	-	-	ID	EX	MEM	WB								
sub r1,r1,r7									IF	ID	EX	MEM	WB							
bnez r1, Repeat										IF	ID	-	-	EX	MEM	WB				
ld r2,100(r3)																IF	ID	EX	MEM	WB

Στο παραπάνω διάγραμμα, οι "-" υποδηλώνουν stalls. Τα stalls στους κύκλους 4,5 οφείλονται στο ότι ο r2 για την εντολή sub r2,r2,r5 (η ανάγνωση του οποίου γίνεται στο στάδιο ID) γίνεται διαθέσιμος στο τέλος του κύκλου 5 (στάδιο WB). Τα stalls στους κύκλους 7,8 οφείλονται πάλι στο ότι ο r2 για την εντολή st r2,100(r3) γίνεται διαθέσιμος στο τέλος του κύκλου 8. Τα stalls στους κύκλους 12,13 οφείλονται στο ότι ο r1 για την εντολή bnez r1,Repeat γίνεται διαθέσιμος στο τέλος του κύκλου 13. Το δεύτερο στιγμιότυπο της εντολής ld r2,100(r3) αρχίζει να εκτελείται από τον κύκλο 16, διότι η απόφαση για την διακλάδωση ελήφθη στον κύκλο 15.

Όπως φαίνεται από το παραπάνω διάγραμμα, ο πρώτος κύκλος για την επανάληψη i επικαλύπτεται με τον τελευταίο κύκλο για την επανάληψη i-1. Επομένως, λαμβάνοντας υπόψη αυτή την επικάλυψη, οι επαναλήψεις για i=1...99 διαρκούν συνολικά 15 κύκλους η κάθε μία, ή 15*99=1485 κύκλους συνολικά. Η τελευταία επανάληψη i=100, διαρκεί 16 κύκλους. Επομένως, απαιτούνται 1485+16 = 1501 κύκλου για την εκτέλεση του βρόχου.

β) Για την ίδια ακολουθία εντολών, δείξτε και εξηγείστε όπως και πριν, τον χρονισμό του pipeline, θεωρώντας όμως τώρα ότι υπάρχει σχήμα προώθησης. Θεωρείστε ότι οι αποφάσεις για τις διακλαδώσεις λαμβάνονται στο στάδιο MEM. Πόσοι κύκλοι απαιτούνται συνολικά για να ολοκληρωθεί ο βρόχος;

Το διάγραμμα χρονισμού του pipeline για τη χρονική διάρκεια που ζητείται είναι το ακόλουθο:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***
ld r2,100(r3)	IF	ID	EX	MEM	WB															
sub r2,r2,r5		IF	ID	-	EX	MEM	WB													
st r2,100(r3)			IF	-	ID	EX	MEM	WB												
sub r3,r3,r6					IF	ID	EX	MEM	WB											
sub r1,r1,r7						IF	ID	EX	MEM	WB										
bnez r1, Repeat							IF	ID	EX	MEM	WB									
ld r2,100(r3)											IF	ID	EX	MEM	WB					

Στον κύκλο 4 υπάρχει stall, διότι η τιμή της θέσης μνήμης 100(r3), που πρόκειται να αποθηκευτεί στον r2, δεν μπορεί να είναι διαθέσιμη πριν το στάδιο MEM. Όταν όμως θα γίνει διαθέσιμη στο τέλος του κύκλου αυτού (και αποθηκευτεί στον ενδιάμεσο καταχωρητή MEM/WB του pipeline), τότε το σχήμα προώθησης θα την προωθήσει κατευθείαν στις εισόδους της ALU, ώστε να μπορεί να εκτελεστεί η εντολή sub r2,r2,r5 χωρίς να χρειάζεται να περιμένουμε μέχρι η τιμή αυτή γραφτεί στον r2. Με αυτόν τον τρόπο, αποφεύγεται το stall που είχαμε στην προηγούμενη περίπτωση στην κύκλο 5.
Το ίδιο ισχύει και για τα stalls που είχαμε στους κύκλους 7,8. Η τιμή του r2, που χρειάζεται από την εντολή st r2,100(r3) , παράγεται στον 5ο κύκλο της εντολής sub r2,r2,r5 κατά το στάδιο EX και αποθηκεύεται στον

ενδιάμεσο καταχωρητή EX/MEM. Επομένως μπορεί να προωθηθεί στις εισόδους της ALU και να αποφευχθούν τα stalls.
Για τον ίδιο λόγο αποφεύγονται και τα stalls στους κύκλους 12,13. Η τιμή του r1 γίνεται διαθέσιμη στο στάδιο EX της εντολής sub r1,r1,r7 , αποθηκεύεται στον ενδιάμεσο καταχωρητή EX/MEM οπότε μπορεί να χρησιμοποιηθεί άμεσα για την εντολή διακλάδωσης που ακολουθεί.

Με ανάλογο σκεπτικό όπως και πριν, οι κύκλοι που απαιτούνται για την εκτέλεση του βρόχου είναι 10*99+11=1001 .

Άσκηση 2

Εξετάζουμε το pipeline για μια αρχιτεκτονική συνόλου εντολών (ISA) καταχωρητή-μνήμης (register-memory). Στην αρχιτεκτονική αυτή υπάρχουν 2 μορφές εντολών: καταχωρητή-καταχωρητή (register-register), και καταχωρητή-μνήμης (register-memory). Στην τελευταία περίπτωση, ένας από τους τελεστές για μία πράξη που εκτελείται στην ALU μπορεί να προέρχεται από τη μνήμη. Για τις αναφορές στη μνήμη, υπάρχει ένας μόνο τρόπος διευθυνσιοδότησης: offset(base_reg) (δηλ. Mem[offset + $base_reg]).
Όλες οι διαθέσιμες εντολές καταχωρητή-μνήμης (που δεν είναι εντολές διακλάδωσης) έχουν τη μορφή:

Operation T, S1, S2
ή
Operation T, S1, Mem

όπου "Operation" είναι κάποια πράξη από τις: add, sub, and, or, ld (σε αυτή την περίπτωση ο S1 αγνοείται), st. Οι T (target), S1 (source 1), S2 (source 2) είναι καταχωρητές, ενώ Mem είναι μια αναφορά στη μνήμη με τον τρόπο διευθυνσιοδότησης που αναφέραμε. Για τις εντολές διακλάδωσης υπό συνθήκη, συγκρίνονται δύο καταχωρητές, και ανάλογα με το αποτέλεσμα της σύγκρισης, γίνεται άλμα στην διεύθυνση-στόχο που υποδεικνύεται. Η διεύθυνση-στόχος, μπορεί να προσδιορίζεται είτε από μία ακέραια σταθερά (που υποδηλώνει το offset σε σχέση με την τρέχουσα τιμή του μετρητή προγράμματος), είτε από έναν καταχωρητή (το περιεχόμενο του οποίου αντιστοιχεί στην απόλυτη διεύθυνση-στόχο).
Υποθέτουμε ότι τα στάδια της σωλήνωσης της αρχιτεκτονικής είναι τα εξής:

IF ID AGU MEM ALU WB

Στο στάδιο AGU (address generation unit) γίνεται ο υπολογισμός των τελικών διευθύνσεων μνήμης, για εντολές αναφοράς στη μνήμη και για εντολές διακλάδωσης. Στο στάδιο ALU γίνεται η εκτέλεση των αριθμητικών πράξεων καθώς και η σύγκριση για εντολές διακλάδωσης υπό συνθήκη. Η εγγραφή σε κάποιον καταχωρητή γίνεται στο πρώτο μισό ενός κύκλου, ενώ η ανάγνωση από τον ίδιον καταχωρητή στο δεύτερο μισό του ίδιου κύκλου.

α) Βρείτε τον αριθμό των αθροιστών που απαιτούνται για την ελαχιστοποίηση των δομικών κινδύνων (structural hazards). Δικαιολογείστε την επιλογή σας αυτή.

Στο pipeline που περιγράφει η άσκηση, οι δομικοί κίνδυνοι που μπορούν να προκύψουν οφείλονται σε "συγκρούσεις" εξαιτίας της χρήσης της ίδιας δομικής μονάδας στον ίδιο κύκλο από διαφορετικά στάδια του pipeline. Τα στάδια που χρησιμοποιούν πράξεις πρόσθεσης, στην περίπτωσή μας, είναι 3: το IF (που προσαυξάνει την τρέχουσα τιμή του PC ώστε να δείχνει στην επόμενη κατά σειρά εντολή), το AGU (που εκτελεί την πρόσθεση offset + $base_reg για να παράξει, είτε την διεύθυνση για μία αναφορά μνήμης, είτε τη διεύθυνση μιας εντολής διακλάδωσης), το ALU (όπου εκτελούνται μεταξύ άλλων πράξεις add και sub). Για να ελαχιστοποιηθούν επομένως οι δομικοί κίνδυνοι, απαιτούνται 3 αθροιστές.

β) Βρείτε τον αριθμό των ports για ανάγνωση και εγγραφή των καταχωρητών, και των ports για ανάγνωση και εγγραφή μνήμης, ώστε να ελαχιστοποιούνται οι δομικοί κίνδυνοι. Δικαιολογείστε αντίστοιχα.

Καταχωρητές:
το αρχείο καταχωρητών χρησιμοποιείται σε δύο στάδια, το ID και το WB.
Στο ID, διαβάζονται οι source registers μιας εντολής. Για το ISA που εξετάζει η άσκηση, ο μέγιστος αριθμός source registers που μπορούν να προσδιοριστούν σε μια εντολή είναι 3: breq r1,r2,10(r3) , για παράδειγμα.
Στο WB, γράφεται κάποιος target register.
Επομένως, για το αρχείο καταχωρητών, απαιτούνται 3 read ports και 1 write port.

Μνήμη:
η προσπέλαση της μνήμης γίνεται μόνο στο στάδιο MEM, και μπορεί να είναι είτε εγγραφή είτε ανάγνωση. Επομένως, απαιτείται 1 read port και 1 write port.

γ) Αν υπήρχε σχήμα προώθησης των αποτελεσμάτων από το στάδιο ALU σε οποιοδήποτε από τα στάδια AGU, MEM, ALU, θα μειώνονταν οι καθυστερήσεις (stalls); 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.

προώθηση ALU→AGU:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
add r1,r1,r2	IF	ID	AGU	MEM	ALU	WB						
instr1		IF	ID	AGU	MEM	ALU	WB					
instr2			IF	ID	AGU	MEM	ALU	WB				
ld r3,0(r1)				IF	ID	-	AGU	MEM	ALU	WB		
Με προώθηση:												
add r1,r1,r2	IF	ID	AGU	MEM	ALU	WB						
instr1		IF	ID	AGU	MEM	ALU	WB					
instr2			IF	ID	AGU	MEM	ALU	WB				
ld r3,0(r1)				IF	ID	AGU	MEM	ALU	WB			

Στο παραπάνω παράδειγμα, οι εντολές instr1 και instr2 είναι τέτοιες ώστε να μην δημιουργούν κινδύνους στο pipeline και επομένως να μην προκαλούν καθυστερήσεις. Στον 5ο κύκλο, το σχήμα προώθησης ανιχνεύει ότι η εντολή load χρειάζεται τον source register r1, και "θυμάται" ότι η τιμή του r1 θα είναι στο τέλος του κύκλου αυτού διαθέσιμη, από την εντολή add. Έτσι, αντί να εμποδίσει την εντολή load (και όλες τις επόμενές της) να προχωρήσει στο pipeline, μέχρι ο r1 να γραφτεί στο στάδιο WB, το σχήμα προώθησης δίνει το δικαίωμα στην εντολή load να συνεχίσει, και ταυτόχρονα προωθεί κατευθείαν την τιμή του r1 από τον ενδιάμεσο καταχωρητή ALU/WB στις εισόδους της AGU.

προώθηση ALU→MEM:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
add r1,r1,r2	IF	ID	AGU	MEM	ALU	WB						
instr1		IF	ID	AGU	MEM	ALU	WB					
st r1,0(r3)			IF	ID	-	-	AGU	MEM	ALU	WB		
Με προώθηση:												
add r1,r1,r2	IF	ID	AGU	MEM	ALU	WB						
instr1		IF	ID	AGU	MEM	ALU	WB					
st r1,0(r3)			IF	ID	AGU	MEM	ALU	WB				

Στο παραπάνω παράδειγμα, οι εντολή instr1 είναι τέτοια ώστε να μην δημιουργεί κινδύνους στο pipeline. Στον 4ο κύκλο, κατά τον οποίον η εντολή store διαβάζει τον source register r1, το σχήμα προώθησης "θυμάται" ότι

η τιμή αυτή θα γίνει διαθέσιμη στο τέλος του 5ου κύκλου, από την εντολή add. Επιπλέον, επειδή ξέρει ότι η εντολή store δε θα χρησιμοποιήσει την τιμή του r1 κατά τη διάρκεια του 5ου κύκλου, της δίνει το δικαίωμα να συνεχίσει και να μην καθυστερήσει στον κύκλο αυτό. Στο τέλος του 5ου κύκλου, το σχήμα προώθησης προωθεί την τιμή του r1, από τον ενδιάμεσο καταχωρητή ALU/WB στις εισόδους της μονάδας MEM, και έτσι αποφεύγεται και η καθυστέρηση και στον κύκλο 6.

προώθηση ALU→ALU:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
add r1,r1,r2	IF	ID	AGU	MEM	ALU	WB						
sub r3,r1,r2		IF	ID	-	-	-	AGU	MEM	ALU	WB		
Με προώθηση:												
add r1,r1,r2	IF	ID	AGU	MEM	ALU	WB						
sub r3,r1,r2		IF	ID	AGU	MEM	ALU	WB					

Στον 3ο κύκλο, κατά τον οποίον η εντολή sub διαβάζει τον source register r1, το σχήμα προώθησης "θυμάται" ότι η τιμή αυτή θα γίνει διαθέσιμη στο τέλος του 5ου κύκλου, από την εντολή add. Επιπλέον, επειδή ξέρει ότι η εντολή sub δε θα χρησιμοποιήσει την τιμή του r1 κατά τη διάρκεια του 4ου και 5ου κύκλου, της δίνει το δικαίωμα να συνεχίσει και να μην καθυστερήσει στους κύκλους αυτούς. Στο τέλος του 5ου κύκλου, το σχήμα προώθησης προωθεί την τιμή του r1, από τον ενδιάμεσο καταχωρητή ALU/WB στις εισόδους της ALU, και έτσι αποφεύγεται και η καθυστέρηση και στον κύκλο 6.

δ) Αν υπήρχε σχήμα προώθησης των αποτελεσμάτων από το στάδιο MEM σε οποιοδήποτε από τα στάδια AGU, MEM, ALU, 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.

προώθηση MEM→AGU:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
ld r1,0(r2)	IF	ID	AGU	MEM	ALU	WB						
instr1		IF	ID	AGU	MEM	ALU	WB					
ld r3,0(r1)			IF	ID	-	-	AGU	MEM	ALU	WB		
Με προώθηση:												
ld r1,0(r2)	IF	ID	AGU	MEM	ALU	WB						
instr1		IF	ID	AGU	MEM	ALU	WB					
ld r3,0(r1)			IF	ID	AGU	MEM	ALU	WB				

Στο παραπάνω παράδειγμα, οι εντολή instr1 είναι τέτοια ώστε να μην δημιουργεί κινδύνους στο pipeline. Στον 4ο κύκλο, κατά τον οποίον η δεύτερη εντολή load διαβάζει τον source register r1, το σχήμα προώθησης "θυμάται" ότι η τιμή αυτή θα γίνει διαθέσιμη στο τέλος του 4ου κύκλου, από την πρώτη εντολή load. Στο τέλος του 4ου κύκλου, το σχήμα προώθησης προωθεί την τιμή που πρόκειται να αποθηκευτεί στον r1 (0(r2)), από τον ενδιάμεσο καταχωρητή MEM/ALU στις εισόδους της μονάδας AGU, και έτσι αποφεύγονται οι

καθυστερήσεις στους κύκλους 5 και 6.

προώθηση MEM→MEM:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
ld r1,0(r2)	IF	ID	AGU	MEM	ALU	WB						
st r1,0(r4)		IF	ID	-	-	-	AGU	MEM	ALU	WB		
Με προώθηση:												
ld r1,0(r2)	IF	ID	AGU	MEM	ALU	WB						
st r1,0(r4)		IF	ID	AGU	MEM	ALU	WB					

Στον 3ο κύκλο, κατά τον οποίον η εντολή store διαβάζει τον source register r1, το σχήμα προώθησης "θυμάται" ότι η τιμή που θα αποθηκευτεί στον r1 (0(r2)) θα γίνει διαθέσιμη στο τέλος του 4ου κύκλου, από την εντολή load. Επιπλέον, επειδή ξέρει ότι η εντολή store δε θα χρησιμοποιήσει την τιμή του r1 κατά τη διάρκεια του 4ου κύκλου, της δίνει το δικαίωμα να συνεχίσει και να μην καθυστερήσει στον κύκλο αυτό. Στο τέλος του 4ου κύκλου, το σχήμα προώθησης προωθεί την τιμή που πρόκειται να αποθηκευτεί στον r1, από τον ενδιάμεσο καταχωρητή MEM/ALU στις εισόδους της μονάδας MEM, και έτσι αποφεύγονται οι καθυστερήσεις.

προώθηση MEM→ALU:

Εφόσον το στάδιο MEM βρίσκεται πριν το στάδιο ALU στο pipeline, δε χρειάζεται να γίνει κάποιο είδος προώθησης. Στην ουσία, τα στάδια αυτά είναι συναπτά, επομένως η προώθηση, τυπικά συμβαίνει σε κάθε κύκλο, εκ κατασκευής του pipeline (το περιεχόμενο του καταχωρητή MEM/ALU "προωθείται", ούτως ή άλλως, στις εισόδους της ALU).

Άσκηση 3

Υποθέτουμε ότι έχουμε αρχιτεκτονική σωλήνωσης (pipelining) αποτελούμενη από τα εξής στάδια: IF ID1 ID2 MEM ALU WB. Τα στάδια IF, MEM, ALU και WB έχουν την ίδια λειτουργικότητα με αυτά της κλασικής σωλήνωσης του MIPS. Το ID1 χρησιμοποιείται για την ανάγνωση καταχωρητών, ενώ το ID2 χρησιμοποιείται για τον υπολογισμό τελικών διευθύνσεων μνήμης, για τον υπολογισμό διευθύνσεων-στόχων σε εντολές διακλάδωσης, καθώς και για τον έλεγχο συνθήκης σε εντολές διακλάδωσης υπό συνθήκη. Επιπλέον, οι αριθμητικές εντολές μπορούν να προσπελάσουν απευθείας μια θέση μνήμης. Όλα τα στάδια διαρκούν 1 κύκλο ρολογιού.

α) Βρείτε όλες τις εξαρτήσεις στο ακόλουθο κομμάτι κώδικα καθώς και την κατηγορία στην οποία ανήκουν (true dependence, output dependence, anti-dependence, control dependence).

```
        Loop:
(1)     ld r1,50(r2)
(2)     add r3,r1,100(r4)
(3)     mul r6,r5,r3
(4)     st r6,50(r2)
(5)     add r4,r4,#100
(6)     sub r2,r2,#8
(7)     bnez r2,Loop
(8)     sub r1,r1,#100
```

True dependence	Output dependence	Anti-dependence	Control dependence
2 από την 1 (στον r1) 3 από την 2 (στον r3) 4 από την 3 (στον r6) 7 από την 6 (στον r2) 8 από την 1 (στον r1)	8 από την 1 (στον r1)	5 από την 2 (στον r4) 6 από την 1 (στον r2) 6 από την 4 (στον r2) 8 από την 2 (στον r1)	1,2,3,4,5,6,8 από την 7

β) Υποθέστε ότι δεν υπάρχει σχήμα προώθησης. Πόσοι κύκλοι απαιτούνται για την εκτέλεση μιας επανάληψης του loop;

Κύκλος	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
ld r1,50(r2)	IF	ID1	ID2	M	AL	WB																
add r3,r1,100(r4)		IF	ID1	-	-	-	ID2	M	AL	WB												
mul r6,r5,r3			IF	-	-	-	ID1	-	-	-	ID2	M	AL	WB								
st r6,50(r2)							IF	-	-	-	ID1	-	-	-	ID2	M	AL	WB				
add r4,r4,#100											IF	-	-	-	ID1	ID2	M	AL	WB			
sub r2,r2,#8															IF	ID1	ID2	M	AL	WB		
bnez r2,Loop																IF	ID1	-	-	-	ID2	M
ld r1,50(r2)																	-	-	-	-	-	IF

Τα stalls στους κύκλους 4-6 οφείλονται στο ότι ο r1 για την εντολή add r3,r1,100(r4) (η ανάγνωση του οποίου γίνεται στο στάδιο ID1) γίνεται διαθέσιμος στο τέλος του κύκλου 6 (στάδιο WB). Τα stalls στους κύκλους 8-10 οφείλονται στο ότι ο r3 για την εντολή mul r6,r5,r3 γίνεται διαθέσιμος στο τέλος του κύκλου 10. Τα stalls στους κύκλους 12-14 οφείλονται στο ότι ο r6 για την εντολή st r6,50(r2) γίνεται διαθέσιμος στο τέλος του κύκλου 14. Τα stalls στους κύκλους 18-20 οφείλονται στο ότι ο r2 για την εντολή bnez r2,Loop γίνεται διαθέσιμος στο τέλος του κύκλου 20. Το δεύτερο στιγμιότυπο της εντολής ld αρχίζει να εκτελείται από τον

κύκλο 22, διότι η απόφαση για την διακλάδωση λαμβάνεται στο προηγούμενο κύκλο (στάδιο ID2). Συνολικά απαιτούνται 21 κύκλοι για την εκτέλεση μιας επανάληψης.

γ) Υποθέστε τώρα ότι υπάρχουν όλα τα δυνατά σχήματα προώθησης. Πόσοι κύκλοι απαιτούνται για την εκτέλεση μιας επανάληψης του loop;

Κύκλος	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
ld r1,50(r2)	IF	ID1	ID2	M	AL	WB																
add r3,r1,100(r4)		IF	ID1	ID2	M	AL	WB															
mul r6,r5,r3			IF	ID1	ID2	M	AL	WB														
st r6,50(r2)				IF	ID1	ID2	-	M	AL	WB												
add r4,r4,#100					IF	ID1	-	ID2	M	AL	WB											
sub r2,r2,#8						IF	-	ID1	ID2	M	AL	WB										
bnez r2,Loop								IF	ID1	-	-	ID2	M	AL	WB							
ld r1,50(r2)									-	-	-	-	IF	ID1	ID2	M	AL	WB				

Στο παραπάνω διάγραμμα, τα κανονικά βέλη δείχνουν τις προωθήσεις που γίνονται, ενώ τα διακεκομμένα δείχνουν τα στάδια όπου χρησιμοποιούνται οι προωθούμενες τιμές.
Στην πρώτη προώθηση, η τιμή που θα αποθηκευτεί στον r1 (και η οποία γίνεται διαθέσιμη στο τέλος του 4^{ου} κύκλου) προωθείται στις εισόδους του ίδιου σταδίου (MEM), έτσι ώστε να προχωρήσει στο pipeline και να χρησιμοποιηθεί όταν ακριβώς την χρειάζεται η add, δηλαδή στον μεθεπόμενο κύκλο, στο στάδιο ALU.
Στη δεύτερη περίπτωση προώθησης, ο r3 προωθείται από την έξοδο της ALU σε κάποια από τις εισόδους της, ώστε να χρησιμοποιηθεί στον επόμενο κύκλο από την mul.
Στην τρίτη περίπτωση, η τιμή του r6 προωθείται αμέσως μόλις γίνει διαθέσιμη (τέλος 7^{ου} κύκλου) στις εισόδους του σταδίου MEM. Δεν μπορεί να γίνει διαθέσιμη νωρίτερα, γι'αυτό και αναγκαστικά εισάγεται stall στον 7^{ο} κύκλο για την εντολή st r6,50(r2) καθώς και όσες επόμενες βρίσκονται στο pipeline.
Ομοίως, στην τέταρτη περίπτωση, η τιμή του r2 (την οποία χρειάζεται η bnez στο στάδιο ID2) δεν μπορεί να γίνει διαθέσιμη πριν τον τέλος του 11^{ου} κύκλου (στάδιο ALU), γι' αυτό και εισάγονται 2 stalls για την bnez στους κύκλους 10 και 11. Μόλις η τιμή του r2 γίνεται διαθέσιμη, προωθείται από την έξοδο του σταδίου ALU στην είσοδο του ID2.
Συνολικά απαιτούνται 12 κύκλοι για την εκτέλεση μιας επανάληψης.

δ) Θεωρώντας την ίδια σωλήνωση με το ερώτημα γ, μπορείτε να επιτύχετε ακόμα καλύτερη επίδοση για την εκτέλεση μιας επανάληψης του loop;

Αρχικά, θα αναδιατάξουμε τις εντολές του κώδικα ώστε να αποφύγουμε τα stalls που εισάγονται λόγω των εξαρτήσεων δεδομένων. Μια αναδιάταξη των εντολών η οποία διατηρεί τη σημασιολογία του κώδικα (σωστή ροή εντολών) και η οποία αποφεύγει την ανάγκη εισαγωγής stalls είναι η ακόλουθη:

```
Loop:
ld r1,50(r2)
add r3,r1,100(r4)
sub r2,r2,#8
mul r6,r5,r3
add r4,r4,#100
st r6,58(r2)
bnez r2,Loop
```

Η λογική είναι να «μπλέξουμε» τις «ανεξάρτητες» εντολές sub r2,r2,#8 και add r4,r4,#100, με τις 4 πρώτες εντολές ld, add, mul, st (οι οποίες αποτελούν στην ουσία μια αλυσίδα από εξαρτήσεις), ώστε εκεί που είχαμε stalls λόγω εξαρτήσεων να παρεμβάλλεται η εκτέλεση των ανεξάρτητων αυτών εντολών. Η παραπάνω μίξη των εντολών είναι τέτοια ώστε να μην προκαλεί stalls στο pipeline λόγω εξαρτήσεων, όπως φαίνεται κι από το ακόλουθο διάγραμμα χρονισμού.

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***	***21***	***22***
ld r1,50(r2)	IF	ID1	ID2	M	AL	WB																
add r3,r1,100(r4)		IF	ID1	ID2	M	AL	WB															
sub r2,r2,#8			IF	ID1	ID2	M	AL	WB														
mul r6,r5,r3				IF	ID1	ID2	M	AL	WB													
add r4,r4,#100					IF	ID1	ID2	M	AL	WB												
st r6,58(r2)						IF	ID1	ID2	M	AL	WB											
bnez r2,Loop							IF	ID1	ID2	M	AL	WB										
ld r1,50(r2)								-	-	IF	ID1	ID2	M	AL	WB							

Τώρα απαιτούνται συνολικά 9 κύκλοι για την εκτέλεση μιας επανάληψης. Το ερώτημα που τίθεται είναι, μπορούμε να επιτύχουμε κάτι καλύτερο από αυτό; Με άλλα λόγια, για το δεδομένο pipeline, μπορούμε να εξαλείψουμε τα stalls στους κύκλους 8,9 που οφείλονται στο γεγονός ότι η απόφαση για μια διακλάδωση υπό συνθήκη λαμβάνεται στο στάδιο ID2;

Για να το επιτύχουμε αυτό, θα χρησιμοποιήσουμε την τεχνική του *delayed branch*. Η λογική αυτής της τεχνικής είναι να χρησιμοποιήσουμε τους κύκλους εκείνους κατά τους οποίους το pipeline «παγώνει» μέχρι να βγει η απόφαση για μια διακλάδωση («branch delay slots»), για να δρομολογήσουμε εντολές οι οποίες είναι σίγουρο ότι θα εκτελεστούν, είτε αν η διακλάδωση είναι TAKEN είτε είναι NOT TAKEN.
Στην περίπτωσή μας, έχουμε 2 branch delay slots, επομένως αυτό που κάνουμε είναι να εκτελέσουμε την εντολή διακλάδωσης 2 εντολές νωρίτερα. Ο νέος κώδικας έχει ως εξής:

```
Loop:
ld r1,50(r2)
add r3,r1,100(r4)
sub r2,r2,#8
mul r6,r5,r3
bnez r2,Loop
add r4,r4,#100
st r6,58(r2)
```

Το διάγραμμα χρονισμού είναι το ακόλουθο:

Κύκλος	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***	***21***	***22***
ld r1,50(r2)	IF	ID1	ID2	M	AL	WB																
add r3,r1,100(r4)		IF	ID1	ID2	M	AL	WB															
sub r2,r2,#8			IF	ID1	ID2	M	AL	WB														
mul r6,r5,r3				IF	ID1	ID2	M	AL	WB													
bnez r2,Loop					IF	ID1	-	ID2	M	AL	WB											
add r4,r4,#100						IF	-	ID1	ID2	M	AL	WB										
st r6,58(r2)								IF	ID1	ID2	M	AL	WB									
ld r1,50(r2)									IF	ID1	ID2	M	AL	WB								

Όπως βλέπουμε, τα stalls που είχαμε λόγω της εντολής διακλάδωσης εξαλείφθηκαν, ενώ οι κύκλοι για την εκτέλεση μιας επανάληψης μειώθηκαν στους 8. Όμως η πρώιμη εκτέλεση της bnez οδηγεί σε RAW κίνδυνο δεδομένων (εξαιτίας της εξάρτησής της από την sub r2,r2,#8) ο οποίος αντιμετωπίζεται με stall στον κύκλο 7. Για να αποφύγουμε το stall αυτό, ανεβάζουμε την (ούτως ή άλλως ανεξάρτητη) sub μια θέση παραπάνω απ'ό,τι ήταν. Ο νέος κώδικας είναι ο εξής:

```
Loop:
ld r1,50(r2)
sub r2,r2,#8
add r3,r1,100(r4)
mul r6,r5,r3
bnez r2,Loop
add r4,r4,#100
st r6,58(r2)
```

Το διάγραμμα χρονισμού είναι το ακόλουθο:

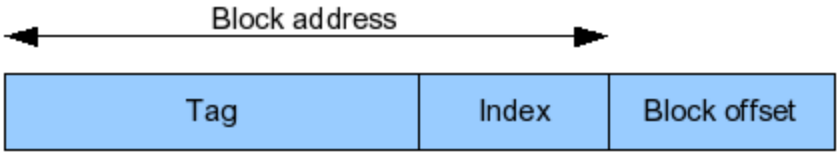
Κύκλος	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
ld r1,50(r2)	IF	ID1	ID2	M	AL	WB																
sub r2,r2,#8		IF	ID1	ID2	M	AL	WB															
add r3,r1,100(r4)			IF	ID1	ID2	M	AL	WB														
mul r6,r5,r3				IF	ID1	ID2	M	AL	WB													
bnez r2,Loop					IF	ID1	ID2	M	AL	WB												
add r4,r4,#100						IF	ID1	ID2	M	AL	WB											
st r6,58(r2)							IF	ID1	ID2	M	AL	WB										
ld r1,50(r2)								IF	ID1	ID2	M	AL	WB									

Όπως βλέπουμε, όλα τα stalls έχουν εξαλειφθεί, ενώ απαιτούνται πλέον 7 κύκλοι για την εκτέλεση μιας επανάληψης.

Άσκηση 4

Θεωρήστε ένα σύστημα μνήμης με μία 4-way set associative cache μεγέθους 256KB, και με cache line 8 λέξεων. Το μέγεθος της λέξης είναι 32 bits. Η μικρότερη μονάδα δεδομένων που μπορεί να διευθυνσιοδοτηθεί είναι το 1 byte, ενώ οι διευθύνσεις μνήμης έχουν εύρος 64 bit.

1) **Για τα επιμέρους πεδία στα οποία χωρίζεται μία διεύθυνση μνήμης σε μία τέτοια οργάνωση cache, υπολογίστε τον αριθμό των bits του καθενός. Παρουσιάστε ένα διάγραμμα που να δείχνει πώς διαχωρίζεται η διεύθυνση στα πεδία αυτά, και εξηγείστε τη σημασία του καθενός.**



Τα επιμέρους πεδία στα οποία χωρίζεται μία διεύθυνση μνήμης σε μία set associative cache παρουσιάζονται στο παραπάνω διάγραμμα. Ο πρώτος διαχωρισμός που γίνεται είναι ανάμεσα στη διεύθυνση του block και στο block offset, το οποίο εκφράζει τη θέση του περιεχομένου της διεύθυνσης μνήμης μέσα στο block. Η διεύθυνση του block διαχωρίζεται περαιτέρω στο tag και στο index. Το index υποδηλώνει το σύνολο της cache στο οποίο απεικονίζεται το block. Το πεδίο tag χρησιμοποιείται για να καθορίσουμε αν το block βρίσκεται στην cache (hit), συγκρίνοντάς το με τα tags όλων των blocks του συνόλου του.

Στην περίπτωσή μας, το μέγεθος ενός block είναι 8*4=32 bytes. Επομένως, αφού η μικρότερη μονάδα που μπορεί να διευθυνσιοδοτηθεί είναι το 1 byte, για το block offset θα απαιτούνται $\log_2 32 = 5$ bits. Κάθε σύνολο στην cache αποτελείται από 4*32 bytes, οπότε υπάρχουν συνολικά 256KB/(4*32) = 2048 σύνολα. Έτσι το μέγεθος του index είναι $\log_2 2048 = 11$ bits. Τα υπόλοιπα 64-5-11=48 bits χρησιμοποιούνται για το tag.

2) **Σε ποιες θέσεις της cache μπορεί να απεικονιστεί το byte στη διεύθυνση μνήμης 5000_{10} ;**

Είναι: $5000_{10} = 1001110001000_2$ η οποία διασπάται στα επιμέρους πεδία tag, index, block offset ως εξής: [000...0000] [00010011100] [01000] . Καταρχήν το byte θα βρίσκεται στην 8η θέση του block. Το block αυτό, μπορεί μετέπειτα να απεικονιστεί σε οποιαδήποτε από τις 4 θέσεις του συνόλου $10011100_2 = 156$ της cache μιας και αυτή είναι set assosiative.

3) **Ποιες θέσεις μνήμης μπορούν να απεικονιστούν στο σύνολο 244 της cache;**

Στο σύνολο 244 απεικονίζονται όλες οι διευθύνσεις που τα bits τους [15:5] (index) ταυτίζονται με το $00011110100_2 = 244$.

4) **Τι ποσοστό του συνολικού μεγέθους της cache αφιερώνεται για τα bits του tag;**

Ένα cache block αποτελείται από 256 bits δεδομένων, και του αντιστοιχεί 1 tag. Επομένως (έστω ότι δε λαμβάνουμε υπόψη το valid bit), το ποσοστό του μεγέθους της cache που αφιερώνεται για τα bits του tag είναι 48/(48+256) = 15.78% .

Άσκηση 5

Έχουμε ένα σύστημα που αποτελείται από έναν επεξεργαστή με in-oder εκτέλεση εντολών, συχνότητα λειτουργίας 1.5GHz και CPI ίσο με 1 (στην περίπτωση όπου δεν υπάρχουν προσβάσεις στη μνήμη). Οι μόνες εντολές που διαβάζουν/γράφουν δεδομένα από/προς τη μνήμη, είναι loads (40% κατά μέσο όρο επί του συνόλου των εντολών) και stores (10% επί του συνόλου των εντολών).
Η ιεραρχία μνήμης αποτελείται από τα εξής στοιχεία:
- Κρυφή μνήμη εντολών επιπέδου 1 (L1I): direct mapped, μεγέθους 32KB, ποσοστό αστοχίας 5%, με μπλοκς των 64 bytes. Ο χρόνος που η L1I ικανοποεί ένα hit ισούται με 2 nsec.
- Κρυφή μνήμη δεδομένων επιπέδου 1 (L1D): direct mapped, write-through, no-write-allocate, μεγέθους 32KB, ποσοστό αστοχίας 8%, με μπλοκς των 64 bytes. Ο χρόνος που η L1D ικανοποεί ένα hit ισούται με 2 nsec.
- Για να μειωθούν τα "write-stalls", δηλαδή οι καθυστερήσεις που οφείλονται στο ότι ο επεξεργαστής πρέπει να περιμένει να ολοκληρωθούν οι "write-through" εγγραφές της L1D, το σύστημα μνήμης διαθέτει έναν ειδικό buffer εγγραφής, που επιτρέπει στον επεξεργαστή να συνεχίσει μόλις τα δεδομένα εγγραφούν σε αυτόν. Με αυτό τον τρόπο γίνεται επικάλυψη της εκτέλεσης και της εγγραφής στη μνήμη. Το 96% των εγγραφών της L1D, βρίσκουν ελεύθερη θέση στον buffer άμεσα. Το υπόλοιπο 4%, πρέπει να περιμένουν μέχρι να ελευθερωθεί κάποια θέση στον buffer. Σε αυτή την περίπτωση, ο buffer κάνει στην ουσία μία αίτηση στην L2 για να στείλει εκεί τα δεδομένα κάποιας θέσης του, και η θέση αυτή αποδεσμεύεται μόλις η L2 είναι έτοιμη να δεχτεί τα δεδομένα αυτά. Αν δεν υπάρχουν ελεύθερες θέσεις στον buffer, ο επεξεργαστής θα πρέπει να περιμένει όποτε κάνει κάποια εγγραφή.
- Ενοποιημένη κρυφή μνήμη εντολών και δεδομένων επιπέδου 2 (L2): write-back, write-allocate, μεγέθους 1024 KB, με μπλοκς των 64 bytes. Ο χρόνος που η L2 ικανοποιεί ένα hit είναι 22nsec. Αυτός είναι και ο χρόνος εγγραφής μιας λέξης στην L2. Το ποσοστό επιτυχίας της είναι 84%. Επιπλέον, το 50% όλων των μπλοκς της που αντικαθιστώνται είναι "dirty".

Οι διευθύνσεις στο σύστημά μας έχουν εύρος 64 bits. Ο χρόνος πρόσβασης σε μία διεύθυνση της κύριας μνήμης (είτε λόγω ανάγνωσης είτε λόγω "write-back") είναι 50nsec. Ο δίαυλος δεδομένων μεταξύ μνήμης-επεξεργαστή έχει εύρος 64 bits, λειτουργεί στα 100 MHz, και μπορεί να μεταφέρει 64 bits σε κάθε κύκλο διαύλου.

Σημείωση: στα ερωτήματα που ακολουθούν, όπου ζητούνται ποσοστά αστοχίας για κάποια κρυφή μνήμη, δώστε τα "τοπικά" ποσοστά (δηλαδή, από την οπτική γωνία της μνήμης αυτής).

Α) Υπολογισμός μέσου χρόνου πρόσβασης στη κύρια μνήμη για ανάγνωση εντολών:

i) **Για την ανάγνωση εντολών, δώστε τις τιμές για τα ακόλουθα μεγέθη:**
χρόνος ικανοποίησης hit από την L1: 2 nsec
ποσοστό αστοχίας στην L1: 5%
χρόνος ικανοποίησης hit από την L2: 22 nsec
ποσοστό αστοχίας στην L2: 16%

ii) **Δώστε έναν τύπο που να υπολογίζει την ποινή αστοχίας στην L2, και υπολογίστε την τιμή της για την περίπτωση που εξετάζουμε. Ως ποινή αστοχίας στην L2, εννοούμε τον χρόνο από τη στιγμή που η L2 ζητήσει τα δεδομένα που δεν βρέθηκαν σε αυτήν, μέχρι τα δεδομένα να έρθουν σε αυτήν από την κύρια μνήμη. Για το υποερώτημα αυτό, λάβετε υπόψη σας το γεγονός ότι το 50% όλων των μπλοκς της που αντικαθιστώνται είναι "dirty".**

Ποινή αστοχίας στην L2 = χρόνος πρόσβασης στη θέση μνήμης όπου βρίσκονται τα ζητούμενα δεδομένα + χρόνος μεταφοράς στην L2 του block στο οποίο περιέχονται τα ζητούμενα δεδομένα.
Ο ρυθμός μεταφοράς στο δίαυλο δεδομένων είναι 64 bits/κύκλο διαύλου = 64 bits / 10 nsec = 8 bytes / 10 nsec = 0.8 bytes / nsec .
Ο χρόνος για τη μεταφορά ενός cache block στην L2 ισούται με 64 bytes/0.8 bytes = 80 nsec.

Επομένως, η ποινή αστοχίας στην L2 ισούται με 50 nsec + 80 nsec = 130 nsec.
Όμως, όταν ένα block έρχεται στην L2, έχει πιθανότητα 50% να αντικαταστήσει ένα dirty block, και στην περίπτωση που γίνει αυτό θα πρέπει να περιμένει μέχρι το dirty block γίνει write-back στην κύρια μνήμη πριν γραφτεί στη θέση του. Το write-back, χρειάζεται κι αυτό άλλα 130 nsec για να ολοκληρωθεί. Επομένως, κατά μέσο όρο, η ποινή αστοχίας στην L2 ισούται με 130 + 50%*130 = 195 nsec.

iii) Χρησιμοποιώντας τα μεγέθη που ζητούνται στα i και ii, δώστε έναν τύπο που να υπολογίζει τον μέσο χρόνο πρόσβασης στη μνήμη. Υπολογίστε την τιμή του για το σύστημα μνήμης που εξετάζουμε.

Μέσος χρόνος πρόσβασης = (χρόνος L1I hit) + (%αστοχίας L1I)*(χρόνος L2 hit) + (%αστοχίας L1I)*(%αστοχίας L2)*(ποινή αστοχίας L2) = 2nsec + 5%*22nsec + 5%*16%*195nsec = 2 +1.1 + 1.56 = 4.66 nsec.

<u>Β) Υπολογισμός μέσου χρόνου πρόσβασης στη κύρια μνήμη για ανάγνωση δεδομένων:</u>

i) Για την ανάγνωση δεδομένων, δώστε την τιμή του ποσοστού αστοχίας στην L1:
8%

ii) Δώστε έναν τύπο που να υπολογίζει τον μέσο χρόνο πρόσβασης στην μνήμη. Υπολογίστε την τιμή του για το σύστημα μνήμης που εξετάζουμε.

Μέσος χρόνος πρόσβασης = (χρόνος L1D hit) + (%αστοχίας L1D)*(χρόνος L2 hit) + (%αστοχίας L1D)*(%αστοχίας L2)*(ποινή αστοχίας L2) = 2nsec + 8%*22nsec + 8%*16%*195nsec = 2 + 1.76 + 2.496 = 6.256 nsec.

<u>Γ) Υπολογισμός μέσου χρόνου πρόσβασης στη κύρια μνήμη για εγγραφή δεδομένων:</u>

i) Για την εγγραφή δεδομένων, υπολογίστε την τιμή της ποινής αστοχίας στην L2.

Εφόσον η L2 είναι write-allocate, στην περίπτωση ενός write miss θα πρέπει να γίνουν στην ουσία οι ίδιες ενέργειες με την περίπτωση ενός read miss, να φορτωθεί δηλαδή το block από την κύρια μνήμη στην L2. Επομένως η ποινή αστοχίας για την εγγραφή ισούται με την ποινή αστοχίας για την ανάγνωση, δηλαδή 195 nsec.

ii) Υπολογίστε τον μέσο χρόνο εγγραφής των δεδομένων μιας θέσης του buffer εγγραφής στην L2.

Το ποσοστό επιτυχίας της L2 ισούται με 84%, πράγμα που σημαίνει ότι το 16% των εγγραφών του buffer θα αστοχούν στην L2 και θα "βαρύνονται" με την ποινή αστοχίας της L2. Επομένως, ο ζητούμενος μέσος χρόνος εγγραφής ισούται με: (χρόνος L2 hit) + (%αστοχίας L2)*(ποινή αστοχίας L2) = 22nsec + 16%*195nsec = 53.2 nsec

iii) Χρησιμοποιώντας τα προηγούμενα μεγέθη, δώστε έναν τύπο και υπολογίστε τον μέσο χρόνο πρόσβασης στη μνήμη για εγγραφές δεδομένων. Σημείωση: λάβετε υπόψη σας τις 2 περιπτώσεις για την κατάσταση του buffer εγγραφής (γεμάτος ή όχι).

Στο 96% των περιπτώσεων ο buffer εγγραφής θα έχει ελεύθερη θέση, οπότε ο επεξεργαστής μπορεί να συνεχίσει, έχοντας περιμένει συνολικά 2nsec που απαιτούνται για πρόσβαση στην L1. Στο υπόλοιπο 4% των περιπτώσεων, ο buffer εγγραφής θα είναι γεμάτος, οπότε ο επεξεργαστής θα πρέπει να περιμένει επιπρόσθετα για τον χρόνο που απαιτείται για να εγγραφούν τα δεδομένα μιας θέσης του buffer εγγραφής στην L2. Δηλαδή, μέσος χρόνος πρόσβασης στη μνήμη = χρόνος πρόσβασης στην L1 + 4%*(μέσος χρόνος εγγραφής μιας θέσης του buffer εγγραφής στην L2) = 2nsec + 4%*53.2nsec = 4.128 nsec.

Άσκηση 6

Θεωρείστε το ακόλουθο κομμάτι κώδικα:

```
 1:   int x,y;
 2:   float tmp1, tmp2, arr1[64][64], arr2[64][64];
 3:   for ( x = 0; x < 64; x++ )
 4:   {
 5:        for ( y = 0; y < 64; y++ )
 6:             tmp1 -= arr1[x][y];
 7:
 8:        for ( y = 0; y < 32; y++ )
 9:             tmp2 -= arr2[x][2*y];
10:
11:   }
```

Κάνουμε τις ακόλουθες υποθέσεις:
- Οι τύποι δεδομένων int και float είναι μεγέθους 4 bytes.
- Όλες οι μεταβλητές, πλην των στοιχείων των 2 πινάκων, μπορούν να αποθηκευτούνσε καταχωρητές του επεξεργαστή, οπότε οποιαδήποτε αναφορά σε αυτές δεν συνεπάγεται προσπέλαση στην κρυφή μνήμη. Επιπλέον, δε λαμβάνονται υπόψη οι προσπελάσεις των εντολών στη μνήμη (υποθέτουμε ότι έχουμε τέλεια κρυφή μνήμη εντολών).
- Η κρυφή μνήμη δεδομένων είναι πλήρως συσχετιστική, με LRU πολιτική αντικαστάστασης, αποτελούμενη από 32 γραμμές, με κάθε γραμμή να αποτελείται από 16 bytes. Αρχικά, είναι εντελώς άδεια.
- Οι πίνακες είναι αποθηκευμένοι στην κύρια μνήμη κατά γραμμές ("row-major layout"). Επιπλέον, είναι "ευθυγραμμισμένοι" ώστε το πρώτο στοιχείο του καθενός να απεικονίζεται στην αρχή μιας γραμμής της κρυφής μνήμης.
- Η εκτέλεση των εντολών γίνεται σειριακά. Ο χρόνος που απαιτείται για να εκτελεστεί κάθε φορά κάποια από τις γραμμές 3,5 και 8 ισούται με 5 κύκλους. Οι γραμμές 6 και 9 χρειάζονται 12 κύκλους η κάθε μία, και 50 κύκλους επιπλέον αν σημειωθεί κάποια αστοχία στην κρυφή μνήνη δεδομένων.
- Το σύνολο εντολών της αρχιτεκτονικής του επεξεργαστή διαθέτει μία ειδική εντολή prf(*addr). Η εντολή αυτή προ-φορτώνει στην κρυφή μνήμη ολόκληρο το μπλοκ που περιέχει τη λέξη που βρίσκεται στη διεύθυνση μνήμης addr. Η εντολή εκτελείται σε 1 κύκλο από τον επεξεργαστή, ενώ από το σημείο αυτό και έπειτα, η προφόρτωση των δεδομένων γίνεται παράλληλα και ανεξάρτητα από την υπόλοιπη εκτέλεση του προγράμματος, χωρίς να προκαλείται κάποιο stall στο pipeline του επεξεργαστή. Με άλλα λόγια, ο επεξεργαστής μπορεί να συνεχίσει με την εκτέλεση των εντολών που ακολουθούν. Αν τα δεδομένα που ζητάει η εντολή prf δεν είναι στην κρυφή μνήμη, τότε θεωρούμε ότι απαιτούνται 50 κύκλοι μέχρι να μεταφερθούν σε αυτήν.

α. Πόσοι κύκλοι απαιτούνται για την εκτέλεση του παραπάνω τμήματος κώδικα αν δεν χρησιμοποιήσουμε προφόρτωση δεδομένων;

Για την γραμμή (3), απαιτούνται συνολικά 64*5 = 320 κύκλοι
Για την γραμμή (5), απαιτούνται συνολικά 64*64*5 = 20480 κύκλοι
Για την γραμμή (6), απαιτούνται συνολικά 64*64*12 + 64*64*50/4 = 100352 κύκλοι
Ο δεύτερος όρος αντιστοιχεί στο γεγονός ότι, ανά 4 διαδοχικές αναφορές στο arr1[x][y] έχουμε cache miss. Το miss αυτό προκαλείται από την πρώτη αναφορά (π.χ. arr1[0][0]), εφόσον η cache είναι αρχικά άδεια. Οι επόμενες 3 αναφορές (π.χ. arr1[0][1], arr1[0][2], arr1[0][3]) δεν αστοχούν στην cache, επειδή τα στοιχεία αυτά φορτώθηκαν με την πρώτη αναφορά, διότι βρίσκονται στην ίδια cache line με το πρώτο (λόγω του ότι βρίσκονται σε διαδοχικές θέσεις μνήμης εξαιτίας του row-major layout).

Για την γραμμή (8), απαιτούνται συνολικά 64*32*5 = 10240 κύκλοι

Για την γραμμή (9), απαιτούνται συνολικά 64*32*12 + 64*32*50/2 = 75776 κύκλοι
Κι εδώ ισχύει το ίδιο με πριν, μόνο που τώρα έχουμε miss και φόρτωμα νέας cache line ανά 2 διαδοχικές αναφορές στο arr2[x][2*y] (δηλαδή γίνονται π.χ. διαδοχικά οι αναφορές arr2[0][0], arr2[0][2], arr2[0][4], arr2[0][6], εκ των οποίων οι 2 πρώτες ανήκουν στην ίδια cache line ενώ οι 2 επόμενες στην επόμενη).

Συνολικά, απαιτούνται 320+20480+100352+10240+75776=207168 κύκλοι.

β. Θεωρείστε την περίπτωση εισαγωγής εντολών prf στα δύο εσωτερικότερα loops του κώδικα. Εξηγείστε γιατί είναι απαραίτητο σε αυτή την περίπτωση να εφαρμόσουμε στα loops αυτά την τεχνική του ξεδιπλώματος βρόχων ("loop unrolling"). Ποιος είναι ο ελάχιστος αριθμός φορών που χρειάζεται να "ξεδιπλώσουμε" κάθε ένα από τα 2 loops για το σκοπό αυτό;

Μία cache line αποτελείται από 16 bytes, οπότε μπορεί να χωρέσει 4 floats. Επομένως, μια εντολή prf, θα φορτώσει 4 διαδοχικά στοιχεία του πίνακα στην cache. Έτσι, για το 1ο εσωτερικό loop του κώδικα, χρειάζεται στην ουσία να κάνουμε προφόρτωση κάθε 4 επαναλήψεις, οπότε αρκεί να ξεδιπλώσουμε το loop 4 φορές. Για το 2ο εσωτερικό loop, χρειάζεται να κάνουμε προφόρτωση κάθε 2 επαναλήψεις, οπότε αρκεί να το ξεδιπλώσουμε 2 φορές.

γ. Εφαρμόστε unrolling στα εσωτερικά loops, για τον αριθμό φορών που απαντήσατε στο προηγούμενο ερώτημα. Εισάγετε τον μικρότερο δυνατό αριθμό εντολών prf στα κατάλληλα σημεία του "ξεδιπλωμένου" κώδικα, ώστε να ελαχιστοποιηθεί ο χρόνος εκτέλεσης. (Σημείωση: μην λάβετε ειδική μέριμνα για τις αρχικές επαναλήψεις των loops)

Ο κώδικας που προκύπτει μετά το ξεδίπλωμα είναι ο ακόλουθος:

```
 1:   int x,y;
 2:   float tmp1, tmp2, arr1[64][64], arr2[64][64];
 3:   for ( x = 0; x < 64; x++ )
 4:   {
 5:        for ( y = 0; y < 64; y+=4 ) {
 6:             prf(arr1[x][y+4]);
 7:             tmp1 -= arr1[x][y];
 8:             tmp1 -= arr1[x][y+1];
 9:             tmp1 -= arr1[x][y+2];
10:             tmp1 -= arr1[x][y+3];
11:        }
12:
13:        for ( y = 0; y < 32; y+=2 ) {
14:             prf(arr2[x][2*(y+4)]);
15:             tmp2 -= arr2[x][2*y];
16:             tmp2 -= arr2[x][2*(y+1)];
17:
18:   }
```

Για το πρώτο loop, η prf προφορτώνει δεδομένα του arr1 που θα χρειαστούν στην επόμενη επανάληψη. Τα δεδομένα αυτά θα έχουν φτάσει όντως στην cache πριν γίνει στην επόμενη επανάληψη η πρώτη αναφορά (arr1[x][y]) σε αυτά.
Δηλαδή, αν υποθέσουμε ότι είμαστε στην επανάληψη x=0, y=8 και στον κύκλο k, τότε:

```
k:      prf(arr1[0][12]);     /*τα δεδομένα θα φτάσουν στην cache στον κύκλο k+50 */
k+1:    tmp1 -= arr1[0][8]; /*τα δεδομένα έχουν προφορτωθεί από την προηγούμενη επανάληψη,*/
k+13:   tmp1 -= arr1[0][9]; /*οπότε κάθε μία από τις γραμμές 7-10 θα εκτελεστεί σε 12 κύκλους*/
k+25:   tmp1 -= arr1[0][10];
k+37:   tmp1 -= arr1[0][11];
```

k+49: for(...)
k+54: prf(arr1[0][16]);
k+55: tmp1 -= arr1[0][12]; /*η cache line που περιέχει τα στοιχεία arr1[0][12],...,arr1[0][15] */
/*βρίσκεται ήδη στην cache από τον κύκλο k+50*/

Για το δεύτερο loop, η prf προφορτώνει δεδομένα του arr2 που θα χρειαστούν στην μεθεπόμενη επανάληψη, διότι το σώμα το 2ου loop, με τις 2 φορές που έχει ξεδιπλωθεί, θα εκτελεστεί σε λιγότερο χρόνο απ'ό,τι χρειάζεται για να φέρει η prf τα δεδομένα. Αντιθέτως, 2 συνεχόμενες επαναλήψεις του loop, διαρκούν περισσότερο από το χρόνο αυτό (λαμβάνοντας υπόψη, όπως και προηγουμένως, το χρόνο για να εκτελεστεί η ίδια η prf και το χρόνο για να εκτελεστούν οι εντολές ελέγχου του loop), οπότε η τρίτη κατά σειρά επανάληψη θα βρει έτοιμα στην cache τα δεδομένα που ζήτησε η prf στην πρώτη επανάληψη.

δ. Πόσοι κύκλοι απαιτούνται τώρα για την εκτέλεση του κώδικα που προέκυψε από το προηγούμενο ερώτημα; Υπολογίστε το ποσοστό % της βελτίωσης του χρόνου εκτέλεσης ("speedup") σε σχέση με τον αρχικό, μη βελτιστοποιημένο κώδικα.

Για τη γραμμή (3), απαιτούνται συνολικά 64*5 κύκλοι
Για τη γραμμή (5), απαιτούνται συνολικά 64*16*5 κύκλοι
Για τη γραμμή (6), απαιτούνται συνολικά 64*16*1 κύκλοι
Για τη γραμμή (7), απαιτούνται συνολικά 64*1*(12+50) + 64*15*12 κύκλοι
Αυτό συμβαίνει διότι κάθε πρώτη επανάληψη του εσωτερικού loop, θα έχουμε cache miss, αφού προφορτώνουμε δεδομένα μόνο για τις επόμενες επαναλήψεις.
Κάθε μία από τις γραμμές (8)-(10), απαιτούν συνολικά 64*16*12 κύκλους.

Για τη γραμμή (13), απαιτούνται συνολικά 64*16*5 κύκλοι
Για τη γραμμή (14), απαιτούνται συνολικά 64*16*1 κύκλοι
Για τη γραμμή (15), απαιτούνται συνολικά 64*2*(12+50) + 64*14*12 κύκλοι
Ομοίως, στις 2 πρώτες επαναλήψεις του 2ου εσωτερικού loop θα έχουμε misses στις αναφορές που κάνει η γραμμή (15). Στις επόμενες επαναλήψεις, οι αναφορές θα βρίσκουν τα δεδομένα προφορτωμένα στην cache.
Για τη γραμμή (16), απαιτούνται συνολικά 64*16*12 κύκλοι

Συνολικά, οι κύκλοι που απαιτούνται είναι 95936.
Το speedup σε σχέση με τον αρχικό κώδικα είναι 207168/95936 = 2.16%

ε. Υπάρχει άλλος τρόπος, εκτός του loop unrolling, να γραφτεί το αρχικό πρόγραμμα, ώστε να επιτυγχάνεται μεν ο σκοπός που εξυπηρετεί το unrolling, αλλά χρησιμοποιώντας αυτή τη φορά λιγότερες εντολές σε σχέση με τον κώδικα του ερωτήματος "γ";

Μπορούμε να χρησιμοποιήσουμε εντολές if ώστε να καθορίσουμε ανά πόσες επαναλήψεις πρέπει να γίνεται η προφόρτωση δεδομένων, χωρίς να ξεδιπλώσουμε το σώμα των loops, χρησιμοποιώντας έτσι λιγότερες εντολές στο πρόγραμμα.

```
1:  int x,y;
2:  float tmp1, tmp2, arr1[64][64], arr2[64][64];
3:  for ( x = 0; x < 64; x++ )
4:  {
5:        for ( y = 0; y < 64; y++ ) {
6:             if(y%4 == 0)
7:                  prf(arr1[x][y+4];
8:             tmp1 -= arr1[x][y];
```

```
 9:        }
10:
11:        for ( y = 0; y < 32; y++ ){
12:             if(y%2 == 0)
13:                  prf(arr2[x][2*(y+4)];
14:             tmp2 -= arr2[x][2*y];
15:
16:   }
```